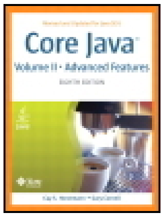


Core Java™ Volume II–Advanced Features, Eighth Edition
by Cay S. Horstmann; Gary Cornell

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.

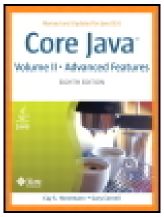
Volume II includes new sections on the StAX API, JDBC 4, compiler API, scripting framework, splash screen and tray APIs, and many other Java SE 6 enhancements. In this book, the authors focus on the more advanced features of the Java language, including complete coverage of

- 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
The Compiler API
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

UNREGISTERED VERSION OF CHM TO PDF CONVERTER PRO BY THETA-SOFTWARE

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

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++.

API

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 Philion, 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.

API

`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:	`b`	The array from which to write the data
	`off`	The offset into `b` to the first byte that will be written
	`len`	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]

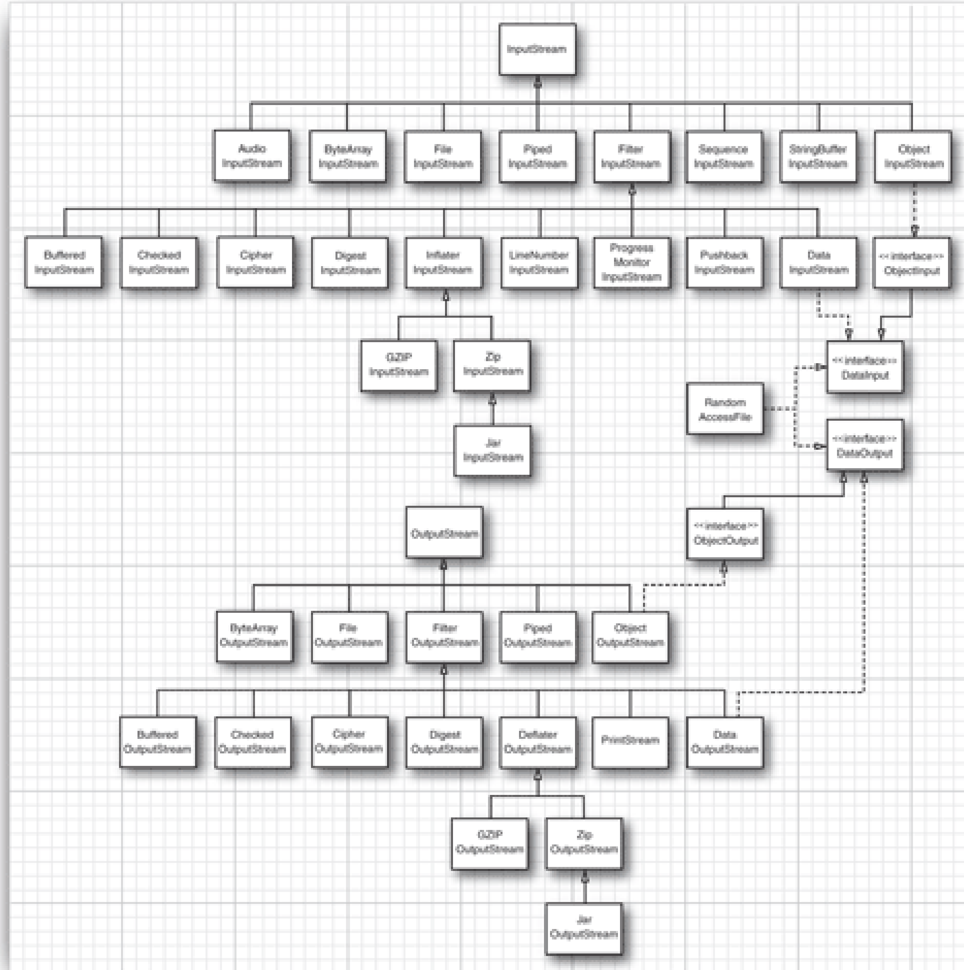
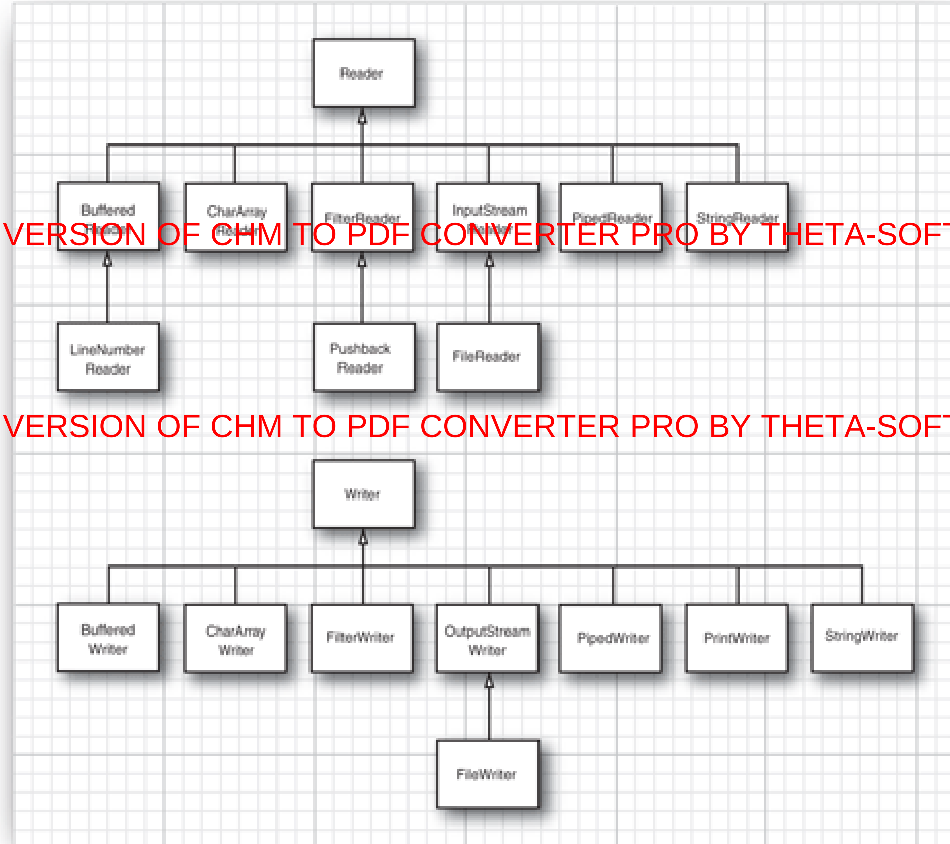


Figure 1-2. Reader and writer hierarchy

[View full size image]



UNREGISTERED VERSION OF CHM TO PDF CONVERTER PRO BY THETA-SOFTWARE

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 hiearchy 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
```

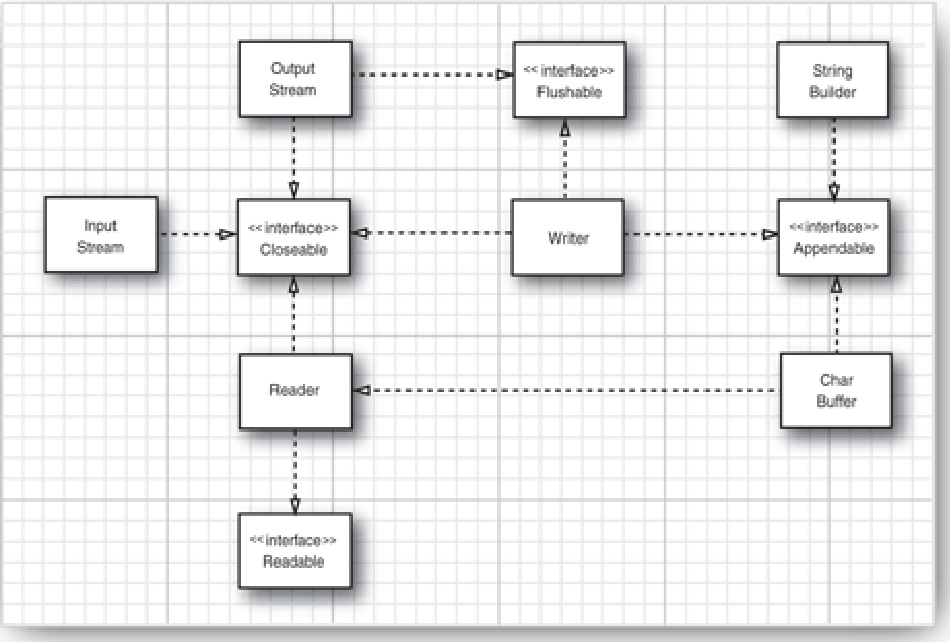
and

```
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.

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.

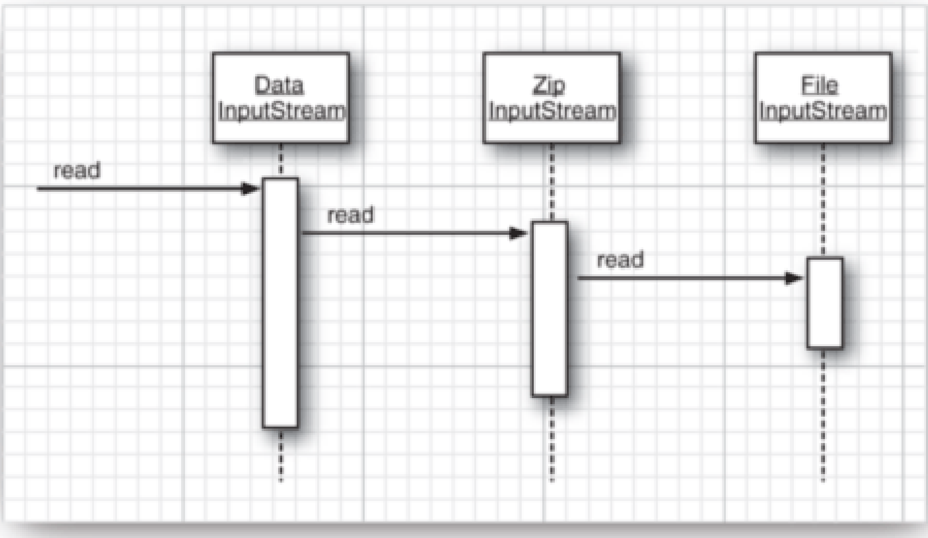
```
DataInputStream din = new DataInputStream(
   pbin = new PushbackInputStream(
      new BufferedInputStream(
         new FileInputStream("employee.dat"))));
```

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.)

API 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.

API 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.

`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.

`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.

`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:	`b`	The array from which to write the data
	`off`	The offset into `b` to the first byte that will be written
	`len`	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]

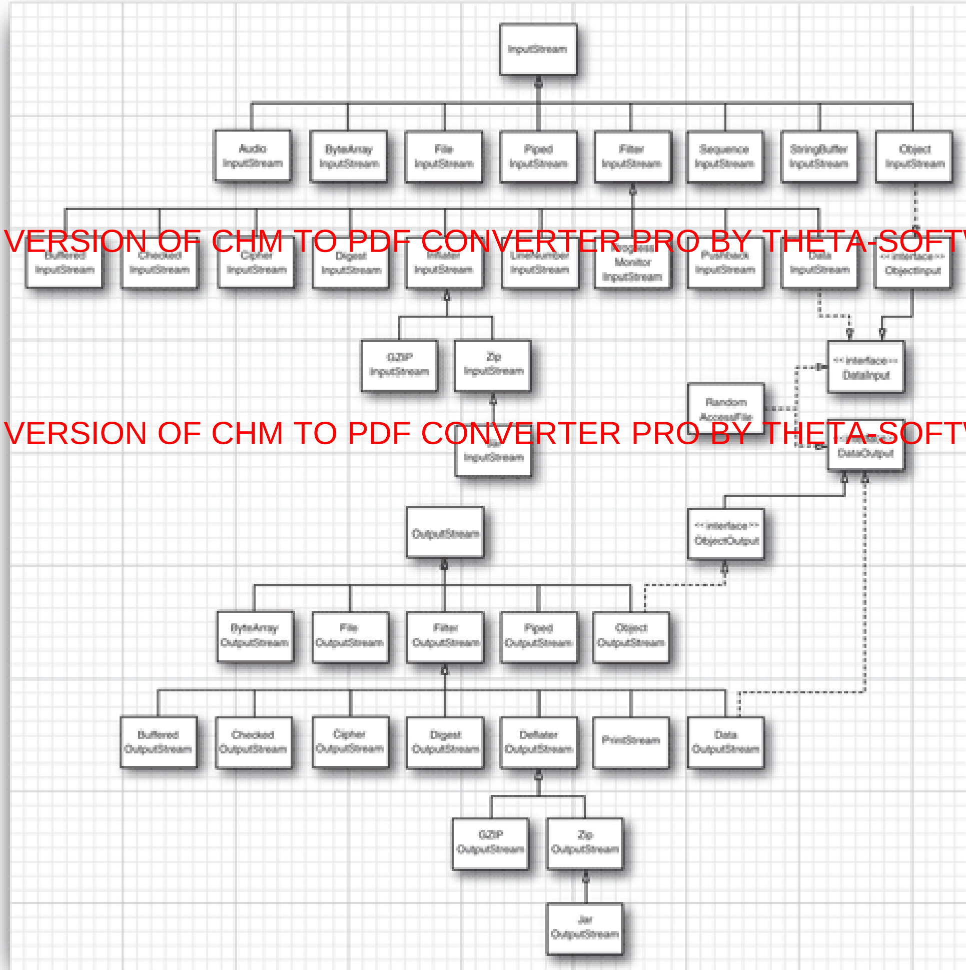
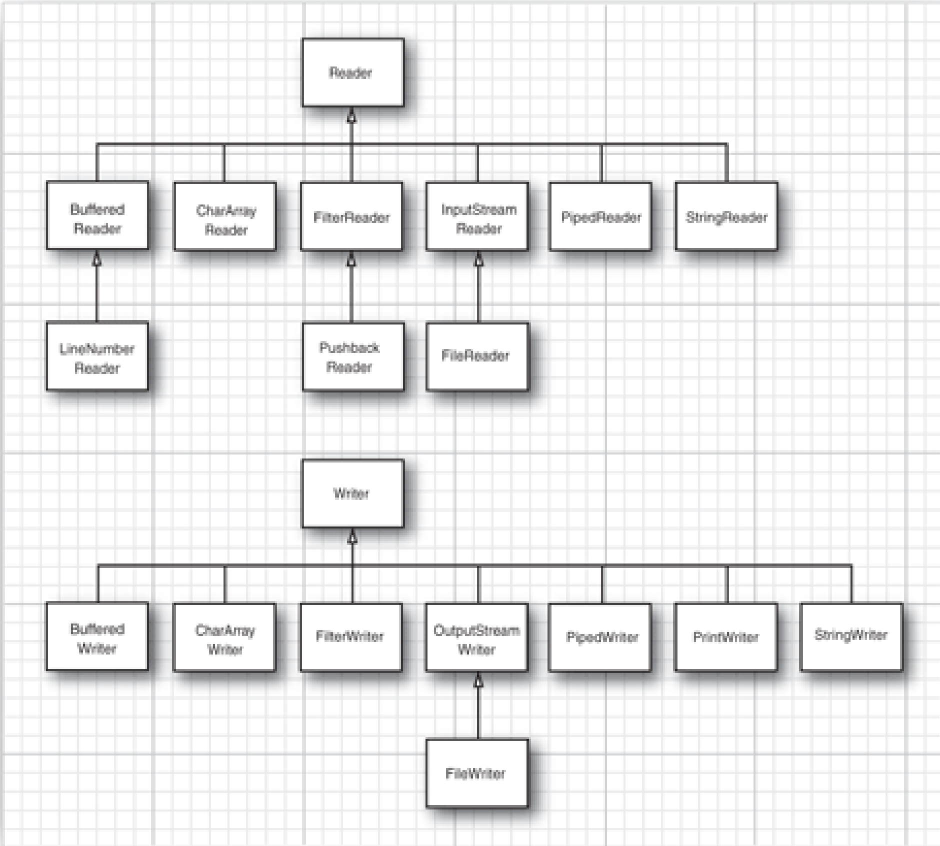


Figure 1-2. Reader and writer hierarchy

[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 hiearchy 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
```

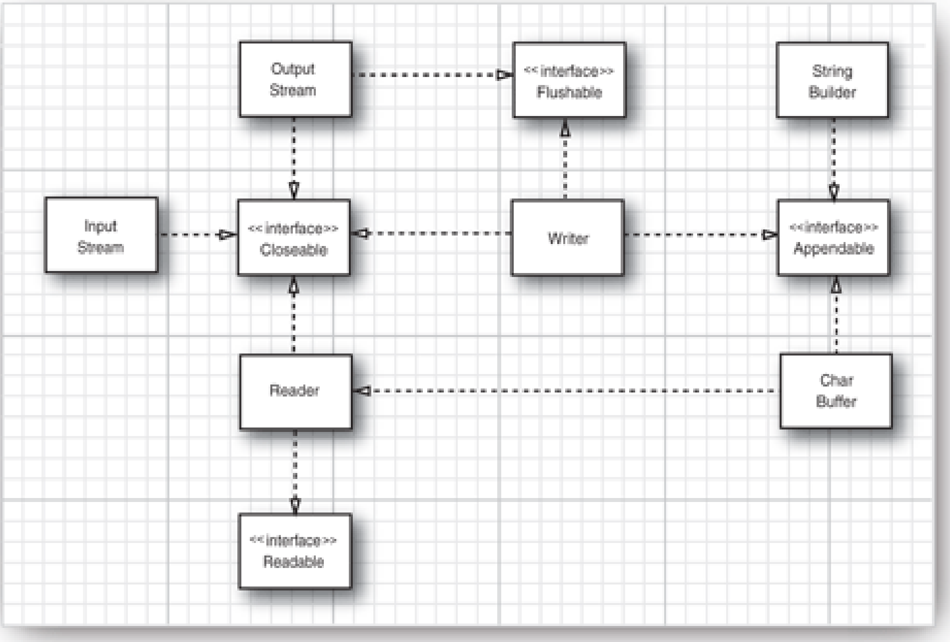
and

```
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`.

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"))));
```

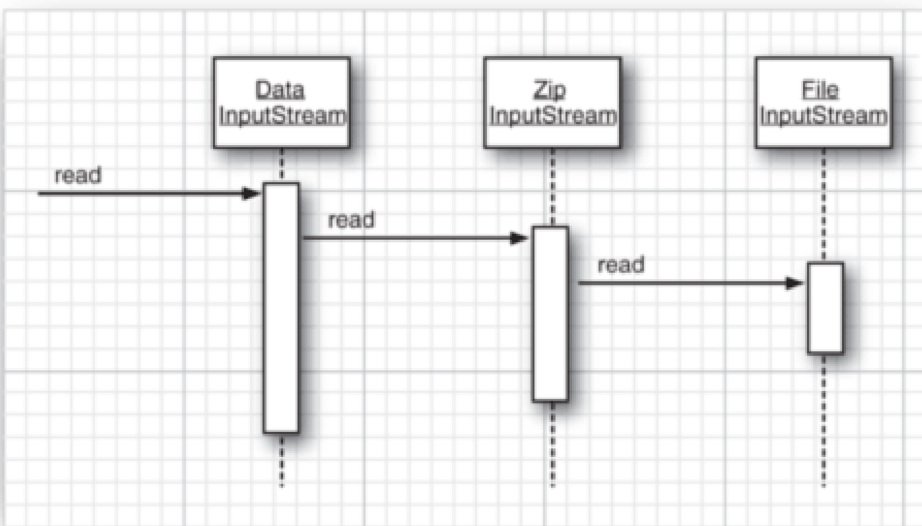
Of course, in the stream libraries of other programming languages, niceties such as buffering and lookahead are automatically taken care of, so it is a bit of a hassle in Java that one has to resort to combining stream filters in these cases. But the ability to mix and match filter classes to construct 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.)

API

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.

API

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.

`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.

`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.

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:

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

/**
 * @version 1.12 2007-06-22
 * @author Cay Horstmann
 */
public class TextFileTest
{
   public static void main(String[] args)
   {
      Employee[] staff = new Employee[3];

```

```
      staff[0] = new Employee("Carl Cracker", 75000, 1987, 12, 15);
      staff[1] = new Employee("Harry Hacker", 50000, 1989, 10, 1);
      staff[2] = new Employee("Tony Tester", 40000, 1990, 3, 15);

      try
      {
         // save all employee records to the file employee.dat
         PrintWriter out = new PrintWriter("employee.dat");
         writeData(staff, out);
         out.close();

         // retrieve all records into a new array
         Scanner in = new Scanner(new FileReader("employee.dat"));
         Employee[] newStaff = readData(in);
         in.close();

         // print the newly read employee records
         for (Employee e : newStaff)
            System.out.println(e);
      }
      catch (IOException exception)
      {
         exception.printStackTrace();
      }
   }

   /**
    * Writes all employees in an array to a print writer
    * @param employees an array of employees
    * @param out a print writer
    */
   private static void writeData(Employee[] employees, PrintWriter out) throws IOException
   {
      // write number of employees
      out.println(employees.length);

      for (Employee e : employees)
         e.writeData(out);
   }
   /**
    * Reads an array of employees from a scanner
    * @param in the scanner
    * @return the array of employees
    */
   private static Employee[] readData(Scanner in)
   {
      // retrieve the array size
      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);
      }
      return employees;
   }
}
```

```

class Employee
{
   public Employee()
   {
   }

   public Employee(String n, double s, int year, int month, int day)
   {
```



```
      name = n;
      salary = s;
      GregorianCalendar calendar = new GregorianCalendar(year, month - 1, day);
      hireDay = calendar.getTime();
   }

   public String getName()
```



```
   {
      return name;
   }

   public double getSalary()
   {
      return salary;
   }

   public Date getHireDay()
   {
      return hireDay;
   }

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

   public String toString()
   {
      return getClass().getName() + "[name=" + name + ",salary=" + salary + ",hireDay="
            + hireDay + "]";
   }

   /**
    * Writes employee data to a print writer
    * @param out the print writer
    */
   public void writeData(PrintWriter out)
   {
      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));
   }

   /**
    * Reads employee data from a buffered reader
    * @param in the scanner
    */
   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();
   }

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

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
l1
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

Charset Standard Name	Legacy Name	Description
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, 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

Charset Standard Name	Legacy Name	Description
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
x-windows-950	MS950	Windows Traditional Chinese

UNREGISTERED VERSION OF CHM TO PDF CONVERTER PRO BY THETA-SOFTWARE

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();
```

`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'`).

`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 x 256 + 13 x 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`	$0a_6a_5a_4a_3a_2a_1a_0$
`80...7FF`	$110a_{10}a_9a_8a_7a_6\ 10a_5a_4a_3a_2a_1a_0$
`800...FFFF`	$1110a_{15}a_{14}a_{13}a_{12}\ 10a_{11}a_{10}a_9a_8a_7a_6\ 10a_5a_4a_3a_2a_1a_0$
`10000...10FFFF`	$11110a_{20}a_{19}a_{18}\ 10a_{17}a_{16}a_{15}a_{14}a_{13}a_{12}\ 10a_{11}a_{10}a_9a_8a_7a_6\ 10a_5a_4a_3a_2a_1a_0$

Table 1-5. UTF-16 Encoding

Character Range	Encoding
`0...FFFF`	$a_{15}a_{14}a_{13}a_{12}a_{11}a_{10}a_9a_8\ a_7a_6a_5a_4a_3a_2a_1a_0$
`10000...10FFFF`	$110110b_{19}b_{18}\ b_{17}b_{16}a_{15}a_{14}a_{13}a_{12}a_{11}a_{10}\ 110111a_9a_8\ a_7a_6a_5a_4a_3a_2a_1a_0$ 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"));
```

API `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

API `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:

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

/**
 * @version 1.11 2004-05-11
 * @author Cay Horstmann
 */

public class RandomFileTest
{
   public static void main(String[] args)
   {
      Employee[] staff = new Employee[3];

      staff[0] = new Employee("Carl Cracker", 75000, 1987, 12, 15);
      staff[1] = new Employee("Harry Hacker", 50000, 1989, 10, 1);
      staff[2] = new Employee("Tony Tester", 40000, 1990, 3, 15);

      try
      {
         // save all employee records to the file employee.dat
         DataOutputStream out = new DataOutputStream(new FileOutputStream("employee.dat"));
         for (Employee e : staff)
            e.writeData(out);
         out.close();

         // retrieve all records into a new array
         RandomAccessFile in = new RandomAccessFile("employee.dat", "r");
         // compute the array size
         int n = (int)(in.length() / Employee.RECORD_SIZE);
         Employee[] newStaff = new Employee[n];

```

```
         // read employees in reverse order
         for (int i = n - 1; i >= 0; i--)
         {
            newStaff[i] = new Employee();
            in.seek(i * Employee.RECORD_SIZE);
            newStaff[i].readData(in);
         }
         in.close();

```



```
         // print the newly read employee records
         for (Employee e : newStaff)
            System.out.println(e);
      }
      catch (IOException e)
      {
```



```
         e.printStackTrace();
      }
   }
}

class Employee
{
   public Employee() {}

   public Employee(String n, double s, int year, int month, int day)
   {
      name = n;
      salary = s;
      GregorianCalendar calendar = new GregorianCalendar(year, month - 1, day);
      hireDay = calendar.getTime();
   }

   public String getName()
   {
      return name;
   }

   public double getSalary()
   {
      return salary;
   }

   public Date getHireDay()
   {
      return hireDay;
   }

   /**
      Raises the salary of this employee.
      @byPercent the percentage of the raise
   */
   public void raiseSalary(double byPercent)
   {
      double raise = salary * byPercent / 100;
      salary += raise;
   }

   public String toString()
   {
```

```
      return getClass().getName()
         + "[name=" + name
         + ",salary=" + salary
         + ",hireDay=" + hireDay
         + "]";
   }

   /**
      Writes employee data to a data output
      @param out the data output
   */
   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));
   }

   /**
      Reads employee data from a data input
      @param in the data input
   */
   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();
   }

   public static final int NAME_SIZE = 40;
   public static final int RECORD_SIZE = 2 * NAME_SIZE + 8 + 4 + 4 + 4;

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

class DataIO
{
   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();
   }

   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);
      }
   }
}
```


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.

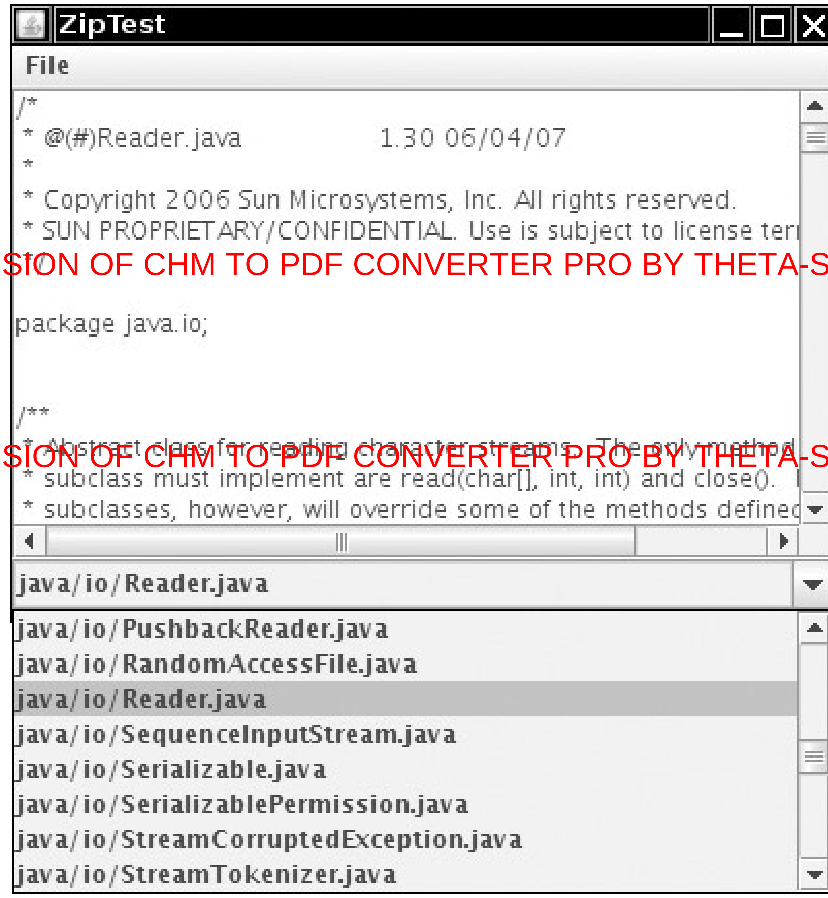
ZIP streams are a good example of the power of the stream abstraction. When you read the data that are stored in compressed form, you don't worry that the data are being decompressed as they are being requested. And the source of the bytes in ZIP formats need not be a file—the ZIP data can come from 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:
```
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.util.*;
import java.util.List;
import java.util.zip.*;
import javax.swing.*;

/**
 * @version 1.32 2007-06-22
 * @author Cay Horstmann
 */
public class ZipTest
{
   public static void main(String[] args)
   {
      EventQueue.invokeLater(new Runnable()
         {
            public void run()
            {
               ZipTestFrame frame = new ZipTestFrame();
               frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
               frame.setVisible(true);
```

```
            }
         });
   }
}

/**
 * A frame with a text area to show the contents of a file inside a ZIP archive, a combo
 * box to select different files in the archive, and a menu to load a new archive.
 */
class ZipTestFrame extends JFrame
{
   public ZipTestFrame()
   {
      setTitle("ZipTest");
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

      // add the menu and the Open and Exit menu items
      JMenuBar menuBar = new JMenuBar();
      JMenu menu = new JMenu("File");

      JMenuItem openItem = new JMenuItem("Open");
      menu.add(openItem);
      openItem.addActionListener(new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               JFileChooser chooser = new JFileChooser();
               chooser.setCurrentDirectory(new File("."));
               int r = chooser.showOpenDialog(ZipTestFrame.this);
               if (r == JFileChooser.APPROVE_OPTION)
               {
                  zipname = chooser.getSelectedFile().getPath();
                  fileCombo.removeAllItems();
                  scanZipFile();
               }
            }
         });

      JMenuItem exitItem = new JMenuItem("Exit");
      menu.add(exitItem);
      exitItem.addActionListener(new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               System.exit(0);
            }
         });

      menuBar.add(menu);
      setJMenuBar(menuBar);

      // add the text area and combo box
      fileText = new JTextArea();
      fileCombo = new JComboBox();
      fileCombo.addActionListener(new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               loadZipFile((String) fileCombo.getSelectedItem());
```

```
            }
         });

      add(fileCombo, BorderLayout.SOUTH);
      add(new JScrollPane(fileText), BorderLayout.CENTER);
   }

   /**
    * Scans the contents of the ZIP archive and populates the combo box.
    */
   public void scanZipFile()
   {
      new SwingWorker<Void, String>()
         {
            protected Void doInBackground() throws Exception
            {
               ZipInputStream zin = new ZipInputStream(new FileInputStream(zipname));
               ZipEntry entry;
               while ((entry = zin.getNextEntry()) != null)
               {
                  publish(entry.getName());
                  zin.closeEntry();
               }
               zin.close();
               return null;
            }

            protected void process(List<String> names)
            {
               for (String name : names)
                  fileCombo.addItem(name);

            }
         }.execute();
   }

   /**
    * Loads a file from the ZIP archive into the text area
    * @param name the name of the file in the archive
    */
    public void loadZipFile(final String name)
    {
       fileCombo.setEnabled(false);
       fileText.setText("");
       new SwingWorker<Void, Void>()
          {
             protected Void doInBackground() throws Exception
             {
                try
                {
                   ZipInputStream zin = new ZipInputStream(new FileInputStream(zipname));
                   ZipEntry entry;

                   // find entry with matching name in archive
                   while ((entry = zin.getNextEntry()) != null)
                   {
                      if (entry.getName().equals(name))
                      {
                         // read entry into text area
```

```
                         Scanner in = new Scanner(zin);
                         while (in.hasNextLine())
                         {
                            fileText.append(in.nextLine());
                            fileText.append("\n");
                         }
                      }
                      zin.closeEntry();
                   }
                   zin.close();
                }
                catch (IOException e)
                {
                   e.printStackTrace();
                }
                return null;
             }

             protected void done()
             {
                fileCombo.setEnabled(true);
             }
          }.execute();
    }

    public static final int DEFAULT_WIDTH = 400;
    public static final int DEFAULT_HEIGHT = 300;
    private JComboBox fileCombo;
    private JTextArea fileText;
    private String zipname;
}
```

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()`.

API 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:

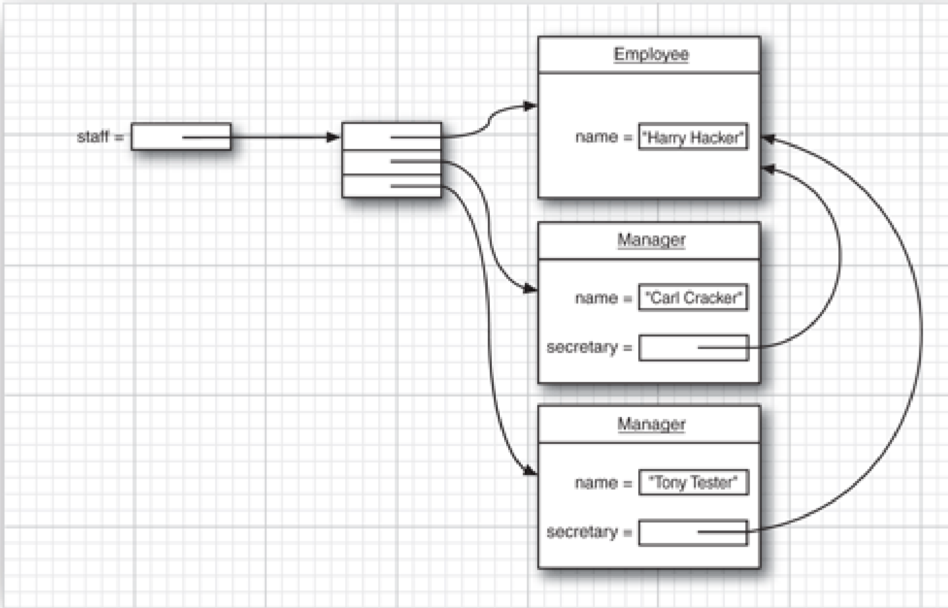
```
class Manager extends Employee
{
   . . .
   private Employee secretary;
}
```

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]



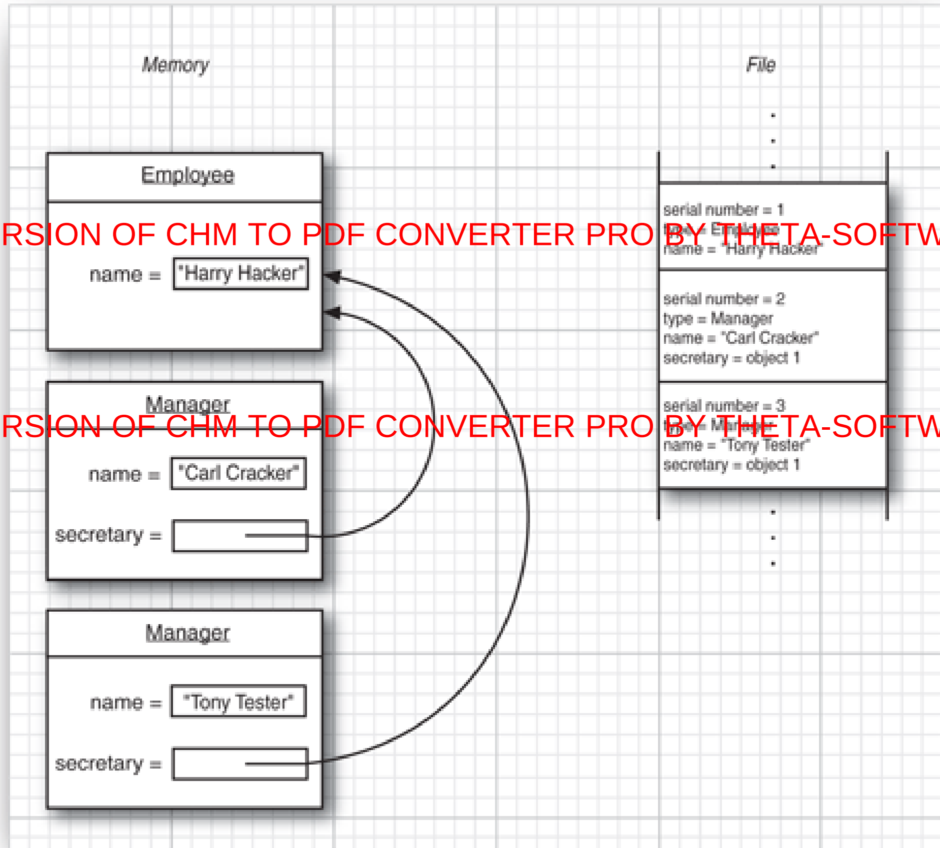
Saving such a network of objects is a challenge. Of course, we cannot save and restore the memory addresses for the secretary objects. When an object is reloaded, it will likely occupy a completely different memory address than it originally did.

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]



UNREGISTERED VERSION OF CHM TO PDF CONVERTER PRO BY THETA-SOFTWARE

- 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:

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

/**
 * @version 1.10 17 Aug 1998
 * @author Cay Horstmann
 */
class ObjectStreamTest
{
   public static void main(String[] args)
   {
      Employee harry = new Employee("Harry Hacker", 50000, 1989, 10, 1);
      Manager carl = new Manager("Carl Cracker", 80000, 1987, 12, 15);
      carl.setSecretary(harry);
      Manager tony = new Manager("Tony Tester", 40000, 1990, 3, 15);
      tony.setSecretary(harry);

      Employee[] staff = new Employee[3];

      staff[0] = carl;
      staff[1] = harry;
      staff[2] = tony;

      try
      {
         // save all employee records to the file employee.dat
        ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream("employee.dat"));
         out.writeObject(staff);
         out.close();

         // retrieve all records into a new array
         ObjectInputStream in = new ObjectInputStream(new FileInputStream("employee.dat"));
         Employee[] newStaff = (Employee[]) in.readObject();
         in.close();

         // raise secretary's salary
         newStaff[1].raiseSalary(10);
```

```

         // print the newly read employee records
         for (Employee e : newStaff)
            System.out.println(e);
      }
      catch (Exception e)
      {
         e.printStackTrace();
      }
```

```
   }
}

class Employee implements Serializable
{
   public Employee()
```

```
   {
   }

   public Employee(String n, double s, int year, int month, int day)
   {
      name = n;
      salary = s;
      GregorianCalendar calendar = new GregorianCalendar(year, month - 1, day);
      hireDay = calendar.getTime();
   }

   public String getName()
   {
      return name;
   }

   public double getSalary()
   {
      return salary;
   }

   public Date getHireDay()
   {
      return hireDay;
   }

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

   public String toString()
   {
      return getClass().getName() + "[name=" + name + ",salary=" + salary + ",hireDay="
            + hireDay + "]";
   }

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

class Manager extends Employee
```

```
{
   /**
    * Constructs a Manager without a secretary
    * @param n the employee's name
    * @param s the salary
    * @param year the hire year
    * @param month the hire month
    * @param day the hire day
    */
   public Manager(String n, double s, int year, int month, int day)
   {
      super(n, s, year, month, day);
      secretary = null;
   }

   /**
    * Assigns a secretary to the manager
    * @param s the secretary
    */
   public void setSecretary(Employee s)
   {
      secretary = s;
   }

   public String toString()
   {
      return super.toString() + "[secretary=" + secretary + "]";
   }

   private Employee secretary;
}
```

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 `java.util.Date`
`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 Employee[]
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 staff (serial #1)
72 00 0B [LEmployee;	New class, string length, class name Employee[] (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	staff[0]—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—`String` (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—`String` (serial #5)
`    L 00 04 name`	Instance field type and name
`    74 00 12 Ljava/lang/String;`	Instance field class name—`String` (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, state, 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.write(value);
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

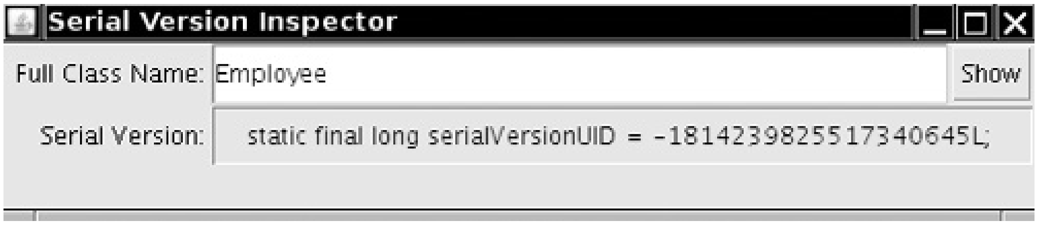
```
serialver Employee
```

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]

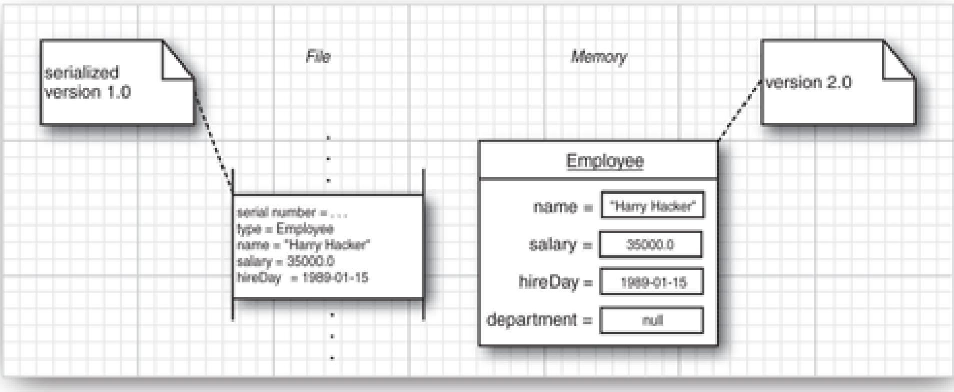
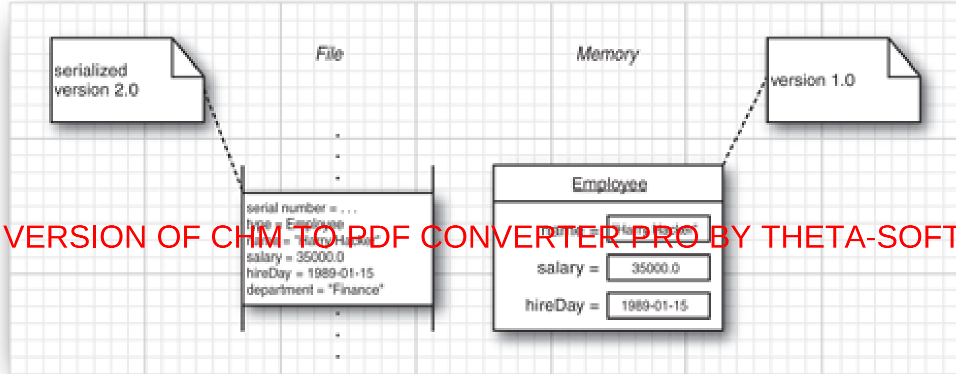


Figure 1-10. Reading an object with more data fields

[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 `SerialCloneable` 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:

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

public class SerialCloneTest
{
   public static void main(String[] args)
   {
      Employee harry = new Employee("Harry Hacker", 35000, 1989, 10, 1);
      // clone harry
      Employee harry2 = (Employee) harry.clone();

      // mutate harry
      harry.raiseSalary(10);

      // now harry and the clone are different
      System.out.println(harry);
      System.out.println(harry2);
   }
}
```

```java

/**
   A class whose clone method uses serialization.
*/
class SerialCloneable implements Cloneable, Serializable
{
   public Object clone()
   {
      try
      {
         // save the object to a byte array
         ByteArrayOutputStream bout = new ByteArrayOutputStream();
         ObjectOutputStream out = new ObjectOutputStream(bout);
         out.writeObject(this);
         out.close();

         // read a clone of the object from the byte array
         ByteArrayInputStream bin = new ByteArrayInputStream(bout.toByteArray());
         ObjectInputStream in = new ObjectInputStream(bin);
         Object ret = in.readObject();
         in.close();

         return ret;
      }
      catch (Exception e)
      {
         return null;
      }
   }
}

/**
   The familiar Employee class, redefined to extend the
   SerialCloneable class.
*/
class Employee extends SerialCloneable
{
   public Employee(String n, double s, int year, int month, int day)
   {
      name = n;
      salary = s;
      GregorianCalendar calendar = new GregorianCalendar(year, month - 1, day);
      hireDay = calendar.getTime();
   }

   public String getName()
   {
      return name;
   }

   public double getSalary()
   {
      return salary;
   }

   public Date getHireDay()
   {
      return hireDay;
   }
```

```java

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

   public String toString()
   {
      return getClass().getName()
         + "[name=" + name
         + ",salary=" + salary
         + ",hireDay=" + hireDay
         + "]";
   }

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

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:

```
import java.io.*;

/**
 * @version 1.00 05 Sep 1997
 * @author Gary Cornell
 */
public class FindDirectories
{
   public static void main(String[] args)
   {
      // if no arguments provided, start at the parent directory
      if (args.length == 0) args = new String[] { ".." };

      try
      {
         File pathName = new File(args[0]);
         String[] fileNames = pathName.list();

         // enumerate all files in the directory
         for (int i = 0; i < fileNames.length; i++)
         {
            File f = new File(pathName.getPath(), fileNames[i]);

            // if the file is again a directory, call the main method recursivel
            if (f.isDirectory())
            {
               System.out.println(f.getCanonicalPath());
               main(new String[] { f.getPath() });
            }
         }
      }
      catch (IOException e)
      {
         e.printStackTrace();
      }
   }
}
```

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.

API

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:	`prefix`	A prefix string that is at least three characters long
	`suffix`	An optional suffix. If `null`, `.tmp` is used
	`directory`	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.

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:

```
import java.io.*;
import java.nio.*;
import java.nio.channels.*;
import java.util.zip.*;

/**
 * This program computes the CRC checksum of a file. <br>
 * Usage: java NIOTest filename
 * @version 1.01 2004-05-11
 * @author Cay Horstmann
 */
public class NIOTest
{
   public static long checksumInputStream(String filename) throws IOException
   {
      InputStream in = new FileInputStream(filename);
      CRC32 crc = new CRC32();

      int c;
      while ((c = in.read()) != -1)
         crc.update(c);
      return crc.getValue();
   }

   public static long checksumBufferedInputStream(String filename) throws IOException
   {
      InputStream in = new BufferedInputStream(new FileInputStream(filename));
      CRC32 crc = new CRC32();

      int c;
      while ((c = in.read()) != -1)
         crc.update(c);
      return crc.getValue();
   }

   public static long checksumRandomAccessFile(String filename) throws IOException
   {
      RandomAccessFile file = new RandomAccessFile(filename, "r");
      long length = file.length();
      CRC32 crc = new CRC32();

      for (long p = 0; p < length; p++)
```

```
      {
         file.seek(p);
         int c = file.readByte();
         crc.update(c);
      }
      return crc.getValue();
   }

   public static long checksumMappedFile(String filename) throws IOException
```

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

      CRC32 crc = new CRC32();
      int length = (int) channel.size();
      MappedByteBuffer buffer = channel.map(FileChannel.MapMode.READ_ONLY, 0, length);
```

```

      for (int p = 0; p < length; p++)
      {
         int c = buffer.get(p);
         crc.update(c);
      }
      return crc.getValue();
   }

   public static void main(String[] args) throws IOException
   {
      System.out.println("Input Stream:");
      long start = System.currentTimeMillis();
      long crcValue = checksumInputStream(args[0]);
      long end = System.currentTimeMillis();
      System.out.println(Long.toHexString(crcValue));
      System.out.println((end - start) + " milliseconds");

      System.out.println("Buffered Input Stream:");
      start = System.currentTimeMillis();
      crcValue = checksumBufferedInputStream(args[0]);
      end = System.currentTimeMillis();
      System.out.println(Long.toHexString(crcValue));
      System.out.println((end - start) + " milliseconds");

      System.out.println("Random Access File:");
      start = System.currentTimeMillis();
      crcValue = checksumRandomAccessFile(args[0]);
      end = System.currentTimeMillis();
      System.out.println(Long.toHexString(crcValue));
      System.out.println((end - start) + " milliseconds");

      System.out.println("Mapped File:");
      start = System.currentTimeMillis();
      crcValue = checksumMappedFile(args[0]);
      end = System.currentTimeMillis();
      System.out.println(Long.toHexString(crcValue));
      System.out.println((end - start) + " milliseconds");
   }
}
```

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:	`mode`	One of the constants `READ_ONLY`, `READ_WRITE`, or `PRIVATE` in the `FileChannel.MapMode` class
	`position`	The start of the mapped region
	`size`	The size of the mapped region

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.

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:	`destination`	The byte array to be filled
	`offset`	The offset of the region to be filled
	`length`	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:	`source`	The byte array to be written
	`offset`	The offset of the region to be written
	`length`	The length of the region to be written

- *Xxx* `get`*Xxx*`()`
- *Xxx* `get`*Xxx*`(int index)`
- `ByteBuffer put`*Xxx*`(`*xxx* `value)`
- `ByteBuffer put`*Xxx*`(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`.

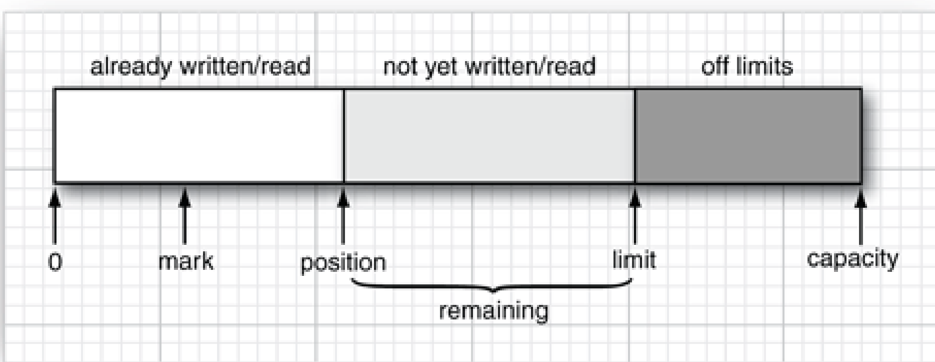
Note

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 ≤ *mark* ≤ *position* ≤ *limit* ≤ *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.

API `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.

API `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:	`position`	The start of the region to be locked
	`size`	The size of the region to be locked
	`shared`	`true` for a shared lock, `false` for an exclusive lock

API 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 <a href="...">.

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	
c	The character *c*
\u*nnnn*, \x*nn*, \0*n*, \0*nn*, \0*nnn*	The code unit with the given hex or octal value
\t, \n, \r, \f, \a, \e	The control characters tab, newline, return, form feed, alert, and escape

Syntax	Explanation
\c*c*	The control character corresponding to the character *c*
Character Classes	
[C_1C_2. . .]	Any of the characters represented by C_1, C_2, . . . The C_i are characters, character ranges (c_1-c_2), or character classes
[^. . .]	Complement of character class
[. . . && . . .]	Intersection of two character classes
Predefined Character Classes	
.	Any character except line terminators (or any character if the DOTALL flag is set)
\d	A digit [0-9]
\D	A nondigit [^0-9]
\s	A whitespace character [\t\n\r\f\x0B]
\S	A nonwhitespace character
\w	A word character [a-zA-Z0-9_]
\W	A nonword character
\p{*name*}	A named character class—see Table 1-8
\P{*name*}	The complement of a named character class
Boundary Matchers	
^ $	Beginning, end of input (or beginning, end of line in multiline mode)
\b	A word boundary
\B	A nonword boundary
\A	Beginning of input
\z	End of input
\Z	End of input except final line terminator
\G	End of previous match
Quantifiers	
X?	Optional X
X*	X, 0 or more times
X+	X, 1 or more times
X{*n*} *X*{*n*,} *X*{*n*,*m*}	X *n* times, at least *n* times, between *n* and *m* times
Quantifier Suffixes	
?	Turn default (greedy) match into reluctant match

Syntax	Explanation
`+`	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 *n*th group
Escapes	
`\`*c*	The character *c* (must not be an alphabetic character)
`\Q` . . . `\E`	Quote . . . verbatim
`(?` . . . `)`	Special construct—see API notes of `Pattern` 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 `Character.isLowerCase()`
`javaUpperCase`	Upper case, as determined by `Character.isUpperCase()`
`javaWhitespace`	Whitespace, as determined by `Character.isWhitespace()`
`javaMirrored`	Mirrored, as determined by `Character.isMirrored()`

Character Class Name	Explanation
In*Block*	*Block* is the name of a Unicode character block, with spaces removed, such as BasicLatin or Mongolian. See http://www.unicode.org for a list of block names.
Category or In*Category*	*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 ava 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:

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

/**
 * This program tests regular expression matching.
 *  Enter a pattern and strings to match, or hit Cancel
 *  to exit. If the pattern contains groups, the group
 *  boundaries are displayed in the match.
 *  @version 1.01 2004-05-11
 *  @author Cay Horstmann
 */
public class RegExTest
{
   public static void main(String[] args)
   {
      Scanner in = new Scanner(System.in);
      System.out.println("Enter pattern: ");
      String patternString = in.nextLine();

      Pattern pattern = null;
      try
      {
         pattern = Pattern.compile(patternString);
```

```
      }
      catch (PatternSyntaxException e)
      {
         System.out.println("Pattern syntax error");
         System.exit(1);
      }

      while (true)
      {
```

```
         System.out.println("Enter string to match: ");
         String input = in.nextLine();
         if (input == null || input.equals("")) return;
         Matcher matcher = pattern.matcher(input);
         if (matcher.matches())
         {
```

```
            System.out.println("Match");
            int g = matcher.groupCount();
            if (g > 0)
            {
               for (int i = 0; i < input.length(); i++)
               {
                  for (int j = 1; j <= g; j++)
                     if (i == matcher.start(j))
                        System.out.print('(');
                  System.out.print(input.charAt(i));
                  for (int j = 1; j <= g; j++)
                     if (i + 1 == matcher.end(j))
                        System.out.print(')');
               }
               System.out.println();
            }
         }
         else
            System.out.println("No match");
      }
   }
}
```

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:

```java
import java.io.*;
import java.net.*;
import java.util.regex.*;

/**
 * This program displays all URLs in a web page by matching a regular expression that describes
 * the <a href=...> HTML tag. Start the program as <br>
 * java HrefMatch URL
 * @version 1.01 2004-06-04
 * @author Cay Horstmann
 */
public class HrefMatch
{
   public static void main(String[] args)
   {
      try
      {
         // get URL string from command line or use default
         String urlString;
         if (args.length > 0) urlString = args[0];
         else urlString = "http://java.sun.com";

         // open reader for URL
         InputStreamReader in = new InputStreamReader(new URL(urlString).openStream());

         // read contents into string builder
         StringBuilder input = new StringBuilder();
         int ch;
         while ((ch = in.read()) != -1)
            input.append((char) ch);

         // search for all occurrences of pattern
         String patternString = "<a\\s+href\\s*=\\s*(\"[^\"]*\"|[^\\s>])\\s*>";
         Pattern pattern = Pattern.compile(patternString, Pattern.CASE_INSENSITIVE);
         Matcher matcher = pattern.matcher(input);

         while (matcher.find())
         {
            int start = matcher.start();
            int end = matcher.end();
            String match = input.substring(start, end);
            System.out.println(match);
         }
      }
      catch (IOException e)
      {
         e.printStackTrace();
      }
      catch (PatternSyntaxException e)
      {
         e.printStackTrace();
      }
   }
```

```
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 *n*th 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);
```

API 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 $\leq$ 0, then the entire input is split. If `limit` is 0, then trailing empty strings are not placed in the returned array

`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 </img> 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

UNREGISTERED VERSION OF CHM TO PDF CONVERTER PRO BY THETA-SOFTWARE

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 `&#x`*hexValue*`;`. 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[< & > 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 </img> 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>
```

UNREGISTERED VERSION OF CHM TO PDF CONVERTER PRO BY THETA-SOFTWARE

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 `&#x`*hexValue*`;`. 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[< & > 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.

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);
```

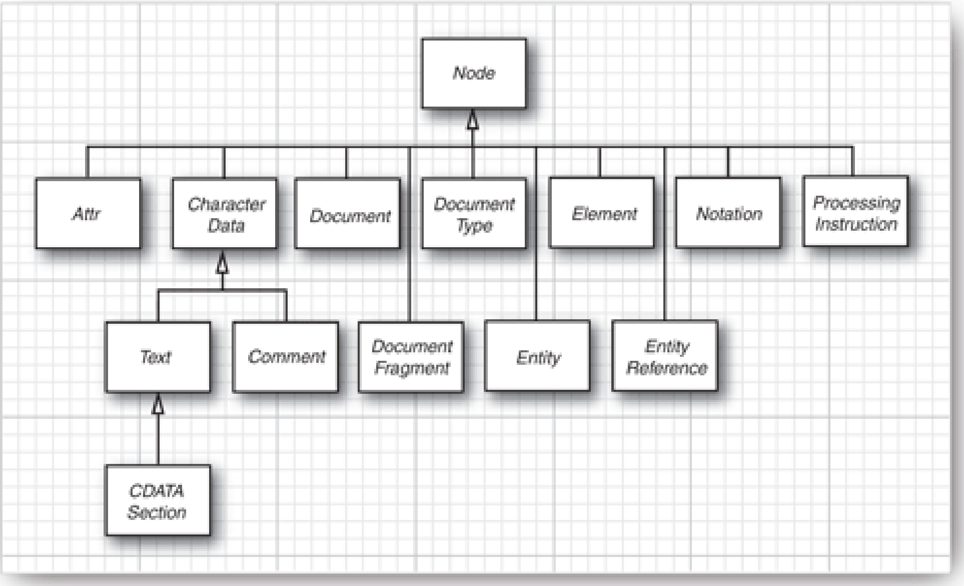
Note

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>
```

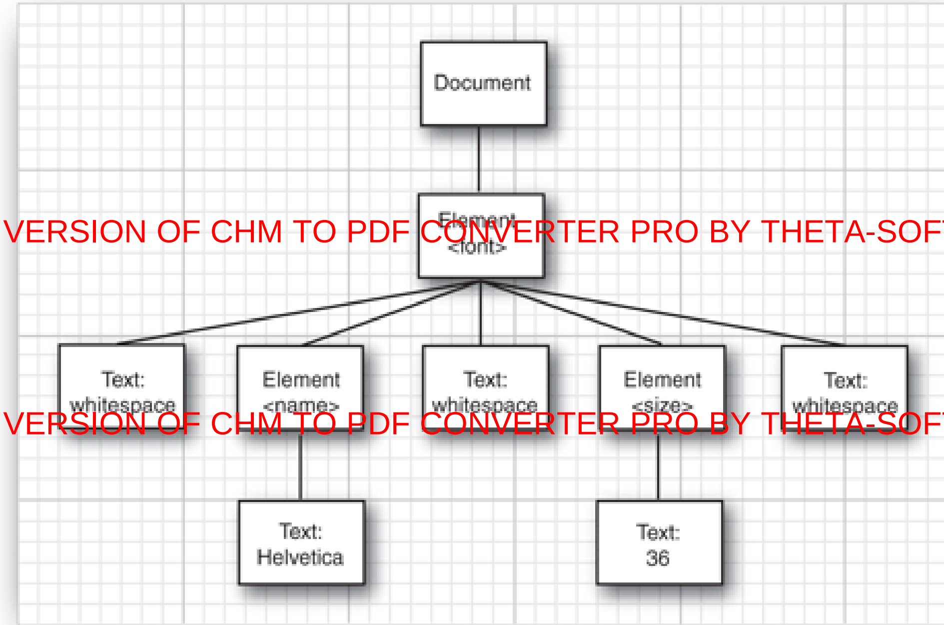
You would expect the `font` element to have two children, but the parser reports five:

- The whitespace between `<font>` and `<name>`
- The `name` element
- The whitespace between `</name>` and `<size>`
- The `size` element
- The whitespace between `</size>` and `</font>`

Figure 2-2 shows the DOM tree.

Figure 2-2. A simple DOM tree

[View full size image]



UNREGISTERED VERSION OF CHM TO PDF CONVERTER PRO BY THETA-SOFTWARE

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.

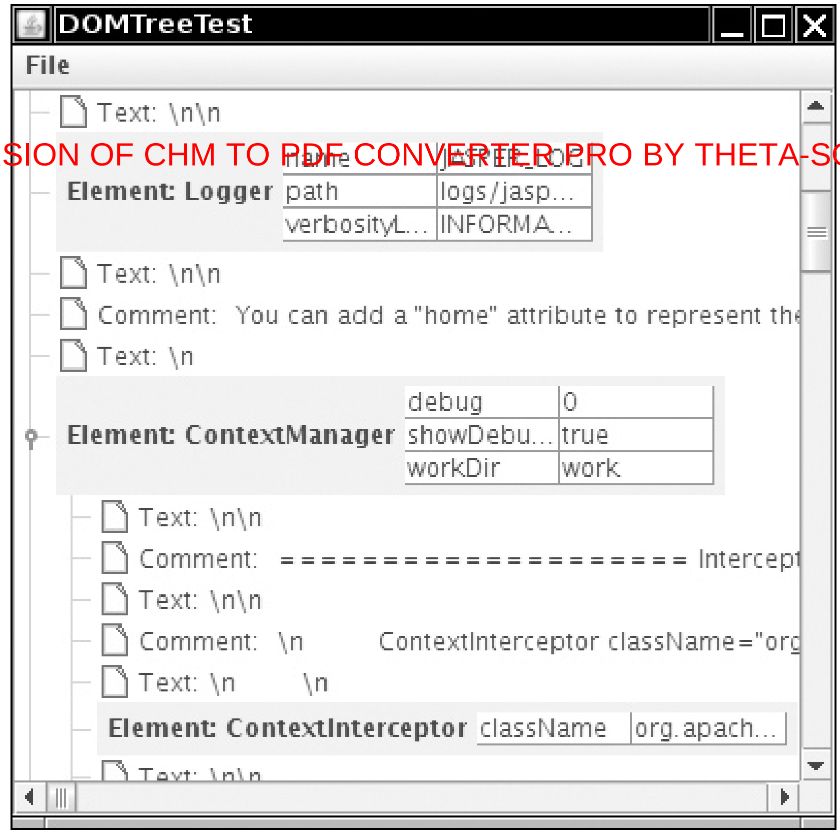
```
NamedNodeMap attributes = element.getAttributes();
for (int i = 0; i < attributes.getLength(); i++)
{
   Node attribute = attributes.item(i);
   String name = attribute.getNodeName();
   String value = attribute.getNodeValue();
   . . .
}
```

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:

```
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import javax.swing.*;
import javax.swing.event.*;
import javax.swing.table.*;
import javax.swing.tree.*;
import javax.xml.parsers.*;
import org.w3c.dom.*;

/**
 * This program displays an XML document as a tree.
 * @version 1.11 2007-06-24
 * @author Cay Horstmann
 */
public class DOMTreeTest
{
   public static void main(String[] args)
   {
      EventQueue.invokeLater(new Runnable()
         {
            public void run()
            {
               JFrame frame = new DOMTreeFrame();
               frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
               frame.setVisible(true);
            }
         });
   }
}

/**
 * This frame contains a tree that displays the contents of an XML document.
 */
class DOMTreeFrame extends JFrame
{
   public DOMTreeFrame()
   {
      setTitle("DOMTreeTest");
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

      JMenu fileMenu = new JMenu("File");
      JMenuItem openItem = new JMenuItem("Open");
      openItem.addActionListener(new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               openFile();
            }
         });
      fileMenu.add(openItem);
```

```
      JMenuItem exitItem = new JMenuItem("Exit");
      exitItem.addActionListener(new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               System.exit(0);
            }
         });
```



```
      fileMenu.add(exitItem);

      JMenuBar menuBar = new JMenuBar();
      menuBar.add(fileMenu);
      setJMenuBar(menuBar);
   }
```



```
   /**
    * Open a file and load the document.
    */
   public void openFile()
   {
      JFileChooser chooser = new JFileChooser();
      chooser.setCurrentDirectory(new File("."));

      chooser.setFileFilter(new javax.swing.filechooser.FileFilter()
         {
            public boolean accept(File f)
            {
               return f.isDirectory() || f.getName().toLowerCase().endsWith(".xml");
            }

            public String getDescription()
            {
               return "XML files";
            }
         });
      int r = chooser.showOpenDialog(this);
      if (r != JFileChooser.APPROVE_OPTION) return;
      final File file = chooser.getSelectedFile();

      new SwingWorker<Document, Void>()
         {
            protected Document doInBackground() throws Exception
            {
               if (builder == null)
               {
                  DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
                  builder = factory.newDocumentBuilder();
               }
               return builder.parse(file);
            }

            protected void done()
            {
               try
               {
                  Document doc = get();
                  JTree tree = new JTree(new DOMTreeModel(doc));
                  tree.setCellRenderer(new DOMTreeCellRenderer());
```

```

                  setContentPane(new JScrollPane(tree));
                  validate();
               }
               catch (Exception e)
               {
                  JOptionPane.showMessageDialog(DOMTreeFrame.this, e);
               }
            }
         }.execute();
   }

   private DocumentBuilder builder;
   private static final int DEFAULT_WIDTH = 400;
   private static final int DEFAULT_HEIGHT = 400;
}

/**
 * This tree model describes the tree structure of an XML document.
 */
class DOMTreeModel implements TreeModel
{
   /**
    * Constructs a document tree model.
    * @param doc the document
    */
   public DOMTreeModel(Document doc)
   {
      this.doc = doc;
   }

   public Object getRoot()
   {
      return doc.getDocumentElement();
   }

   public int getChildCount(Object parent)
   {
      Node node = (Node) parent;
      NodeList list = node.getChildNodes();
      return list.getLength();
   }

   public Object getChild(Object parent, int index)
   {
      Node node = (Node) parent;
      NodeList list = node.getChildNodes();
      return list.item(index);
   }

   public int getIndexOfChild(Object parent, Object child)
   {
      Node node = (Node) parent;
      NodeList list = node.getChildNodes();
      for (int i = 0; i < list.getLength(); i++)
         if (getChild(node, i) == child) return i;
      return -1;
   }

```

```
   public boolean isLeaf(Object node)
   {
      return getChildCount(node) == 0;
   }

   public void valueForPathChanged(TreePath path, Object newValue)
   {
   }

```

```
   public void addTreeModelListener(TreeModelListener l)
   {
   }

   public void removeTreeModelListener(TreeModelListener l)
   {
```

```
   }

   private Document doc;
}

/**
 * This class renders an XML node.
 */
class DOMTreeCellRenderer extends DefaultTreeCellRenderer
{
   public Component getTreeCellRendererComponent(JTree tree, Object value, boolean selected,
         boolean expanded, boolean leaf, int row, boolean hasFocus)
   {
      Node node = (Node) value;
      if (node instanceof Element) return elementPanel((Element) node);

      super.getTreeCellRendererComponent(tree, value, selected, expanded, leaf, row, hasFocus);
      if (node instanceof CharacterData) setText(characterString((CharacterData) node));
      else setText(node.getClass() + ": " + node.toString());
      return this;
   }

   public static JPanel elementPanel(Element e)
   {
      JPanel panel = new JPanel();
      panel.add(new JLabel("Element: " + e.getTagName()));
      final NamedNodeMap map = e.getAttributes();
      panel.add(new JTable(new AbstractTableModel()
         {
            public int getRowCount()
            {
               return map.getLength();
            }

            public int getColumnCount()
            {
               return 2;
            }

            public Object getValueAt(int r, int c)
            {
               return c == 0 ? map.item(r).getNodeName() : map.item(r).getNodeValue();
            }
         }));
```

```
      return panel;
   }

   public static String characterString(CharacterData node)
   {
      StringBuilder builder = new StringBuilder(node.getData());
      for (int i = 0; i < builder.length(); i++)
      {
         if (builder.charAt(i) == '\r')
         {
            builder.replace(i, i + 1, "\\r");
            i++;
         }
         else if (builder.charAt(i) == '\n')
         {
            builder.replace(i, i + 1, "\\n");
            i++;
         }
         else if (builder.charAt(i) == '\t')
         {
            builder.replace(i, i + 1, "\\t");
            i++;
         }
      }
      if (node instanceof CDATASection) builder.insert(0, "CDATASection: ");
      else if (node instanceof Text) builder.insert(0, "Text: ");
      else if (node instanceof Comment) builder.insert(0, "Comment: ");

      return builder.toString();
   }
}
```

API

`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.

API

`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.

API

`org.w3c.dom.Document` 1.4

- `Element getDocumentElement()`

  returns the root element of the document.

API

`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.

`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`.

`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 contair 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 eact 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 annoye by the complexity of XML Schema that they use alternative validation languages. The most common choice is Rela 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, w 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>
```


UNREGISTERED VERSION OF CHM TO PDF CONVERTER PRO BY THETA-SOFTWARE

As you can see, the rules are included inside a `DOCTYPE` declaration, in a block delimited by `[. . .]`. The docume 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 mor 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
E *	0 or more occurrences of E
E +	1 or more occurrences of E
E ?	0 or 1 occurrences of E
E_1 | E_2 | . . . | E_n	One of E_1 , E_2 , . . . , E_n
E_1 , E_2 , . . . , E_n	E_1 followed by E_2 , . . . , E_n
#PCDATA	Text
(#PCDATA| E_1 | E_2 | . . . | E_n)*	0 or more occurrences of text and E_1 , E_2 , . . ., E_n in any order (mixed content)
ANY	Any children allowed
EMPTY	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 interpret

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 on 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)|(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|z))` . However, some expressions can't be reformulated, such as `((x,y)*|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 | A_2 | . . . | A_n)	One of the string attributes A_1 A_2 . . . | A_n
NMTOKEN , NMTOKENS	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 th 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 ar 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 th 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, on 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, o 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)
```

UNREGISTERED VERSION OF CHM TO PDF CONVERTER PRO BY THETA-SOFTWARE

You install the error handler with the `setErrorHandler` method of the `DocumentBuilder` class:

```
builder.setErrorHandler(handler);
```

API `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.

API `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.

API

`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).

API

`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.

API

`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.

API 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 documer 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—c 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 an 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, is feared not just for its complexity but also for the programming tedium. It would be much more convenient to pu the layout instructions into a text file instead of producing large amounts of repetitive code. In this section, you se 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 i 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 speci them by hand. Supplying them is optional:

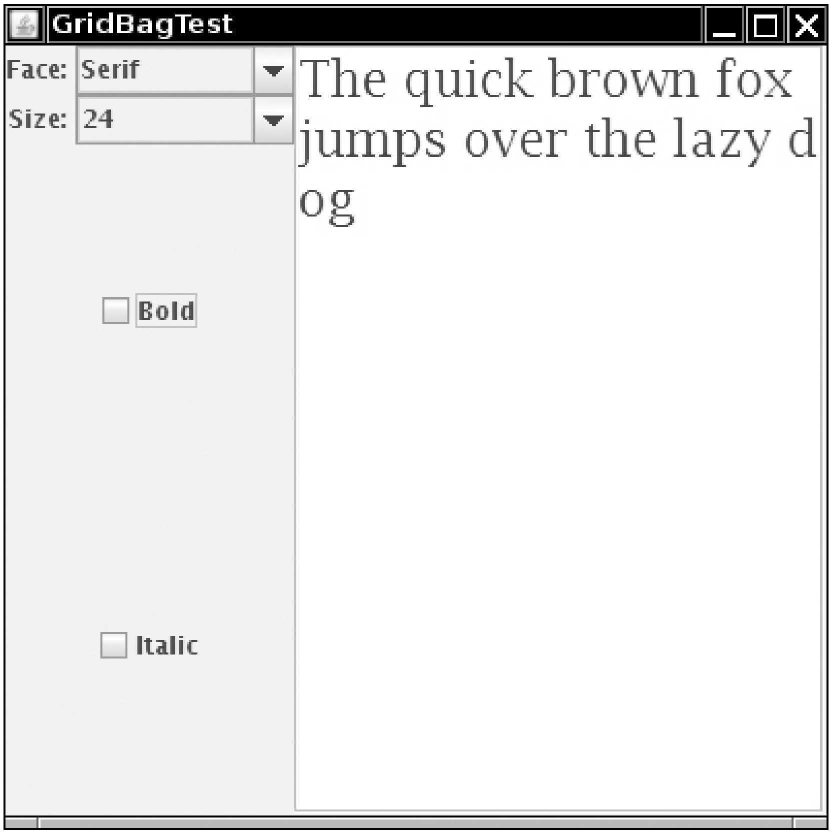
```
<!ATTLIST cell gridx CDATA #IMPLIED>
<!ATTLIST cell gridy CDATA #IMPLIED>
```

If they are not supplied, the program determines them according to the following heuristic: In column 0, the defau 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.gridy = 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 defau 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 `valu` 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 ad event listeners, we have to locate the components. For that reason, we support an optional attribute of type `ID` fo 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 GridBagPane("fontdialog.xml");
setContentPane(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 `GridBagPane` class to do all the boring work of setting up the gric bag layout. The layout is defined in Listing 2-3 . Figure 2-4 shows the result. The program only initializes the comk 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, anc 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:

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

/**
 * This program shows how to use an XML file to describe a gridbag layout
 * @version 1.01 2007-06-25
 * @author Cay Horstmann
 */
public class GridBagTest
{
   public static void main(final String[] args)
   {
      EventQueue.invokeLater(new Runnable()
         {
            public void run()
```

```
            {
               String filename = args.length == 0 ? "fontdialog.xml" : args[0];
               JFrame frame = new FontFrame(filename);
               frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
               frame.setVisible(true);
            }
         });
   }
}
```



```

/**
 * This frame contains a font selection dialog that is described by an XML file.
 * @param filename the file containing the user interface components for the dialog.
 */
class FontFrame extends JFrame
{
```



```
   public FontFrame(String filename)
   {
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
      setTitle("GridBagTest");

      gridbag = new GridBagPane(filename);
      add(gridbag);

      face = (JComboBox) gridbag.get("face");
      size = (JComboBox) gridbag.get("size");
      bold = (JCheckBox) gridbag.get("bold");
      italic = (JCheckBox) gridbag.get("italic");

      face.setModel(new DefaultComboBoxModel(new Object[] { "Serif", "SansSerif",
            "Monospaced", "Dialog", "DialogInput" }));

      size.setModel(new DefaultComboBoxModel(new Object[] { "8", "10", "12", "15", "18", "24",
            "36", "48" }));

      ActionListener listener = new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               setSample();
            }
         };

      face.addActionListener(listener);
      size.addActionListener(listener);
      bold.addActionListener(listener);
      italic.addActionListener(listener);
      setSample();
   }

   /**
    * This method sets the text sample to the selected font.
    */
   public void setSample()
   {
      String fontFace = (String) face.getSelectedItem();
      int fontSize = Integer.parseInt((String) size.getSelectedItem());
      JTextArea sample = (JTextArea) gridbag.get("sample");
      int fontStyle = (bold.isSelected() ? Font.BOLD : 0)
```

```
            + (italic.isSelected() ? Font.ITALIC : 0);

      sample.setFont(new Font(fontFace, fontStyle, fontSize));
      sample.repaint();
   }

   private GridBagPane gridbag;
   private JComboBox face;
   private JComboBox size;
   private JCheckBox bold;
   private JCheckBox italic;
   private static final int DEFAULT_WIDTH = 400;
   private static final int DEFAULT_HEIGHT = 400;
}
```

Listing 2-3. **fontdialog.xml**

Code View:

```
<?xml version="1.0"?>
<!DOCTYPE gridbag SYSTEM "gridbag.dtd">
<gridbag>
   <row>
      <cell anchor="EAST">
         <bean>
            <class>javax.swing.JLabel</class>
            <property>
               <name>text</name>
               <value><string>Face: </string></value>
            </property>
         </bean>
      </cell>
      <cell fill="HORIZONTAL" weightx="100">
         <bean id="face">
            <class>javax.swing.JComboBox</class>
         </bean>
      </cell>
      <cell gridheight="4" fill="BOTH" weightx="100" weighty="100">
         <bean id="sample">
            <class>javax.swing.JTextArea</class>
            <property>
               <name>text</name>
               <value><string>The quick brown fox jumps over the lazy dog</string></value
            </property>
            <property>
               <name>editable</name>
               <value><boolean>false</boolean></value>
            </property>
            <property>
               <name>lineWrap</name>
               <value><boolean>true</boolean></value>
            </property>
            <property>
```

```
               <name>border</name>
               <value>
                  <bean>
                     <class>javax.swing.border.EtchedBorder</class>
                  </bean>
               </value>
            </property>
         </bean>
      </cell>
   </row>
   <row>
      <cell anchor="EAST">
         <bean>
            <class>javax.swing.JLabel</class>
            <property>
               <name>text</name>
               <value><string>Size: </string></value>
            </property>
         </bean>
      </cell>
      <cell fill="HORIZONTAL" weightx="100">
         <bean id="size">
            <class>javax.swing.JComboBox</class>
         </bean>
      </cell>
   </row>
   <row>
      <cell gridwidth="2" weighty="100">
         <bean id="bold">
            <class>javax.swing.JCheckBox</class>
            <property>
               <name>text</name>
               <value><string>Bold</string></value>
            </property>
         </bean>
      </cell>
   </row>
   <row>
      <cell gridwidth="2" weighty="100">
         <bean id="italic">
            <class>javax.swing.JCheckBox</class>
            <property>
               <name>text</name>
               <value><string>Italic</string></value>
            </property>
         </bean>
      </cell>
   </row>
</gridbag>
```

Listing 2-4. **GridBagPane.java**

Code View:

```
import java.awt.*;
import java.beans.*;
import java.io.*;
import java.lang.reflect.*;
import javax.swing.*;
import javax.xml.parsers.*;
import org.w3c.dom.*;

/**
 * This panel uses an XML file to describe its components and their grid bag layout positions.
 * @version 1.10 2004-09-04
 * @author Cay Horstmann
 */
public class GridBagPane extends JPanel
{
   /**
    * Constructs a grid bag pane.
    * @param filename the name of the XML file that describes the pane's components and their
    * positions
    */
   public GridBagPane(String filename)
   {
      setLayout(new GridBagLayout());
      constraints = new GridBagConstraints();

      try
      {
         DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
         factory.setValidating(true);

         if (filename.contains("-schema"))
         {
            factory.setNamespaceAware(true);
            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);
         }

         factory.setIgnoringElementContentWhitespace(true);

         DocumentBuilder builder = factory.newDocumentBuilder();
         Document doc = builder.parse(new File(filename));

         if (filename.contains("-schema"))
         {
            int count = removeElementContentWhitespace(doc.getDocumentElement());
            System.out.println(count + " whitespace nodes removed.");
         }

         parseGridbag(doc.getDocumentElement());
      }
      catch (Exception e)
      {
         e.printStackTrace();
      }
   }

```

```java
   /**
    * Removes all (heuristically determined) element content whitespace nodes
    * @param e the root element
    * @return the number of whitespace nodes that were removed.
    */
   private int removeElementContentWhitespace(Element e)
   {
      NodeList children = e.getChildNodes();
      int count = 0;
      boolean allTextChildrenAreWhiteSpace = true;
      int elements = 0;
      for (int i = 0; i < children.getLength() && allTextChildrenAreWhiteSpace; i++)
      {
         Node child = children.item(i);
         if (child instanceof Text && ((Text) child).getData().trim().length() > 0)
            allTextChildrenAreWhiteSpace = false;
         else if (child instanceof Element)
         {
            elements++;
            count += removeElementContentWhitespace((Element) child);
         }
      }
      if (elements > 0 && allTextChildrenAreWhiteSpace) // heuristics for element content
      {
         for (int i = children.getLength() - 1; i >= 0; i--)
         {
            Node child = children.item(i);
            if (child instanceof Text)
            {
               e.removeChild(child);
               count++;
            }
         }
      }
      return count;
   }

   /**
    * Gets a component with a given name
    * @param name a component name
    * @return the component with the given name, or null if no component in this grid bag
    * pane has the given name
    */
   public Component get(String name)
   {
      Component[] components = getComponents();
      for (int i = 0; i < components.length; i++)
      {
         if (components[i].getName().equals(name)) return components[i];
      }
      return null;
   }

   /**
    * Parses a gridbag element.
    * @param e a gridbag element
    */
   private void parseGridbag(Element e)
   {
```

```
      NodeList rows = e.getChildNodes();
      for (int i = 0; i < rows.getLength(); i++)
      {
         Element row = (Element) rows.item(i);
         NodeList cells = row.getChildNodes();
         for (int j = 0; j < cells.getLength(); j++)
         {
            Element cell = (Element) cells.item(j);
            parseCell(cell, i, j);
         }
      }
   }

   /**
    * Parses a cell element.
    * @param e a cell element
    * @param r the row of the cell
    * @param c the column of the cell
    */
   private void parseCell(Element e, int r, int c)
   {
      // get attributes

      String value = e.getAttribute("gridx");
      if (value.length() == 0) // use default
      {
         if (c == 0) constraints.gridx = 0;
         else constraints.gridx += constraints.gridwidth;
      }
      else constraints.gridx = Integer.parseInt(value);

      value = e.getAttribute("gridy");
      if (value.length() == 0) // use default
      constraints.gridy = r;
      else constraints.gridy = Integer.parseInt(value);

      constraints.gridwidth = Integer.parseInt(e.getAttribute("gridwidth"));
      constraints.gridheight = Integer.parseInt(e.getAttribute("gridheight"));
      constraints.weightx = Integer.parseInt(e.getAttribute("weightx"));
      constraints.weighty = Integer.parseInt(e.getAttribute("weighty"));
      constraints.ipadx = Integer.parseInt(e.getAttribute("ipadx"));
      constraints.ipady = Integer.parseInt(e.getAttribute("ipady"));

      // use reflection to get integer values of static fields
      Class<GridBagConstraints> cl = GridBagConstraints.class;

      try
      {
         String name = e.getAttribute("fill");
         Field f = cl.getField(name);
         constraints.fill = f.getInt(cl);

         name = e.getAttribute("anchor");
         f = cl.getField(name);
         constraints.anchor = f.getInt(cl);
      }
      catch (Exception ex) // the reflection methods can throw various exceptions
      {
         ex.printStackTrace();
```

```
      }

      Component comp = (Component) parseBean((Element) e.getFirstChild());
      add(comp, constraints);
   }

   /**
    * Parses a bean element.
    * @param e a bean element
    */
   private Object parseBean(Element e)
   {
      try
      {
         NodeList children = e.getChildNodes();
         Element classElement = (Element) children.item(0);
         String className = ((Text) classElement.getFirstChild()).getData();

         Class<?> cl = Class.forName(className);

         Object obj = cl.newInstance();

         if (obj instanceof Component) ((Component) obj).setName(e.getAttribute("id"));

         for (int i = 1; i < children.getLength(); i++)
         {
            Node propertyElement = children.item(i);
            Element nameElement = (Element) propertyElement.getFirstChild();
            String propertyName = ((Text) nameElement.getFirstChild()).getData();

            Element valueElement = (Element) propertyElement.getLastChild();
            Object value = parseValue(valueElement);
            BeanInfo beanInfo = Introspector.getBeanInfo(cl);
            PropertyDescriptor[] descriptors = beanInfo.getPropertyDescriptors();
            boolean done = false;
            for (int j = 0; !done && j < descriptors.length; j++)
            {
               if (descriptors[j].getName().equals(propertyName))
               {
                  descriptors[j].getWriteMethod().invoke(obj, value);
                  done = true;
               }
            }

         }
         return obj;
      }
      catch (Exception ex) // the reflection methods can throw various exceptions
      {
         ex.printStackTrace();
         return null;
      }
   }

   /**
    * Parses a value element.
    * @param e a value element
    */
   private Object parseValue(Element e)
```

```
   {
      Element child = (Element) e.getFirstChild();
      if (child.getTagName().equals("bean")) return parseBean(child);
      String text = ((Text) child.getFirstChild()).getData();
      if (child.getTagName().equals("int")) return new Integer(text);
      else if (child.getTagName().equals("boolean")) return new Boolean(text);
      else if (child.getTagName().equals("string")) return text;
      else return null;
   }

   private GridBagConstraints constraints;
}
```

Listing 2-5. **gridbag.dtd**

Code View:

```
<!ELEMENT gridbag (row)*>
<!ELEMENT row (cell)*>
<!ELEMENT cell (bean)>
<!ATTLIST cell gridx CDATA #IMPLIED>
<!ATTLIST cell gridy CDATA #IMPLIED>
<!ATTLIST cell gridwidth CDATA "1">
<!ATTLIST cell gridheight CDATA "1">
<!ATTLIST cell weightx CDATA "0">
<!ATTLIST cell weighty CDATA "0">
<!ATTLIST cell fill (NONE|BOTH|HORIZONTAL|VERTICAL) "NONE">
<!ATTLIST cell anchor
   (CENTER|NORTH|NORTHEAST|EAST|SOUTHEAST|SOUTH|SOUTHWEST|WEST|NORTHWEST) "CENTER">
<!ATTLIST cell ipadx CDATA "0">
<!ATTLIST cell ipady CDATA "0">

<!ELEMENT bean (class, property*)>
<!ATTLIST bean id ID #IMPLIED>

<!ELEMENT class (#PCDATA)>
<!ELEMENT property (name, value)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT value (int|string|boolean|bean)>
<!ELEMENT int (#PCDATA)>
<!ELEMENT string (#PCDATA)>
<!ELEMENT boolean (#PCDATA)>
```

Listing 2-6. **gridbag.xsd**

Code View:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

```

```
<xsd:element name="gridbag" type="GridBagType"/>

<xsd:element name="bean" type="BeanType"/>

<xsd:complexType name="GridBagType">
   <xsd:sequence>
      <xsd:element name="row" type="RowType" minOccurs="0" maxOccurs="unbounded"/>
   </xsd:sequence>
</xsd:complexType>
```



```
<xsd:complexType name="RowType">
   <xsd:sequence>
      <xsd:element name="cell" type="CellType" minOccurs="0" maxOccurs="unbounded"/>
   </xsd:sequence>
</xsd:complexType>
```



```
<xsd:complexType name="CellType">
   <xsd:sequence>
      <xsd:element ref="bean"/>
   </xsd:sequence>
   <xsd:attribute name="gridx" type="xsd:int" use="optional"/>
   <xsd:attribute name="gridy" type="xsd:int" use="optional"/>
   <xsd:attribute name="gridwidth" type="xsd:int" use="optional" default="1" />
   <xsd:attribute name="gridheight" type="xsd:int" use="optional" default="1" />
   <xsd:attribute name="weightx" type="xsd:int" use="optional" default="0" />
   <xsd:attribute name="weighty" type="xsd:int" use="optional" default="0" />
   <xsd:attribute name="fill" use="optional" default="NONE">
     <xsd:simpleType>
       <xsd:restriction base="xsd:string">
         <xsd:enumeration value="NONE" />
         <xsd:enumeration value="BOTH" />
         <xsd:enumeration value="HORIZONTAL" />
         <xsd:enumeration value="VERTICAL" />
       </xsd:restriction>
     </xsd:simpleType>
   </xsd:attribute>
   <xsd:attribute name="anchor" use="optional" default="CENTER">
     <xsd:simpleType>
       <xsd:restriction base="xsd:string">
         <xsd:enumeration value="CENTER" />
         <xsd:enumeration value="NORTH" />
         <xsd:enumeration value="NORTHEAST" />
         <xsd:enumeration value="EAST" />
         <xsd:enumeration value="SOUTHEAST" />
         <xsd:enumeration value="SOUTH" />
         <xsd:enumeration value="SOUTHWEST" />
         <xsd:enumeration value="WEST" />
         <xsd:enumeration value="NORTHWEST" />
       </xsd:restriction>
     </xsd:simpleType>
   </xsd:attribute>
   <xsd:attribute name="ipady" type="xsd:int" use="optional" default="0" />
   <xsd:attribute name="ipadx" type="xsd:int" use="optional" default="0" />
</xsd:complexType>

<xsd:complexType name="BeanType">
   <xsd:sequence>
      <xsd:element name="class" type="xsd:string"/>
     <xsd:element name="property" type="PropertyType" minOccurs="0" maxOccurs="unbounded"/>
```

```
      </xsd:sequence>
      <xsd:attribute name="id" type="xsd:ID" use="optional" />
   </xsd:complexType>

   <xsd:complexType name="PropertyType">
      <xsd:sequence>
         <xsd:element name="name" type="xsd:string"/>
         <xsd:element name="value" type="ValueType"/>
      </xsd:sequence>
   </xsd:complexType>

   <xsd:complexType name="ValueType">
      <xsd:choice>
         <xsd:element ref="bean"/>
         <xsd:element name="int" type="xsd:int"/>
         <xsd:element name="string" type="xsd:string"/>
         <xsd:element name="boolean" type="xsd:boolean"/>
      </xsd:choice>
   </xsd:complexType>
</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/xxl/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`:

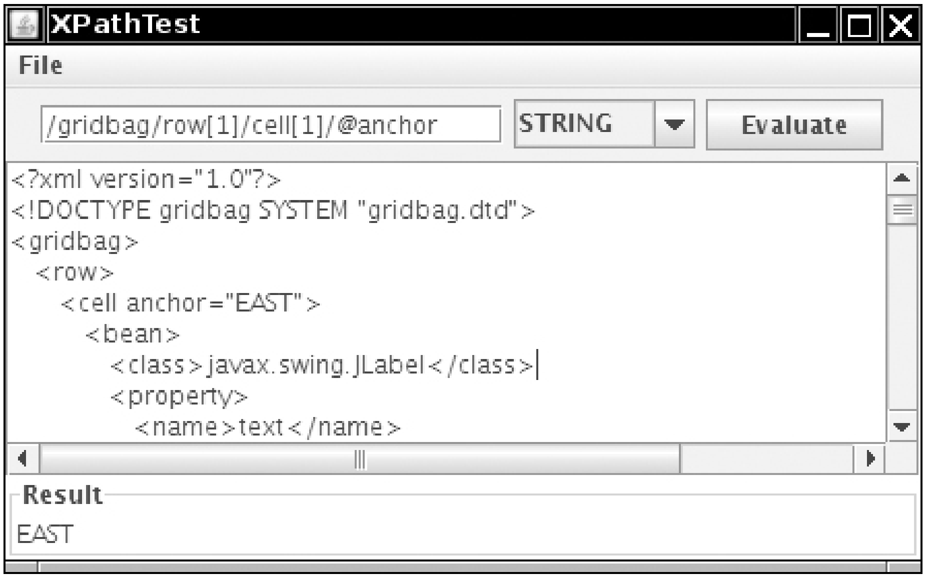
Code View:
```
int count = ((Number) path.evaluate("count(/gridbag/row)", doc, XPathConstants.NUM-
BER)).intValue();
```


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:
```
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import javax.swing.*;
import javax.swing.border.*;
import javax.xml.namespace.*;
import javax.xml.parsers.*;
import javax.xml.xpath.*;
import org.w3c.dom.*;
import org.xml.sax.*;

```

```
/**
 * This program evaluates XPath expressions
 * @version 1.01 2007-06-25
 * @author Cay Horstmann
 */
public class XPathTest
{
   public static void main(String[] args)
   {
      EventQueue.invokeLater(new Runnable()
         {
            public void run()
            {
               JFrame frame = new XPathFrame();
               frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
               frame.setVisible(true);
            }
         });
   }
}

/**
 * This frame shows an XML document, a panel to type an XPath expression, and a text field
 * to display the result.
 */
class XPathFrame extends JFrame
{
   public XPathFrame()
   {
      setTitle("XPathTest");

      JMenu fileMenu = new JMenu("File");
      JMenuItem openItem = new JMenuItem("Open");
      openItem.addActionListener(new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               openFile();
            }
         });
      fileMenu.add(openItem);

      JMenuItem exitItem = new JMenuItem("Exit");
      exitItem.addActionListener(new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               System.exit(0);
            }
         });
      fileMenu.add(exitItem);

      JMenuBar menuBar = new JMenuBar();
      menuBar.add(fileMenu);
      setJMenuBar(menuBar);

      ActionListener listener = new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
```

```
            {
               evaluate();
            }
         };
      expression = new JTextField(20);
      expression.addActionListener(listener);
      JButton evaluateButton = new JButton("Evaluate");
      evaluateButton.addActionListener(listener);
```



```

      typeCombo = new JComboBox(new Object[] { "STRING", "NODE", "NODESET", "NUMBER",
                                          "BOOLEAN" });
      typeCombo.setSelectedItem("STRING");

      JPanel panel = new JPanel();
      panel.add(expression);
```



```
      panel.add(typeCombo);
      panel.add(evaluateButton);
      docText = new JTextArea(10, 40);
      result = new JTextField();
      result.setBorder(new TitledBorder("Result"));

      add(panel, BorderLayout.NORTH);
      add(new JScrollPane(docText), BorderLayout.CENTER);
      add(result, BorderLayout.SOUTH);

      try
      {
         DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
         builder = factory.newDocumentBuilder();
      }
      catch (ParserConfigurationException e)
      {
         JOptionPane.showMessageDialog(this, e);
      }

      XPathFactory xpfactory = XPathFactory.newInstance();
      path = xpfactory.newXPath();
      pack();
   }

   /**
    * Open a file and load the document.
    */
   public void openFile()
   {
      JFileChooser chooser = new JFileChooser();
      chooser.setCurrentDirectory(new File("."));

      chooser.setFileFilter(new javax.swing.filechooser.FileFilter()
         {
            public boolean accept(File f)
            {
               return f.isDirectory() || f.getName().toLowerCase().endsWith(".xml");
            }

            public String getDescription()
            {
               return "XML files";
            }
```

```
         });
      int r = chooser.showOpenDialog(this);
      if (r != JFileChooser.APPROVE_OPTION) return;
      File f = chooser.getSelectedFile();
      try
      {
         byte[] bytes = new byte[(int) f.length()];
         new FileInputStream(f).read(bytes);
         docText.setText(new String(bytes));
         doc = builder.parse(f);
      }
      catch (IOException e)
      {
         JOptionPane.showMessageDialog(this, e);
      }
      catch (SAXException e)
      {
         JOptionPane.showMessageDialog(this, e);
      }
   }

   public void evaluate()
   {
      try
      {
         String typeName = (String) typeCombo.getSelectedItem();
         QName returnType = (QName) XPathConstants.class.getField(typeName).get(null);
         Object evalResult = path.evaluate(expression.getText(), doc, returnType);
         if (typeName.equals("NODESET"))
         {
            NodeList list = (NodeList) evalResult;
            StringBuilder builder = new StringBuilder();
            builder.append("{");
            for (int i = 0; i < list.getLength(); i++)
            {
               if (i > 0) builder.append(", ");
               builder.append("" + list.item(i));
            }
            builder.append("}");
            result.setText("" + builder);
         }
         else result.setText("" + evalResult);
      }
      catch (XPathExpressionException e)
      {
         result.setText("" + e);
      }
      catch (Exception e) // reflection exception
      {
         e.printStackTrace();
      }
   }

   private DocumentBuilder builder;
   private Document doc;
   private XPath path;
   private JTextField expression;
   private JTextField result;
   private JTextArea docText;
```

```
    private JComboBox typeCombo;
}
```


API

`javax.xml.xpath.XPathFactory` **5.0**

- `static XPathFactory newInstance()`

UNREGISTERED VERSION OF CHM TO PDF CONVERTER PRO BY THETA-SOFTWARE

  returns an `XPathFactory` instance for creating `XPath` objects.

- `XPath newXpath()`

  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="namespaceURI1">
   <child xmlns="namespaceURI2">
      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`.

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, D However, it can be inefficient if the document is large and if your processing algorithm is simple enough that you c 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 modern StAX parser that was added to Java SE 6. The SAX parser uses event callbacks, and the StAX parser provi 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 to the event handlers whether they want to build a data structure. In fact, the DOM parser is built on top of the SA 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. T interface defines several callback methods that the parser executes as it parses the document. Here are the most

- `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
The program at the end of this section prints all links `<a href="...">` in an HTML file. It simply overrides the `star`

of the handler to check for links with name `a` and an attribute with name `href` . This is potentially useful for implen
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 n
be able to parse them. As already mentioned, the W3C recommends that web designers use XHTML,
HTML dialect that can be displayed by current web browsers and that is also proper XML. Because th
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 oc
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` . Th
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
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 th
the form alias:localname . If namespace processing is turned on, then the namespaceURI and lname parameters
namespace and local (unqualified) name.

As with the DOM parser, namespace processing is turned off by default. You activate namespace processing by cal
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 u
way of turning a non-XML data source into XML is to report the SAX events that an XML parser would report. See t
Transformations " on page 157 for details.

Listing 2-8. **SAXTest.java**

Code View:

```
import java.io.*;
import java.net.*;
import javax.xml.parsers.*;
import org.xml.sax.*;
import org.xml.sax.helpers.*;

/**
```



```
 * This program demonstrates how to use a SAX parser. The program prints all hyperlinks links
 * of an XHTML web page. <br>
 * Usage: java SAXTest url
 * @version 1.00 2001-09-29
 * @author Cay Horstmann
 */
public class SAXTest
```



```
{
   public static void main(String[] args) throws Exception
   {
      String url;
      if (args.length == 0)
      {
         url = "http://www.w3c.org";
         System.out.println("Using " + url);
      }
      else url = args[0];

      DefaultHandler handler = new DefaultHandler()
         {
            public void startElement(String namespaceURI, String lname, String qname,
                  Attributes attrs)
            {
               if (lname.equals("a") && attrs != null)
               {
                  for (int i = 0; i < attrs.getLength(); i++)
                  {
                     String aname = attrs.getLocalName(i);
                     if (aname.equals("href")) System.out.println(attrs.getValue(i));
                  }
               }
            }
         };

      SAXParserFactory factory = SAXParserFactory.newInstance();
      factory.setNamespaceAware(true);
      SAXParser saxParser = factory.newSAXParser();
      InputStream in = new URL(url).openStream();
      saxParser.parse(in, handler);
   }
}
```

`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, us

```
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 than the equivalent SAX code because you don't have to worry about event handling.

Listing 2-9. **StAXTest.java**

Code View:

```
import java.io.*;
import java.net.*;
import javax.xml.stream.*;

/**
 * This program demonstrates how to use a StAX parser. The program prints all hyperlinks links
 * of an XHTML web page. <br>
 * Usage: java StAXTest url
 * @author Cay Horstmann
 * @version 1.0 2007-06-23
 */
public class StAXTest
{
   public static void main(String[] args) throws Exception
   {
      String urlString;
      if (args.length == 0)
      {
         urlString = "http://www.w3c.org";
         System.out.println("Using " + urlString);
      }
      else urlString = args[0];
      URL url = new URL(urlString);
      InputStream in = url.openStream();
      XMLInputFactory factory = XMLInputFactory.newInstance();
      XMLStreamReader parser = factory.createXMLStreamReader(in);
      while (parser.hasNext())
      {
         int event = parser.next();
         if (event == XMLStreamConstants.START_ELEMENT)
         {
            if (parser.getLocalName().equals("a"))
            {
               String href = parser.getAttributeValue(null, "href");
               if (href != null)
                  System.out.println(href);
            }
         }
      }
   }
}
```

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 support cannot be set to the given value. The Java SE implementation supports the following `Boolean` valued p

`"javax.xml.stream.isValidating"`	When false (the default), the document is not Not required by the specification.
`"javax.xml.stream.isNamespaceAware"`	When true (the default), namespaces are pro required by the specification.
`"javax.xml.stream.isCoalescing"`	When false (the default), adjacent character c coalesced.
`"javax.xml.stream.isReplacingEntityReferences"`	When true (the default), entity references are and reported as character data.
`"javax.xml.stream.isSupportingExternalEntities"`	When true (the default), external entities are The specification gives no default for this prop
`"javax.xml.stream.supportDTD"`	When true (the default), DTDs are reported a

- `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.

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.DOCTYPE_SYSTEM, systemIdentifier);
t.setOutputProperty(OutputKeys.DOCTYPE_PUBLIC, 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)));
```

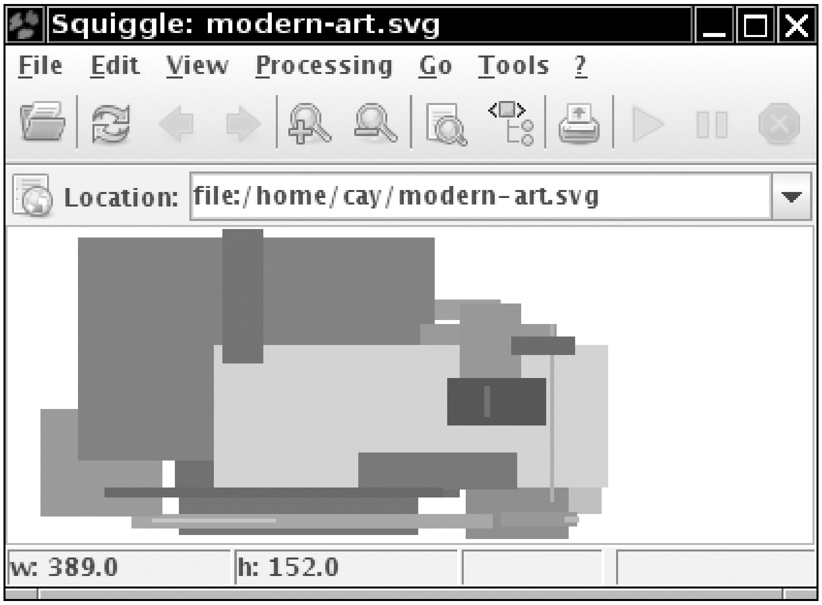

Note

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.

API

`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 `null`
	`qname`	The qualified name. If it has an alias prefix, then `uri` must not be `null`
	`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:

`doctype-public`	The public ID to be used in the `DOCTYPE` declaration
`doctype-system`	The system ID to be used in the `DOCTYPE` declaration
`indent`	`"yes"` or `"no"`
`method`	`"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 `<img .../>`), 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:

```
import java.awt.*;
import java.awt.geom.*;
import java.io.*;
import java.util.*;
import java.awt.event.*;
```



```
import javax.swing.*;
import javax.xml.parsers.*;
import javax.xml.stream.*;
import javax.xml.transform.*;
import javax.xml.transform.dom.*;
import javax.xml.transform.stream.*;
import org.w3c.dom.*;
```



```

/**
 * This program shows how to write an XML file. It saves a file describing a modern drawing
 * in SVG format.
 * @version 1.10 2004-09-04
 * @author Cay Horstmann
 */
public class XMLWriteTest
{
   public static void main(String[] args)
   {
      EventQueue.invokeLater(new Runnable()
         {
            public void run()
            {
               XMLWriteFrame frame = new XMLWriteFrame();
               frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
               frame.setVisible(true);
            }
         });
   }
}

/**
 * A frame with a component for showing a modern drawing.
 */
class XMLWriteFrame extends JFrame
{
   public XMLWriteFrame()
   {
      setTitle("XMLWriteTest");
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

      chooser = new JFileChooser();

      // add component to frame

      comp = new RectangleComponent();
      add(comp);

      // set up menu bar

      JMenuBar menuBar = new JMenuBar();
      setJMenuBar(menuBar);
```

```

      JMenu menu = new JMenu("File");
      menuBar.add(menu);

      JMenuItem newItem = new JMenuItem("New");
      menu.add(newItem);
      newItem.addActionListener(new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               comp.newDrawing();
            }
         });

      JMenuItem saveItem = new JMenuItem("Save with DOM/XSLT");
      menu.add(saveItem);
      saveItem.addActionListener(new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               try
               {
                  saveDocument();
               }
               catch (Exception e)
               {
                  JOptionPane.showMessageDialog(XMLWriteFrame.this, e.toString());
               }
            }
         });

      JMenuItem saveStAXItem = new JMenuItem("Save with StAX");
      menu.add(saveStAXItem);
      saveStAXItem.addActionListener(new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               try
               {
                  saveStAX();
               }
               catch (Exception e)
               {
                  JOptionPane.showMessageDialog(XMLWriteFrame.this, e.toString());
               }
            }
         });

      JMenuItem exitItem = new JMenuItem("Exit");
      menu.add(exitItem);
      exitItem.addActionListener(new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               System.exit(0);
            }
         });
   }

```

```
   /**
    * Saves the drawing in SVG format, using DOM/XSLT
    */
   public void saveDocument() throws TransformerException, IOException
   {
      if (chooser.showSaveDialog(this) != JFileChooser.APPROVE_OPTION) return;
      File f = chooser.getSelectedFile();
      Document doc = comp.buildDocument();
      Transformer t = TransformerFactory.newInstance().newTransformer();
```

```
      t.setOutputProperty(OutputKeys.DOCTYPE_SYSTEM,
            "http://www.w3.org/TR/2000/CR-SVG-20000802/DTD/svg-20000802.dtd");
      t.setOutputProperty(OutputKeys.DOCTYPE_PUBLIC, "-//W3C//DTD SVG 20000802//EN");
      t.setOutputProperty(OutputKeys.INDENT, "yes");
      t.setOutputProperty(OutputKeys.METHOD, "xml");
      t.setOutputProperty("{http://xml.apache.org/xslt}indent-amount", "2");
      t.transform(new DOMSource(doc), new StreamResult(new FileOutputStream(f)));
```

```
   }

   /**
    * Saves the drawing in SVG format, using StAX
    */
   public void saveStAX() throws FileNotFoundException, XMLStreamException
   {
      if (chooser.showSaveDialog(this) != JFileChooser.APPROVE_OPTION) return;
      File f = chooser.getSelectedFile();
      XMLOutputFactory factory = XMLOutputFactory.newInstance();
      XMLStreamWriter writer = factory.createXMLStreamWriter(new FileOutputStream(f));
      comp.writeDocument(writer);
      writer.close();
   }

   public static final int DEFAULT_WIDTH = 300;
   public static final int DEFAULT_HEIGHT = 200;

   private RectangleComponent comp;
   private JFileChooser chooser;
}

/**
 * A component that shows a set of colored rectangles
 */
class RectangleComponent extends JComponent
{
   public RectangleComponent()
   {
      rects = new ArrayList<Rectangle2D>();
      colors = new ArrayList<Color>();
      generator = new Random();

      DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
      try
      {
         builder = factory.newDocumentBuilder();
      }
      catch (ParserConfigurationException e)
      {
         e.printStackTrace();
      }
   }
```

```
   /**
    * Create a new random drawing.
    */
   public void newDrawing()
   {
      int n = 10 + generator.nextInt(20);
      rects.clear();
      colors.clear();
      for (int i = 1; i <= n; i++)
      {
         int x = generator.nextInt(getWidth());
         int y = generator.nextInt(getHeight());
         int width = generator.nextInt(getWidth() - x);
         int height = generator.nextInt(getHeight() - y);
         rects.add(new Rectangle(x, y, width, height));
         int r = generator.nextInt(256);
         int g = generator.nextInt(256);
         int b = generator.nextInt(256);
         colors.add(new Color(r, g, b));
      }
      repaint();
   }

   public void paintComponent(Graphics g)
   {
      if (rects.size() == 0) newDrawing();
      Graphics2D g2 = (Graphics2D) g;
      // draw all rectangles
      for (int i = 0; i < rects.size(); i++)
      {
         g2.setPaint(colors.get(i));
         g2.fill(rects.get(i));
      }
   }

   /**
    * Creates an SVG document of the current drawing.
    * @return the DOM tree of the SVG document
    */
   public Document buildDocument()
   {
      Document doc = builder.newDocument();
      Element svgElement = doc.createElement("svg");
      doc.appendChild(svgElement);
      svgElement.setAttribute("width", "" + getWidth());
      svgElement.setAttribute("height", "" + getHeight());
      for (int i = 0; i < rects.size(); i++)
      {
         Color c = colors.get(i);
         Rectangle2D r = rects.get(i);
         Element rectElement = doc.createElement("rect");
         rectElement.setAttribute("x", "" + r.getX());
         rectElement.setAttribute("y", "" + r.getY());
         rectElement.setAttribute("width", "" + r.getWidth());
         rectElement.setAttribute("height", "" + r.getHeight());
         rectElement.setAttribute("fill", colorToString(c));
         svgElement.appendChild(rectElement);
      }
```

```
      return doc;
   }

   /**
    * Writers an SVG document of the current drawing.
    * @param writer the document destination
    */
   public void writeDocument(XMLStreamWriter writer) throws XMLStreamException
   {
      writer.writeStartDocument();
      writer.writeDTD("<!DOCTYPE svg PUBLIC \"-//W3C//DTD SVG 20000802//EN\" "
            + "\"http://www.w3.org/TR/2000/CR-SVG-20000802/DTD/svg-20000802.dtd\">");
      writer.writeStartElement("svg");
      writer.writeAttribute("width", "" + getWidth());
      writer.writeAttribute("height", "" + getHeight());
      for (int i = 0; i < rects.size(); i++)
      {
         Color c = colors.get(i);
         Rectangle2D r = rects.get(i);
         writer.writeEmptyElement("rect");
         writer.writeAttribute("x", "" + r.getX());
         writer.writeAttribute("y", "" + r.getY());
         writer.writeAttribute("width", "" + r.getWidth());
         writer.writeAttribute("height", "" + r.getHeight());
         writer.writeAttribute("fill", colorToString(c));
      }
      writer.writeEndDocument(); // closes svg element
   }

   /**
    * Converts a color to a hex value.
    * @param c a color
    * @return a string of the form #rrggbb
    */
   private static String colorToString(Color c)
   {
      StringBuffer buffer = new StringBuffer();
      buffer.append(Integer.toHexString(c.getRGB() & 0xFFFFFF));
      while (buffer.length() < 6)
         buffer.insert(0, '0');
      buffer.insert(0, '#');
      return buffer.toString();
   }

   private ArrayList<Rectangle2D> rects;
   private ArrayList<Color> colors;
   private Random generator;
   private DocumentBuilder builder;
}
```

`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()`

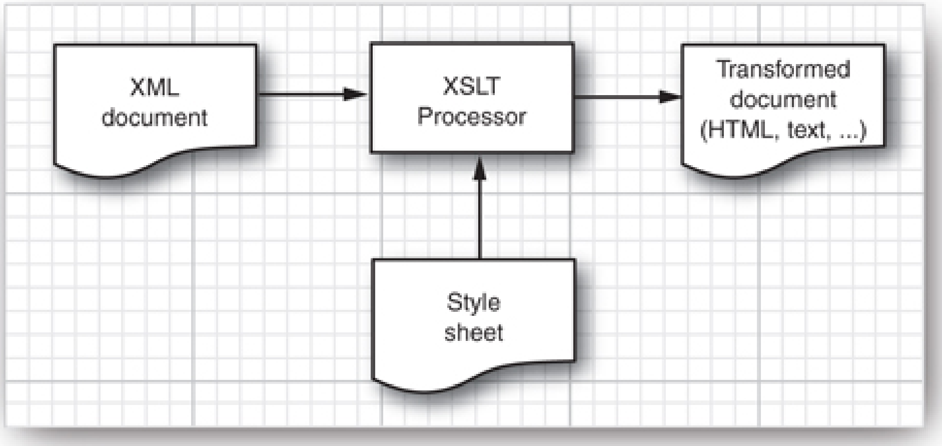
closes this writer.

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:

```
<?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"/>

   <xsl:template match="/staff">
      <table border="1"><xsl:apply-templates/></table>
   </xsl:template>

   <xsl:template match="/staff/employee">
      <tr><xsl:apply-templates/></tr>
   </xsl:template>

   <xsl:template match="/staff/employee/name">
      <td><xsl:apply-templates/></td>
   </xsl:template>

   <xsl:template match="/staff/employee/salary">
      <td>$<xsl:apply-templates/></td>
   </xsl:template>

   <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>

</xsl:stylesheet>
```

Listing 2-12. **makeprop.xsl**

Code View:

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

<xsl:stylesheet
   xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
   version="1.0">

   <xsl:output method="text"/>
   <xsl:template match="/staff/employee">
employee.<xsl:value-of select="position()"/>.name=<xsl:value-of select="name/text()"/>
employee.<xsl:value-of select="position()"/>.salary=<xsl:value-of select="salary/text()"/>
employee.<xsl:value-of select="position()"/>.hiredate=<xsl:value-of select="hiredate/@year"/>
        -<xsl:value-of select="hiredate/@month"/>-<xsl:value-of select="hiredate/@day"/>

   </xsl:template>

</xsl:stylesheet>
```


Listing 2-13. **TransformTest.java**

Code View:

```
import java.io.*;
import java.util.*;
import javax.xml.transform.*;
import javax.xml.transform.sax.*;
import javax.xml.transform.stream.*;
import org.xml.sax.*;
import org.xml.sax.helpers.*;

/**
 * This program demonstrates XSL transformations. It applies a transformation to a set
 * of employee records. The records are stored in the file employee.dat and turned into XML
 * format. Specify the stylesheet on the command line, e.g. java TransformTest makeprop.xsl
 * @version 1.01 2007-06-25
 * @author Cay Horstmann
 */
public class TransformTest
{
   public static void main(String[] args) throws Exception
   {
      String filename;
      if (args.length > 0) filename = args[0];
      else filename = "makehtml.xsl";
      File styleSheet = new File(filename);
      StreamSource styleSource = new StreamSource(styleSheet);

      Transformer t = TransformerFactory.newInstance().newTransformer(styleSource);
      t.setOutputProperty(OutputKeys.INDENT, "yes");
      t.setOutputProperty(OutputKeys.METHOD, "xml");
      t.setOutputProperty("{http://xml.apache.org/xslt}indent-amount", "2");

      t.transform(new SAXSource(new EmployeeReader(), new InputSource(new FileInputStream(
            "employee.dat"))), new StreamResult(System.out));
   }
```

```
}
/**
 * This class reads the flat file employee.dat and reports SAX parser events to act as if
 * it was parsing an XML file.
 */
class EmployeeReader implements XMLReader
{
   public void parse(InputSource source) throws IOException, SAXException
   {
      InputStream stream = source.getByteStream();
      BufferedReader in = new BufferedReader(new InputStreamReader(stream));
      String rootElement = "staff";
      AttributesImpl atts = new AttributesImpl();

      if (handler == null) throw new SAXException("No content handler");

      handler.startDocument();
      handler.startElement("", rootElement, rootElement, atts);
      String line;
      while ((line = in.readLine()) != null)
      {
         handler.startElement("", "employee", "employee", atts);
         StringTokenizer t = new StringTokenizer(line, "|");

         handler.startElement("", "name", "name", atts);
         String s = t.nextToken();
         handler.characters(s.toCharArray(), 0, s.length());
         handler.endElement("", "name", "name");

         handler.startElement("", "salary", "salary", atts);
         s = t.nextToken();
         handler.characters(s.toCharArray(), 0, s.length());
         handler.endElement("", "salary", "salary");

         atts.addAttribute("", "year", "year", "CDATA", t.nextToken());
         atts.addAttribute("", "month", "month", "CDATA", t.nextToken());
         atts.addAttribute("", "day", "day", "CDATA", t.nextToken());
         handler.startElement("", "hiredate", "hiredate", atts);
         handler.endElement("", "hiredate", "hiredate");
         atts.clear();

         handler.endElement("", "employee", "employee");
      }

      handler.endElement("", rootElement, rootElement);
      handler.endDocument();
   }

   public void setContentHandler(ContentHandler newValue)
   {
      handler = newValue;
   }

   public ContentHandler getContentHandler()
   {
      return handler;
   }

   // the following methods are just do-nothing implementations
```

```
   public void parse(String systemId) throws IOException, SAXException
   {
   }

   public void setErrorHandler(ErrorHandler handler)
   {
   }

   public ErrorHandler getErrorHandler()
   {
      return null;
   }

   public void setDTDHandler(DTDHandler handler)
   {
   }

   public DTDHandler getDTDHandler()
   {
      return null;
   }

   public void setEntityResolver(EntityResolver resolver)
   {
   }

   public EntityResolver getEntityResolver()
   {
      return null;
   }

   public void setProperty(String name, Object value)
   {
   }

   public Object getProperty(String name)
   {
      return null;
   }

   public void setFeature(String name, boolean value)
   {
   }

   public boolean getFeature(String name)
   {
      return false;
   }

   private ContentHandler handler;
}
```


UNREGISTERED VERSION OF CHM TO PDF CONVERTER PRO BY THETA-SOFTWARE

`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.

javax.xml.transform.dom.DOMResult 1.4

- `DOMResult(Node n)`

  constructs a source from the given node. Usually, `n` is a new document node.

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

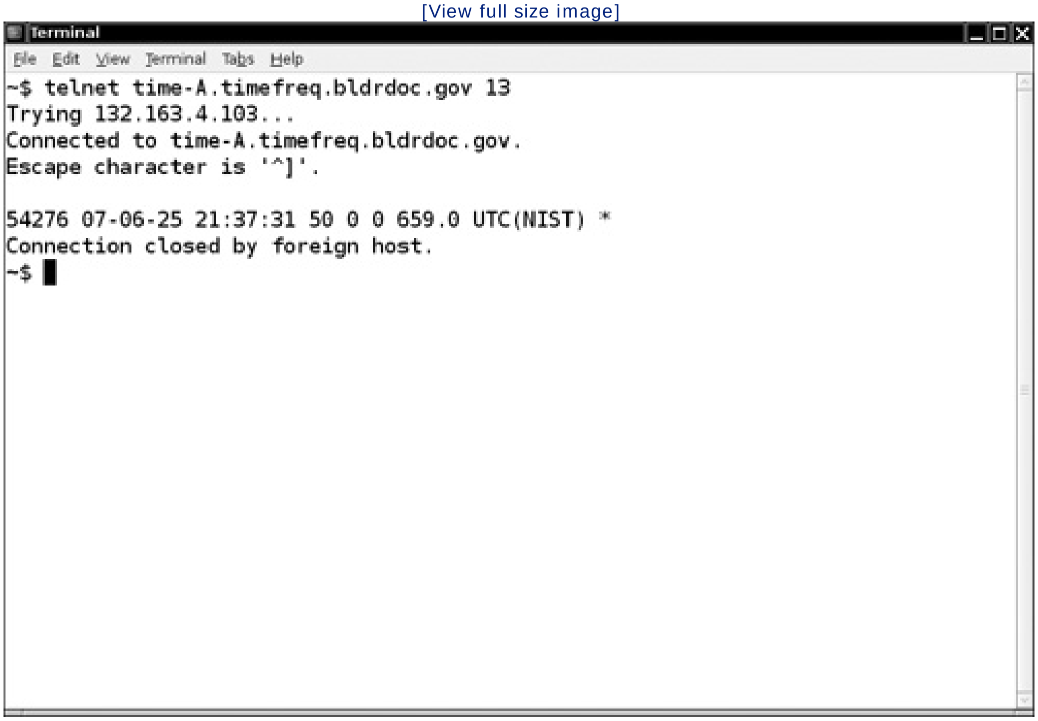
```
telnet time-A.timefreq.bldrdoc.gov 13
```

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]



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.)

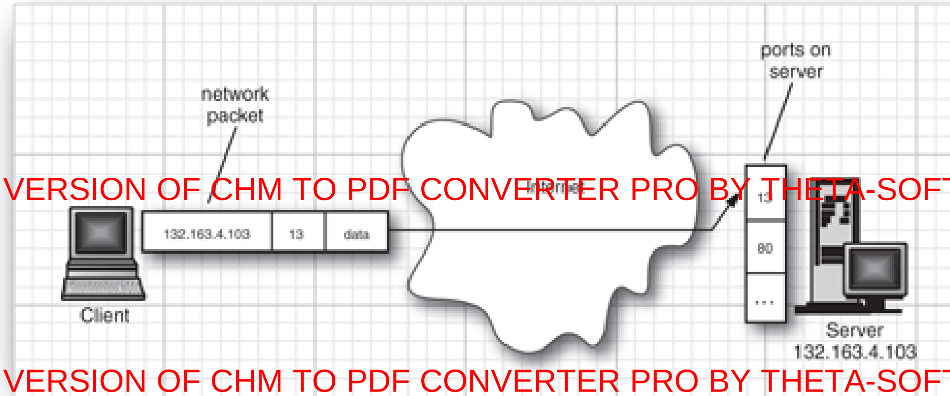
By convention, the "time of day" service is always attached to "port" number 13.

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]



UNREGISTERED VERSION OF CHM TO PDF CONVERTER PRO BY THETA-SOFTWARE

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]

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:

```
import java.io.*;
import java.net.*;
import java.util.*;

/**
 * This program makes a socket connection to the atomic clock in Boulder, Colorado, anc
 * prints the time that the server sends.
 * @version 1.20 2004-08-03
 * @author Cay Horstmann
 */
public class SocketTest
{
   public static void main(String[] args)
   {
      try
      {
         Socket s = new Socket("time-A.timefreq.bldrdoc.gov", 13);
         try
         {
            InputStream inStream = s.getInputStream();
            Scanner in = new Scanner(inStream);

            while (in.hasNextLine())
            {
               String line = in.nextLine();
               System.out.println(line);
            }
         }
         finally
         {
            s.close();
         }
      }
      catch (IOException e)
      {
         e.printStackTrace();
      }
   }
}
```


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.

API

`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.

API `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:

```
import java.net.*;

/**
 * This program demonstrates the InetAddress class. Supply a host name as command line
 * argument, or run without command line arguments to see the address of the local host.
 * @version 1.01 2001-06-26
 * @author Cay Horstmann
 */
public class InetAddressTest
{
   public static void main(String[] args)
   {
      try
      {
         if (args.length > 0)
         {
            String host = args[0];
            InetAddress[] addresses = InetAddress.getAllByName(host);
            for (InetAddress a : addresses)
               System.out.println(a);
         }
         else
         {
            InetAddress localHostAddress = InetAddress.getLocalHost();
            System.out.println(localHostAddress);
         }
      }
      catch (Exception e)
      {
         e.printStackTrace();
      }
   }
}
```

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.

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

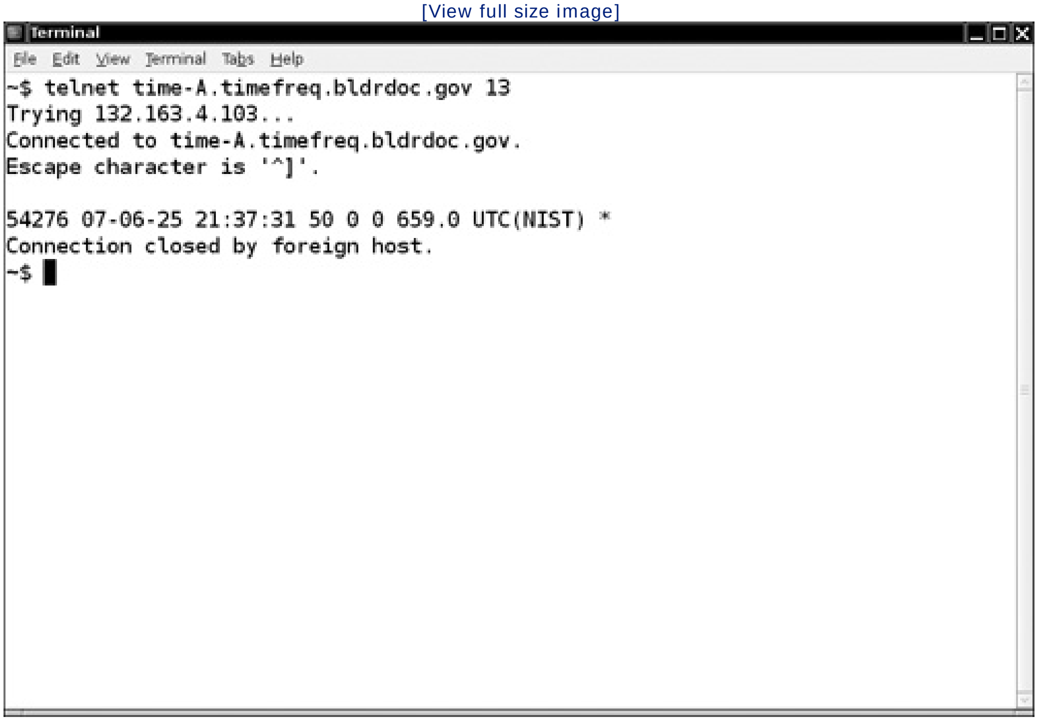
```
telnet time-A.timefreq.bldrdoc.gov 13
```

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]



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.)

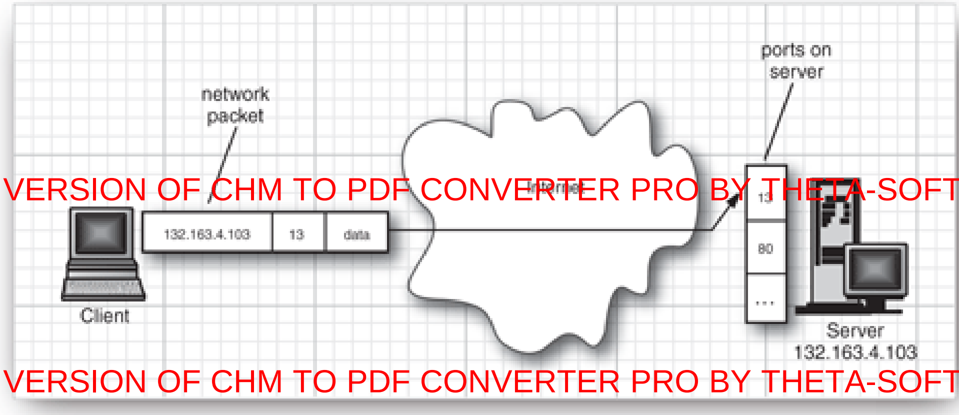
By convention, the "time of day" service is always attached to "port" number 13.

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]

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:

```
import java.io.*;
import java.net.*;
import java.util.*;

/**
 * This program makes a socket connection to the atomic clock in Boulder, Colorado, and
 * prints the time that the server sends.
 * @version 1.20 2004-08-03
 * @author Cay Horstmann
 */
public class SocketTest
{
   public static void main(String[] args)
   {
      try
      {
         Socket s = new Socket("time-A.timefreq.bldrdoc.gov", 13);
         try
         {
            InputStream inStream = s.getInputStream();
            Scanner in = new Scanner(inStream);

            while (in.hasNextLine())
            {
               String line = in.nextLine();
               System.out.println(line);
            }
         }
         finally
         {
            s.close();
         }
      }
      catch (IOException e)
      {
         e.printStackTrace();
      }
   }
}
```

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.

API

`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.

API **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:
```
import java.net.*;

/**
 * This program demonstrates the InetAddress class. Supply a host name as command line
 * argument, or run without command line arguments to see the address of the local host.
 * @version 1.01 2001-06-26
 * @author Cay Horstmann
 */
public class InetAddressTest
{
   public static void main(String[] args)
   {
      try
      {
         if (args.length > 0)
         {
            String host = args[0];
            InetAddress[] addresses = InetAddress.getAllByName(host);
            for (InetAddress a : addresses)
               System.out.println(a);
         }
         else
         {
            InetAddress localHostAddress = InetAddress.getLocalHost();
            System.out.println(localHostAddress);
         }
      }
      catch (Exception e)
      {
         e.printStackTrace();
      }
   }
}
```

`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:
```
import java.io.*;
import java.net.*;
import java.util.*;

/**
 * This program implements a simple server that listens to port 8189 and echoes back all
 * client input.
 * @version 1.20 2004-08-03
 * @author Cay Horstmann
 */
public class EchoServer
{
   public static void main(String[] args)
   {
      try
      {
         // establish server socket
         ServerSocket s = new ServerSocket(8189);

         // wait for client connection
         Socket incoming = s.accept();
         try
         {
            InputStream inStream = incoming.getInputStream();
            OutputStream outStream = incoming.getOutputStream();

            Scanner in = new Scanner(inStream);
            PrintWriter out = new PrintWriter(outStream, true /* autoFlush */);

            out.println("Hello! Enter BYE to exit.");

            // echo client input
            boolean done = false;
            while (!done && in.hasNextLine())
            {
               String line = in.nextLine();
```

```
                  out.println("Echo: " + line);
                  if (line.trim().equals("BYE")) done = true;
               }
            }
            finally
            {
               incoming.close();
            }
         }
         catch (IOException e)
         {
            e.printStackTrace();
         }
      }
   }
```


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.

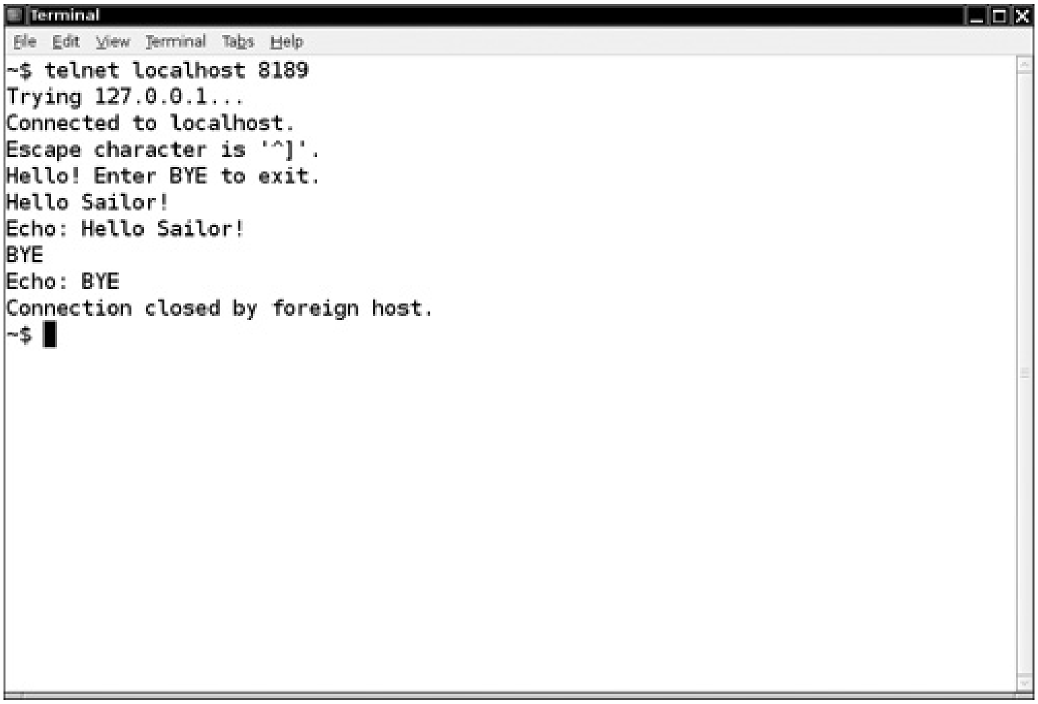
If you are connected directly to the Internet, then anyone in the world can access your echo server, provided they know your IP address and the magic port number.

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.

API

`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
      }
   }
}
```

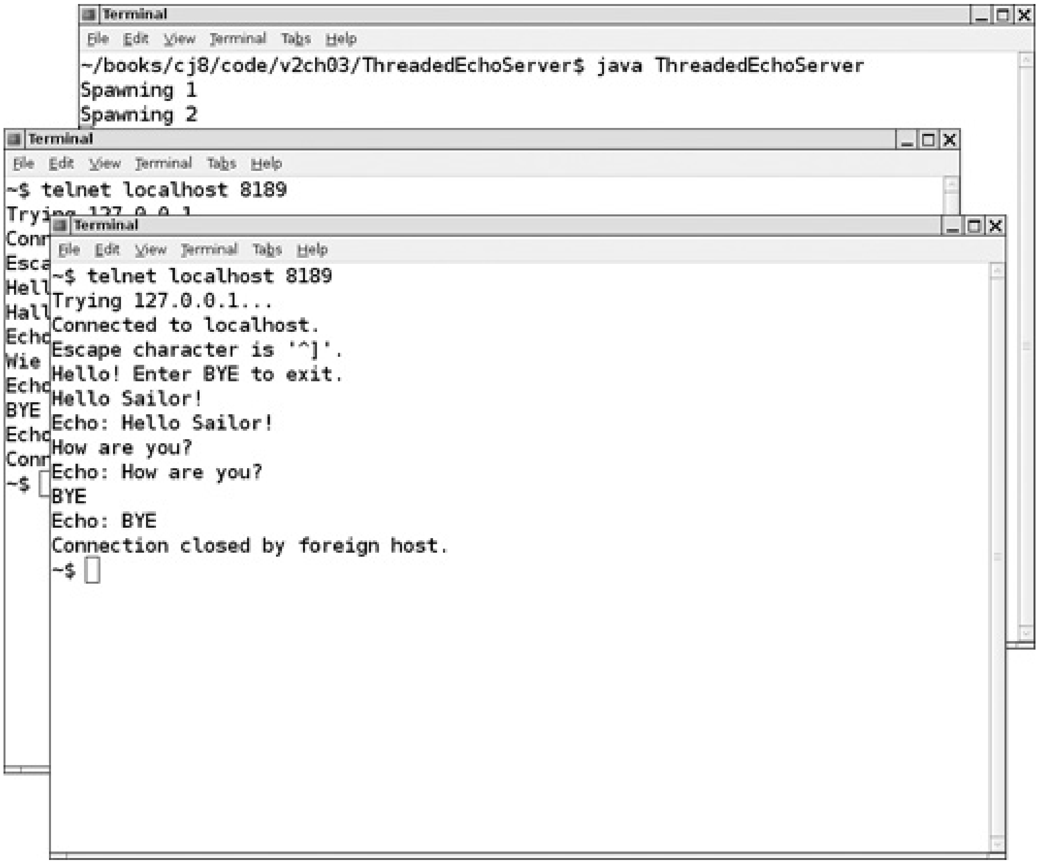
Because each connection starts a new thread, multiple clients can connect to the server at the same time. You can easily check this out.

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:
```
import java.io.*;
import java.net.*;
import java.util.*;
```

```java

/**
   This program implements a multithreaded server that listens to port 8189 and echoes back
   all client input.
   @author Cay Horstmann
   @version 1.20 2004-08-03
*/
public class ThreadedEchoServer
{
```

```java
   public static void main(String[] args )
   {
      try
      {
         int i = 1;
         ServerSocket s = new ServerSocket(8189);
```

```java
         while (true)
         {
            Socket incoming = s.accept();
            System.out.println("Spawning " + i);
            Runnable r = new ThreadedEchoHandler(incoming);
            Thread t = new Thread(r);
            t.start();
            i++;
         }
      }
      catch (IOException e)
      {
         e.printStackTrace();
      }
   }
}

/**
   This class handles the client input for one server socket connection.
*/
class ThreadedEchoHandler implements Runnable
{
   /**
      Constructs a handler.
      @param i the incoming socket
      @param c the counter for the handlers (used in prompts)
   */
   public ThreadedEchoHandler(Socket i)
   {
      incoming = i;
   }

   public void run()
   {
      try
      {
         try
         {
            InputStream inStream = incoming.getInputStream();
            OutputStream outStream = incoming.getOutputStream();

            Scanner in = new Scanner(inStream);
            PrintWriter out = new PrintWriter(outStream, true /* autoFlush */);
```

```

            out.println( "Hello! Enter BYE to exit." );

            // echo client input
            boolean done = false;
            while (!done && in.hasNextLine())
            {
               String line = in.nextLine();
               out.println("Echo: " + line);
               if (line.trim().equals("BYE"))
                  done = true;
            }
         }
         finally
         {
            incoming.close();
         }
      }
      catch (IOException e)
      {
         e.printStackTrace();
      }
   }

   private Socket incoming;
}
```

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);
```

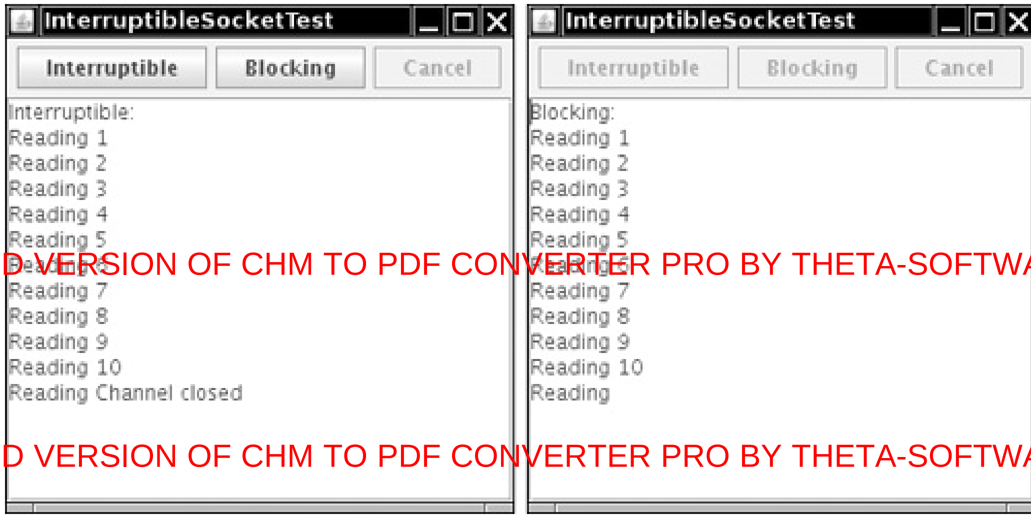
That's all you need to do. Whenever a thread is interrupted during an open, read, or write operation, the operation does not block, but is terminated with an exception.

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

[View full size image]



UNREGISTERED VERSION OF CHM TO PDF CONVERTER PRO BY THETA-SOFTWARE

Listing 3-5. **InterruptibleSocketTest.java**

Code View:
```
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import java.net.*;
import java.io.*;
import java.nio.channels.*;
import javax.swing.*;

/**
 * This program shows how to interrupt a socket channel.
 * @author Cay Horstmann
 * @version 1.01 2007-06-25
 */
public class InterruptibleSocketTest
{
   public static void main(String[] args)
   {
      EventQueue.invokeLater(new Runnable()
         {
            public void run()
            {
               JFrame frame = new InterruptibleSocketFrame();
               frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
               frame.setVisible(true);
            }
         });
   }
}

class InterruptibleSocketFrame extends JFrame
{
   public InterruptibleSocketFrame()
   {
      setSize(WIDTH, HEIGHT);
      setTitle("InterruptibleSocketTest");

      JPanel northPanel = new JPanel();
```

```
      add(northPanel, BorderLayout.NORTH);

      messages = new JTextArea();
      add(new JScrollPane(messages));

      interruptibleButton = new JButton("Interruptible");
      blockingButton = new JButton("Blocking");

      northPanel.add(interruptibleButton);
      northPanel.add(blockingButton);

      interruptibleButton.addActionListener(new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               interruptibleButton.setEnabled(false);
               blockingButton.setEnabled(false);
               cancelButton.setEnabled(true);
               connectThread = new Thread(new Runnable()
                  {
                     public void run()
                     {
                        try
                        {
                           connectInterruptibly();
                        }
                        catch (IOException e)
                        {
                           messages.append("\nInterruptibleSocketTest.connectInterruptibly: "
                                           + e);
                        }
                     }
                  });
               connectThread.start();
            }
         });

      blockingButton.addActionListener(new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               interruptibleButton.setEnabled(false);
               blockingButton.setEnabled(false);
               cancelButton.setEnabled(true);
               connectThread = new Thread(new Runnable()
                  {
                     public void run()
                     {
                        try
                        {
                           connectBlocking();
                        }
                        catch (IOException e)
                        {
                          messages.append("\nInterruptibleSocketTest.connectBlocking: " + e);
                        }
                     }
                  });
               connectThread.start();
```

```
            }
         });

      cancelButton = new JButton("Cancel");
      cancelButton.setEnabled(false);
      northPanel.add(cancelButton);
      cancelButton.addActionListener(new ActionListener()
         {
```

```
            public void actionPerformed(ActionEvent event)
            {
               connectThread.interrupt();
               cancelButton.setEnabled(false);
            }
         });
      server = new TestServer();
```
UNREGISTERED VERSION OF CHM TO PDF CONVERTER PRO BY THETA-SOFTWARE
```
      new Thread(server).start();
   }

   /**
    * Connects to the test server, using interruptible I/O
    */
   public void connectInterruptibly() throws IOException
   {
      messages.append("Interruptible:\n");
      SocketChannel channel = SocketChannel.open(new InetSocketAddress("localhost", 8189));
      try
      {
         in = new Scanner(channel);
         while (!Thread.currentThread().isInterrupted())
         {
            messages.append("Reading ");
            if (in.hasNextLine())
            {
               String line = in.nextLine();
               messages.append(line);
               messages.append("\n");
            }
         }
      }
      finally
      {
         channel.close();
         EventQueue.invokeLater(new Runnable()
         {
            public void run()
            {
               messages.append("Channel closed\n");
               interruptibleButton.setEnabled(true);
               blockingButton.setEnabled(true);
            }
         });
      }
   }

   /**
    * Connects to the test server, using blocking I/O
    */
   public void connectBlocking() throws IOException
   {
```

```
      messages.append("Blocking:\n");
      Socket sock = new Socket("localhost", 8189);
      try
      {
         in = new Scanner(sock.getInputStream());
         while (!Thread.currentThread().isInterrupted())
         {
            messages.append("Reading ");
            if (in.hasNextLine())
            {
               String line = in.nextLine();
               messages.append(line);
               messages.append("\n");
            }
         }
      }
      finally
      {
         sock.close();
         EventQueue.invokeLater(new Runnable()
         {
            public void run()
            {
               messages.append("Socket closed\n");
               interruptibleButton.setEnabled(true);
               blockingButton.setEnabled(true);
            }
         });
      }
   }

   /**
    * A multithreaded server that listens to port 8189 and sends numbers to the client,
    * simulating a hanging server after 10 numbers.
    */
   class TestServer implements Runnable
   {
      public void run()
      {
         try
         {
            ServerSocket s = new ServerSocket(8189);

            while (true)
            {
               Socket incoming = s.accept();
               Runnable r = new TestServerHandler(incoming);
               Thread t = new Thread(r);
               t.start();
            }
         }
         catch (IOException e)
         {
            messages.append("\nTestServer.run: " + e);
         }
      }
   }

   /**
```

```
    * This class handles the client input for one server socket connection.
    */
   class TestServerHandler implements Runnable
   {
      /**
       * Constructs a handler.
       * @param i the incoming socket
       */
      public TestServerHandler(Socket i)
```



```
      {
         incoming = i;
      }

      public void run()
      {
         try
```



```
         {
            OutputStream outStream = incoming.getOutputStream();
            PrintWriter out = new PrintWriter(outStream, true /* autoFlush */);
            while (counter < 100)
            {
               counter++;
               if (counter <= 10) out.println(counter);
               Thread.sleep(100);
            }
            incoming.close();
            messages.append("Closing server\n");
         }
         catch (Exception e)
         {
            messages.append("\nTestServerHandler.run: " + e);
         }
      }

      private Socket incoming;
      private int counter;
   }

   private Scanner in;
   private JButton interruptibleButton;
   private JButton blockingButton;
   private JButton cancelButton;
   private JTextArea messages;
   private TestServer server;
   private Thread connectThread;

   public static final int WIDTH = 300;
   public static final int HEIGHT = 300;
}
```

API

`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 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:

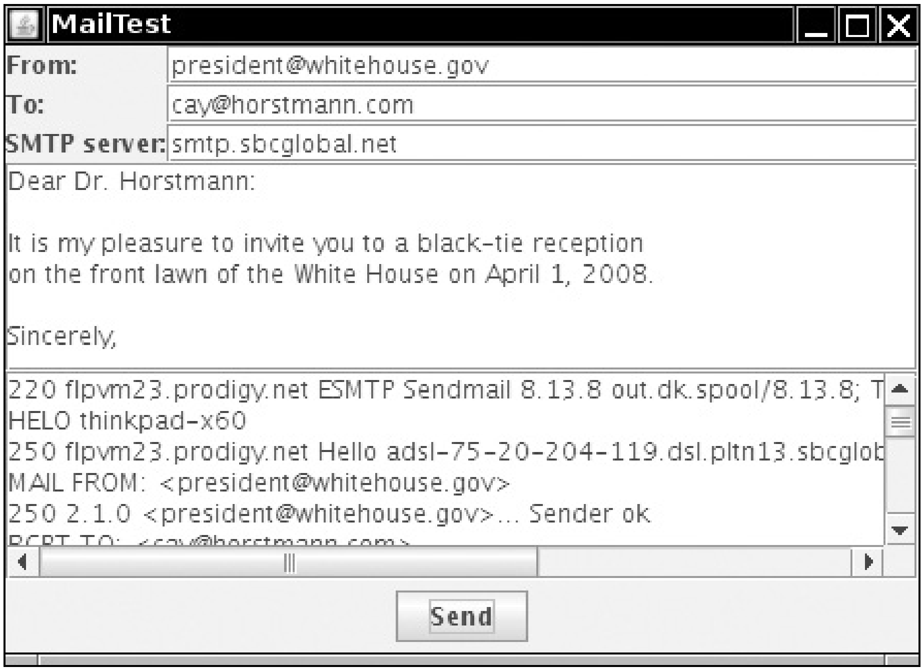
   ```
   HELO sending host
   MAIL FROM: <sender e-mail address>
   RCPT TO: <recipient e-mail address>
   DATA
   mail message
   (any number of lines)
   .
   QUIT
   ```

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:


```
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import java.net.*;
import java.io.*;
import javax.swing.*;

/**
 * This program shows how to use sockets to send plain text mail messages.
 * @author Cay Horstmann
 * @version 1.11 2007-06-25
 */
public class MailTest
{
   public static void main(String[] args)
   {
      EventQueue.invokeLater(new Runnable()
         {
            public void run()
            {
               JFrame frame = new MailTestFrame();
               frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
               frame.setVisible(true);
            }
         });
   }
}

/**
 * The frame for the mail GUI.
 */
class MailTestFrame extends JFrame
{
   public MailTestFrame()
   {
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
      setTitle("MailTest");

      setLayout(new GridBagLayout());

      // we use the GBC convenience class of Core Java Volume I, Chapter 9
      add(new JLabel("From:"), new GBC(0, 0).setFill(GBC.HORIZONTAL));

      from = new JTextField(20);
      add(from, new GBC(1, 0).setFill(GBC.HORIZONTAL).setWeight(100, 0));

      add(new JLabel("To:"), new GBC(0, 1).setFill(GBC.HORIZONTAL));

      to = new JTextField(20);
      add(to, new GBC(1, 1).setFill(GBC.HORIZONTAL).setWeight(100, 0));

      add(new JLabel("SMTP server:"), new GBC(0, 2).setFill(GBC.HORIZONTAL));

```

```java
      smtpServer = new JTextField(20);
      add(smtpServer, new GBC(1, 2).setFill(GBC.HORIZONTAL).setWeight(100, 0));

      message = new JTextArea();
      add(new JScrollPane(message), new GBC(0, 3, 2, 1).setFill(GBC.BOTH).setWeight(100, 100));

      comm = new JTextArea();
      add(new JScrollPane(comm), new GBC(0, 4, 2, 1).setFill(GBC.BOTH).setWeight(100, 100));

      JPanel buttonPanel = new JPanel();
      add(buttonPanel, new GBC(0, 5, 2, 1));

      JButton sendButton = new JButton("Send");
      buttonPanel.add(sendButton);
      sendButton.addActionListener(new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               new SwingWorker<Void, Void>()
               {
                  protected Void doInBackground() throws Exception
                  {
                     comm.setText("");
                     sendMail();
                     return null;
                  }
               }.execute();
            }
         });
   }

   /**
    * Sends the mail message that has been authored in the GUI.
    */
   public void sendMail()
   {
      try
      {
         Socket s = new Socket(smtpServer.getText(), 25);

         InputStream inStream = s.getInputStream();
         OutputStream outStream = s.getOutputStream();

         in = new Scanner(inStream);
         out = new PrintWriter(outStream, true /* autoFlush */);

         String hostName = InetAddress.getLocalHost().getHostName();

         receive();
         send("HELO " + hostName);
         receive();
         send("MAIL FROM: <" + from.getText() + ">");
         receive();
         send("RCPT TO: <" + to.getText() + ">");
         receive();
         send("DATA");
         receive();
         send(message.getText());
         send(".");
```

```
         receive();
         s.close();
      }
      catch (IOException e)
      {
         comm.append("Error: " + e);
      }
   }
```



```
   /**
    * Sends a string to the socket and echoes it in the comm text area.
    * @param s the string to send.
    */
   public void send(String s) throws IOException
   {
```



```
      comm.append(s);
      comm.append("\n");
      out.print(s.replaceAll("\n", "\r\n"));
      out.print("\r\n");
      out.flush();
   }

   /**
    * Receives a string from the socket and displays it in the comm text area.
    */
   public void receive() throws IOException
   {
      String line = in.nextLine();
      comm.append(line);
      comm.append("\n");
   }

   private Scanner in;
   private PrintWriter out;
   private JTextField from;
   private JTextField to;
   private JTextField smtpServer;
   private JTextArea message;
   private JTextArea comm;

   public static final int DEFAULT_WIDTH = 300;
   public static final int DEFAULT_HEIGHT = 300;
}
```

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:

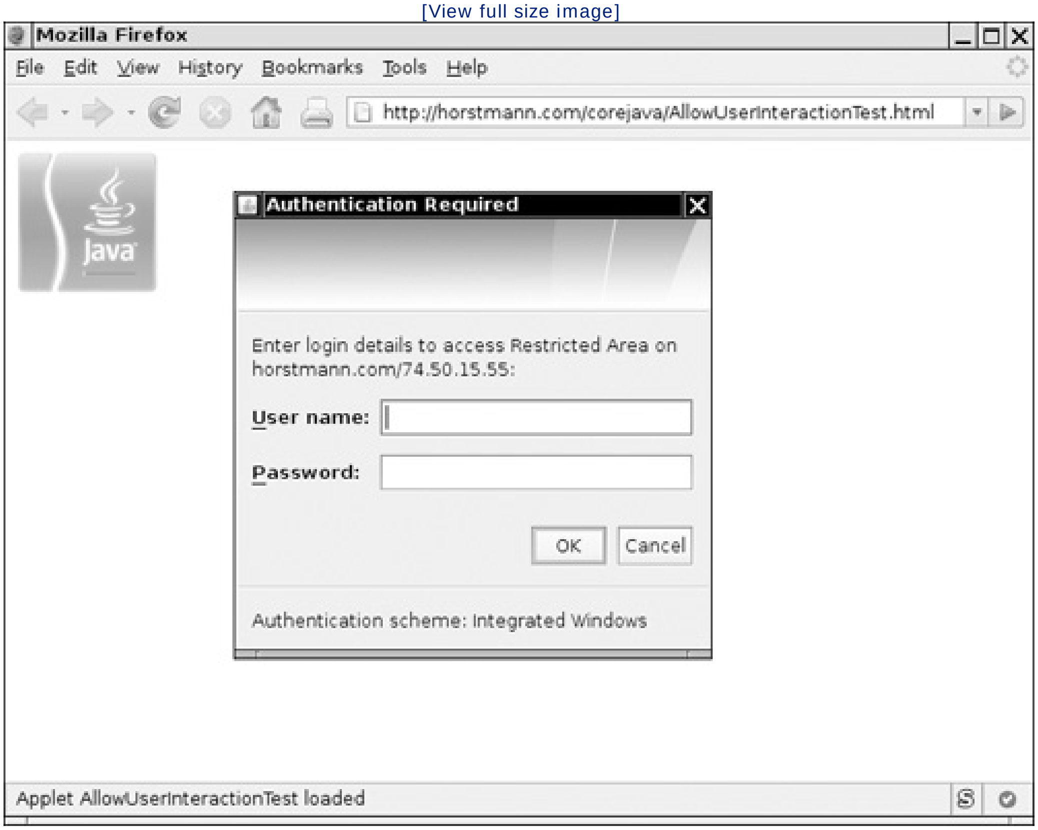
```
GET www.server.com/index.html HTTP/1.0
Referer: http://www.somewhere.com/links.html
Proxy-Connection: Keep-Alive
User-Agent: Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.8.1.4)
Host: www.server.com
Accept: text/html, image/gif, image/jpeg, image/png, */*
Accept-Language: en
Accept-Charset: iso-8859-1,*,utf-8
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:

```
import java.io.*;
import java.net.*;
import java.util.*;

/**
 * This program connects to a URL and displays the response header data and the first 10
 * lines of the requested data.
 *
 * Supply the URL and an optional username and password (for HTTP basic authentication) on
 * the command line.
 * @version 1.11 2007-06-26
 * @author Cay Horstmann
 */
public class URLConnectionTest
{
   public static void main(String[] args)
   {
      try
      {
         String urlName;
         if (args.length > 0) urlName = args[0];
         else urlName = "http://java.sun.com";

         URL url = new URL(urlName);
         URLConnection connection = url.openConnection();

         // set username, password if specified on command line

         if (args.length > 2)
         {
            String username = args[1];
            String password = args[2];
            String input = username + ":" + password;
            String encoding = base64Encode(input);
            connection.setRequestProperty("Authorization", "Basic " + encoding);
         }

         connection.connect();

         // print header fields

         Map<String, List<String>> headers = connection.getHeaderFields();
         for (Map.Entry<String, List<String>> entry : headers.entrySet())
         {
            String key = entry.getKey();
            for (String value : entry.getValue())
               System.out.println(key + ": " + value);
         }

         // print convenience functions

         System.out.println("----------");
         System.out.println("getContentType: " + connection.getContentType());
         System.out.println("getContentLength: " + connection.getContentLength());
         System.out.println("getContentEncoding: " + connection.getContentEncoding());
         System.out.println("getDate: " + connection.getDate());
```

```
         System.out.println("getExpiration: " + connection.getExpiration());
         System.out.println("getLastModifed: " + connection.getLastModified());
         System.out.println("----------");

         Scanner in = new Scanner(connection.getInputStream());

         // print first ten lines of contents

         for (int n = 1; in.hasNextLine() && n <= 10; n++)
            System.out.println(in.nextLine());
         if (in.hasNextLine()) System.out.println(". . .");
      }
      catch (IOException e)
      {
         e.printStackTrace();
      }
   }
   /**
    * Computes the Base64 encoding of a string
    * @param s a string
    * @return the Base 64 encoding of s
    */
   public static String base64Encode(String s)
   {
      ByteArrayOutputStream bOut = new ByteArrayOutputStream();
      Base64OutputStream out = new Base64OutputStream(bOut);
      try
      {
         out.write(s.getBytes());
         out.flush();
      }
      catch (IOException e)
      {
      }
      return bOut.toString();
   }
}

/**
 * This stream filter converts a stream of bytes to their Base64 encoding.
 *
 * Base64 encoding encodes 3 bytes into 4 characters. |11111122|22223333|33444444| Each set
 * of 6 bits is encoded according to the toBase64 map. If the number of input bytes is not a
 * multiple of 3, then the last group of 4 characters is padded with one or two = signs. Eacl
 * output line is at most 76 characters.
 */
class Base64OutputStream extends FilterOutputStream
{
   /**
    * Constructs the stream filter
    * @param out the stream to filter
    */
   public Base64OutputStream(OutputStream out)
   {
      super(out);
   }

   public void write(int c) throws IOException
   {
```

```
      inbuf[i] = c;
      i++;
      if (i == 3)
      {
         super.write(toBase64[(inbuf[0] & 0xFC) >> 2]);
         super.write(toBase64[((inbuf[0] & 0x03) << 4) | ((inbuf[1] & 0xF0) >> 4)]);
         super.write(toBase64[((inbuf[1] & 0x0F) << 2) | ((inbuf[2] & 0xC0) >> 6)]);
         super.write(toBase64[inbuf[2] & 0x3F]);
         col += 4;
         i = 0;
         if (col >= 76)
         {
            super.write('\n');
            col = 0;
         }
      }
   }

   public void flush() throws IOException
   {
      if (i == 1)
      {
         super.write(toBase64[(inbuf[0] & 0xFC) >> 2]);
         super.write(toBase64[(inbuf[0] & 0x03) << 4]);
         super.write('=');
         super.write('=');
      }
      else if (i == 2)
      {
         super.write(toBase64[(inbuf[0] & 0xFC) >> 2]);
         super.write(toBase64[((inbuf[0] & 0x03) << 4) | ((inbuf[1] & 0xF0) >> 4)]);
         super.write(toBase64[(inbuf[1] & 0x0F) << 2]);
         super.write('=');
      }
      if (col > 0)
      {
         super.write('\n');
         col = 0;
      }
   }

   private static char[] toBase64 = { 'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K',
         'L', 'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z', 'a', 'b',
         'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r', 's',
         't', 'u', 'v', 'w', 'x', 'y', 'z', '0', '1', '2', '3', '4', '5', '6', '7', '8', '9',
         '+', '/' };

   private int col = 0;
   private int i = 0;
   private int[] inbuf = new int[3];
}
```

Note

A commonly asked question is whether the Java platform supports access of secure web pages (`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.

API

`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.

API

`java.net.URLConnection` 1.0

- `void setDoInput(boolean doInput)`

- `boolean getDoInput()`

  If `doInput` is `true`, then the user can receive input from this `URLConnection`.

- `void setDoOutput(boolean doOutput)`

- `boolean getDoOutput()`

  If `doOutput` is `true`, then the user can send output to this `URLConnection`.

- `void setIfModifiedSince(long time)`

- 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)

- boolean getUseCaches()

  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.

- void setAllowUserInteraction(boolean allowUserInteraction)

- boolean getAllowsUserInteraction()

  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.

- 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 $\leq$ 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 $\leq$ 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 getLastModifed()`

  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.

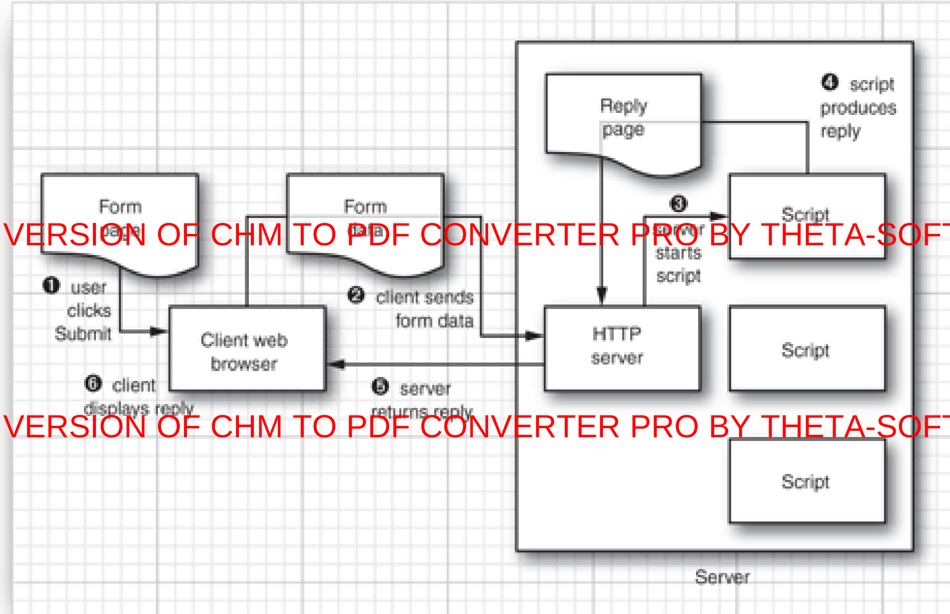
Many technologies enable web servers to invoke programs. Among the best known ones are Java servlets, JavaServer Faces, Microsoft Active Server Pages (ASP), and Common Gateway Interface (CGI) scripts. For simplicity, we use the generic term *script* for a server-side program, no matter what technology 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]

UNREGISTERED VERSION OF CHM TO PDF CONVERTER PRO BY THETA-SOFTWARE



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&Varient=2&StartYear=1950&EndYear=2050&
   DoWhat=Download+as+%2eCSV+File
```

Send the string to the URL

```
http://esa.un.org/unpp/p2k0data.asp
```

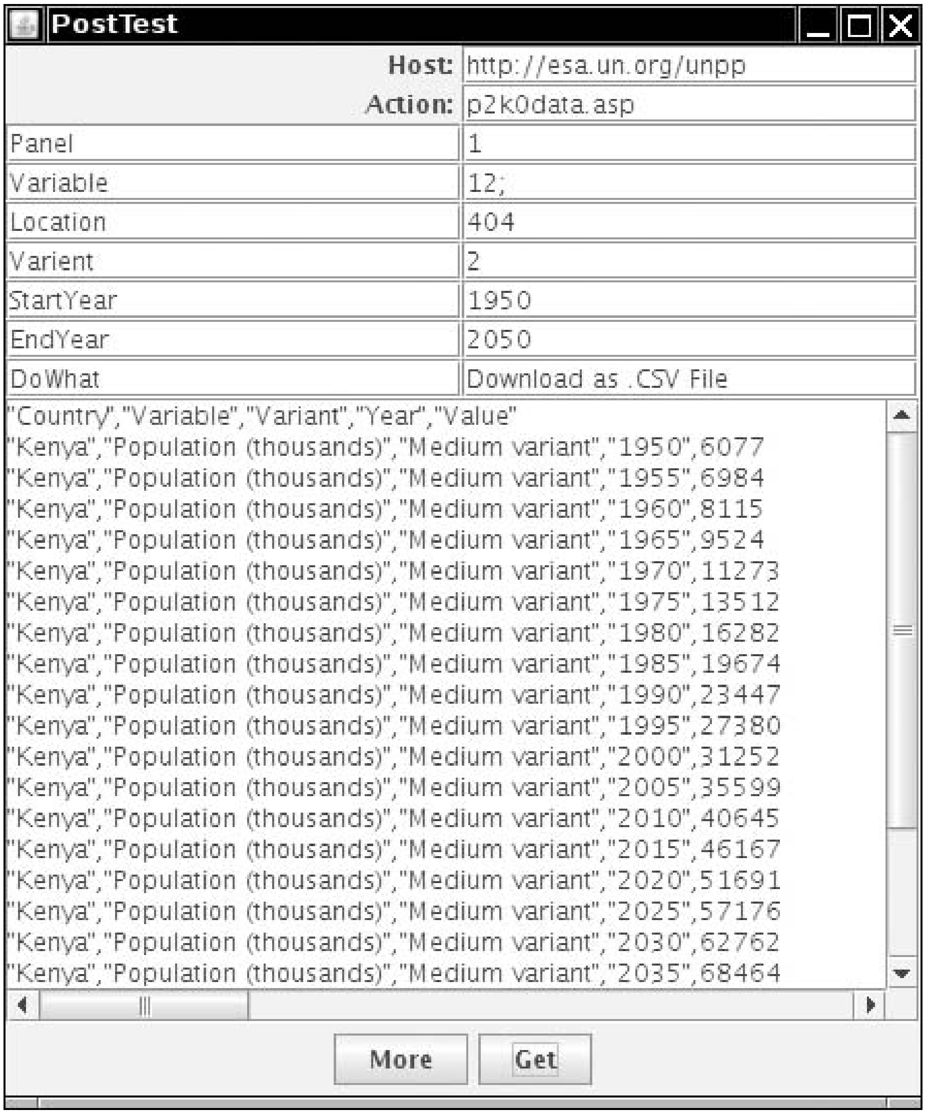
The script sends back the following reply:

```
"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:

```
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.net.*;
import java.util.*;
import javax.swing.*;

/**
 * This program demonstrates how to use the URLConnection class for a POST request.
 * @version 1.20 2007-06-25
 * @author Cay Horstmann
 */
public class PostTest
{
   public static void main(String[] args)
   {
      EventQueue.invokeLater(new Runnable()
         {
            public void run()
            {
               JFrame frame = new PostTestFrame();
               frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
               frame.setVisible(true);
            }
         });
   }
}
```

```

class PostTestFrame extends JFrame
{
   /**
    * Makes a POST request and returns the server response.
    * @param urlString the URL to post to
    * @param nameValuePairs a map of name/value pairs to supply in the request.
    * @return the server reply (either from the input stream or the error stream)
    */
   public static String doPost(String urlString, Map<String, String> nameValuePairs)
         throws IOException
   {
      URL url = new URL(urlString);
      URLConnection connection = url.openConnection();
      connection.setDoOutput(true);

      PrintWriter out = new PrintWriter(connection.getOutputStream());
      boolean first = true;
      for (Map.Entry<String, String> pair : nameValuePairs.entrySet())
      {
          if (first) first = false;
          else out.print('&');
          String name = pair.getKey();
          String value = pair.getValue();
          out.print(name);
          out.print('=');
          out.print(URLEncoder.encode(value, "UTF-8"));
      }

      out.close();
      Scanner in;
      StringBuilder response = new StringBuilder();
      try
      {
         in = new Scanner(connection.getInputStream());
      }
      catch (IOException e)
      {
         if (!(connection instanceof HttpURLConnection)) throw e;
         InputStream err = ((HttpURLConnection) connection).getErrorStream();
         if (err == null) throw e;
         in = new Scanner(err);
      }

      while (in.hasNextLine())
      {
         response.append(in.nextLine());
         response.append("\n");
      }

      in.close();
      return response.toString();
   }

   public PostTestFrame()
   {
      setTitle("PostTest");

      northPanel = new JPanel();
```

```
      add(northPanel, BorderLayout.NORTH);
      northPanel.setLayout(new GridLayout(0, 2));
      northPanel.add(new JLabel("Host: ", SwingConstants.TRAILING));
      final JTextField hostField = new JTextField();
      northPanel.add(hostField);
      northPanel.add(new JLabel("Action: ", SwingConstants.TRAILING));
      final JTextField actionField = new JTextField();
      northPanel.add(actionField);
      for (int i = 1; i <= 8; i++)
         northPanel.add(new JTextField());

      final JTextArea result = new JTextArea(20, 40);
      add(new JScrollPane(result));

      JPanel southPanel = new JPanel();
      add(southPanel, BorderLayout.SOUTH);
      JButton addButton = new JButton("More");
      southPanel.add(addButton);
      addButton.addActionListener(new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               northPanel.add(new JTextField());
               northPanel.add(new JTextField());
               pack();
            }
         });

      JButton getButton = new JButton("Get");
      southPanel.add(getButton);
      getButton.addActionListener(new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               result.setText("");
               final Map<String, String> post = new HashMap<String, String>();
               for (int i = 4; i < northPanel.getComponentCount(); i += 2)
               {
                  String name = ((JTextField) northPanel.getComponent(i)).getText();
                  if (name.length() > 0)
                  {
                     String value = ((JTextField) northPanel.getComponent(i + 1)).getText();
                     post.put(name, value);
                  }
               }
               new SwingWorker<Void, Void>()
                  {
                     protected Void doInBackground() throws Exception
                     {
                        try
                        {
                           String urlString = hostField.getText() + "/" + actionField.getText();
                           result.setText(doPost(urlString, post));
                        }
                        catch (IOException e)
                        {
                           result.setText("" + e);
                        }
                        return null;
```

```
                     }
                  }.execute();
            }
         });

      pack();
   }

   private JPanel northPanel;
}
```

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 `0x`*XY* 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

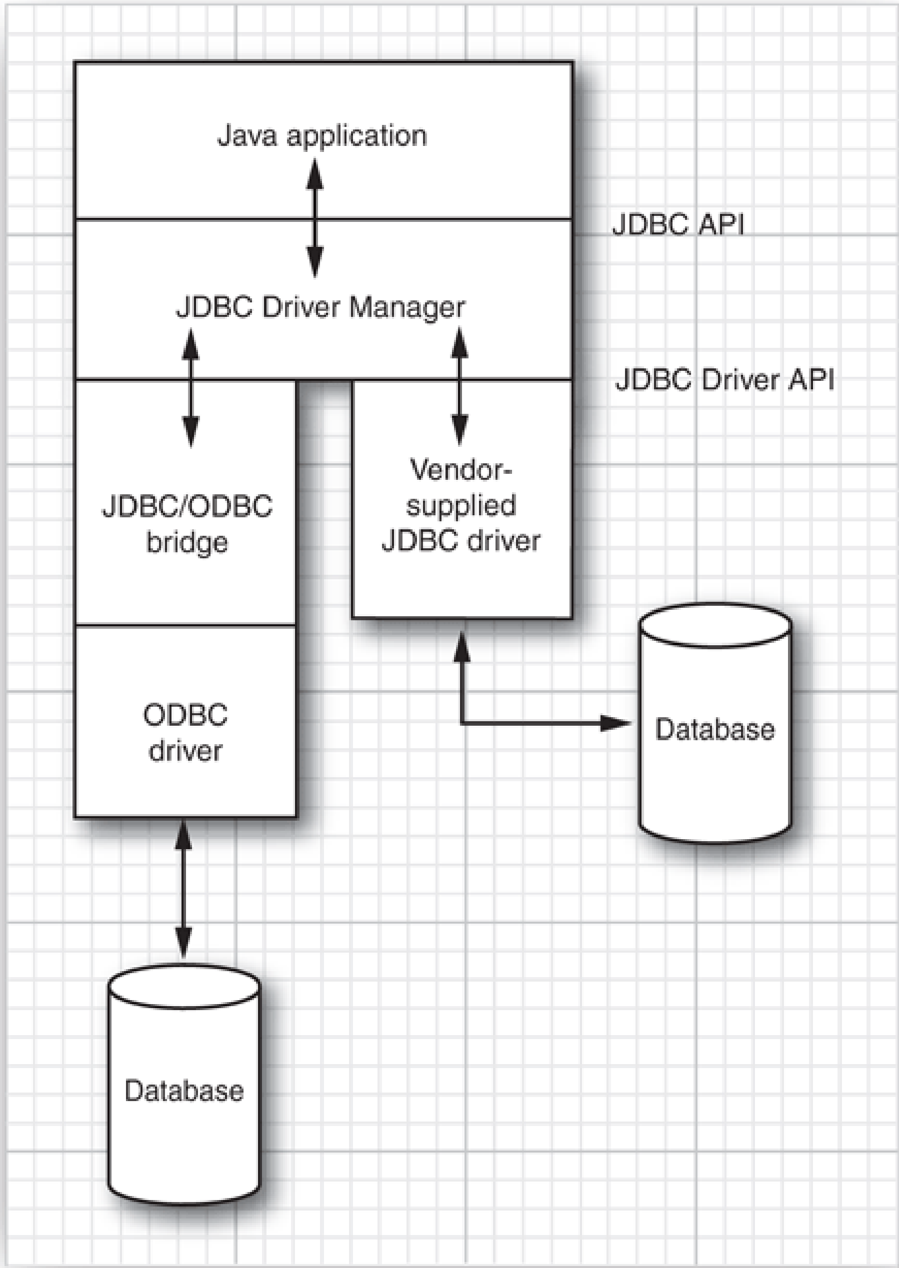
From the start, the developers of the Java technology at Sun were aware of the potential that Java showed for working with databases. In 1995, they began working on extending the standard Java library to deal with SQL access to databases. What they first hoped to do was to extend Java so that it could 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

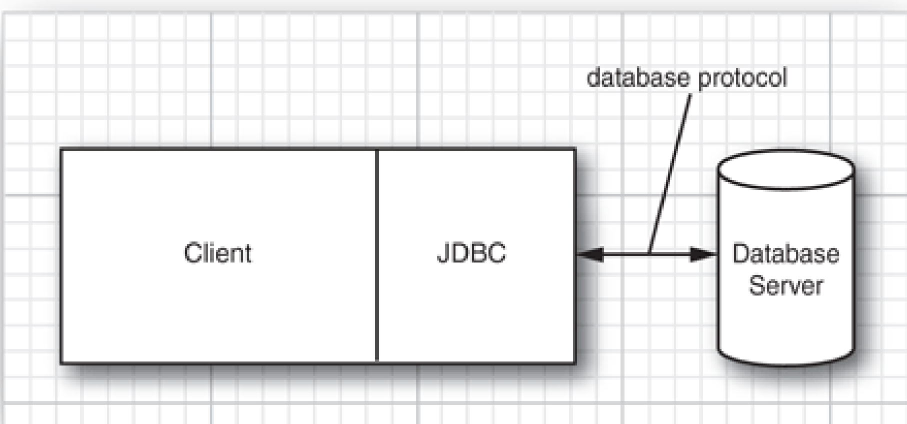
If you are curious as to why Sun just didn't adopt the ODBC model, their response, as given at the JavaOne conference in May 1996, was this:

- 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

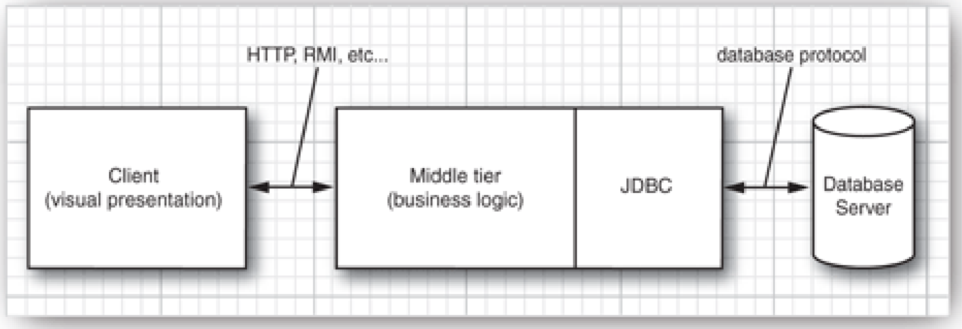


However, the world is moving away from client/server and toward a three-tier model or even more advanced *n*-tier models. In the three-tier model, the client does not make database calls. Instead, it calls on a middleware layer on the server that in turn makes the database queries. The three-tier model 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

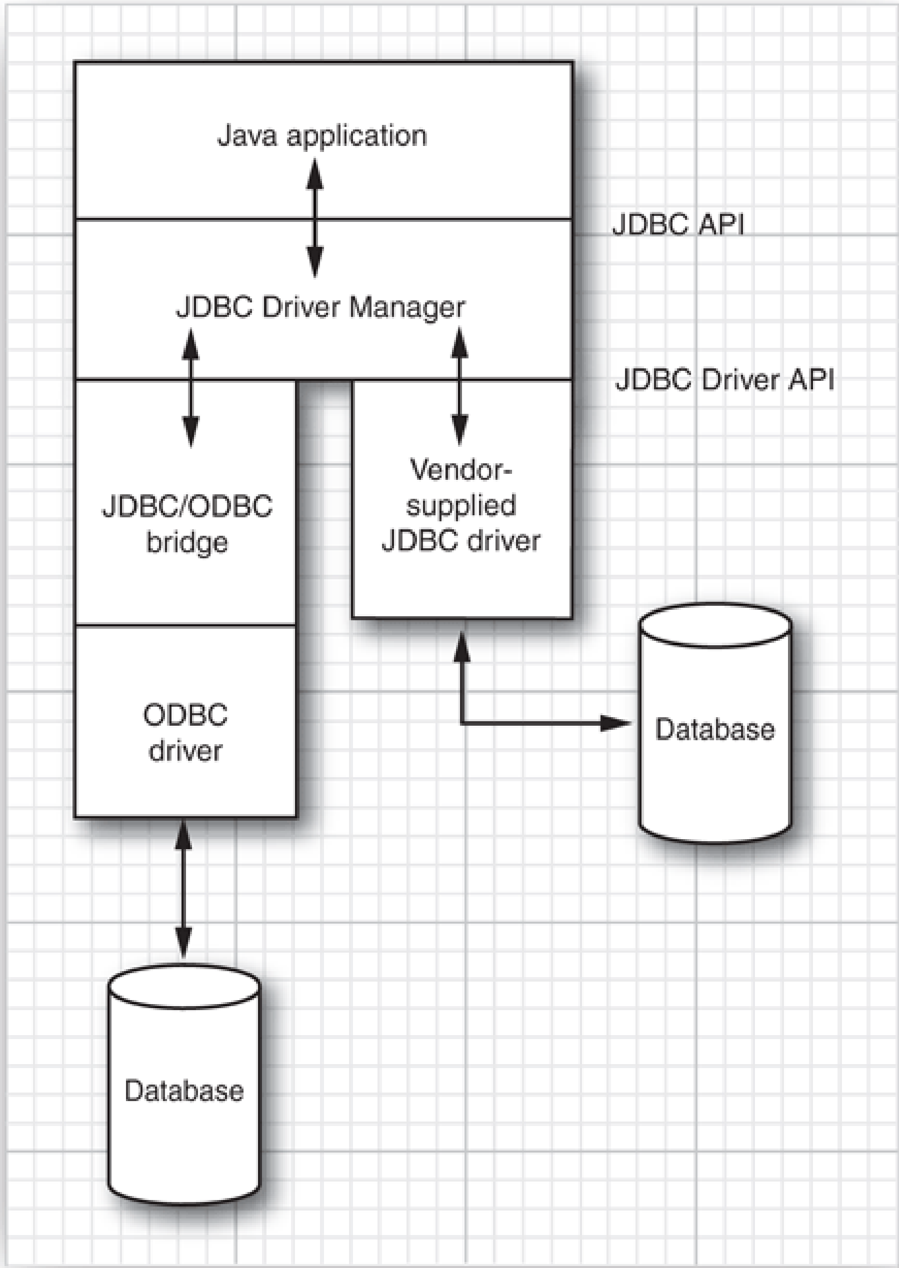
From the start, the developers of the Java technology at Sun were aware of the potential that Java showed for working with databases. In 1995, they began working on extending the standard Java library to deal with SQL access to databases. What they first hoped to do was to extend Java so that it could 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

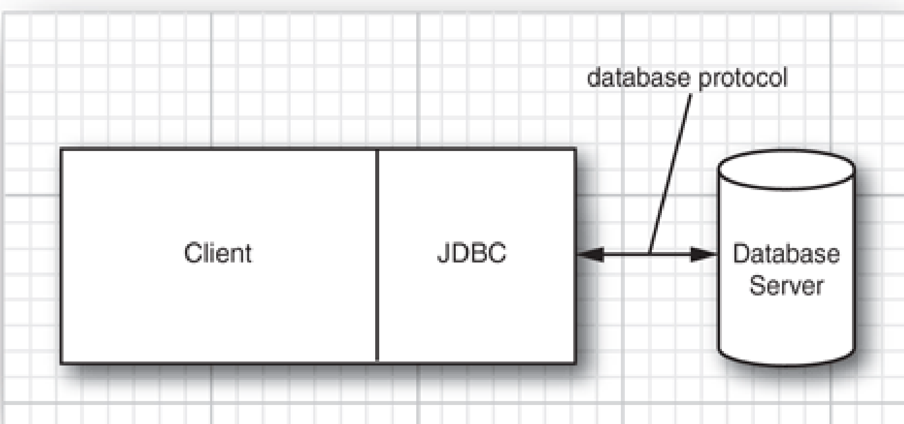
If you are curious as to why Sun just didn't adopt the ODBC model, their response, as given at the JavaOne conference in May 1996, was this:

- 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

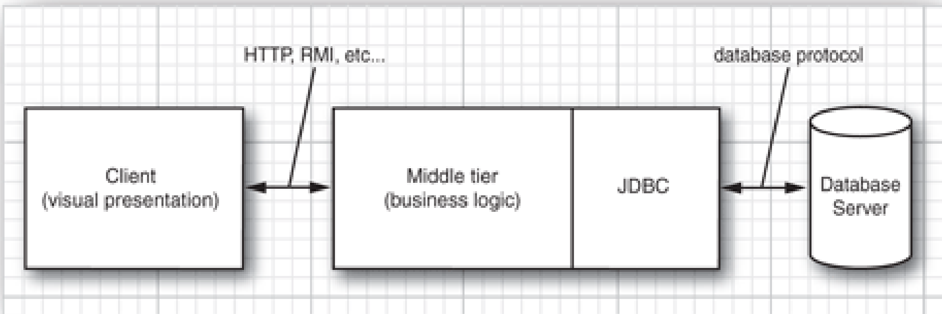


However, the world is moving away from client/server and toward a three-tier model or even more advanced *n*-tier models. In the three-tier model, the client does not make database calls. Instead, it calls on a middleware layer on the server that in turn makes the database queries. The three-tier model 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
...	...	...

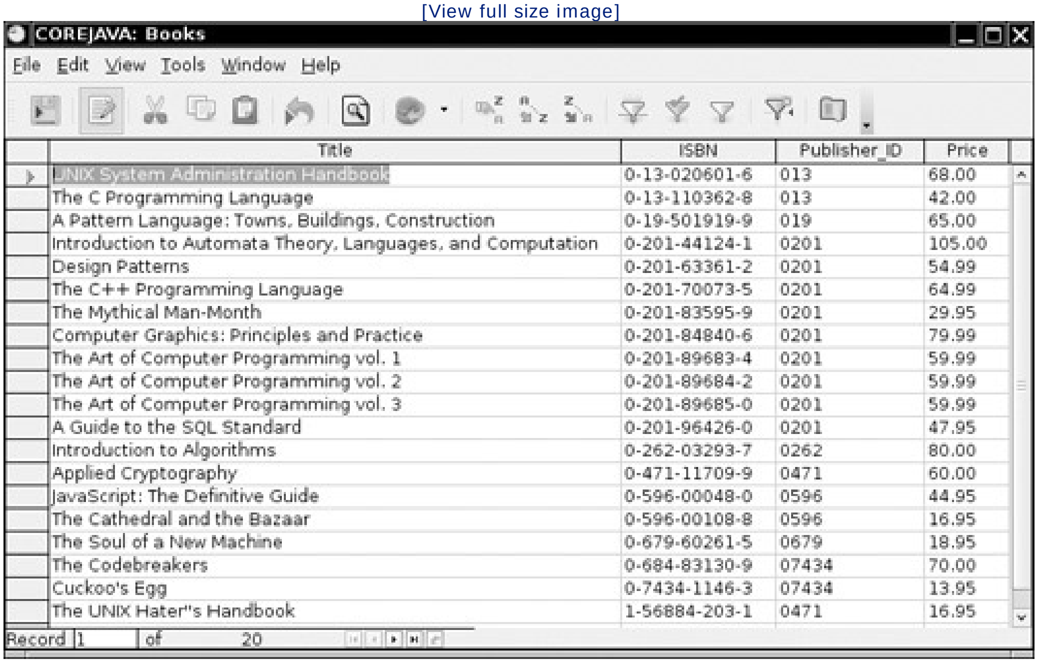
Table 4-4. The **Publishers** Table

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]


COREJAVA: Books

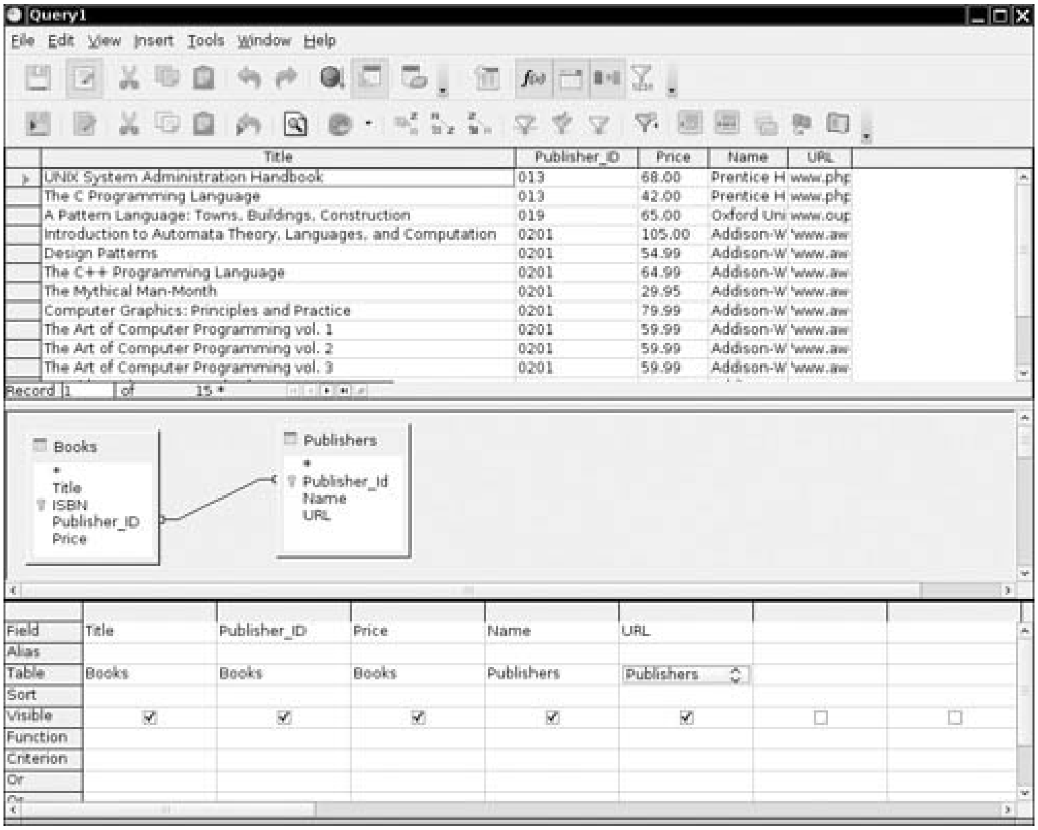
File Edit View Tools Window Help

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	0-201-83595-9	0201	29.95
Computer Graphics: Principles and Practice	0-201-84840-6	0201	79.99
The Art of Computer Programming vol. 1	0-201-89683-4	0201	59.99
The Art of Computer Programming vol. 2	0-201-89684-2	0201	59.99
The Art of Computer Programming vol. 3	0-201-89685-0	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

Figure 4-5. Two tables joined together

[View full size image]



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
```

UNREGISTERED VERSION OF CHM TO PDF CONVERTER PRO BY THETA-SOFTWARE

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:

```
import java.sql.*;
import java.io.*;
import java.util.*;

/**
 * This program tests that the database and the JDBC driver are correctly configured.
 * @version 1.01 2004-09-24
 * @author Cay Horstmann
 */
class TestDB
{
   public static void main(String args[])
   {
      try
      {
         runTest();
      }
      catch (SQLException ex)
      {
         for (Throwable t : ex)
            t.printStackTrace();
      }
      catch (IOException ex)
      {
         ex.printStackTrace();
      }
   }

```

```java
   /**
    * Runs a test by creating a table, adding a value, showing the table contents, and
    * removing the table.
    */
   public static void runTest() throws SQLException, IOException
   {
      Connection conn = getConnection();
      try
      {
```



```java
         Statement stat = conn.createStatement();

         stat.executeUpdate("CREATE TABLE Greetings (Message CHAR(20))");
         stat.executeUpdate("INSERT INTO Greetings VALUES ('Hello, World!')");

         ResultSet result = stat.executeQuery("SELECT * FROM Greetings");
         if (result.next())
```



```java
            System.out.println(result.getString(1));
         result.close();
         stat.executeUpdate("DROP TABLE Greetings");
      }
      finally
      {
         conn.close();
      }
   }

   /**
    * Gets a connection from the properties specified in the file database.properties
    * @return the database connection
    */
   public static Connection getConnection() throws SQLException, IOException
   {
      Properties props = new Properties();
      FileInputStream in = new FileInputStream("database.properties");
      props.load(in);
      in.close();

      String drivers = props.getProperty("jdbc.drivers");
      if (drivers != null) System.setProperty("jdbc.drivers", drivers);
      String url = props.getProperty("jdbc.url");
      String username = props.getProperty("jdbc.username");
      String password = props.getProperty("jdbc.password");

      return DriverManager.getConnection(url, username, password);
   }
}
```

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.

API

`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.

`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()` **6**

  returns `true` if this statement is closed.

`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* `get`*Xxx*`(int columnNumber)`

- *Xxx* `get`*Xxx*`(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:

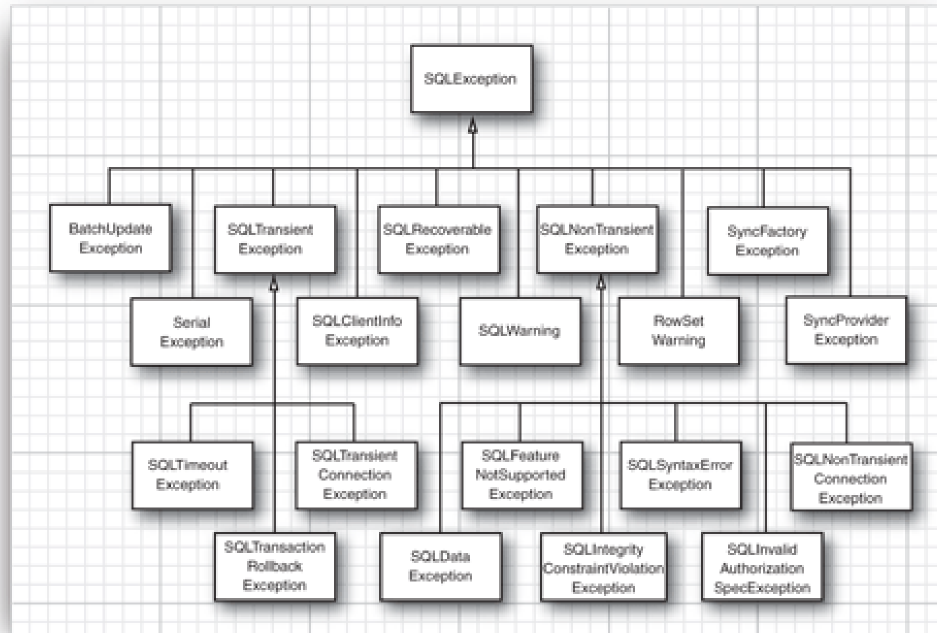
```
for (Throwable t : sqlException)
{
   do something with t
}
```

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() **6**

  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

- QLWarning getWarnings()

- SQLWarning getWarnings()

  returns the first of the pending warnings, or null if no warnings are pending.

API

`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 provde 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:
```
import java.io.*;
import java.util.*;
import java.sql.*;

/**
 * Executes all SQL statements in a file. Call this program as <br>
 * java -classpath driverPath:. ExecSQL commandFile
 * @version 1.30 2004-08-05
 * @author Cay Horstmann
 */
class ExecSQL
{
   public static void main(String args[])
   {
      try
      {
         Scanner in;
         if (args.length == 0) in = new Scanner(System.in);
         else in = new Scanner(new File(args[0]));

         Connection conn = getConnection();
         try
         {
            Statement stat = conn.createStatement();

            while (true)
            {
               if (args.length == 0) System.out.println("Enter command or EXIT to exit:");

               if (!in.hasNextLine()) return;

               String line = in.nextLine();
               if (line.equalsIgnoreCase("EXIT")) return;
               if (line.trim().endsWith(";")) // remove trailing semicolon
               {
                  line = line.trim();
                  line = line.substring(0, line.length() - 1);
               }
               try
               {
                  boolean hasResultSet = stat.execute(line);
                  if (hasResultSet) showResultSet(stat);
               }
               catch (SQLException ex)
               {
                  for (Throwable e : ex)
                     e.printStackTrace();
               }
            }
         }
         finally
         {
            conn.close();
         }
```

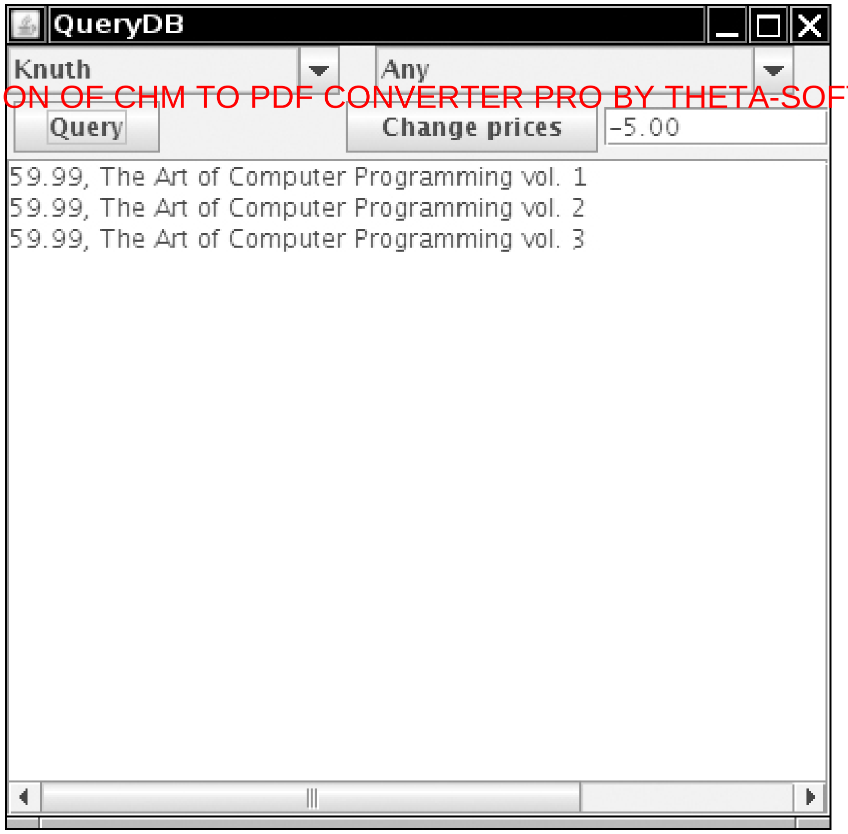
```
      }
      catch (SQLException e)
      {
         for (Throwable t : e)
            t.printStackTrace();
      }
      catch (IOException e)
      {
         e.printStackTrace();
      }
   }

   /**
    * Gets a connection from the properties specified in the file database.properties
    * @return the database connection
    */
   public static Connection getConnection() throws SQLException, IOException
   {
      Properties props = new Properties();
      FileInputStream in = new FileInputStream("database.properties");
      props.load(in);
      in.close();

      String drivers = props.getProperty("jdbc.drivers");
      if (drivers != null) System.setProperty("jdbc.drivers", drivers);

      String url = props.getProperty("jdbc.url");
      String username = props.getProperty("jdbc.username");
      String password = props.getProperty("jdbc.password");

      return DriverManager.getConnection(url, username, password);
   }

   /**
    * Prints a result set.
    * @param stat the statement whose result set should be printed
    */
   public static void showResultSet(Statement stat) throws SQLException
   {
      ResultSet result = stat.getResultSet();
      ResultSetMetaData metaData = result.getMetaData();
      int columnCount = metaData.getColumnCount();

      for (int i = 1; i <= columnCount; i++)
      {
         if (i > 1) System.out.print(", ");
         System.out.print(metaData.getColumnLabel(i));
      }
      System.out.println();

      while (result.next())
      {
         for (int i = 1; i <= columnCount; i++)
         {
            if (i > 1) System.out.print(", ");
            System.out.print(result.getString(i));
         }
         System.out.println();
      }
```

```
        result.close();
    }
}
```

Query Execution

In this section, we write a program that executes queries against the `COREJAVA` database. For this program to wor 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 kee 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 `set` *Xxx* metho 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 publish names in the database.

- The listener for the Query button checks which query type is requested. If this is the first time this query typ 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- 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 rov 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 wi 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:
```
import java.sql.*;
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.util.*;
import javax.swing.*;

/**
 * This program demonstrates several complex database queries.
 * @version 1.23 2007-06-28
 * @author Cay Horstmann
 */
public class QueryDB
{
   public static void main(String[] args)
   {
      EventQueue.invokeLater(new Runnable()
         {
            public void run()
            {
               JFrame frame = new QueryDBFrame();
               frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
               frame.setVisible(true);
            }
         });
   }
```

```
}

/**
 * This frame displays combo boxes for query parameters, a text area for command results,
 * and buttons to launch a query and an update.
 */
class QueryDBFrame extends JFrame
{
```

```
   public QueryDBFrame()
   {
      setTitle("QueryDB");
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
      setLayout(new GridBagLayout());

      authors = new JComboBox();
```

```
      authors.setEditable(false);
      authors.addItem("Any");

      publishers = new JComboBox();
      publishers.setEditable(false);
      publishers.addItem("Any");

      result = new JTextArea(4, 50);
      result.setEditable(false);

      priceChange = new JTextField(8);
      priceChange.setText("-5.00");

      try
      {
         conn = getConnection();
         Statement stat = conn.createStatement();
         String query = "SELECT Name FROM Authors";
         ResultSet rs = stat.executeQuery(query);
         while (rs.next())
            authors.addItem(rs.getString(1));
         rs.close();

         query = "SELECT Name FROM Publishers";
         rs = stat.executeQuery(query);
         while (rs.next())
            publishers.addItem(rs.getString(1));
         rs.close();
         stat.close();
      }
      catch (SQLException e)
      {
         for (Throwable t : e)
            result.append(t.getMessage());
      }
      catch (IOException e)
      {
         result.setText("" + e);
      }

      // we use the GBC convenience class of Core Java Volume I, Chapter 9
      add(authors, new GBC(0, 0, 2, 1));

      add(publishers, new GBC(2, 0, 2, 1));
```

```

      JButton queryButton = new JButton("Query");
      queryButton.addActionListener(new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               executeQuery();
            }
         });
      add(queryButton, new GBC(0, 1, 1, 1).setInsets(3));

      JButton changeButton = new JButton("Change prices");
      changeButton.addActionListener(new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               changePrices();
            }
         });
      add(changeButton, new GBC(2, 1, 1, 1).setInsets(3));

      add(priceChange, new GBC(3, 1, 1, 1).setFill(GBC.HORIZONTAL));

      add(new JScrollPane(result), new GBC(0, 2, 4, 1).setFill(GBC.BOTH).setWeight(100, 100));

      addWindowListener(new WindowAdapter()
         {
            public void windowClosing(WindowEvent event)
            {
               try
               {
                  if (conn != null) conn.close();
               }
               catch (SQLException e)
               {
                  for (Throwable t : e)
                     t.printStackTrace();
               }
            }
         });
   }

   /**
    * Executes the selected query.
    */
   private void executeQuery()
   {
      ResultSet rs = null;
      try
      {
         String author = (String) authors.getSelectedItem();
         String publisher = (String) publishers.getSelectedItem();
         if (!author.equals("Any") && !publisher.equals("Any"))
         {
            if (authorPublisherQueryStmt == null) authorPublisherQueryStmt = conn
                  .prepareStatement(authorPublisherQuery);
            authorPublisherQueryStmt.setString(1, author);
            authorPublisherQueryStmt.setString(2, publisher);
            rs = authorPublisherQueryStmt.executeQuery();
```

```
         }
         else if (!author.equals("Any") && publisher.equals("Any"))
         {
            if (authorQueryStmt == null) authorQueryStmt = conn.prepareStatement(authorQuery);
            authorQueryStmt.setString(1, author);
            rs = authorQueryStmt.executeQuery();
         }
         else if (author.equals("Any") && !publisher.equals("Any"))
         {
```

```
            if (publisherQueryStmt == null) publisherQueryStmt = conn
                  .prepareStatement(publisherQuery);
            publisherQueryStmt.setString(1, publisher);
            rs = publisherQueryStmt.executeQuery();
         }
         else
         {
```

```
            if (allQueryStmt == null) allQueryStmt = conn.prepareStatement(allQuery);
            rs = allQueryStmt.executeQuery();
         }

         result.setText("");
         while (rs.next())
         {
            result.append(rs.getString(1));
            result.append(", ");
            result.append(rs.getString(2));
            result.append("\n");
         }
         rs.close();
      }
      catch (SQLException e)
      {
         for (Throwable t : e)
            result.append(t.getMessage());
      }
   }

   /**
    * Executes an update statement to change prices.
    */
   public void changePrices()
   {
      String publisher = (String) publishers.getSelectedItem();
      if (publisher.equals("Any"))
      {
         result.setText("I am sorry, but I cannot do that.");
         return;
      }
      try
      {
         if (priceUpdateStmt == null) priceUpdateStmt = conn.prepareStatement(priceUpdate);
         priceUpdateStmt.setString(1, priceChange.getText());
         priceUpdateStmt.setString(2, publisher);
         int r = priceUpdateStmt.executeUpdate();
         result.setText(r + " records updated.");
      }
      catch (SQLException e)
      {
         for (Throwable t : e)
```

```
            result.append(t.getMessage());
      }
   }

   /**
    * Gets a connection from the properties specified in the file database.properties
    * @return the database connection
    */
   public static Connection getConnection() throws SQLException, IOException
   {
      Properties props = new Properties();
      FileInputStream in = new FileInputStream("database.properties");
      props.load(in);
      in.close();

      String drivers = props.getProperty("jdbc.drivers");
      if (drivers != null) System.setProperty("jdbc.drivers", drivers);
      String url = props.getProperty("jdbc.url");
      String username = props.getProperty("jdbc.username");
      String password = props.getProperty("jdbc.password");

      return DriverManager.getConnection(url, username, password);
   }

   public static final int DEFAULT_WIDTH = 400;
   public static final int DEFAULT_HEIGHT = 400;

   private JComboBox authors;
   private JComboBox publishers;
   private JTextField priceChange;
   private JTextArea result;
   private Connection conn;
   private PreparedStatement authorQueryStmt;
   private PreparedStatement authorPublisherQueryStmt;
   private PreparedStatement publisherQueryStmt;
   private PreparedStatement allQueryStmt;
   private PreparedStatement priceUpdateStmt;

   private static final String authorPublisherQuery = "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 = ?";

   private static final String authorQuery = "SELECT Books.Price, Books.Title FROM Books,
         BooksAuthors, Authors" + " WHERE Authors.Author_Id =
         BooksAuthors.Author_Id AND BooksAuthors.ISBN = Books.ISBN"
         + " AND Authors.Name = ?";

   private static final String publisherQuery = "SELECT Books.Price, Books.Title FROM Books,
         Publishers" + " WHERE Books.Publisher_Id = Publishers.Publisher_Id
         AND Publishers.Name = ?";

   private static final String allQuery = "SELECT Books.Price, Books.Title FROM Books";

   private static final String priceUpdate = "UPDATE Books " + "SET Price = Price + ? "
         + " WHERE Books.Publisher_Id = (SELECT Publisher_Id FROM Publishers WHERE Name = ?)";
}
```

`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 set` *Xxx* `(int n,` *Xxx* `x)`

  (*Xxx* is a type such as `int` , `double` , `String` , `Date` , etc.)

  sets the value of the `n` th 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 othe 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` . F 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.set(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.set(1, isbn);
stat.set(2, coverBlob);
stat.executeUpdate();
```

API `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 860: 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 wi 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();
}
```

API `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.

Scrollable 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.

Scrollable 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
TYPE_FORWARD_ONLY	The result set is not scrollable (default).
TYPE_SCROLL_INSENSITIVE	The result set is scrollable but not sensitive to database changes.
TYPE_SCROLL_SENSITIVE	The result set is scrollable and sensitive to database changes.

Table 4-7. **ResultSet** Concurrency Values

Value	Explanation
`CONCUR_READ_ONLY`	The result set cannot be used to update the database (default).
`CONCUR_UPDATABLE`	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 `update`*Xxx* methods for all data types that correspond to SQL types, such as `updateDouble`, `updateString`, and so on. As with the `get`*Xxx* 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 `update`*Xxx* 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 `update`*Xxx* 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 `update`*Xxx* 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:	`command`	The command to prepare
	`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

`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 `update`*`Xxx`* 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 update`*Xxx*`(int column, `*Xxx*` data)` **1.2**

- `void update`*Xxx*`(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.

API

`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

Scrollable 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.

API `javax.sql.RowSet` 1.4

- `String getURL()`
- `void setURL(String url)`

  gets or sets the database URL.

- `String getUsername()`
- `void setUsername(String username)`

  gets or sets the user name for connecting to the database.

- `String getPassword()`
- `void setPassword(String password)`

  gets or sets the password for connecting to the database.

- `String getCommand()`
- `void setCommand(String command)`

  gets or sets the command that is executed to populate this row set.

- `void execute()`

  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.

`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.nullPlusNonNullIsNull()
```

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:

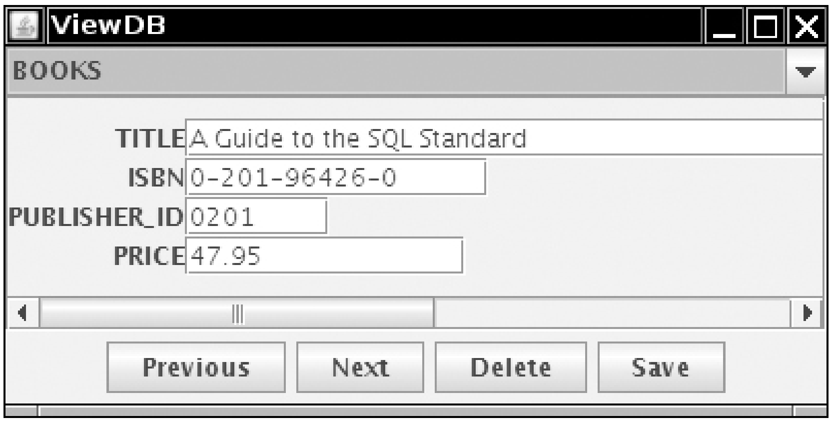
```
ResultSet mrs = stat.executeQuery("SELECT * FROM " + tableName);
ResultSetMetaData meta = mrs.getMetaData();
```

```
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:

```
import com.sun.rowset.*;
```

```
import java.sql.*;
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.util.*;
import javax.swing.*;
import javax.sql.*;
import javax.sql.rowset.*;

/**
 * This program uses metadata to display arbitrary tables in a database.
 * @version 1.31 2007-06-28
 * @author Cay Horstmann
 */
public class ViewDB
{
   public static void main(String[] args)
   {
      EventQueue.invokeLater(new Runnable()
         {
            public void run()
            {
               JFrame frame = new ViewDBFrame();
               frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
               frame.setVisible(true);
            }
         });
   }
}

/**
 * The frame that holds the data panel and the navigation buttons.
 */
class ViewDBFrame extends JFrame
{
   public ViewDBFrame()
   {
      setTitle("ViewDB");
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

      tableNames = new JComboBox();
      tableNames.addActionListener(new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               showTable((String) tableNames.getSelectedItem());
            }
         });
      add(tableNames, BorderLayout.NORTH);

      try
      {
         readDatabaseProperties();
         Connection conn = getConnection();
         try
         {
            DatabaseMetaData meta = conn.getMetaData();
            ResultSet mrs = meta.getTables(null, null, null, new String[] { "TABLE" });
            while (mrs.next())
```

```
               tableNames.addItem(mrs.getString(3));
         }
         finally
         {
            conn.close();
         }
      }
      catch (SQLException e)
      {
```

```
         JOptionPane.showMessageDialog(this, e);
      }
      catch (IOException e)
      {
         JOptionPane.showMessageDialog(this, e);
      }
```

```
      JPanel buttonPanel = new JPanel();
      add(buttonPanel, BorderLayout.SOUTH);

      previousButton = new JButton("Previous");
      previousButton.addActionListener(new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               showPreviousRow();
            }
         });
      buttonPanel.add(previousButton);

      nextButton = new JButton("Next");
      nextButton.addActionListener(new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               showNextRow();
            }
         });
      buttonPanel.add(nextButton);

      deleteButton = new JButton("Delete");
      deleteButton.addActionListener(new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               deleteRow();
            }
         });
      buttonPanel.add(deleteButton);

      saveButton = new JButton("Save");
      saveButton.addActionListener(new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               saveChanges();
            }
         });
      buttonPanel.add(saveButton);
   }
```

```
   /**
    * Prepares the text fields for showing a new table, and shows the first row.
    * @param tableName the name of the table to display
    */
   public void showTable(String tableName)
   {
      try
      {
         // open connection
         Connection conn = getConnection();
         try
         {
            // get result set
            Statement stat = conn.createStatement();
            ResultSet result = stat.executeQuery("SELECT * FROM " + tableName);
            // copy into cached row set
            crs = new CachedRowSetImpl();
            crs.setTableName(tableName);
            crs.populate(result);
         }
         finally
         {
            conn.close();
         }

         if (scrollPane != null) remove(scrollPane);
         dataPanel = new DataPanel(crs);
         scrollPane = new JScrollPane(dataPanel);
         add(scrollPane, BorderLayout.CENTER);
         validate();
         showNextRow();
      }
      catch (SQLException e)
      {
         JOptionPane.showMessageDialog(this, e);
      }
   }

   /**
    * Moves to the previous table row.
    */
   public void showPreviousRow()
   {
      try
      {
         if (crs == null || crs.isFirst()) return;
         crs.previous();
         dataPanel.showRow(crs);
      }
      catch (SQLException e)
      {
         for (Throwable t : e)
            t.printStackTrace();
      }
   }

   /**
    * Moves to the next table row.
```

```
    */
   public void showNextRow()
   {
      try
      {
         if (crs == null || crs.isLast()) return;
         crs.next();
         dataPanel.showRow(crs);
      }
```

```
      catch (SQLException e)
      {
         JOptionPane.showMessageDialog(this, e);
      }
   }

   /**
```

```
    * Deletes current table row.
    */
   public void deleteRow()
   {
      try
      {
         Connection conn = getConnection();
         try
         {
            crs.deleteRow();
            crs.acceptChanges(conn);
            if (!crs.isLast()) crs.next();
            else if (!crs.isFirst()) crs.previous();
            else crs = null;
            dataPanel.showRow(crs);
         }
         finally
         {
            conn.close();
         }
      }
      catch (SQLException e)
      {
         JOptionPane.showMessageDialog(this, e);
      }
   }

   /**
    * Saves all changes.
    */
   public void saveChanges()
   {
      try
      {
         Connection conn = getConnection();
         try
         {
            dataPanel.setRow(crs);
            crs.acceptChanges(conn);
         }
         finally
         {
            conn.close();
```

```
         }
      }
      catch (SQLException e)
      {
         JOptionPane.showMessageDialog(this, e);
      }
   }

   private void readDatabaseProperties() throws IOException
   {
      props = new Properties();
      FileInputStream in = new FileInputStream("database.properties");
      props.load(in);
      in.close();
      String drivers = props.getProperty("jdbc.drivers");
      if (drivers != null) System.setProperty("jdbc.drivers", drivers);
   }

   /**
    * Gets a connection from the properties specified in the file database.properties
    * @return the database connection
    */
   private Connection getConnection() throws SQLException
   {
      String url = props.getProperty("jdbc.url");
      String username = props.getProperty("jdbc.username");
      String password = props.getProperty("jdbc.password");

      return DriverManager.getConnection(url, username, password);
   }

   public static final int DEFAULT_WIDTH = 400;
   public static final int DEFAULT_HEIGHT = 200;

   private JButton previousButton;
   private JButton nextButton;
   private JButton deleteButton;
   private JButton saveButton;
   private DataPanel dataPanel;
   private Component scrollPane;
   private JComboBox tableNames;
   private Properties props;
   private CachedRowSet crs;
}

/**
 * This panel displays the contents of a result set.
 */
class DataPanel extends JPanel
{
   /**
    * Constructs the data panel.
    * @param rs the result set whose contents this panel displays
    */
   public DataPanel(RowSet rs) throws SQLException
   {
      fields = new ArrayList<JTextField>();
      setLayout(new GridBagLayout());
      GridBagConstraints gbc = new GridBagConstraints();
```

```
      gbc.gridwidth = 1;
      gbc.gridheight = 1;

      ResultSetMetaData rsmd = rs.getMetaData();
      for (int i = 1; i <= rsmd.getColumnCount(); i++)
      {
         gbc.gridy = i - 1;

         String columnName = rsmd.getColumnLabel(i);
         gbc.gridx = 0;
         gbc.anchor = GridBagConstraints.EAST;
         add(new JLabel(columnName), gbc);

         int columnWidth = rsmd.getColumnDisplaySize(i);
         JTextField tb = new JTextField(columnWidth);
         if (!rsmd.getColumnClassName(i).equals("java.lang.String"))
            tb.setEditable(false);

         fields.add(tb);

         gbc.gridx = 1;
         gbc.anchor = GridBagConstraints.WEST;
         add(tb, gbc);
      }
   }

   /**
    * Shows a database row by populating all text fields with the column values.
    */
   public void showRow(ResultSet rs) throws SQLException
   {
      for (int i = 1; i <= fields.size(); i++)
      {
         String field = rs.getString(i);
         JTextField tb = (JTextField) fields.get(i - 1);
         tb.setText(field);
      }
   }

   /**
    * Updates changed data into the current row of the row set
    */
   public void setRow(RowSet rs) throws SQLException
   {
      for (int i = 1; i <= fields.size(); i++)
      {
         String field = rs.getString(i);
         JTextField tb = (JTextField) fields.get(i - 1);
         if (!field.equals(tb.getText()))
            rs.updateString(i, tb.getText());
      }
      rs.updateRow();
   }

   private ArrayList<JTextField> fields;
}
```

UNREGISTERED VERSION OF CHM TO PDF CONVERTER PRO BY THETA-SOFTWARE

![API] `java.sql.Connection` 1.1

- `DatabaseMetaData getMetaData()`

  returns the metadata for the connection as a `DatabaseMetaData` object.

![API] `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 `null`)
2	TABLE_SCHEM	Table schema (may be `null`)
3	TABLE_NAME	Table name
4	TABLE_TYPE	Table type
5	REMARKS	Comment on the table

- `int getJDBCMajorVersion()` **1.4**

- `int getJDBCMinorVersion()` **1.4**

  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.

- `int getMaxConnections()`

  returns the maximum number of concurrent connections allowed to this database.

- `int getMaxStatements()`

  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.getStatement();
. . .
// keep calling stat.addBatch(. . .);
. . .
stat.executeBatch();
conn.commit();
conn.setAutoCommit(autoCommit);
```


API 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.

API

`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.

API

`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.

API `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` intefaces, 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
`dc`	Domain component
`cn`	Common name
`sn`	Surname
`dn`	Distinguished name
`o`	Organization
`ou`	Organizational unit
`uid`	Unique identifier

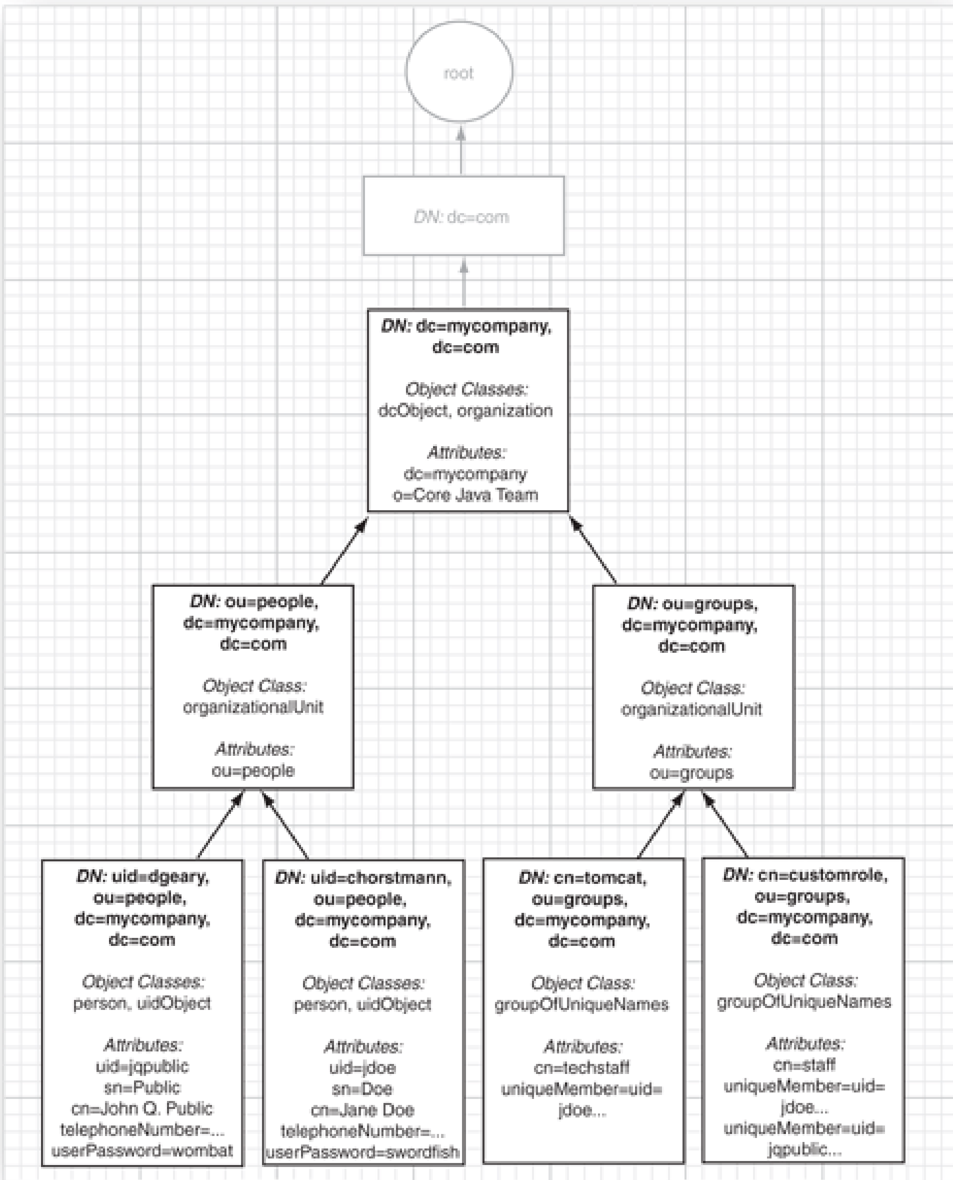
- One or more *object classes*. An object class defines the set of required and optional attributes for this element. For example, the object class `person` defines a required attribute `cn` and an optional attribute `telephoneNumber`. Of course, the object classes are different from Java classes, but 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:

```
# Define top-level entry
dn: dc=mycompany,dc=com
objectClass: dcObject
objectClass: organization
dc: mycompany
o: Core Java Team

# Define an entry to contain people
# searches for users are based on this entry
dn: ou=people,dc=mycompany,dc=com
objectClass: organizationalUnit
ou: people

# Define a user entry for John Q. Public
dn: uid=jqpublic,ou=people,dc=mycompany,dc=com
objectClass: person
objectClass: uidObject
uid: jqpublic
sn: Public
cn: John Q. Public
telephoneNumber: +1 408 555 0017
userPassword: wombat

# Define a user entry for Jane Doe
dn: uid=jdoe,ou=people,dc=mycompany,dc=com
objectClass: person
objectClass: uidObject
uid: jdoe
sn: Doe
cn: Jane Doe
telephoneNumber: +1 408 555 0029
userPassword: heffalump

# Define an entry to contain LDAP groups
# searches for roles are based on this entry
dn: ou=groups,dc=mycompany,dc=com
objectClass: organizationalUnit
ou: groups

# Define an entry for the "techstaff" group
```

```
dn: cn=techstaff,ou=groups,dc=mycompany,dc=com
objectClass: groupOfUniqueNames
cn: techstaff
uniqueMember: uid=jdoe,ou=people,dc=mycompany,dc=com

# Define an entry for the "staff" group
dn: cn=staff,ou=groups,dc=mycompany,dc=com
objectClass: groupOfUniqueNames
cn: staff
uniqueMember: uid=jqpublic,ou=people,dc=mycompany,dc=com
uniqueMember: uid=jdoe,ou=people,dc=mycompany,dc=com
```

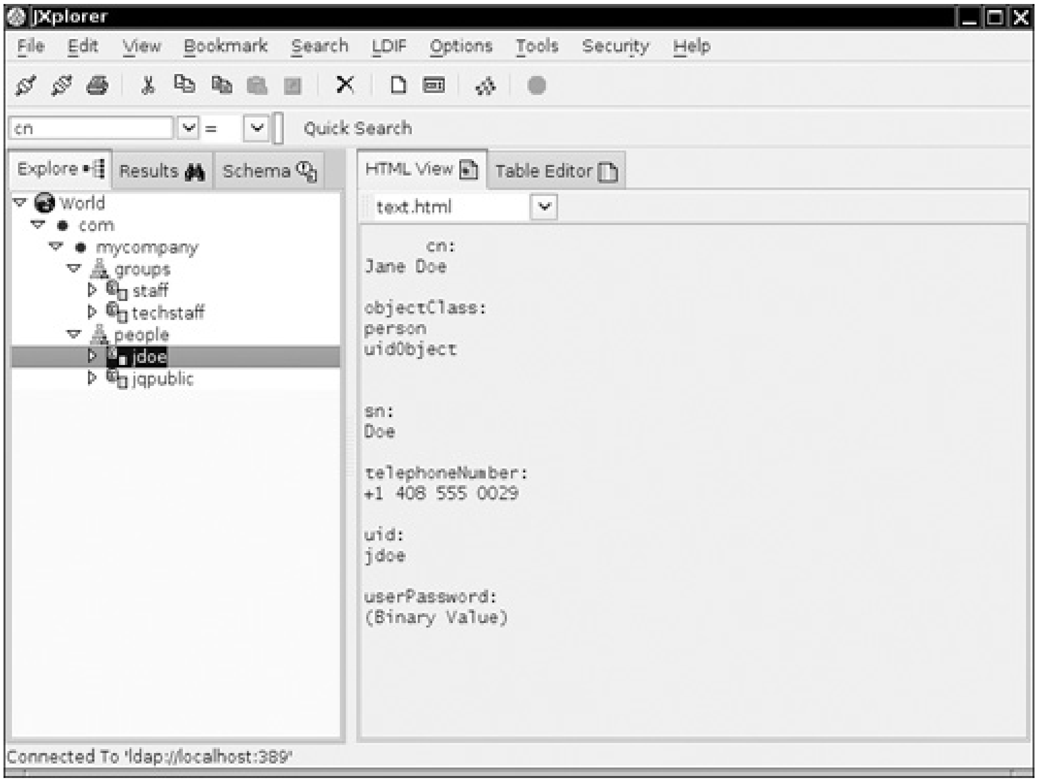

Before proceeding, it is a good idea to double-check that the directory contains the data that you need. We suggest that you download JXplorer (http://www.jxplorer.org) or Jarek Gawor's LDAP Browser/Editor (http://www-unix.mcs.anl.gov/~gawor/ldap). These convenient Java programs let you browse the contents 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

[View full size image]



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()));
```

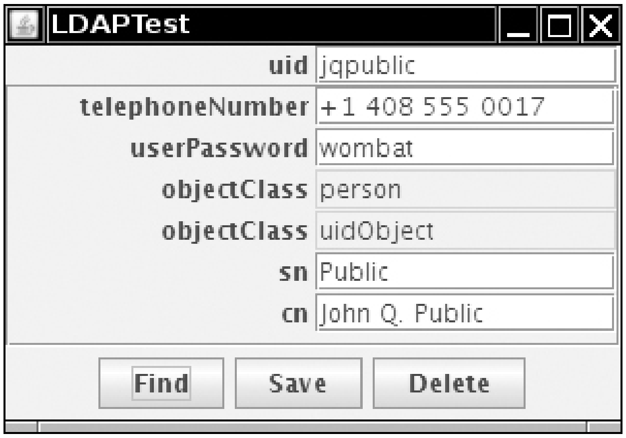
Finally, when you are done with a context, you should close it:

```
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:

```java
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.util.*;
import javax.naming.*;
```



```java
import javax.naming.directory.*;
import javax.swing.*;

/**
 * This program demonstrates access to a hierarchical database through LDAP
 * @version 1.01 2007-06-28
 * @author Cay Horstmann
```



```java
 */
public class LDAPTest
{
   public static void main(String[] args)
   {
      EventQueue.invokeLater(new Runnable()
         {
            public void run()
            {
               JFrame frame = new LDAPFrame();
               frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
               frame.setVisible(true);
            }
         });
   }
}

/**
 * The frame that holds the data panel and the navigation buttons.
 */
class LDAPFrame extends JFrame
{
   public LDAPFrame()
   {
      setTitle("LDAPTest");
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

      JPanel northPanel = new JPanel();
      northPanel.setLayout(new java.awt.GridLayout(1, 2, 3, 1));
      northPanel.add(new JLabel("uid", SwingConstants.RIGHT));
      uidField = new JTextField();
      northPanel.add(uidField);
      add(northPanel, BorderLayout.NORTH);

      JPanel buttonPanel = new JPanel();
      add(buttonPanel, BorderLayout.SOUTH);

      findButton = new JButton("Find");
      findButton.addActionListener(new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               findEntry();
            }
```

```
         });
      buttonPanel.add(findButton);

      saveButton = new JButton("Save");
      saveButton.addActionListener(new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               saveEntry();
            }
         });
      buttonPanel.add(saveButton);

      deleteButton = new JButton("Delete");
      deleteButton.addActionListener(new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               deleteEntry();
            }
         });
      buttonPanel.add(deleteButton);

      addWindowListener(new WindowAdapter()
         {
            public void windowClosing(WindowEvent event)
            {
               try
               {
                  if (context != null) context.close();
               }
               catch (NamingException e)
               {
                  e.printStackTrace();
               }
            }
         });
   }

   /**
    * Finds the entry for the uid in the text field.
    */
   public void findEntry()
   {
      try
      {
         if (scrollPane != null) remove(scrollPane);
         String dn = "uid=" + uidField.getText() + ",ou=people,dc=mycompany,dc=com";
         if (context == null) context = getContext();
         attrs = context.getAttributes(dn);
         dataPanel = new DataPanel(attrs);
         scrollPane = new JScrollPane(dataPanel);
         add(scrollPane, BorderLayout.CENTER);
         validate();
         uid = uidField.getText();
      }
      catch (NamingException e)
      {
         JOptionPane.showMessageDialog(this, e);
```

```
      }
      catch (IOException e)
      {
         JOptionPane.showMessageDialog(this, e);
      }
   }

   /**
    * Saves the changes that the user made.
```

```
    */
   public void saveEntry()
   {
      try
      {
         if (dataPanel == null) return;
         if (context == null) context = getContext();
```

```
         if (uidField.getText().equals(uid)) // update existing entry
         {
            String dn = "uid=" + uidField.getText() + ",ou=people,dc=mycompany,dc=com";
            Attributes editedAttrs = dataPanel.getEditedAttributes();
            NamingEnumeration<? extends Attribute> attrEnum = attrs.getAll();
            while (attrEnum.hasMore())
            {
               Attribute attr = attrEnum.next();
               String id = attr.getID();
               Attribute editedAttr = editedAttrs.get(id);
               if (editedAttr != null && !attr.get().equals(editedAttr.get())) context
                     .modifyAttributes(dn, DirContext.REPLACE_ATTRIBUTE,
                           new BasicAttributes(id, editedAttr.get()));
            }
         }
         else
         // create new entry
         {
            String dn = "uid=" + uidField.getText() + ",ou=people,dc=mycompany,dc=com";
            attrs = dataPanel.getEditedAttributes();
            Attribute objclass = new BasicAttribute("objectClass");
            objclass.add("uidObject");
            objclass.add("person");
            attrs.put(objclass);
            attrs.put("uid", uidField.getText());
            context.createSubcontext(dn, attrs);
         }

         findEntry();
      }
      catch (NamingException e)
      {
         JOptionPane.showMessageDialog(LDAPFrame.this, e);
         e.printStackTrace();
      }
      catch (IOException e)
      {
         JOptionPane.showMessageDialog(LDAPFrame.this, e);
         e.printStackTrace();
      }
   }

   /**
```

```
     * Deletes the entry for the uid in the text field.
     */
    public void deleteEntry()
    {
       try
       {
          String dn = "uid=" + uidField.getText() + ",ou=people,dc=mycompany,dc=com";
          if (context == null) context = getContext();
          context.destroySubcontext(dn);
          uidField.setText("");
          remove(scrollPane);
          scrollPane = null;
          repaint();
       }
       catch (NamingException e)
       {
          JOptionPane.showMessageDialog(LDAPFrame.this, e);
          e.printStackTrace();
       }
       catch (IOException e)
       {
          JOptionPane.showMessageDialog(LDAPFrame.this, e);
          e.printStackTrace();
       }
    }

    /**
     * Gets a context from the properties specified in the file ldapserver.properties
     * @return the directory context
     */
    public static DirContext getContext() throws NamingException, IOException
    {
       Properties props = new Properties();
       FileInputStream in = new FileInputStream("ldapserver.properties");
       props.load(in);
       in.close();

       String url = props.getProperty("ldap.url");
       String username = props.getProperty("ldap.username");
       String password = props.getProperty("ldap.password");

       Hashtable<String, String> env = new Hashtable<String, String>();
       env.put(Context.SECURITY_PRINCIPAL, username);
       env.put(Context.SECURITY_CREDENTIALS, password);
       DirContext initial = new InitialDirContext(env);
       DirContext context = (DirContext) initial.lookup(url);

       return context;
    }

    public static final int DEFAULT_WIDTH = 300;
    public static final int DEFAULT_HEIGHT = 200;

    private JButton findButton;
    private JButton saveButton;
    private JButton deleteButton;

    private JTextField uidField;
    private DataPanel dataPanel;
```

```java
   private Component scrollPane;

   private DirContext context;
   private String uid;
   private Attributes attrs;
}

/**
 * This panel displays the contents of a result set.
```

```java
 */
class DataPanel extends JPanel
{
   /**
    * Constructs the data panel.
    * @param attributes the attributes of the given entry
```

```java
    */
   public DataPanel(Attributes attrs) throws NamingException
   {
      setLayout(new java.awt.GridLayout(0, 2, 3, 1));

      NamingEnumeration<? extends Attribute> attrEnum = attrs.getAll();
      while (attrEnum.hasMore())
      {
         Attribute attr = attrEnum.next();
         String id = attr.getID();

         NamingEnumeration<?> valueEnum = attr.getAll();
         while (valueEnum.hasMore())
         {
            Object value = valueEnum.next();
            if (id.equals("userPassword")) value = new String((byte[]) value);

            JLabel idLabel = new JLabel(id, SwingConstants.RIGHT);
            JTextField valueField = new JTextField("" + value);
            if (id.equals("objectClass")) valueField.setEditable(false);
            if (!id.equals("uid"))
            {
               add(idLabel);
               add(valueField);
            }
         }
      }
   }

   public Attributes getEditedAttributes()
   {
      Attributes attrs = new BasicAttributes();
      for (int i = 0; i < getComponentCount(); i += 2)
      {
         JLabel idLabel = (JLabel) getComponent(i);
         JTextField valueField = (JTextField) getComponent(i + 1);
         String id = idLabel.getText();
         String value = valueField.getText();
         if (id.equals("userPassword")) attrs.put("userPassword", value.getBytes());
         else if (!id.equals("") && !id.equals("objectClass")) attrs.put(id, value);
      }
      return attrs;
   }
}
```

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.

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`.

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.

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

Country	Code
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

UNREGISTERED VERSION OF CHM TO PDF CONVERTER PRO BY THETA-SOFTWARE

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

UNREGISTERED VERSION OF CHM TO PDF CONVERTER PRO BY THETA-SOFTWARE

```
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`

Country	Code
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.

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 get*Xxx*Instance 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:
```
import java.awt.*;
import java.awt.event.*;
import java.text.*;
import java.util.*;
```

```java

import javax.swing.*;

/**
 * This program demonstrates formatting numbers under various locales.
 * @version 1.13 2007-07-25
 * @author Cay Horstmann
 */
public class NumberFormatTest
```



```java
{
   public static void main(String[] args)
   {
      EventQueue.invokeLater(new Runnable()
         {
            public void run()
            {
```



```java
               JFrame frame = new NumberFormatFrame();
               frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
               frame.setVisible(true);
            }
         });
   }
}

/**
 * This frame contains radio buttons to select a number format, a combo box to pick a locale,
 * a text field to display a formatted number, and a button to parse the text field contents.
 */
class NumberFormatFrame extends JFrame
{
   public NumberFormatFrame()
   {
      setLayout(new GridBagLayout());

      ActionListener listener = new ActionListener()
         {
      setTitle("NumberFormatTest");
            public void actionPerformed(ActionEvent event)
            {
               updateDisplay();
            }
         };

      JPanel p = new JPanel();
      addRadioButton(p, numberRadioButton, rbGroup, listener);
      addRadioButton(p, currencyRadioButton, rbGroup, listener);
      addRadioButton(p, percentRadioButton, rbGroup, listener);

      add(new JLabel("Locale:"), new GBC(0, 0).setAnchor(GBC.EAST));
      add(p, new GBC(1, 1));
      add(parseButton, new GBC(0, 2).setInsets(2));
      add(localeCombo, new GBC(1, 0).setAnchor(GBC.WEST));
      add(numberText, new GBC(1, 2).setFill(GBC.HORIZONTAL));
      locales = (Locale[]) NumberFormat.getAvailableLocales().clone();
      Arrays.sort(locales, new Comparator<Locale>()
         {
            public int compare(Locale l1, Locale l2)
            {
               return l1.getDisplayName().compareTo(l2.getDisplayName());
```

```
            }
         });
      for (Locale loc : locales)
         localeCombo.addItem(loc.getDisplayName());
      localeCombo.setSelectedItem(Locale.getDefault().getDisplayName());
      currentNumber = 123456.78;
      updateDisplay();

      localeCombo.addActionListener(listener);

      parseButton.addActionListener(new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               String s = numberText.getText().trim();
               try
               {
                  Number n = currentNumberFormat.parse(s);
                  if (n != null)
                  {
                     currentNumber = n.doubleValue();
                     updateDisplay();
                  }
                  else
                  {
                     numberText.setText("Parse error: " + s);
                  }
               }
               catch (ParseException e)
               {
                  numberText.setText("Parse error: " + s);
               }
            }
         });
      pack();
   }

   /**
    * Adds a radio button to a container.
    * @param p the container into which to place the button
    * @param b the button
    * @param g the button group
    * @param listener the button listener
    */
   public void addRadioButton(Container p, JRadioButton b, ButtonGroup g,
                              ActionListener listener)
   {
      b.setSelected(g.getButtonCount() == 0);
      b.addActionListener(listener);
      g.add(b);
      p.add(b);
   }

   /**
    * Updates the display and formats the number according to the user settings.
    */
   public void updateDisplay()
   {
      Locale currentLocale = locales[localeCombo.getSelectedIndex()];
```

```
         currentNumberFormat = null;
         if (numberRadioButton.isSelected()) currentNumberFormat = NumberFormat
               .getNumberInstance(currentLocale);
         else if (currencyRadioButton.isSelected()) currentNumberFormat = NumberFormat
               .getCurrencyInstance(currentLocale);
         else if (percentRadioButton.isSelected()) currentNumberFormat = NumberFormat
               .getPercentInstance(currentLocale);
         String n = currentNumberFormat.format(currentNumber);
         numberText.setText(n);
      }
```



```
      private Locale[] locales;
      private double currentNumber;
      private JComboBox localeCombo = new JComboBox();
      private JButton parseButton = new JButton("Parse");
      private JTextField numberText = new JTextField(30);
      private JRadioButton numberRadioButton = new JRadioButton("Number");
      private JRadioButton currencyRadioButton = new JRadioButton("Currency");
      private JRadioButton percentRadioButton = new JRadioButton("Percent");
      private ButtonGroup rbGroup = new ButtonGroup();
      private NumberFormat currentNumberFormat;
   }
```


API

`java.text.NumberFormat` 1.1

- `static Locale[] getAvailableLocales()`

  returns an array of `Locale` objects for which `NumberFormat` formatters are available.

- `static NumberFormat getNumberInstance()`
- `static NumberFormat getNumberInstance(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

API

`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. Firs 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.

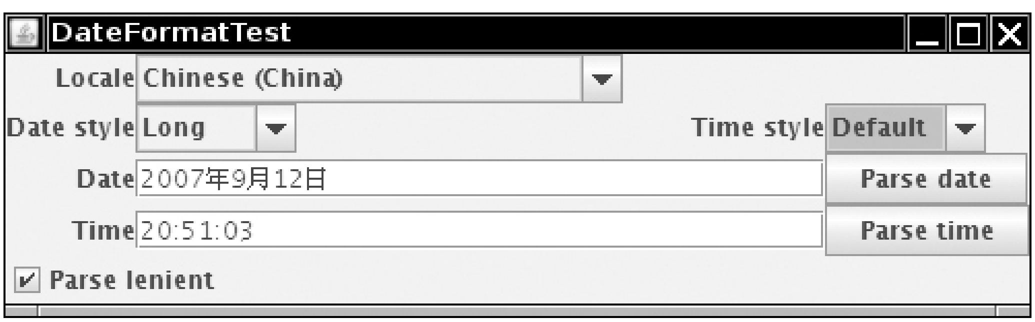
A lenient flag interprets dates leniently. For example, February 30, 2007 will be automatically converted to March 2, 2007. This seems dangerous, but, unfortunately, it is the default. You should probably turn off this feature. The calendar object that interprets the parsed date will throw IllegalArgumentException 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 th 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:

```
import java.awt.*;
import java.awt.event.*;
import java.text.*;
import java.util.*;

import javax.swing.*;

/**
 * This program demonstrates formatting dates under various locales.
 * @version 1.13 2007-07-25
 * @author Cay Horstmann
 */
public class DateFormatTest
{
   public static void main(String[] args)
   {
      EventQueue.invokeLater(new Runnable()
         {
            public void run()
            {
               JFrame frame = new DateFormatFrame();
               frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
               frame.setVisible(true);
            }
         });
   }
}

/**
 * This frame contains combo boxes to pick a locale, date and time formats, text fields
 * to display formatted date and time, buttons to parse the text field contents, and a
 * "lenient" checkbox.
 */
class DateFormatFrame extends JFrame
{
   public DateFormatFrame()
```

```
   {
      setTitle("DateFormatTest");

      setLayout(new GridBagLayout());
      add(new JLabel("Locale"), new GBC(0, 0).setAnchor(GBC.EAST));
      add(new JLabel("Date style"), new GBC(0, 1).setAnchor(GBC.EAST));
      add(new JLabel("Time style"), new GBC(2, 1).setAnchor(GBC.EAST));
      add(new JLabel("Date"), new GBC(0, 2).setAnchor(GBC.EAST));
      add(new JLabel("Time"), new GBC(0, 3).setAnchor(GBC.EAST));
      add(localeCombo, new GBC(1, 0, 2, 1).setAnchor(GBC.WEST));
      add(dateStyleCombo, new GBC(1, 1).setAnchor(GBC.WEST));
      add(timeStyleCombo, new GBC(3, 1).setAnchor(GBC.WEST));
      add(dateParseButton, new GBC(3, 2).setAnchor(GBC.WEST));
      add(timeParseButton, new GBC(3, 3).setAnchor(GBC.WEST));
      add(lenientCheckbox, new GBC(0, 4, 2, 1).setAnchor(GBC.WEST));
      add(dateText, new GBC(1, 2, 2, 1).setFill(GBC.HORIZONTAL));
      add(timeText, new GBC(1, 3, 2, 1).setFill(GBC.HORIZONTAL));

      locales = (Locale[]) DateFormat.getAvailableLocales().clone();
      Arrays.sort(locales, new Comparator<Locale>()
         {
            public int compare(Locale l1, Locale l2)
            {
               return l1.getDisplayName().compareTo(l2.getDisplayName());
            }
         });
      for (Locale loc : locales)
         localeCombo.addItem(loc.getDisplayName());
      localeCombo.setSelectedItem(Locale.getDefault().getDisplayName());
      currentDate = new Date();
      currentTime = new Date();
      updateDisplay();

      ActionListener listener = new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               updateDisplay();
            }
         };

      localeCombo.addActionListener(listener);
      dateStyleCombo.addActionListener(listener);
      timeStyleCombo.addActionListener(listener);

      dateParseButton.addActionListener(new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               String d = dateText.getText().trim();
               try
               {
                  currentDateFormat.setLenient(lenientCheckbox.isSelected());
                  Date date = currentDateFormat.parse(d);
                  currentDate = date;
                  updateDisplay();
               }
               catch (ParseException e)
               {
```

```
                  dateText.setText("Parse error: " + d);
               }
               catch (IllegalArgumentException e)
               {
                  dateText.setText("Argument error: " + d);
               }
            }
         });

      timeParseButton.addActionListener(new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               String t = timeText.getText().trim();
               try
               {
                  currentDateFormat.setLenient(lenientCheckbox.isSelected());
                  Date date = currentTimeFormat.parse(t);
                  currentTime = date;
                  updateDisplay();
               }
               catch (ParseException e)
               {
                  timeText.setText("Parse error: " + t);
               }
               catch (IllegalArgumentException e)
               {
                  timeText.setText("Argument error: " + t);
               }
            }
         });
      pack();
   }

   /**
    * Updates the display and formats the date according to the user settings.
    */
   public void updateDisplay()
   {
      Locale currentLocale = locales[localeCombo.getSelectedIndex()];
      int dateStyle = dateStyleCombo.getValue();
      currentDateFormat = DateFormat.getDateInstance(dateStyle, currentLocale);
      String d = currentDateFormat.format(currentDate);
      dateText.setText(d);
      int timeStyle = timeStyleCombo.getValue();
      currentTimeFormat = DateFormat.getTimeInstance(timeStyle, currentLocale);
      String t = currentTimeFormat.format(currentTime);
      timeText.setText(t);
   }

   private Locale[] locales;
   private Date currentDate;
   private Date currentTime;
   private DateFormat currentDateFormat;
   private DateFormat currentTimeFormat;
   private JComboBox localeCombo = new JComboBox();
   private EnumCombo dateStyleCombo = new EnumCombo(DateFormat.class, new String[] { "Default",
         "Full", "Long", "Medium", "Short" });
   private EnumCombo timeStyleCombo = new EnumCombo(DateFormat.class, new String[] { "Default",
```

```
         "Full", "Long", "Medium", "Short" });
   private JButton dateParseButton = new JButton("Parse date");
   private JButton timeParseButton = new JButton("Parse time");
   private JTextField dateText = new JTextField(30);
   private JTextField timeText = new JTextField(30);
   private JCheckBox lenientCheckbox = new JCheckBox("Parse lenient", true);
}
```

Listing 5-3. **EnumCombo.java**

Code View:

```
import java.util.*;
import javax.swing.*;

/**
   A combo box that lets users choose from among static field
   values whose names are given in the constructor.
   @version 1.13 2007-07-25
   @author Cay Horstmann
*/
public class EnumCombo extends JComboBox
{
   /**
      Constructs an EnumCombo.
      @param cl a class
      @param labels an array of static field names of cl
   */
   public EnumCombo(Class<?> cl, String[] labels)
   {
      for (String label : labels)
      {
         String name = label.toUpperCase().replace(' ', '_');
         int value = 0;
         try
         {
            java.lang.reflect.Field f = cl.getField(name);
            value = f.getInt(cl);
         }
         catch (Exception e)
         {
            label = "(" + label + ")";
         }
         table.put(label, value);
         addItem(label);
      }
      setSelectedItem(labels[0]);
   }

   /**
      Returns the value of the field that the user selected.
      @return the static field value
   */
```

```
   public int getValue()
   {
      return table.get(getSelectedItem());
   }

   private Map<String, Integer> table = new TreeMap<String, Integer>();
}
```


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.

API `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.

- `String getID()`

  gets the ID of this time zone.

- `String getDisplayName()`

- `String getDisplayName(Locale locale)`

- `String getDisplayName(boolean daylight, int style)`

- `String getDisplayName(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 lette 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 ≠ Ångström	Angstrom ≠ Ångström
Able = able	Able = able	Able ≠ 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 ≠ A°	Å = A°	Å = A°
™ ≠ TM	™ ≠ 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:

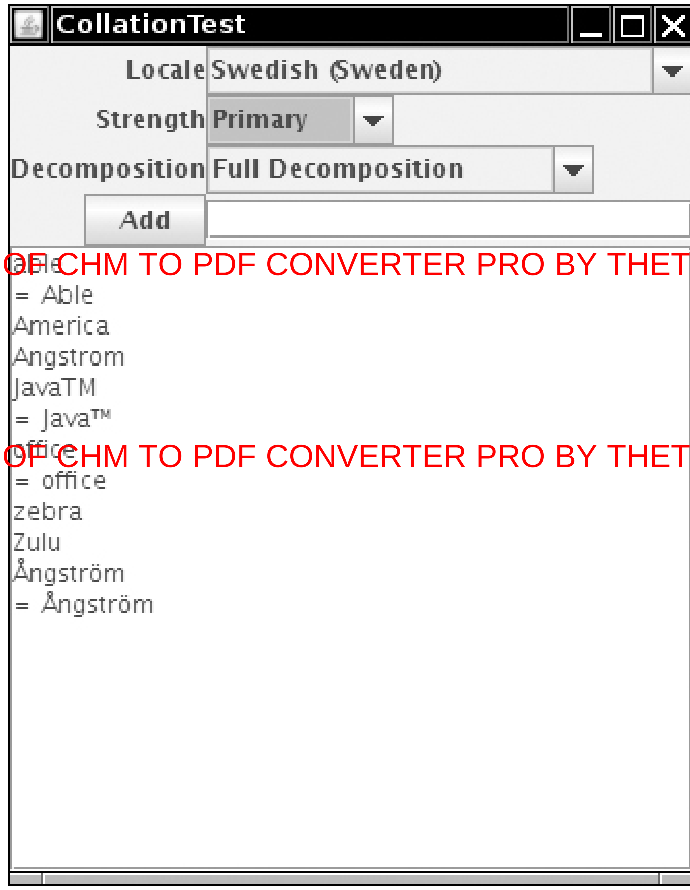
```
String name = "Ångström";
String normalized = Normalizer.normalize(name, Normalizer.Form.NFD); // uses normalization form
```

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



UNREGISTERED VERSION OF CHM TO PDF CONVERTER PRO BY THETA-SOFTWARE

The locale names in the combo box are displayed in sorted order, using the collator of the default locale. If you ru 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 closin parenthesis.

Listing 5-4. **CollationTest.java**

Code View:
```
import java.awt.*;
import java.awt.event.*;
import java.text.*;
import java.util.*;
import java.util.List;

import javax.swing.*;

/**
 * This program demonstrates collating strings under various locales.
 * @version 1.13 2007-07-25
 * @author Cay Horstmann
 */
public class CollationTest
{
   public static void main(String[] args)
   {
```

```
      EventQueue.invokeLater(new Runnable()
         {
            public void run()
            {

               JFrame frame = new CollationFrame();
               frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
               frame.setVisible(true);
            }
         });
   }
}

/**
 * This frame contains combo boxes to pick a locale, collation strength and decomposition
 * rules, a text field and button to add new strings, and a text area to list the
 * collated strings.
 */
class CollationFrame extends JFrame
{
   public CollationFrame()
   {
      setTitle("CollationTest");

      setLayout(new GridBagLayout());
      add(new JLabel("Locale"), new GBC(0, 0).setAnchor(GBC.EAST));
      add(new JLabel("Strength"), new GBC(0, 1).setAnchor(GBC.EAST));
      add(new JLabel("Decomposition"), new GBC(0, 2).setAnchor(GBC.EAST));
      add(addButton, new GBC(0, 3).setAnchor(GBC.EAST));
      add(localeCombo, new GBC(1, 0).setAnchor(GBC.WEST));
      add(strengthCombo, new GBC(1, 1).setAnchor(GBC.WEST));
      add(decompositionCombo, new GBC(1, 2).setAnchor(GBC.WEST));
      add(newWord, new GBC(1, 3).setFill(GBC.HORIZONTAL));
      add(new JScrollPane(sortedWords), new GBC(0, 4, 2, 1).setFill(GBC.BOTH));

      locales = (Locale[]) Collator.getAvailableLocales().clone();
      Arrays.sort(locales, new Comparator<Locale>()
         {
            private Collator collator = Collator.getInstance(Locale.getDefault());

            public int compare(Locale l1, Locale l2)
            {
               return collator.compare(l1.getDisplayName(), l2.getDisplayName());
            }
         });
      for (Locale loc : locales)
         localeCombo.addItem(loc.getDisplayName());
      localeCombo.setSelectedItem(Locale.getDefault().getDisplayName());

      strings.add("America");
      strings.add("able");
      strings.add("Zulu");
      strings.add("zebra");
      strings.add("\u00C5ngstr\u00F6m");
      strings.add("A\u030angstro\u0308m");
      strings.add("Angstrom");
      strings.add("Able");
      strings.add("office");
      strings.add("o\uFB03ce");
```

```
      strings.add("Java\u2122");
      strings.add("JavaTM");
      updateDisplay();

      addButton.addActionListener(new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               strings.add(newWord.getText());
               updateDisplay();
            }
         });

      ActionListener listener = new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               updateDisplay();
            }
         };

      localeCombo.addActionListener(listener);
      strengthCombo.addActionListener(listener);
      decompositionCombo.addActionListener(listener);
      pack();
   }

   /**
    * Updates the display and collates the strings according to the user settings.
    */
   public void updateDisplay()
   {
      Locale currentLocale = locales[localeCombo.getSelectedIndex()];
      localeCombo.setLocale(currentLocale);

      currentCollator = Collator.getInstance(currentLocale);
      currentCollator.setStrength(strengthCombo.getValue());
      currentCollator.setDecomposition(decompositionCombo.getValue());

      Collections.sort(strings, currentCollator);

      sortedWords.setText("");
      for (int i = 0; i < strings.size(); i++)
      {
         String s = strings.get(i);
         if (i > 0 && currentCollator.compare(s, strings.get(i - 1)) == 0) sortedWords
               .append("= ");
         sortedWords.append(s + "\n");
      }
      pack();
   }

   private List<String> strings = new ArrayList<String>();
   private Collator currentCollator;
   private Locale[] locales;
   private JComboBox localeCombo = new JComboBox();

   private EnumCombo strengthCombo = new EnumCombo(Collator.class, new String[] { "Primary",
         "Secondary", "Tertiary", "Identical" });
```

```
    private EnumCombo decompositionCombo = new EnumCombo(Collator.class, new String[] {
          "Canonical Decomposition", "Full Decomposition", "No Decomposition" });
    private JTextField newWord = new JTextField(20);
    private JTextArea sortedWords = new JTextArea(20, 20);
    private JButton addButton = new JButton("Add");
}
```

API

`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 });
```

API

`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.

API

`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 ≤ symbol (expressed as the Unicode character code `\u2264`) as a synonym for `#`. If you like, you can even specify a lower bound of -∞ as `-\u221E` for the first value.

For example,

```
-∞<no houses|0<one house|2≤{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, value2},
      { 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)
- static ResourceBundle getBundle(String baseName)

  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)

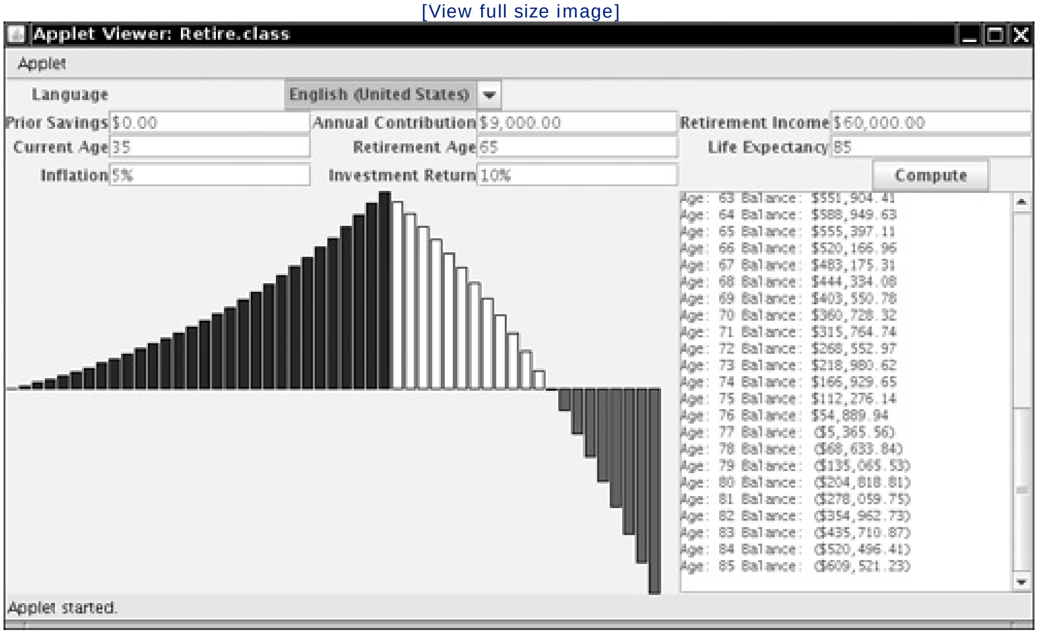
  should be overridden to look up the resource value associated with the given key if you define your own resource lookup mechanism.

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 5-12 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]

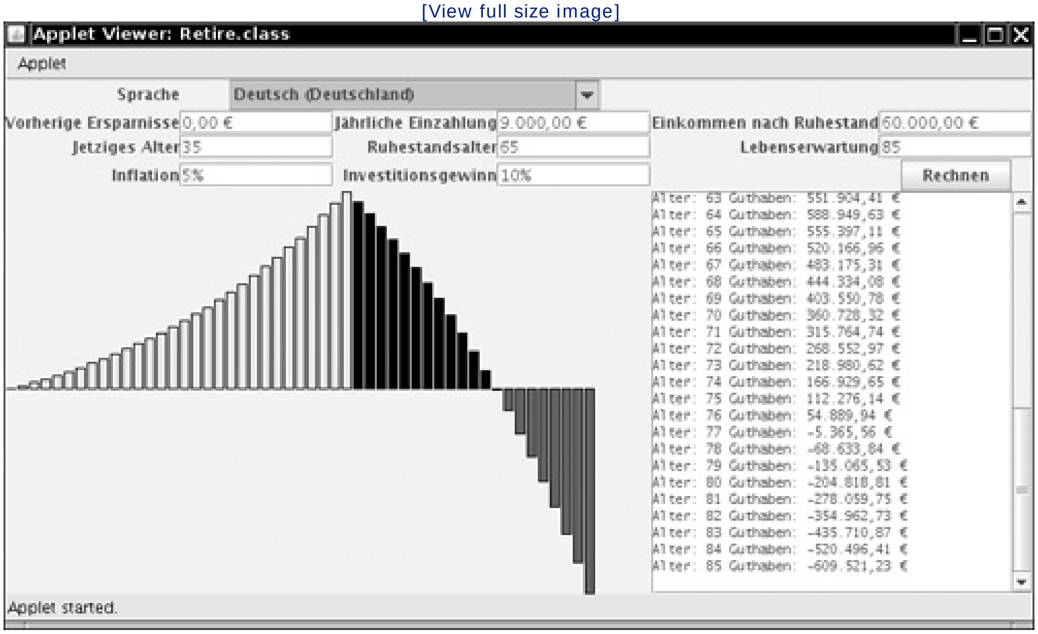
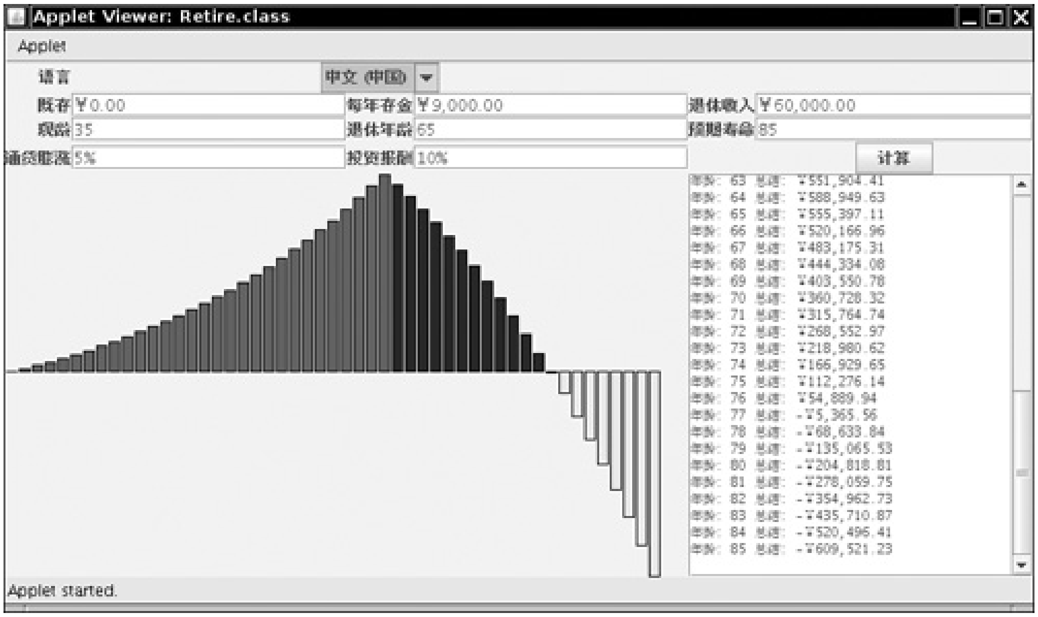


Figure 5-6. The retirement calculator in Chinese

[View full size image]



Listing 5-5. **Retire.java**

Code View:
```
import java.awt.*;
import java.awt.event.*;
import java.awt.geom.*;
import java.util.*;
import java.text.*;
import javax.swing.*;

/**
 * This applet shows a retirement calculator. The UI is displayed in English, German,
 * and Chinese.
 * @version 1.22 2007-07-25
 * @author Cay Horstmann
 */
public class Retire extends JApplet
{
   public void init()
   {
      EventQueue.invokeLater(new Runnable()
         {
            public void run()
            {
               initUI();
            }
         });
   }

   public void initUI()
   {
      setLayout(new GridBagLayout());
      add(languageLabel, new GBC(0, 0).setAnchor(GBC.EAST));
      add(savingsLabel, new GBC(0, 1).setAnchor(GBC.EAST));
      add(contribLabel, new GBC(2, 1).setAnchor(GBC.EAST));
```

```
      add(incomeLabel, new GBC(4, 1).setAnchor(GBC.EAST));
      add(currentAgeLabel, new GBC(0, 2).setAnchor(GBC.EAST));
      add(retireAgeLabel, new GBC(2, 2).setAnchor(GBC.EAST));
      add(deathAgeLabel, new GBC(4, 2).setAnchor(GBC.EAST));
      add(inflationPercentLabel, new GBC(0, 3).setAnchor(GBC.EAST));
      add(investPercentLabel, new GBC(2, 3).setAnchor(GBC.EAST));
      add(localeCombo, new GBC(1, 0, 3, 1));
      add(savingsField, new GBC(1, 1).setWeight(100, 0).setFill(GBC.HORIZONTAL));
      add(contribField, new GBC(3, 1).setWeight(100, 0).setFill(GBC.HORIZONTAL));
```

```
      add(incomeField, new GBC(5, 1).setWeight(100, 0).setFill(GBC.HORIZONTAL));
      add(currentAgeField, new GBC(1, 2).setWeight(100, 0).setFill(GBC.HORIZONTAL));
      add(retireAgeField, new GBC(3, 2).setWeight(100, 0).setFill(GBC.HORIZONTAL));
      add(deathAgeField, new GBC(5, 2).setWeight(100, 0).setFill(GBC.HORIZONTAL));
      add(inflationPercentField, new GBC(1, 3).setWeight(100, 0).setFill(GBC.HORIZONTAL));
      add(investPercentField, new GBC(3, 3).setWeight(100, 0).setFill(GBC.HORIZONTAL));
      add(retireCanvas, new GBC(0, 4, 4, 1).setWeight(100, 100).setFill(GBC.BOTH));
```

```
      add(new JScrollPane(retireText), new GBC(4, 4, 2, 1).setWeight(0, 100).setFill(GBC.BOTH))

      computeButton.setName("computeButton");
      computeButton.addActionListener(new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               getInfo();
               updateData();
               updateGraph();
            }
         });
      add(computeButton, new GBC(5, 3));

      retireText.setEditable(false);
      retireText.setFont(new Font("Monospaced", Font.PLAIN, 10));

      info.setSavings(0);
      info.setContrib(9000);
      info.setIncome(60000);
      info.setCurrentAge(35);
      info.setRetireAge(65);
      info.setDeathAge(85);
      info.setInvestPercent(0.1);
      info.setInflationPercent(0.05);

      int localeIndex = 0; // US locale is default selection
      for (int i = 0; i < locales.length; i++)
         // if current locale one of the choices, select it
         if (getLocale().equals(locales[i])) localeIndex = i;
      setCurrentLocale(locales[localeIndex]);

      localeCombo.addActionListener(new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               setCurrentLocale((Locale) localeCombo.getSelectedItem());
               validate();
            }
         });
   }

   /**
```

```
    * Sets the current locale.
    * @param locale the desired locale
    */
   public void setCurrentLocale(Locale locale)
      currentLocale = locale;
      localeCombo.setSelectedItem(currentLocale);
      localeCombo.setLocale(currentLocale);
   {

      res = ResourceBundle.getBundle("RetireResources", currentLocale);
      resStrings = ResourceBundle.getBundle("RetireStrings", currentLocale);
      currencyFmt = NumberFormat.getCurrencyInstance(currentLocale);
      numberFmt = NumberFormat.getNumberInstance(currentLocale);
      percentFmt = NumberFormat.getPercentInstance(currentLocale);

      updateDisplay();
      updateInfo();
      updateData();
      updateGraph();
   }

   /**
    * Updates all labels in the display.
    */
   public void updateDisplay()
   {
      languageLabel.setText(resStrings.getString("language"));
      savingsLabel.setText(resStrings.getString("savings"));
      contribLabel.setText(resStrings.getString("contrib"));
      incomeLabel.setText(resStrings.getString("income"));
      currentAgeLabel.setText(resStrings.getString("currentAge"));
      retireAgeLabel.setText(resStrings.getString("retireAge"));
      deathAgeLabel.setText(resStrings.getString("deathAge"));
      inflationPercentLabel.setText(resStrings.getString("inflationPercent"));
      investPercentLabel.setText(resStrings.getString("investPercent"));
      computeButton.setText(resStrings.getString("computeButton"));
   }

   /**
    * Updates the information in the text fields.
    */
   public void updateInfo()
   {
      savingsField.setText(currencyFmt.format(info.getSavings()));
      contribField.setText(currencyFmt.format(info.getContrib()));
      incomeField.setText(currencyFmt.format(info.getIncome()));
      currentAgeField.setText(numberFmt.format(info.getCurrentAge()));
      retireAgeField.setText(numberFmt.format(info.getRetireAge()));
      deathAgeField.setText(numberFmt.format(info.getDeathAge()));
      investPercentField.setText(percentFmt.format(info.getInvestPercent()));
      inflationPercentField.setText(percentFmt.format(info.getInflationPercent()));
   }

   /**
    * Updates the data displayed in the text area.
    */
   public void updateData()
   {
      retireText.setText("");
```

```
      MessageFormat retireMsg = new MessageFormat("");
      retireMsg.setLocale(currentLocale);
      retireMsg.applyPattern(resStrings.getString("retire"));

      for (int i = info.getCurrentAge(); i <= info.getDeathAge(); i++)
      {
         Object[] args = { i, info.getBalance(i) };
         retireText.append(retireMsg.format(args) + "\n");
      }
```

```
   }

   /**
    * Updates the graph.
    */
   public void updateGraph()
```

```
   {
      retireCanvas.setColorPre((Color) res.getObject("colorPre"));
      retireCanvas.setColorGain((Color) res.getObject("colorGain"));
      retireCanvas.setColorLoss((Color) res.getObject("colorLoss"));
      retireCanvas.setInfo(info);
      repaint();
   }

   /**
    * Reads the user input from the text fields.
    */
   public void getInfo()
   {
      try
      {
         info.setSavings(currencyFmt.parse(savingsField.getText()).doubleValue());
         info.setContrib(currencyFmt.parse(contribField.getText()).doubleValue());
         info.setIncome(currencyFmt.parse(incomeField.getText()).doubleValue());
         info.setCurrentAge(numberFmt.parse(currentAgeField.getText()).intValue());
         info.setRetireAge(numberFmt.parse(retireAgeField.getText()).intValue());
         info.setDeathAge(numberFmt.parse(deathAgeField.getText()).intValue());
         info.setInvestPercent(percentFmt.parse(investPercentField.getText()).doubleValue());
         info.setInflationPercent(percentFmt.parse(
               inflationPercentField.getText()).doubleValue());
      }
      catch (ParseException e)
      {
      }
   }

   private JTextField savingsField = new JTextField(10);
   private JTextField contribField = new JTextField(10);
   private JTextField incomeField = new JTextField(10);
   private JTextField currentAgeField = new JTextField(4);
   private JTextField retireAgeField = new JTextField(4);
   private JTextField deathAgeField = new JTextField(4);
   private JTextField inflationPercentField = new JTextField(6);
   private JTextField investPercentField = new JTextField(6);
   private JTextArea retireText = new JTextArea(10, 25);
   private RetireCanvas retireCanvas = new RetireCanvas();
   private JButton computeButton = new JButton();
   private JLabel languageLabel = new JLabel();
   private JLabel savingsLabel = new JLabel();
   private JLabel contribLabel = new JLabel();
```

```
   private JLabel incomeLabel = new JLabel();
   private JLabel currentAgeLabel = new JLabel();
   private JLabel retireAgeLabel = new JLabel();
   private JLabel deathAgeLabel = new JLabel();
   private JLabel inflationPercentLabel = new JLabel();
   private JLabel investPercentLabel = new JLabel();

   private RetireInfo info = new RetireInfo();

   private Locale[] locales = { Locale.US, Locale.CHINA, Locale.GERMANY };
   private Locale currentLocale;
   private JComboBox localeCombo = new LocaleCombo(locales);
   private ResourceBundle res;
   private ResourceBundle resStrings;
   private NumberFormat currencyFmt;
   private NumberFormat numberFmt;
   private NumberFormat percentFmt;
}

/**
 * The information required to compute retirement income data.
 */
class RetireInfo
{
   /**
    * Gets the available balance for a given year.
    * @param year the year for which to compute the balance
    * @return the amount of money available (or required) in that year
    */
   public double getBalance(int year)
   {
      if (year < currentAge) return 0;
      else if (year == currentAge)
      {
         age = year;
         balance = savings;
         return balance;
      }
      else if (year == age) return balance;
      if (year != age + 1) getBalance(year - 1);
      age = year;
      if (age < retireAge) balance += contrib;
      else balance -= income;
      balance = balance * (1 + (investPercent - inflationPercent));
      return balance;
   }

   /**
    * Gets the amount of prior savings.
    * @return the savings amount
    */
   public double getSavings()
   {
      return savings;
   }

   /**
    * Sets the amount of prior savings.
    * @param newValue the savings amount
```

```
    */
   public void setSavings(double newValue)
   {
      savings = newValue;
   }

   /**
    * Gets the annual contribution to the retirement account.
    * @return the contribution amount
    */
   public double getContrib()
   {
      return contrib;
   }

   /**
    * Sets the annual contribution to the retirement account.
    * @param newValue the contribution amount
    */
   public void setContrib(double newValue)
   {
      contrib = newValue;
   }

   /**
    * Gets the annual income.
    * @return the income amount
    */
   public double getIncome()
   {
      return income;
   }

   /**
    * Sets the annual income.
    * @param newValue the income amount
    */
   public void setIncome(double newValue)
   {
      income = newValue;
   }

   /**
    * Gets the current age.
    * @return the age
    */
   public int getCurrentAge()
   {
      return currentAge;
   }

   /**
    * Sets the current age.
    * @param newValue the age
    */
   public void setCurrentAge(int newValue)
   {
      currentAge = newValue;
   }
```

```
    /**
     * Gets the desired retirement age.
     * @return the age
     */
    public int getRetireAge()
    {
       return retireAge;
    }

    /**
     * Sets the desired retirement age.
     * @param newValue the age
     */
    public void setRetireAge(int newValue)
    {
       retireAge = newValue;
    }

    /**
     * Gets the expected age of death.
     * @return the age
     */
    public int getDeathAge()
    {
       return deathAge;
    }

    /**
     * Sets the expected age of death.
     * @param newValue the age
     */
    public void setDeathAge(int newValue)
    {
       deathAge = newValue;
    }

    /**
     * Gets the estimated percentage of inflation.
     * @return the percentage
     */
    public double getInflationPercent()
    {
       return inflationPercent;
    }

    /**
     * Sets the estimated percentage of inflation.
     * @param newValue the percentage
     */
    public void setInflationPercent(double newValue)
    {
       inflationPercent = newValue;
    }

    /**
     * Gets the estimated yield of the investment.
     * @return the percentage
     */
```

```
   public double getInvestPercent()
   {
      return investPercent;
   }

   /**
    * Sets the estimated yield of the investment.
    * @param newValue the percentage
    */
```

```
   public void setInvestPercent(double newValue)
   {
      investPercent = newValue;
   }

   private double savings;
```

```
   private double contrib;
   private double income;
   private int currentAge;
   private int retireAge;
   private int deathAge;
   private double inflationPercent;
   private double investPercent;

   private int age;
   private double balance;
}

/**
 * This panel draws a graph of the investment result.
 */
class RetireCanvas extends JPanel
{
   public RetireCanvas()
   {
      setSize(PANEL_WIDTH, PANEL_HEIGHT);
   }

   /**
    * Sets the retirement information to be plotted.
    * @param newInfo the new retirement info.
    */
   public void setInfo(RetireInfo newInfo)
   {
      info = newInfo;
      repaint();
   }

   public void paintComponent(Graphics g)
   {
      Graphics2D g2 = (Graphics2D) g;
      if (info == null) return;

      double minValue = 0;
      double maxValue = 0;
      int i;
      for (i = info.getCurrentAge(); i <= info.getDeathAge(); i++)
      {
         double v = info.getBalance(i);
         if (minValue > v) minValue = v;
```

```
         if (maxValue < v) maxValue = v;
      }
      if (maxValue == minValue) return;

      int barWidth = getWidth() / (info.getDeathAge() - info.getCurrentAge() + 1);
      double scale = getHeight() / (maxValue - minValue);

      for (i = info.getCurrentAge(); i <= info.getDeathAge(); i++)
      {
         int x1 = (i - info.getCurrentAge()) * barWidth + 1;
         int y1;
         double v = info.getBalance(i);
         int height;
         int yOrigin = (int) (maxValue * scale);

         if (v >= 0)
         {
            y1 = (int) ((maxValue - v) * scale);
            height = yOrigin - y1;
         }
         else
         {
            y1 = yOrigin;
            height = (int) (-v * scale);
         }

         if (i < info.getRetireAge()) g2.setPaint(colorPre);
         else if (v >= 0) g2.setPaint(colorGain);
         else g2.setPaint(colorLoss);
         Rectangle2D bar = new Rectangle2D.Double(x1, y1, barWidth - 2, height);
         g2.fill(bar);
         g2.setPaint(Color.black);
         g2.draw(bar);
      }
   }

   /**
    * Sets the color to be used before retirement.
    * @param color the desired color
    */
   public void setColorPre(Color color)
   {
      colorPre = color;
      repaint();
   }

   /**
    * Sets the color to be used after retirement while the account balance is positive.
    * @param color the desired color
    */
   public void setColorGain(Color color)
   {
      colorGain = color;
      repaint();
   }

   /**
    * Sets the color to be used after retirement when the account balance is negative.
    * @param color the desired color
```

```
     */
    public void setColorLoss(Color color)
    {
       colorLoss = color;
       repaint();
    }

    private RetireInfo info = null;
    private Color colorPre;
    private Color colorGain;
    private Color colorLoss;
    private static final int PANEL_WIDTH = 400;
    private static final int PANEL_HEIGHT = 200;
}
```


Listing 5-6. **RetireResources.java**

Code View:

```
import java.awt.*;

/**
 * These are the English non-string resources for the retirement calculator.
 * @version 1.21 2001-08-27
 * @author Cay Horstmann
 */
public class RetireResources extends java.util.ListResourceBundle
{
   public Object[][] getContents()
   {
      return contents;
   }

   static final Object[][] contents = {
   // BEGIN LOCALIZE
         { "colorPre", Color.blue }, { "colorGain", Color.white }, { "colorLoss", Color.red }
   // END LOCALIZE
   };
}
```

Listing 5-7. **RetireResources_de.java**

```
Code View:
import java.awt.*;

/**
 * These are the German non-string resources for the retirement calculator.
 * @version 1.21 2001-08-27
 * @author Cay Horstmann
 */
public class RetireResources_de extends java.util.ListResourceBundle
{
   public Object[][] getContents()
   {
      return contents;
   }

   static final Object[][] contents = {
   // BEGIN LOCALIZE
         { "colorPre", Color.yellow }, { "colorGain", Color.black }, { "colorLoss", Color.red }
   // END LOCALIZE
   };
}
```

Listing 5-8. **RetireResources_zh.java**

```
Code View:
import java.awt.*;

/**
 * These are the Chinese non-string resources for the retirement calculator.
 * @version 1.21 2001-08-27
 * @author Cay Horstmann
 */
public class RetireResources_zh extends java.util.ListResourceBundle
{
   public Object[][] getContents()
   {
      return contents;
   }

   static final Object[][] contents = {
   // BEGIN LOCALIZE
         { "colorPre", Color.red }, { "colorGain", Color.blue }, { "colorLoss", Color.yellow }
   // END LOCALIZE
   };
}
```

Listing 5-9. **RetireStrings.properties**

```
language=Language
computeButton=Compute
savings=Prior Savings
contrib=Annual Contribution
income=Retirement Income
currentAge=Current Age
retireAge=Retirement Age
deathAge=Life Expectancy
inflationPercent=Inflation
investPercent=Investment Return
retire=Age: {0,number} Balance: {1,number,currency}
```


Listing 5-10. **RetireStrings_de.properties**


```
language=Sprache
computeButton=Rechnen
savings=Vorherige Ersparnisse
contrib=J\u00e4hrliche Einzahlung
income=Einkommen nach Ruhestand
currentAge=Jetziges Alter
retireAge=Ruhestandsalter
deathAge=Lebenserwartung
inflationPercent=Inflation
investPercent=Investitionsgewinn
retire=Alter: {0,number} Guthaben: {1,number,currency}
```

Listing 5-11. **RetireStrings_zh.properties**

```
language=\u8bed\u8a00
computeButton=\u8ba1\u7b97
savings=\u65e2\u5b58
contrib=\u6bcf\u5e74\u5b58\u91d1
income=\u9000\u4f11\u6536\u5165
currentAge=\u73b0\u9f84
retireAge=\u9000\u4f11\u5e74\u9f84
deathAge=\u9884\u671f\u5bff\u547d
inflationPercent=\u901a\u8d27\u81a8\u6da8
investPercent=\u6295\u8d44\u62a5\u916c
retire=\u5e74\u9f84: {0,number} \u603b\u7ed3: {1,number,currency}
```

API

`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.

Chapter 6. Advanced Swing

- Lists

- Tables

- Trees

- Text Components

- Progress Indicators

- Component Organizers

In this chapter, we continue our discussion of the Swing user interface toolkit from Volume I. Swing is a rich toolkit, and Volume I covered only basic and commonly used components. That leaves us with three significantly more complex components for lists, tables, and trees, the exploration of which occupies a large part of this chapter. We then turn to text components and go beyond the simple text fields and text areas that you have seen in Volume I. We show you how to add validations and spinners to text fields and how you can display structured text such as HTML. Next, you will see a number of components for displaying progress of a slow activity. We finish the chapter by covering component organizers such as tabbed panes and desktop panes with internal frames.

Lists

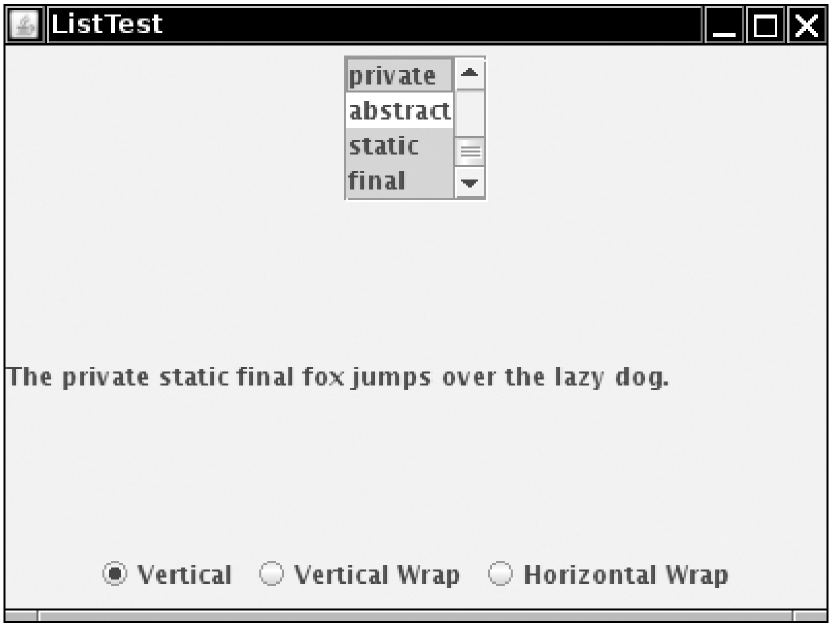
If you want to present a set of choices to a user, and a radio button or checkbox set consumes too much space, you can use a combo box or a list. Combo boxes were covered in Volume I because they are relatively simple. The `JList` component has many more features, and its design is similar to that of the tree and table components. For that reason, it is our starting point for the discussion of complex Swing components.

Of course, you can have lists of strings, but you can also have lists of arbitrary objects, with full control of how they appear. The internal architecture of the list component that makes this generality possible is rather elegant. Unfortunately, the designers at Sun felt that they needed to show off that elegance, rather than hiding it from the programmer who just wants to use the component. You will find that the list control is somewhat awkward to use for common cases because you need to manipulate some of the machinery that makes the general cases possible. We walk you through the simple and most common case, a list box of strings, and then give a more complex example that shows off the flexibility of the list component.

The **`JList`** Component

The JList component shows a number of items inside a single box. Figure 6-1 shows an admittedly silly example. The user can select the attributes for the fox, such as "quick," "brown," "hungry," "wild," and, because we ran out of attributes, "static," "private," and "final." You can thus have the *static final* fox jump over the lazy dog.

Figure 6-1. A list box



To construct this list component, you first start out with an array of strings, then pass the array to the JList constructor:

```
String[] words= { "quick", "brown", "hungry", "wild", ... };
JList wordList = new JList(words);
```

Alternatively, you can use an anonymous array:

Code View:
```
JList wordList = new JList(new String[] {"quick", "brown", "hungry", "wild", ... });
```

List boxes do not scroll automatically. To make a list box scroll, you must insert it into a scroll pane:

```
JScrollPane scrollPane = new JScrollPane(wordList);
```

You then add the scroll pane, not the list, into the surrounding panel.

We must admit that the separation of the list display and the scrolling mechanism is elegant in theory, but it is a pain in practice. Essentially all lists that we ever encountered needed scrolling. It seems cruel to force programmers to go through hoops in the default case just so they can appreciate that elegance.

By default, the list component displays eight items; use the `setVisibleRowCount` method to change that value:

```
wordList.setVisibleRowCount(4); // display 4 items
```

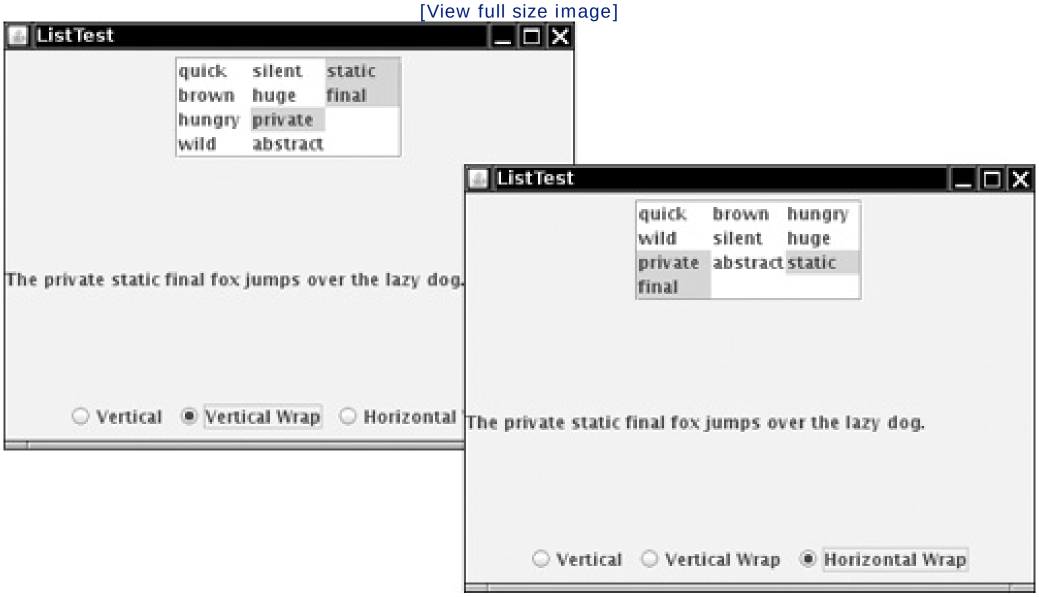
You can set the *layout orientation* to one of three values:

- `JList.VERTICAL` (the default)— Arrange all items vertically.

- `JList.VERTICAL_WRAP`— Start new columns if there are more items than the visible row count (see Figure 6-2).

Figure 6-2. Lists with vertical and horizontal wrap

[View full size image]



- `JList.HORIZONTAL_WRAP`— Start new columns if there are more items than the visible row count, but fill them horizontally. Look at the placement of the words "quick," "brown," and "hungry" in Figure 6-2 to see the difference between vertical and horizontal wrap.

By default, a user can select multiple items. To add more items to a selection, press the CTRL key while clicking on each item. To select a contiguous range of items, click on the first one, then hold down the SHIFT key and click on the last one.

You can also restrict the user to a more limited selection mode with the `setSelectionMode` method:

```
wordList.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
   // select one item at a time
wordList.setSelectionMode(ListSelectionModel.SINGLE_INTERVAL_SELECTION);
   // select one item or one range of items
```

You might recall from Volume I that the basic user interface components send out action events when the user activates them. List boxes use a different notification mechanism. Rather than listening to action events, you need to listen to list selection events. Add a list selection listener to the list component, and implement the method

```
public void valueChanged(ListSelectionEvent evt)
```

in the listener.

When the user selects items, a flurry of list selection events is generated. For example, suppose the user clicks on a new item. When the mouse button goes down, an event reports a change in selection. This is a transitional event—the call

```
event.isAdjusting()
```

returns `true` if the selection is not yet final. Then, when the mouse button goes up, there is another event, this time with `isAdjusting` returning `false`. If you are not interested in the transitional events, then you can wait for the event for which `isAdjusting` is `false`. However, if you want to give the user instant feedback as soon as the mouse button is clicked, you need to process all events.

Once you are notified that an event has happened, you will want to find out what items are currently selected. The `getSelectedValues` method returns an *array of objects* containing all selected items. Cast *each* array element to a string.

```
Object[] values = list.getSelectedValues();
for (Object value : values)
   do something with (String) value;
```

Caution

You cannot cast the return value of `getSelectedValues` from an `Object[]` array to a `String[]` array. The return value was not created as an array of strings, but as an array of objects, each of which happens to be a string. To process the return value as an array of strings, use the following code:

```
int length = values.length;
String[] words = new String[length];
System.arrayCopy(values, 0, words, 0, length);
```

If your list does not allow multiple selections, you can call the convenience method `getSelectedValue`. It returns the first selected value (which you know to be the only value if multiple selections are disallowed).

```
String value = (String) list.getSelectedValue();
```

Note

List components do not react to double clicks from a mouse. As envisioned by the designers of Swing, you use a list to select an item, and then you click a button to make something happen. However, some user interfaces allow a user to double-click on a list item as a shortcut for item selection and acceptance of a default action. If you want to implement this behavior, you have to add a mouse listener to the list box, then trap the mouse event as follows:


```
public void mouseClicked(MouseEvent evt)
{
   if (evt.getClickCount() == 2)
   {
      JList source = (JList) evt.getSource();
      Object[] selection = source.getSelectedValues();
      doAction(selection);
   }
}
```


Listing 6-1 is the listing of the program that demonstrates a list box filled with strings. Notice how the `valueChanged` method builds up the message string from the selected items.

Listing 6-1. **`ListTest.java`**

Code View:

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

/**
 * This program demonstrates a simple fixed list of strings.
 * @version 1.23 2007-08-01
 * @author Cay Horstmann
 */
public class ListTest
{
   public static void main(String[] args)
   {
      EventQueue.invokeLater(new Runnable()
         {
            public void run()
            {
               JFrame frame = new ListFrame();
               frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
               frame.setVisible(true);
            }
         });
   }
}

/**
 * This frame contains a word list and a label that shows a sentence made up from the chosen
 * words. Note that you can select multiple words with Ctrl+click and Shift+click.
 */
```

```java
class ListFrame extends JFrame
{
   public ListFrame()
   {
      setTitle("ListTest");
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

      String[] words = { "quick", "brown", "hungry", "wild", "silent", "huge", "private",
            "abstract", "static", "final" };

      wordList = new JList(words);
      wordList.setVisibleRowCount(4);
      JScrollPane scrollPane = new JScrollPane(wordList);

      listPanel = new JPanel();
      listPanel.add(scrollPane);
      wordList.addListSelectionListener(new ListSelectionListener()
         {
            public void valueChanged(ListSelectionEvent event)
            {
               Object[] values = wordList.getSelectedValues();

               StringBuilder text = new StringBuilder(prefix);
               for (int i = 0; i < values.length; i++)
               {
                  String word = (String) values[i];
                  text.append(word);
                  text.append(" ");
               }
               text.append(suffix);

               label.setText(text.toString());
            }
         });

      buttonPanel = new JPanel();
      group = new ButtonGroup();
      makeButton("Vertical", JList.VERTICAL);
      makeButton("Vertical Wrap", JList.VERTICAL_WRAP);
      makeButton("Horizontal Wrap", JList.HORIZONTAL_WRAP);

      add(listPanel, BorderLayout.NORTH);
      label = new JLabel(prefix + suffix);
      add(label, BorderLayout.CENTER);
      add(buttonPanel, BorderLayout.SOUTH);
   }

   /**
    * Makes a radio button to set the layout orientation.
    * @param label the button label
    * @param orientation the orientation for the list
    */
   private void makeButton(String label, final int orientation)
   {
      JRadioButton button = new JRadioButton(label);
      buttonPanel.add(button);
      if (group.getButtonCount() == 0) button.setSelected(true);
      group.add(button);
      button.addActionListener(new ActionListener()
```

```
            {
                public void actionPerformed(ActionEvent event)
                {
                    wordList.setLayoutOrientation(orientation);
                    listPanel.revalidate();
                }
            });
    }
```



```
    private static final int DEFAULT_WIDTH = 400;
    private static final int DEFAULT_HEIGHT = 300;
    private JPanel listPanel;
    private JList wordList;
    private JLabel label;
    private JPanel buttonPanel;
```



```
    private ButtonGroup group;
    private String prefix = "The ";
    private String suffix = "fox jumps over the lazy dog.";
}
```

`javax.swing.JList` 1.2

- `JList(Object[] items)`

  constructs a list that displays these items.

- `int getVisibleRowCount()`

- `void setVisibleRowCount(int c)`

  gets or sets the preferred number of rows in the list that can be displayed without a scroll bar.

- `int getLayoutOrientation()` **1.4**

- `void setLayoutOrientation(int orientation)` **1.4**

  gets or sets the layout orientation

Parameters:	`orientation`	One of `VERTICAL`, `VERTICAL_WRAP`, `HORIZONTAL_WRAP`

- `int getSelectionMode()`

- `void setSelectionMode(int mode)`

  gets or sets the mode that determines whether single-item or multiple-item selections are allowed.

Parameters:	`mode`	One of `SINGLE_SELECTION`, `SINGLE_INTERVAL_SELECTION`, `MULTIPLE_INTERVAL_SELECTION`

- `void addListSelectionListener(ListSelectionListener listener)`

  adds to the list a listener that's notified each time a change to the selection occurs.

- `Object[] getSelectedValues()`

  returns the selected values or an empty array if the selection is empty.

- `Object getSelectedValue()`

  returns the first selected value or `null` if the selection is empty.

API `javax.swing.event.ListSelectionListener` 1.2

- `void valueChanged(ListSelectionEvent e)`

  is called whenever the list selection changes.

List Models

In the preceding section, you saw the most common method for using a list component:

1. Specify a fixed set of strings for display in the list.
2. Place the list inside a scroll pane.
3. Trap the list selection events.

In the remainder of the section on lists, we cover more complex situations that require a bit more finesse:

- Very long lists
- Lists with changing contents
- Lists that don't contain strings

In the first example, we constructed a JList component that held a fixed collection of strings. However, the collection of choices in a list box is not always fixed. How do we add or remove items in the list box? Somewhat surprisingly, there are no methods in the JList class to achieve this. Instead, you have to understand a little more about the internal design of the list component. The list component uses the model-view-controller design pattern to separate the visual appearance (a column of items that are rendered in some way) from the underlying data (a collection of objects).

The JList class is responsible for the visual appearance of the data. It actually knows very little about how the data are stored—all it knows is that it can retrieve the data through some object that implements the ListModel interface:

```
public interface ListModel
{
   int getSize();
   Object getElementAt(int i);
   void addListDataListener(ListDataListener l);
   void removeListDataListener(ListDataListener l);
}
```

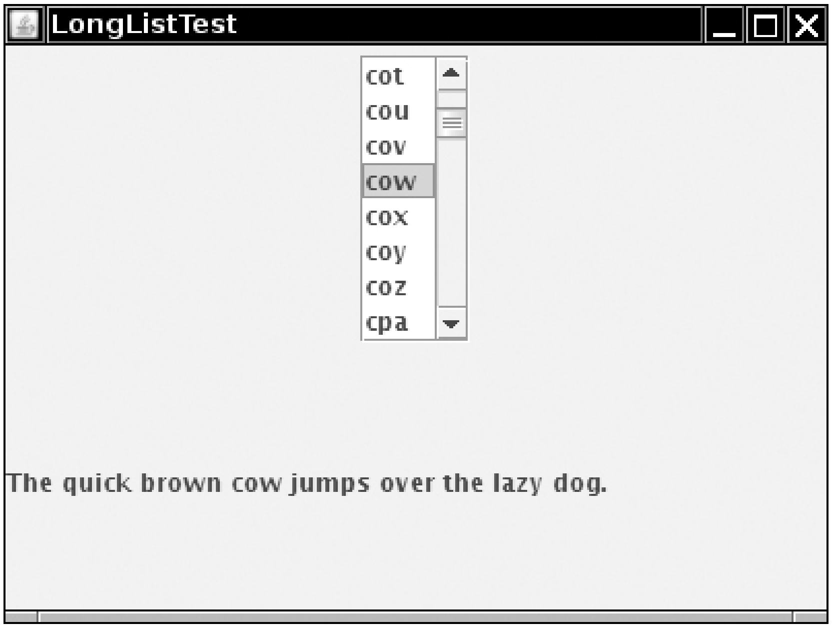
Through this interface, the JList can get a count of elements and retrieve each one of the elements. Also, the JList object can add itself as a ListDataListener. That way, if the collection of elements changes, the JList gets notified so that it can repaint itself.

Why is this generality useful? Why doesn't the JList object simply store an array of objects?

Note that the interface doesn't specify how the objects are stored. In particular, it doesn't force them to be stored at all! The getElementAt method is free to recompute each value whenever it is called. This is potentially useful if you want to show a very large collection without having to store the values.

Here is a somewhat silly example: We let the user choose among *all three-letter words* in a list box (see Figure 6-3).

Figure 6-3. Choosing from a very long list of selections



There are 26 x 26 x 26 = 17,576 three-letter combinations. Rather than storing all these combinations, we recompute them as requested when the user scrolls through them.

This turns out to be easy to implement. The tedious part, adding and removing listeners, has been done for us in the `AbstractListModel` class, which we extend. We only need to supply the `getSize` and `getElementAt` methods:

```
class WordListModel extends AbstractListModel
{
   public WordListModel(int n) { length = n; }
   public int getSize() { return (int) Math.pow(26, length); }
   public Object getElementAt(int n)
   {
      // compute nth string
      . . .
   }
   . . .
}
```

The computation of the *n*th string is a bit technical—you'll find the details in Listing 6-2.

Now that we have supplied a model, we can simply build a list that lets the user scroll through the elements supplied by the model:

```
JList wordList = new JList(new WordListModel(3));
wordList.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
JScrollPane scrollPane = new JScrollPane(wordList);
```

The point is that the strings are never *stored*. Only those strings that the user actually requests to see are generated.

We must make one other setting. We must tell the list component that all items have a fixed width and height. The easiest way to set the cell dimensions is to specify a *prototype cell value*:

```
wordList.setPrototypeCellValue("www");
```

The prototype cell value is used to determine the size for all cells. (We use the string "www" because "w" is the widest lowercase letter in most fonts.)

Alternatively, you can set a fixed cell size:

```
wordList.setFixedCellWidth(50);
wordList.setFixedCellHeight(15);
```


If you don't set a prototype value or a fixed cell size, the list component computes the width and height of each item. That can take a long time.

As a practical matter, very long lists are rarely useful. It is extremely cumbersome for a user to scroll through a huge selection. For that reason, we believe that the list control has been completely overengineered. A selection that a user can comfortably manage on the screen is certainly small enough to be stored directly in the list component. That arrangement would have saved programmers from the pain of having to deal with the list model as a separate entity. On the other hand, the `JList` class is consistent with the `JTree` and `JTable` class where this generality is useful.

Listing 6-2. **`LongListTest.java`**

Code View:

```
import java.awt.*;

import javax.swing.*;
import javax.swing.event.*;

/**
 * This program demonstrates a list that dynamically computes list entries.
 * @version 1.23 2007-08-01
 * @author Cay Horstmann
 */
public class LongListTest
{
   public static void main(String[] args)
   {
      EventQueue.invokeLater(new Runnable()
         {
            public void run()
            {
               JFrame frame = new LongListFrame();
               frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
               frame.setVisible(true);
            }
         });
   }
}

/**
 * This frame contains a long word list and a label that shows a sentence made up from
 * the chosen word.
 */
```

```
class LongListFrame extends JFrame
{
   public LongListFrame()
   {
      setTitle("LongListTest");
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

      wordList = new JList(new WordListModel(3));
      wordList.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
      wordList.setPrototypeCellValue("www");
      JScrollPane scrollPane = new JScrollPane(wordList);

      JPanel p = new JPanel();
      p.add(scrollPane);
      wordList.addListSelectionListener(new ListSelectionListener()
         {
            public void valueChanged(ListSelectionEvent evt)
            {
               StringBuilder word = (StringBuilder) wordList.getSelectedValue();
               setSubject(word.toString());
            }

         });

      Container contentPane = getContentPane();
      contentPane.add(p, BorderLayout.NORTH);
      label = new JLabel(prefix + suffix);
      contentPane.add(label, BorderLayout.CENTER);
      setSubject("fox");
   }

   /**
    * Sets the subject in the label.
    * @param word the new subject that jumps over the lazy dog
    */
   public void setSubject(String word)
   {
      StringBuilder text = new StringBuilder(prefix);
      text.append(word);
      text.append(suffix);
      label.setText(text.toString());
   }

   private static final int DEFAULT_WIDTH = 400;
   private static final int DEFAULT_HEIGHT = 300;
   private JList wordList;
   private JLabel label;
   private String prefix = "The quick brown ";
   private String suffix = " jumps over the lazy dog.";
}

/**
 * A model that dynamically generates n-letter words.
 */
class WordListModel extends AbstractListModel
{
   /**
    * Constructs the model.
    * @param n the word length
```

```
    */
   public WordListModel(int n)
   {
      length = n;
   }

   public int getSize()
   {
      return (int) Math.pow(LAST - FIRST + 1, length);
   }

   public Object getElementAt(int n)
   {
      StringBuilder r = new StringBuilder();
      ;
      for (int i = 0; i < length; i++)
      {
         char c = (char) (FIRST + n % (LAST - FIRST + 1));
         r.insert(0, c);
         n = n / (LAST - FIRST + 1);
      }
      return r;
   }

   private int length;
   public static final char FIRST = 'a';
   public static final char LAST = 'z';
}
```


API

`javax.swing.JList` 1.2

- `JList(ListModel dataModel)`

  constructs a list that displays the elements in the specified model.

- `Object getPrototypeCellValue()`

- `void setPrototypeCellValue(Object newValue)`

  gets or sets the prototype cell value that is used to determine the width and height of each cell in the list. The default is `null`, which forces the size of each cell to be measured.

- `void setFixedCellWidth(int width)`

  if the width is greater than zero, specifies the width (in pixels) of every cell in the list. The default value is -1, which forces the size of each cell to be measured.

- `void setFixedCellHeight(int height)`

  if the height is greater than zero, specifies the height (in pixels) of every cell in the list. The default value is -1, which forces the size of each cell to be measured.

API `javax.swing.ListModel` 1.2

- `int getSize()`

  returns the number of elements of the model.

- `Object getElementAt(int position)`

  returns an element of the model at the given position.

Inserting and Removing Values

You cannot directly edit the collection of list values. Instead, you must access the *model* and then add or remove elements. That, too, is easier said than done. Suppose you want to add more values to a list. You can obtain a reference to the model:

```
ListModel model = list.getModel();
```

But that does you no good—as you saw in the preceding section, the `ListModel` interface has no methods to insert or remove elements because, after all, the whole point of having a list model is that it need not *store* the elements.

Let's try it the other way around. One of the constructors of `JList` takes a vector of objects:

```
Vector<String> values = new Vector<String>();
values.addElement("quick");
values.addElement("brown");
. . .
JList list = new JList(values);
```

You can now edit the vector and add or remove elements, but the list does not know that this is happening, so it cannot react to the changes. In particular, the list cannot update its view when you add the values. Therefore, this constructor is not very useful.

Instead, you should construct a `DefaultListModel` object, fill it with the initial values, and associate it with the list. The `DefaultListModel` class implements the `ListModel` interface and manages a collection of objects.

```
DefaultListModel model = new DefaultListModel();
model.addElement("quick");
model.addElement("brown");
. . .
JList list = new JList(model);
```

Now you can add or remove values from the `model` object. The `model` object then notifies the list of the changes, and the list repaints itself.


```
model.removeElement("quick");
model.addElement("slow");
```

For historical reasons, the `DefaultListModel` class doesn't use the same method names as the collection classes.

The default list model uses a vector internally to store the values.

Caution

There are `JList` constructors that construct a list from an array or vector of objects or strings. You might think that these constructors use a `DefaultListModel` to store these values. That is not the case—the constructors build a trivial model that can access the values without any provisions for notification if the content changes. For example, here is the code for the constructor that constructs a `JList` from a `Vector`:

```
public JList(final Vector<?> listData)
{
   this (new AbstractListModel()
   {
      public int getSize() { return listData.size(); }
      public Object getElementAt(int i) { return listData.elementAt(i); }
   });
}
```

That means, if you change the contents of the vector after the list is constructed, then the list might show a confusing mix of old and new values until it is completely repainted. (The keyword `final` in the preceding constructor does not prevent you from changing the vector elsewhere—it only means that the constructor itself won't modify the value of the `listData` reference; the keyword is required because the `listData` object is used in the inner class.)

API `javax.swing.JList` 1.2

- `ListModel getModel()`

  gets the model of this list.

API `javax.swing.DefaultListModel` 1.2

- `void addElement(Object obj)`

  adds the object to the end of the model.

- `boolean removeElement(Object obj)`

  removes the first occurrence of the object from the model. Returns `true` if the object was contained in the model, `false` otherwise.

Rendering Values

So far, all lists that you have seen in this chapter contained strings. It is actually just as easy to show a list of icons—simply pass an array or vector filled with `Icon` objects. More interestingly, you can easily represent your list values with any drawing whatsoever.

Although the `JList` class can display strings and icons automatically, you need to install a *list cell renderer* into the `JList` object for all custom drawing. A list cell renderer is any class that implements the following interface:

Code View:
```
interface ListCellRenderer
{
   Component getListCellRendererComponent(JList list, Object value, int index,
      boolean isSelected, boolean cellHasFocus);
}
```

This method is called for each cell. It returns a component that paints the cell contents. The component is placed at the appropriate location whenever a cell needs to be rendered.

One way to implement a cell renderer is to create a class that extends `JComponent`, like this:

Code View:
```
class MyCellRenderer extends JComponent implements ListCellRenderer
{
   public Component getListCellRendererComponent(JList list, Object value, int index,
      boolean isSelected, boolean cellHasFocus)
```

```
    {
       // stash away information that is needed for painting and size measurement
       return this;
    }
    public void paintComponent(Graphics g)
    {
       // paint code goes here
    }
    public Dimension getPreferredSize()
    {
       // size measurement code goes here
    }
    // instance fields
 }
```


In Listing 6-3, we display the font choices graphically by showing the actual appearance of each font (see Figure 6-4). In the `paintComponent` method, we display each name in its own font. We also need to make sure to match the usual colors of the look and feel of the `JList` class. We obtain these colors by calling the `getForeground`/`getBackground` and `getSelectionForeground`/`getSelectionBackground` methods of the `JList` class. In the `getPreferredSize` method, we need to measure the size of the string, using the techniques that you saw in Volume I, Chapter 7.

Figure 6-4. A list box with rendered cells

To install the cell renderer, simply call the `setCellRenderer` method:

```
fontList.setCellRenderer(new FontCellRenderer());
```

Now all list cells are drawn with the custom renderer.

Actually, a simpler method for writing custom renderers works in many cases. If the rendered image just contains text, an icon, and possibly a change of color, then you can get by with configuring a `JLabel`. For example, to show the font name in its own font, we can use the following renderer:

Code View:
```
class FontCellRenderer extends JLabel implements ListCellRenderer
{
   public Component getListCellRendererComponent(JList list, Object value, int index,
      boolean isSelected, boolean cellHasFocus)
   {
      JLabel label = new JLabel();
      Font font = (Font) value;
      setText(font.getFamily());
      setFont(font);
      setOpaque(true);
      setBackground(isSelected ? list.getSelectionBackground() : list.getBackground());
      setForeground(isSelected ? list.getSelectionForeground() : list.getForeground());
      return this;
   }
}
```

Note that here we don't write any `paintComponent` or `getPreferredSize` methods; the `JLabel` class already implements these methods to our satisfaction. All we do is configure the label appropriately by setting its text, font, and color.

This code is a convenient shortcut for those cases in which an existing component—in this case, `JLabel`—already provides all functionality needed to render a cell value.

We could have used a `JLabel` in our sample program, but we gave you the more general code so that you can modify it when you need to do arbitrary drawings in list cells.

Caution

It is not a good idea to construct a new component in each call to `getListCellRendererComponent`. If the user scrolls through many list entries, a new component would be constructed every time. Reconfiguring an existing component is safe and much more efficient.

Listing 6-3. **ListRenderingTest.java**

Code View:
```
import java.util.*;
```

```java
import java.awt.*;

import javax.swing.*;
import javax.swing.event.*;

/**
 * This program demonstrates the use of cell renderers in a list box.
 * @version 1.23 2007-08-01
 * @author Cay Horstmann
 */
```

```java
public class ListRenderingTest
{
   public static void main(String[] args)
   {
      EventQueue.invokeLater(new Runnable()
         {
```

```java
            public void run()
            {
               JFrame frame = new ListRenderingFrame();
               frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
               frame.setVisible(true);
            }
         });
   }
}

/**
 * This frame contains a list with a set of fonts and a text area that is set to the
 * selected font.
 */
class ListRenderingFrame extends JFrame
{
   public ListRenderingFrame()
   {
      setTitle("ListRenderingTest");
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

      ArrayList<Font> fonts = new ArrayList<Font>();
      final int SIZE = 24;
      fonts.add(new Font("Serif", Font.PLAIN, SIZE));
      fonts.add(new Font("SansSerif", Font.PLAIN, SIZE));
      fonts.add(new Font("Monospaced", Font.PLAIN, SIZE));
      fonts.add(new Font("Dialog", Font.PLAIN, SIZE));
      fonts.add(new Font("DialogInput", Font.PLAIN, SIZE));
      fontList = new JList(fonts.toArray());
      fontList.setVisibleRowCount(4);
      fontList.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
      fontList.setCellRenderer(new FontCellRenderer());
      JScrollPane scrollPane = new JScrollPane(fontList);

      JPanel p = new JPanel();
      p.add(scrollPane);
      fontList.addListSelectionListener(new ListSelectionListener()
         {
            public void valueChanged(ListSelectionEvent evt)
            {
               Font font = (Font) fontList.getSelectedValue();
               text.setFont(font);
            }
```

```
         });

      Container contentPane = getContentPane();
      contentPane.add(p, BorderLayout.SOUTH);
      text = new JTextArea("The quick brown fox jumps over the lazy dog");
      text.setFont((Font) fonts.get(0));
      text.setLineWrap(true);
      text.setWrapStyleWord(true);
      contentPane.add(text, BorderLayout.CENTER);
   }

   private JTextArea text;
   private JList fontList;
   private static final int DEFAULT_WIDTH = 400;
   private static final int DEFAULT_HEIGHT = 300;
}

/**
 * A cell renderer for Font objects that renders the font name in its own font.
 */
class FontCellRenderer extends JComponent implements ListCellRenderer
{
   public Component getListCellRendererComponent(JList list, Object value, int index,
         boolean isSelected, boolean cellHasFocus)
   {
      font = (Font) value;
      background = isSelected ? list.getSelectionBackground() : list.getBackground();
      foreground = isSelected ? list.getSelectionForeground() : list.getForeground();
      return this;
   }

   public void paintComponent(Graphics g)
   {
      String text = font.getFamily();
      FontMetrics fm = g.getFontMetrics(font);
      g.setColor(background);
      g.fillRect(0, 0, getWidth(), getHeight());
      g.setColor(foreground);
      g.setFont(font);
      g.drawString(text, 0, fm.getAscent());
   }

   public Dimension getPreferredSize()
   {
      String text = font.getFamily();
      Graphics g = getGraphics();
      FontMetrics fm = g.getFontMetrics(font);
      return new Dimension(fm.stringWidth(text), fm.getHeight());
   }

   private Font font;
   private Color background;
   private Color foreground;
}
```

API

javax.swing.JList 1.2

- Color getBackground()

  returns the background color for unselected cells.

- Color getSelectionBackground()

  returns the background color for selected cells.

- Color getForeground()

  returns the foreground color for unselected cells.

- Color getSelectionForeground()

  returns the foreground color for selected cells.

- void setCellRenderer(ListCellRenderer cellRenderer)

  sets the renderer that paints the cells in the list.

API

javax.swing.ListCellRenderer 1.2

- Component getListCellRendererComponent(JList list, Object item, int index, boolean isSelected, boolean hasFocus)

  returns a component whose paint method draws the cell contents. If the list cells do not have fixed size, that component must also implement getPreferredSize.

Parameters:	list	The list whose cell is being drawn
	item	The item to be drawn
	index	The index where the item is stored in the model
	isSelected	true if the specified cell was selected
	hasFocus	true if the specified cell has the focus

Chapter 6. Advanced Swing

- Lists

- Tables

- Trees
- Text Components
- Progress Indicators
- Component Organizers

In this chapter, we continue our discussion of the Swing user interface toolkit from Volume I. Swing is a rich toolkit, and Volume I covered only basic and commonly used components. That leaves us with three significantly more complex components for lists, tables, and trees, the exploration of which occupies a large part of this chapter. We then turn to text components and go beyond the simple text fields and text areas that you have seen in Volume I. We show you how to add validations and spinners to text fields and how you can display structured text such as HTML. Next, you will see a number of components for displaying progress of a slow activity. We finish the chapter by covering component organizers such as tabbed panes and desktop panes with internal frames.

Lists

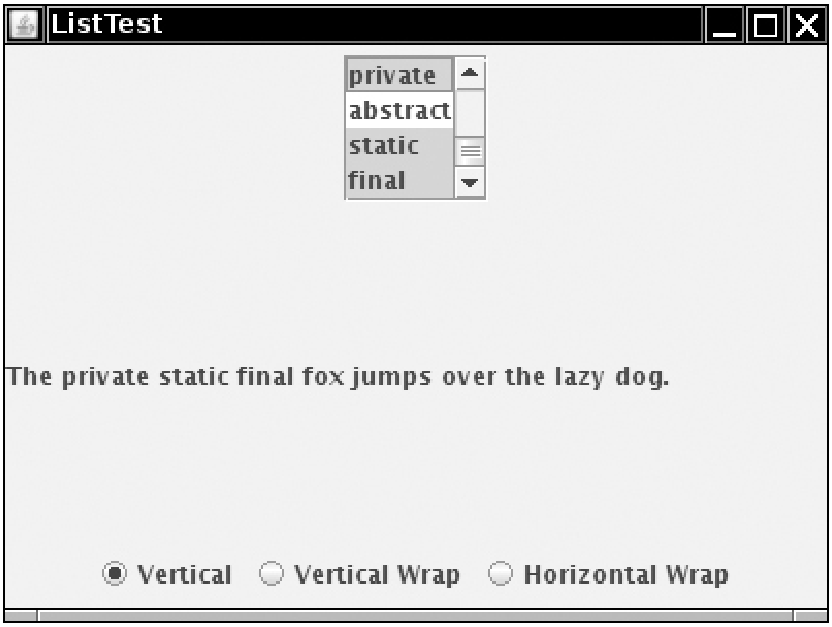
If you want to present a set of choices to a user, and a radio button or checkbox set consumes too much space, you can use a combo box or a list. Combo boxes were covered in Volume I because they are relatively simple. The `JList` component has many more features, and its design is similar to that of the tree and table components. For that reason, it is our starting point for the discussion of complex Swing components.

Of course, you can have lists of strings, but you can also have lists of arbitrary objects, with full control of how they appear. The internal architecture of the list component that makes this generality possible is rather elegant. Unfortunately, the designers at Sun felt that they needed to show off that elegance, rather than hiding it from the programmer who just wants to use the component. You will find that the list control is somewhat awkward to use for common cases because you need to manipulate some of the machinery that makes the general cases possible. We walk you through the simple and most common case, a list box of strings, and then give a more complex example that shows off the flexibility of the list component.

The **`JList`** Component

The JList component shows a number of items inside a single box. Figure 6-1 shows an admittedly silly example. The user can select the attributes for the fox, such as "quick," "brown," "hungry," "wild," and, because we ran out of attributes, "static," "private," and "final." You can thus have the *static final* fox jump over the lazy dog.

Figure 6-1. A list box



To construct this list component, you first start out with an array of strings, then pass the array to the JList constructor:

```
String[] words= { "quick", "brown", "hungry", "wild", ... };
JList wordList = new JList(words);
```

Alternatively, you can use an anonymous array:

Code View:
```
JList wordList = new JList(new String[] {"quick", "brown", "hungry", "wild", ... });
```

List boxes do not scroll automatically. To make a list box scroll, you must insert it into a scroll pane:

```
JScrollPane scrollPane = new JScrollPane(wordList);
```

You then add the scroll pane, not the list, into the surrounding panel.

We must admit that the separation of the list display and the scrolling mechanism is elegant in theory, but it is a pain in practice. Essentially all lists that we ever encountered needed scrolling. It seems cruel to force programmers to go through hoops in the default case just so they can appreciate that elegance.

By default, the list component displays eight items; use the `setVisibleRowCount` method to change that value:

```
wordList.setVisibleRowCount(4); // display 4 items
```

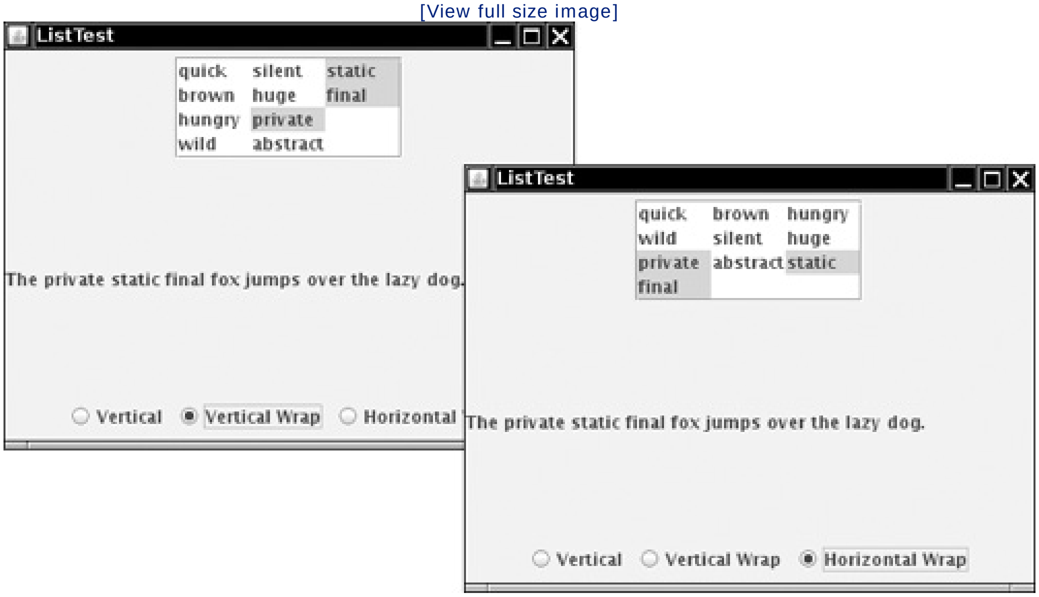
You can set the *layout orientation* to one of three values:

- `JList.VERTICAL` (the default)— Arrange all items vertically.

- `JList.VERTICAL_WRAP`— Start new columns if there are more items than the visible row count (see Figure 6-2).

Figure 6-2. Lists with vertical and horizontal wrap

[View full size image]



- `JList.HORIZONTAL_WRAP`— Start new columns if there are more items than the visible row count, but fill them horizontally. Look at the placement of the words "quick," "brown," and "hungry" in Figure 6-2 to see the difference between vertical and horizontal wrap.

By default, a user can select multiple items. To add more items to a selection, press the CTRL key while clicking on each item. To select a contiguous range of items, click on the first one, then hold down the SHIFT key and click on the last one.

You can also restrict the user to a more limited selection mode with the `setSelectionMode` method:

```
wordList.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
   // select one item at a time
wordList.setSelectionMode(ListSelectionModel.SINGLE_INTERVAL_SELECTION);
   // select one item or one range of items
```

You might recall from Volume I that the basic user interface components send out action events when the user activates them. List boxes use a different notification mechanism. Rather than listening to action events, you need to listen to list selection events. Add a list selection listener to the list component, and implement the method

```
public void valueChanged(ListSelectionEvent evt)
```

in the listener.

When the user selects items, a flurry of list selection events is generated. For example, suppose the user clicks on a new item. When the mouse button goes down, an event reports a change in selection. This is a transitional event—the call

```
event.isAdjusting()
```

returns `true` if the selection is not yet final. Then, when the mouse button goes up, there is another event, this time with `isAdjusting` returning `false`. If you are not interested in the transitional events, then you can wait for the event for which `isAdjusting` is `false`. However, if you want to give the user instant feedback as soon as the mouse button is clicked, you need to process all events.

Once you are notified that an event has happened, you will want to find out what items are currently selected. The `getSelectedValues` method returns an *array of objects* containing all selected items. Cast *each* array element to a string.

```
Object[] values = list.getSelectedValues();
for (Object value : values)
   do something with (String) value;
```

Caution

You cannot cast the return value of `getSelectedValues` from an `Object[]` array to a `String[]` array. The return value was not created as an array of strings, but as an array of objects, each of which happens to be a string. To process the return value as an array of strings, use the following code:

```
int length = values.length;
String[] words = new String[length];
System.arrayCopy(values, 0, words, 0, length);
```

If your list does not allow multiple selections, you can call the convenience method `getSelectedValue`. It returns the first selected value (which you know to be the only value if multiple selections are disallowed).

```
String value = (String) list.getSelectedValue();
```

Note

List components do not react to double clicks from a mouse. As envisioned by the designers of Swing, you use a list to select an item, and then you click a button to make something happen. However, some user interfaces allow a user to double-click on a list item as a shortcut for item selection and acceptance of a default action. If you want to implement this behavior, you have to add a mouse listener to the list box, then trap the mouse event as follows:

```
public void mouseClicked(MouseEvent evt)
{
   if (evt.getClickCount() == 2)
   {
      JList source = (JList) evt.getSource();
      Object[] selection = source.getSelectedValues();
      doAction(selection);
   }
}
```


Listing 6-1 is the listing of the program that demonstrates a list box filled with strings. Notice how the `valueChanged` method builds up the message string from the selected items.

Listing 6-1. **ListTest.java**

Code View:

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

/**
 * This program demonstrates a simple fixed list of strings.
 * @version 1.23 2007-08-01
 * @author Cay Horstmann
 */
public class ListTest
{
   public static void main(String[] args)
   {
      EventQueue.invokeLater(new Runnable()
         {
            public void run()
            {
               JFrame frame = new ListFrame();
               frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
               frame.setVisible(true);
            }
         });
   }
}

/**
 * This frame contains a word list and a label that shows a sentence made up from the chosen
 * words. Note that you can select multiple words with Ctrl+click and Shift+click.
 */
```

```
class ListFrame extends JFrame
{
   public ListFrame()
   {
      setTitle("ListTest");
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

      String[] words = { "quick", "brown", "hungry", "wild", "silent", "huge", "private",
            "abstract", "static", "final" };

      wordList = new JList(words);
      wordList.setVisibleRowCount(4);
      JScrollPane scrollPane = new JScrollPane(wordList);

      listPanel = new JPanel();
      listPanel.add(scrollPane);
      wordList.addListSelectionListener(new ListSelectionListener()
         {
            public void valueChanged(ListSelectionEvent event)
            {
               Object[] values = wordList.getSelectedValues();

               StringBuilder text = new StringBuilder(prefix);
               for (int i = 0; i < values.length; i++)
               {
                  String word = (String) values[i];
                  text.append(word);
                  text.append(" ");
               }
               text.append(suffix);

               label.setText(text.toString());
            }
         });

      buttonPanel = new JPanel();
      group = new ButtonGroup();
      makeButton("Vertical", JList.VERTICAL);
      makeButton("Vertical Wrap", JList.VERTICAL_WRAP);
      makeButton("Horizontal Wrap", JList.HORIZONTAL_WRAP);

      add(listPanel, BorderLayout.NORTH);
      label = new JLabel(prefix + suffix);
      add(label, BorderLayout.CENTER);
      add(buttonPanel, BorderLayout.SOUTH);
   }

   /**
    * Makes a radio button to set the layout orientation.
    * @param label the button label
    * @param orientation the orientation for the list
    */
   private void makeButton(String label, final int orientation)
   {
      JRadioButton button = new JRadioButton(label);
      buttonPanel.add(button);
      if (group.getButtonCount() == 0) button.setSelected(true);
      group.add(button);
      button.addActionListener(new ActionListener()
```

```
         {
            public void actionPerformed(ActionEvent event)
            {
               wordList.setLayoutOrientation(orientation);
               listPanel.revalidate();
            }
         });
   }

   private static final int DEFAULT_WIDTH = 400;
   private static final int DEFAULT_HEIGHT = 300;
   private JPanel listPanel;
   private JList wordList;
   private JLabel label;
   private JPanel buttonPanel;
   private ButtonGroup group;
   private String prefix = "The ";
   private String suffix = "fox jumps over the lazy dog.";
}
```


`javax.swing.JList` 1.2

- `JList(Object[] items)`

  constructs a list that displays these items.

- `int getVisibleRowCount()`

- `void setVisibleRowCount(int c)`

  gets or sets the preferred number of rows in the list that can be displayed without a scroll bar.

- `int getLayoutOrientation()` **1.4**

- `void setLayoutOrientation(int orientation)` **1.4**

  gets or sets the layout orientation

Parameters:	`orientation`	One of `VERTICAL`, `VERTICAL_WRAP`, `HORIZONTAL_WRAP`

- `int getSelectionMode()`

- `void setSelectionMode(int mode)`

  gets or sets the mode that determines whether single-item or multiple-item selections are allowed.

Parameters:	`mode`	One of `SINGLE_SELECTION`, `SINGLE_INTERVAL_SELECTION`, `MULTIPLE_INTERVAL_SELECTION`

- `void addListSelectionListener(ListSelectionListener listener)`

  adds to the list a listener that's notified each time a change to the selection occurs.

- `Object[] getSelectedValues()`

  returns the selected values or an empty array if the selection is empty.

- `Object getSelectedValue()`

  returns the first selected value or `null` if the selection is empty.

API `javax.swing.event.ListSelectionListener` 1.2

- `void valueChanged(ListSelectionEvent e)`

  is called whenever the list selection changes.

List Models

In the preceding section, you saw the most common method for using a list component:

1. Specify a fixed set of strings for display in the list.
2. Place the list inside a scroll pane.
3. Trap the list selection events.

In the remainder of the section on lists, we cover more complex situations that require a bit more finesse:

- Very long lists
- Lists with changing contents
- Lists that don't contain strings

In the first example, we constructed a `JList` component that held a fixed collection of strings. However, the collection of choices in a list box is not always fixed. How do we add or remove items in the list box? Somewhat surprisingly, there are no methods in the `JList` class to achieve this. Instead, you have to understand a little more about the internal design of the list component. The list component uses the model-view-controller design pattern to separate the visual appearance (a column of items that are rendered in some way) from the underlying data (a collection of objects).

The `JList` class is responsible for the visual appearance of the data. It actually knows very little about how the data are stored—all it knows is that it can retrieve the data through some object that implements the `ListModel` interface:

```
public interface ListModel
{
   int getSize();
   Object getElementAt(int i);
   void addListDataListener(ListDataListener l);
   void removeListDataListener(ListDataListener l);
}
```

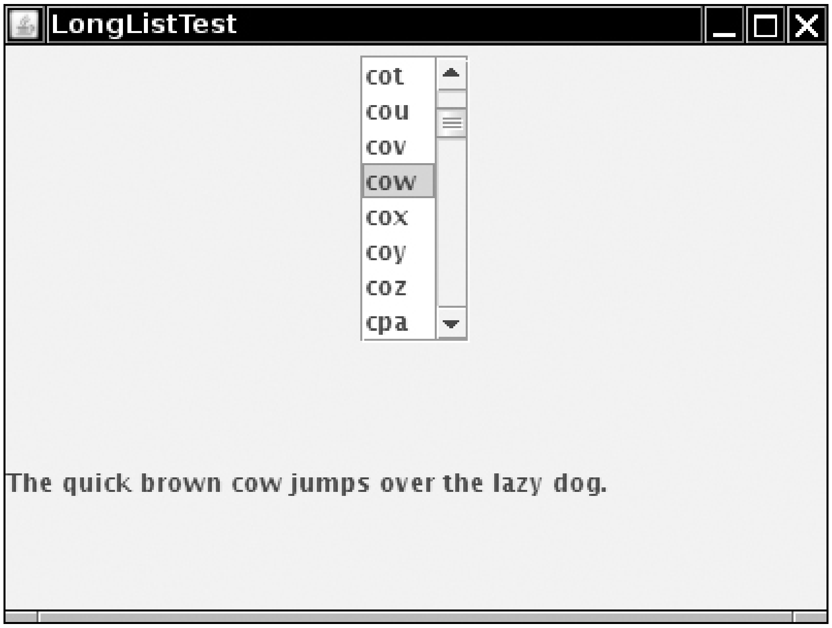
Through this interface, the `JList` can get a count of elements and retrieve each one of the elements. Also, the `JList` object can add itself as a `ListDataListener`. That way, if the collection of elements changes, the `JList` gets notified so that it can repaint itself.

Why is this generality useful? Why doesn't the `JList` object simply store an array of objects?

Note that the interface doesn't specify how the objects are stored. In particular, it doesn't force them to be stored at all! The `getElementAt` method is free to recompute each value whenever it is called. This is potentially useful if you want to show a very large collection without having to store the values.

Here is a somewhat silly example: We let the user choose among *all three-letter words* in a list box (see Figure 6-3).

Figure 6-3. Choosing from a very long list of selections



There are 26 x 26 x 26 = 17,576 three-letter combinations. Rather than storing all these combinations, we recompute them as requested when the user scrolls through them.

This turns out to be easy to implement. The tedious part, adding and removing listeners, has been done for us in the AbstractListModel class, which we extend. We only need to supply the getSize and getElementAt methods:

```
class WordListModel extends AbstractListModel
{
   public WordListModel(int n) { length = n; }
   public int getSize() { return (int) Math.pow(26, length); }
   public Object getElementAt(int n)
   {
      // compute nth string
      . . .
   }
   . . .
}
```

The computation of the *n*th string is a bit technical—you'll find the details in Listing 6-2.

Now that we have supplied a model, we can simply build a list that lets the user scroll through the elements supplied by the model:

```
JList wordList = new JList(new WordListModel(3));
wordList.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
JScrollPane scrollPane = new JScrollPane(wordList);
```

The point is that the strings are never *stored*. Only those strings that the user actually requests to see are generated.

We must make one other setting. We must tell the list component that all items have a fixed width and height. The easiest way to set the cell dimensions is to specify a *prototype cell value*:

```
wordList.setPrototypeCellValue("www");
```

The prototype cell value is used to determine the size for all cells. (We use the string "www" because "w" is the widest lowercase letter in most fonts.)

Alternatively, you can set a fixed cell size:

```
wordList.setFixedCellWidth(50);
wordList.setFixedCellHeight(15);
```


If you don't set a prototype value or a fixed cell size, the list component computes the width and height of each item. That can take a long time.

As a practical matter, very long lists are rarely useful. It is extremely cumbersome for a user to scroll through a huge selection. For that reason, we believe that the list control has been completely overengineered. A selection that a user can comfortably manage on the screen is certainly small enough to be stored directly in the list component. That arrangement would have saved programmers from the pain of having to deal with the list model as a separate entity. On the other hand, the `JList` class is consistent with the `JTree` and `JTable` class where this generality is useful.

Listing 6-2. **`LongListTest.java`**

Code View:

```
import java.awt.*;

import javax.swing.*;
import javax.swing.event.*;

/**
 * This program demonstrates a list that dynamically computes list entries.
 * @version 1.23 2007-08-01
 * @author Cay Horstmann
 */
public class LongListTest
{
   public static void main(String[] args)
   {
      EventQueue.invokeLater(new Runnable()
         {
            public void run()
            {
               JFrame frame = new LongListFrame();
               frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
               frame.setVisible(true);
            }
         });
   }
}

/**
 * This frame contains a long word list and a label that shows a sentence made up from
 * the chosen word.
 */
```

```
class LongListFrame extends JFrame
{
   public LongListFrame()
   {
      setTitle("LongListTest");
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

      wordList = new JList(new WordListModel(3));
      wordList.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
      wordList.setPrototypeCellValue("www");
      JScrollPane scrollPane = new JScrollPane(wordList);

      JPanel p = new JPanel();
      p.add(scrollPane);
      wordList.addListSelectionListener(new ListSelectionListener()
         {
            public void valueChanged(ListSelectionEvent evt)
            {
               StringBuilder word = (StringBuilder) wordList.getSelectedValue();
               setSubject(word.toString());
            }

         });

      Container contentPane = getContentPane();
      contentPane.add(p, BorderLayout.NORTH);
      label = new JLabel(prefix + suffix);
      contentPane.add(label, BorderLayout.CENTER);
      setSubject("fox");
   }

   /**
    * Sets the subject in the label.
    * @param word the new subject that jumps over the lazy dog
    */
   public void setSubject(String word)
   {
      StringBuilder text = new StringBuilder(prefix);
      text.append(word);
      text.append(suffix);
      label.setText(text.toString());
   }

   private static final int DEFAULT_WIDTH = 400;
   private static final int DEFAULT_HEIGHT = 300;
   private JList wordList;
   private JLabel label;
   private String prefix = "The quick brown ";
   private String suffix = " jumps over the lazy dog.";
}

/**
 * A model that dynamically generates n-letter words.
 */
class WordListModel extends AbstractListModel
{
   /**
    * Constructs the model.
    * @param n the word length
```

```
    */
   public WordListModel(int n)
   {
      length = n;
   }

   public int getSize()
   {
      return (int) Math.pow(LAST - FIRST + 1, length);
```



```
   }

   public Object getElementAt(int n)
   {
      StringBuilder r = new StringBuilder();
      ;
      for (int i = 0; i < length; i++)
```



```
      {
         char c = (char) (FIRST + n % (LAST - FIRST + 1));
         r.insert(0, c);
         n = n / (LAST - FIRST + 1);
      }
      return r;
   }

   private int length;
   public static final char FIRST = 'a';
   public static final char LAST = 'z';
}
```

API

`javax.swing.JList` 1.2

- `JList(ListModel dataModel)`

  constructs a list that displays the elements in the specified model.

- `Object getPrototypeCellValue()`

- `void setPrototypeCellValue(Object newValue)`

  gets or sets the prototype cell value that is used to determine the width and height of each cell in the list. The default is `null`, which forces the size of each cell to be measured.

- `void setFixedCellWidth(int width)`

  if the width is greater than zero, specifies the width (in pixels) of every cell in the list. The default value is -1, which forces the size of each cell to be measured.

- `void setFixedCellHeight(int height)`

  if the height is greater than zero, specifies the height (in pixels) of every cell in the list. The default value is -1, which forces the size of each cell to be measured.

API `javax.swing.ListModel` 1.2

- `int getSize()`

  returns the number of elements of the model.

- `Object getElementAt(int position)`

  returns an element of the model at the given position.

Inserting and Removing Values

You cannot directly edit the collection of list values. Instead, you must access the *model* and then add or remove elements. That, too, is easier said than done. Suppose you want to add more values to a list. You can obtain a reference to the model:

```
ListModel model = list.getModel();
```

But that does you no good—as you saw in the preceding section, the `ListModel` interface has no methods to insert or remove elements because, after all, the whole point of having a list model is that it need not *store* the elements.

Let's try it the other way around. One of the constructors of `JList` takes a vector of objects:

```
Vector<String> values = new Vector<String>();
values.addElement("quick");
values.addElement("brown");
. . .
JList list = new JList(values);
```

You can now edit the vector and add or remove elements, but the list does not know that this is happening, so it cannot react to the changes. In particular, the list cannot update its view when you add the values. Therefore, this constructor is not very useful.

Instead, you should construct a `DefaultListModel` object, fill it with the initial values, and associate it with the list. The `DefaultListModel` class implements the `ListModel` interface and manages a collection of objects.

```
DefaultListModel model = new DefaultListModel();
model.addElement("quick");
model.addElement("brown");
. . .
JList list = new JList(model);
```

Now you can add or remove values from the `model` object. The `model` object then notifies the list of the changes, and the list repaints itself.


```
model.removeElement("quick");
model.addElement("slow");
```

For historical reasons, the `DefaultListModel` class doesn't use the same method names as the collection classes.

The default list model uses a vector internally to store the values.

Caution

There are `JList` constructors that construct a list from an array or vector of objects or strings. You might think that these constructors use a `DefaultListModel` to store these values. That is not the case—the constructors build a trivial model that can access the values without any provisions for notification if the content changes. For example, here is the code for the constructor that constructs a `JList` from a `Vector`:

```
public JList(final Vector<?> listData)
{
   this (new AbstractListModel()
   {
      public int getSize() { return listData.size(); }
      public Object getElementAt(int i) { return listData.elementAt(i); }
   });
}
```

That means, if you change the contents of the vector after the list is constructed, then the list might show a confusing mix of old and new values until it is completely repainted. (The keyword `final` in the preceding constructor does not prevent you from changing the vector elsewhere—it only means that the constructor itself won't modify the value of the `listData` reference; the keyword is required because the `listData` object is used in the inner class.)

javax.swing.JList 1.2

- `ListModel getModel()`

  gets the model of this list.

javax.swing.DefaultListModel 1.2

- `void addElement(Object obj)`

  adds the object to the end of the model.

- `boolean removeElement(Object obj)`

  removes the first occurrence of the object from the model. Returns `true` if the object was contained in the model, `false` otherwise.

Rendering Values

So far, all lists that you have seen in this chapter contained strings. It is actually just as easy to show a list of icons—simply pass an array or vector filled with `Icon` objects. More interestingly, you can easily represent your list values with any drawing whatsoever.

Although the `JList` class can display strings and icons automatically, you need to install a *list cell renderer* into the `JList` object for all custom drawing. A list cell renderer is any class that implements the following interface:

Code View:
```
interface ListCellRenderer
{
   Component getListCellRendererComponent(JList list, Object value, int index,
      boolean isSelected, boolean cellHasFocus);
}
```

This method is called for each cell. It returns a component that paints the cell contents. The component is placed at the appropriate location whenever a cell needs to be rendered.

One way to implement a cell renderer is to create a class that extends `JComponent`, like this:

Code View:
```
class MyCellRenderer extends JComponent implements ListCellRenderer
{
   public Component getListCellRendererComponent(JList list, Object value, int index,
      boolean isSelected, boolean cellHasFocus)
```

```
   {
      // stash away information that is needed for painting and size measurement
      return this;
   }
   public void paintComponent(Graphics g)
   {
      // paint code goes here
   }
   public Dimension getPreferredSize()
```



```
   {
      // size measurement code goes here
   }
   // instance fields
}
```


In Listing 6-3, we display the font choices graphically by showing the actual appearance of each font (see Figure 6-4). In the `paintComponent` method, we display each name in its own font. We also need to make sure to match the usual colors of the look and feel of the `JList` class. We obtain these colors by calling the `getForeground`/`getBackground` and `getSelectionForeground`/`getSelectionBackground` methods of the `JList` class. In the `getPreferredSize` method, we need to measure the size of the string, using the techniques that you saw in Volume I, Chapter 7.

Figure 6-4. A list box with rendered cells

To install the cell renderer, simply call the `setCellRenderer` method:

```
fontList.setCellRenderer(new FontCellRenderer());
```

Now all list cells are drawn with the custom renderer.

Actually, a simpler method for writing custom renderers works in many cases. If the rendered image just contains text, an icon, and possibly a change of color, then you can get by with configuring a `JLabel`. For example, to show the font name in its own font, we can use the following renderer:

Code View:
```
class FontCellRenderer extends JLabel implements ListCellRenderer
{
   public Component getListCellRendererComponent(JList list, Object value, int index,
      boolean isSelected, boolean cellHasFocus)
   {
      JLabel label = new JLabel();
      Font font = (Font) value;
      setText(font.getFamily());
      setFont(font);
      setOpaque(true);
      setBackground(isSelected ? list.getSelectionBackground() : list.getBackground());
      setForeground(isSelected ? list.getSelectionForeground() : list.getForeground());
      return this;
   }
}
```

Note that here we don't write any `paintComponent` or `getPreferredSize` methods; the `JLabel` class already implements these methods to our satisfaction. All we do is configure the label appropriately by setting its text, font, and color.

This code is a convenient shortcut for those cases in which an existing component—in this case, `JLabel`—already provides all functionality needed to render a cell value.

We could have used a `JLabel` in our sample program, but we gave you the more general code so that you can modify it when you need to do arbitrary drawings in list cells.

Caution

It is not a good idea to construct a new component in each call to `getListCellRendererComponent`. If the user scrolls through many list entries, a new component would be constructed every time. Reconfiguring an existing component is safe and much more efficient.

Listing 6-3. **ListRenderingTest.java**

Code View:
```
import java.util.*;
```

```
import java.awt.*;

import javax.swing.*;
import javax.swing.event.*;

/**
 * This program demonstrates the use of cell renderers in a list box.
 * @version 1.23 2007-08-01
 * @author Cay Horstmann
```

```
 */
public class ListRenderingTest
{
   public static void main(String[] args)
   {
      EventQueue.invokeLater(new Runnable()
```

```
         {
            public void run()
            {
               JFrame frame = new ListRenderingFrame();
               frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
               frame.setVisible(true);
            }
         });
   }
}

/**
 * This frame contains a list with a set of fonts and a text area that is set to the
 * selected font.
 */
class ListRenderingFrame extends JFrame
{
   public ListRenderingFrame()
   {
      setTitle("ListRenderingTest");
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

      ArrayList<Font> fonts = new ArrayList<Font>();
      final int SIZE = 24;
      fonts.add(new Font("Serif", Font.PLAIN, SIZE));
      fonts.add(new Font("SansSerif", Font.PLAIN, SIZE));
      fonts.add(new Font("Monospaced", Font.PLAIN, SIZE));
      fonts.add(new Font("Dialog", Font.PLAIN, SIZE));
      fonts.add(new Font("DialogInput", Font.PLAIN, SIZE));
      fontList = new JList(fonts.toArray());
      fontList.setVisibleRowCount(4);
      fontList.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
      fontList.setCellRenderer(new FontCellRenderer());
      JScrollPane scrollPane = new JScrollPane(fontList);

      JPanel p = new JPanel();
      p.add(scrollPane);
      fontList.addListSelectionListener(new ListSelectionListener()
         {
            public void valueChanged(ListSelectionEvent evt)
            {
               Font font = (Font) fontList.getSelectedValue();
               text.setFont(font);
            }
```

```
         });

      Container contentPane = getContentPane();
      contentPane.add(p, BorderLayout.SOUTH);
      text = new JTextArea("The quick brown fox jumps over the lazy dog");
      text.setFont((Font) fonts.get(0));
      text.setLineWrap(true);
      text.setWrapStyleWord(true);
      contentPane.add(text, BorderLayout.CENTER);
   }

   private JTextArea text;
   private JList fontList;
   private static final int DEFAULT_WIDTH = 400;
   private static final int DEFAULT_HEIGHT = 300;
}

/**
 * A cell renderer for Font objects that renders the font name in its own font.
 */
class FontCellRenderer extends JComponent implements ListCellRenderer
{
   public Component getListCellRendererComponent(JList list, Object value, int index,
         boolean isSelected, boolean cellHasFocus)
   {
      font = (Font) value;
      background = isSelected ? list.getSelectionBackground() : list.getBackground();
      foreground = isSelected ? list.getSelectionForeground() : list.getForeground();
      return this;
   }

   public void paintComponent(Graphics g)
   {
      String text = font.getFamily();
      FontMetrics fm = g.getFontMetrics(font);
      g.setColor(background);
      g.fillRect(0, 0, getWidth(), getHeight());
      g.setColor(foreground);
      g.setFont(font);
      g.drawString(text, 0, fm.getAscent());
   }

   public Dimension getPreferredSize()
   {
      String text = font.getFamily();
      Graphics g = getGraphics();
      FontMetrics fm = g.getFontMetrics(font);
      return new Dimension(fm.stringWidth(text), fm.getHeight());
   }

   private Font font;
   private Color background;
   private Color foreground;
}
```

javax.swing.JList 1.2

- `Color getBackground()`

  returns the background color for unselected cells.

- `Color getSelectionBackground()`

  returns the background color for selected cells.

- `Color getForeground()`

  returns the foreground color for unselected cells.

- `Color getSelectionForeground()`

  returns the foreground color for selected cells.

- `void setCellRenderer(ListCellRenderer cellRenderer)`

  sets the renderer that paints the cells in the list.

javax.swing.ListCellRenderer 1.2

- `Component getListCellRendererComponent(JList list, Object item, int index, boolean isSelected, boolean hasFocus)`

  returns a component whose `paint` method draws the cell contents. If the list cells do not have fixed size, that component must also implement `getPreferredSize`.

Parameters:	`list`	The list whose cell is being drawn
	`item`	The item to be drawn
	`index`	The index where the item is stored in the model
	`isSelected`	`true` if the specified cell was selected
	`hasFocus`	`true` if the specified cell has the focus

Tables

The `JTable` component displays a two-dimensional grid of objects. Of course, tables are common in user interfaces. The Swing team has put a lot of effort into the table control. Tables are inherently complex, but—perhaps more successfully than with other Swing classes—the `JTable` component hides much of that complexity. You can produce fully functional tables with rich behavior by writing a few lines of code. Of course, you can write more code and customize the display and behavior for your specific applications.

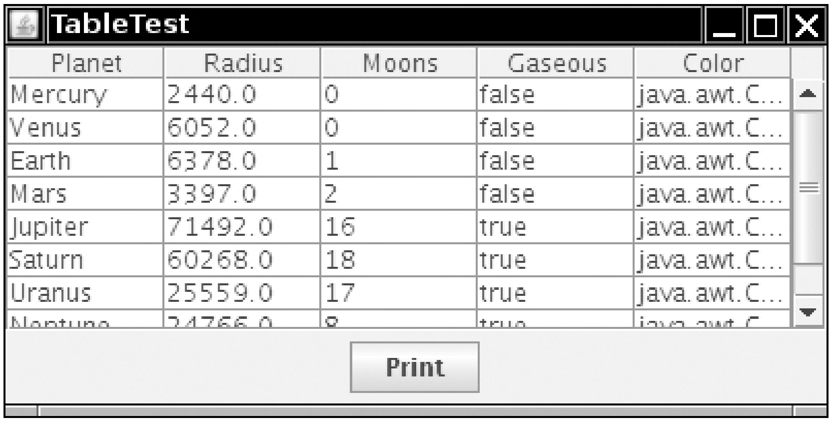
In this section, we explain how to make simple tables, how the user interacts with them, and how to make some of the most common adjustments. As with the other complex Swing controls, it is impossible to cover all aspects in complete detail. For more information, look in *Graphic Java 2: Mastering the JFC, Volume II: Swing*, 3rd ed., by David M. Geary (Prentice Hall PTR 1999) or *Core Java Foundation Classes* by Kim Topley (Prentice Hall 1998).

A Simple Table

Similar to the `JList` component, a `JTable` does not store its own data but obtains its data from a *table model*. The `JTable` class has a constructor that wraps a two-dimensional array of objects into a default model. That is the strategy that we use in our first example. Later in this chapter, we turn to table models.

Figure 6-5 shows a typical table, describing properties of the planets of the solar system. (A planet is *gaseous* if it consists mostly of hydrogen and helium. You should take the "Color" entries with a grain of salt—that column was added because it will be useful in later code examples.)

Figure 6-5. A simple table



As you can see from the code in Listing 6-4, the data of the table is stored as a two-dimensional array of `Object` values:

```
Object[][] cells =
{
   { "Mercury", 2440.0, 0, false, Color.YELLOW },
   { "Venus", 6052.0, 0, false, Color.YELLOW },
   . . .
}
```

Note

Here, we take advantage of autoboxing. The entries in the second, third, and fourth columns are automatically converted into objects of type `Double`, `Integer`, and `Boolean`.

The table simply invokes the `toString` method on each object to display it. That's why the colors show up as `java.awt.Color[r=...,g=...,b=...]`.

You supply the column names in a separate array of strings:

```
String[] columnNames = { "Planet", "Radius", "Moons", "Gaseous", "Color" };
```

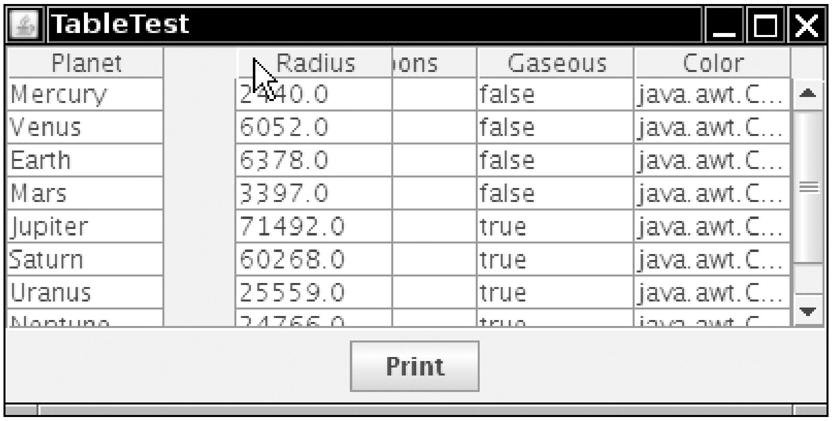
Then, you construct a table from the cell and column name arrays. Finally, add scroll bars in the usual way, by wrapping the table in a `JScrollPane`.

```
JTable table = new JTable(cells, columnNames);
JScrollPane pane = new JScrollPane(table);
```

The resulting table already has surprisingly rich behavior. Resize the table vertically until the scroll bar shows up. Then, scroll the table. Note that the column headers don't scroll out of view!

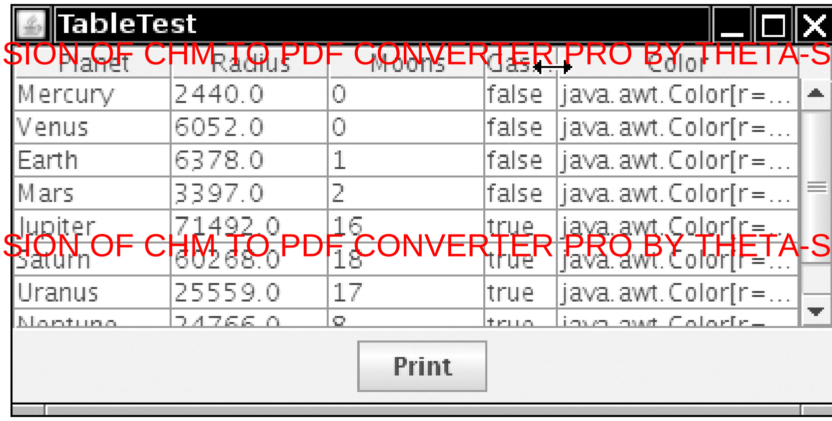
Next, click on one of the column headers and drag it to the left or right. See how the entire column becomes detached (see Figure 6-6). You can drop it to a different location. This rearranges the columns *in the view only*. The data model is not affected.

Figure 6-6. Moving a column



To *resize* columns, simply place the cursor between two columns until the cursor shape changes to an arrow. Then, drag the column boundary to the desired place (see Figure 6-7).

Figure 6-7. Resizing columns



Users can select rows by clicking anywhere in a row. The selected rows are highlighted; you will see later how to get selection events. Users can also edit the table entries by clicking on a cell and typing into it. However, in this code example, the edits do not change the underlying data. In your programs, you should either make cells uneditable or handle cell editing events and update your model. We discuss those topics later in this section.

Finally, click on a column header. The rows are automatically sorted. Click again, and the sort order is reversed. This behavior is activated by the call

```
table.setAutoCreateRowSorter(true);
```

You can print a table with the call

```
table.print();
```

A print dialog box appears, and the table is sent to the printer. We discuss custom printing options in Chapter 7.

Note

If you resize the `TableTest` frame so that its height is taller than the table height, you will see a gray area below the table. Unlike `JList` and `JTree` components, the table does not fill the scroll pane's viewport. This can be a problem if you want to support drag and drop. (For more information on drag and drop, see Chapter 7.) In that case, call

```
table.setFillsViewportHeight(true);
```

Listing 6-4. **PlanetTable.java**

Code View:
```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

/**
 * This program demonstrates how to show a simple table
 * @version 1.11 2007-08-01
 * @author Cay Horstmann
 */
public class PlanetTable
{
   public static void main(String[] args)
   {
      EventQueue.invokeLater(new Runnable()
         {
            public void run()
            {
               JFrame frame = new PlanetTableFrame();
               frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
               frame.setVisible(true);
            }
         });
   }
}

/**
 * This frame contains a table of planet data.
 */
class PlanetTableFrame extends JFrame
{
   public PlanetTableFrame()
   {
      setTitle("PlanetTable");
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
      final JTable table = new JTable(cells, columnNames);
      table.setAutoCreateRowSorter(true);
      add(new JScrollPane(table), BorderLayout.CENTER);
      JButton printButton = new JButton("Print");
      printButton.addActionListener(new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               try
               {
                  table.print();
               }
               catch (java.awt.print.PrinterException e)
               {
                  e.printStackTrace();
               }
            }
         });
      JPanel buttonPanel = new JPanel();
      buttonPanel.add(printButton);
      add(buttonPanel, BorderLayout.SOUTH);
   }
```

```
    private Object[][] cells = { { "Mercury", 2440.0, 0, false, Color.yellow },
        { "Venus", 6052.0, 0, false, Color.yellow }, { "Earth", 6378.0, 1, false, Color.blue },
        { "Mars", 3397.0, 2, false, Color.red }, { "Jupiter", 71492.0, 16, true, Color.orange },
        { "Saturn", 60268.0, 18, true, Color.orange },
        { "Uranus", 25559.0, 17, true, Color.blue }, { "Neptune", 24766.0, 8, true, Color.blue },
        { "Pluto", 1137.0, 1, false, Color.black } };

    private String[] columnNames = { "Planet", "Radius", "Moons", "Gaseous", "Color" };

    private static final int DEFAULT_WIDTH = 400;
    private static final int DEFAULT_HEIGHT = 200;
}
```


API **javax.swing.JTable 1.2**

- `JTable(Object[][] entries, Object[] columnNames)`

  constructs a table with a default table model.

- `void print()` **5.0**

  displays a print dialog box and prints the table.

- `boolean getAutoCreateRowSorter()` **6**

- `void setAutoCreateRowSorter(boolean newValue)` **6**

  gets or sets the `autoCreateRowSorter` property. The default is `false`. When set, a default row sorter is automatically set whenever the model changes.

- `boolean getFillsViewportHeight()` **6**

- `void setFillsViewportHeight(boolean newValue)` **6**

  gets or sets the `fillsViewportHeight` property. The default is `false`. When set, the table always fills an enclosing viewport.

Table Models

In the preceding example, the table data were stored in a two-dimensional array. However, you should generally not use that strategy in your own code. If you find yourself dumping data into an array to display it as a table, you should instead think about implementing your own table model.

Table models are particularly simple to implement because you can take advantage of the `AbstractTableModel` class that implements most of the required methods. You only need to supply three methods:

```
public int getRowCount();
public int getColumnCount();
public Object getValueAt(int row, int column);
```

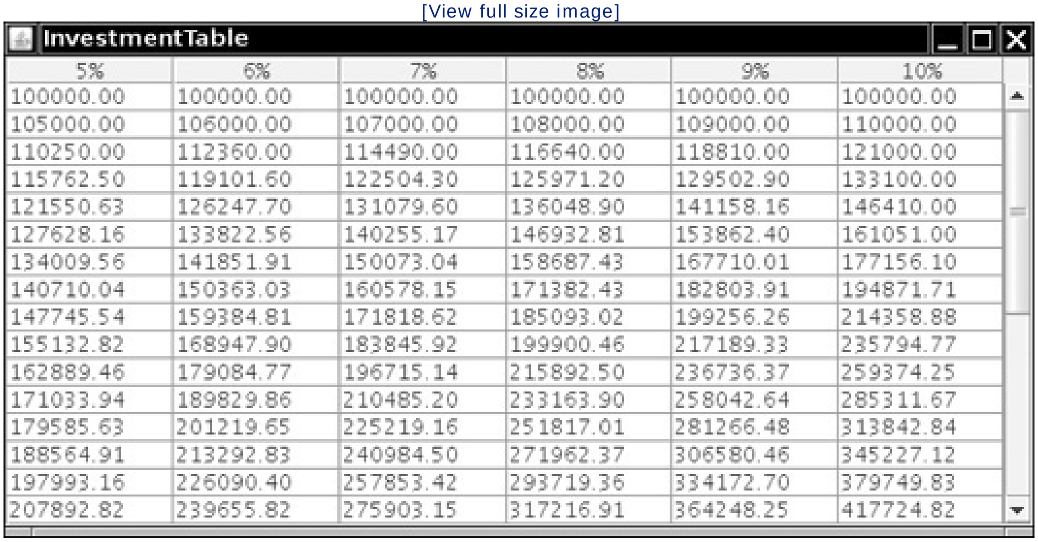
There are many ways of implementing the `getValueAt` method. For example, if you want to display the contents of a `RowSet` that contains the result of a database query, you simply provide this method:

```
   public Object getValueAt(int r, int c)
   {
      try
      {
         rowSet.absolute(r + 1);
         return rowSet.getObject(c + 1);
      }
      catch (SQLException e)
      {
         e.printStackTrace();
         return null;
      }
   }
```

Our sample program is even simpler. We construct a table that shows some computed values, namely, the growth of an investment under different interest rate scenarios (see Figure 6-8).

Figure 6-8. Growth of an investment

[View full size image]



5%	6%	7%	8%	9%	10%
100000.00	100000.00	100000.00	100000.00	100000.00	100000.00
105000.00	106000.00	107000.00	108000.00	109000.00	110000.00
110250.00	112360.00	114490.00	116640.00	118810.00	121000.00
115762.50	119101.60	122504.30	125971.20	129502.90	133100.00
121550.63	126247.70	131079.60	136048.90	141158.16	146410.00
127628.16	133822.56	140255.17	146932.81	153862.40	161051.00
134009.56	141851.91	150073.04	158687.43	167710.01	177156.10
140710.04	150363.03	160578.15	171382.43	182803.91	194871.71
147745.54	159384.81	171818.62	185093.02	199256.26	214358.88
155132.82	168947.90	183845.92	199900.46	217189.33	235794.77
162889.46	179084.77	196715.14	215892.50	236736.37	259374.25
171033.94	189829.86	210485.20	233163.90	258042.64	285311.67
179585.63	201219.65	225219.16	251817.01	281266.48	313842.84
188564.91	213292.83	240984.50	271962.37	306580.46	345227.12
197993.16	226090.40	257853.42	293719.36	334172.70	379749.83
207892.82	239655.82	275903.15	317216.91	364248.25	417724.82

The getValueAt method computes the appropriate value and formats it:

```
public Object getValueAt(int r, int c)
{
   double rate = (c + minRate) / 100.0;
   int nperiods = r;
   double futureBalance = INITIAL_BALANCE * Math.pow(1 + rate, nperiods);
   return String.format("%.2f", futureBalance);
}
```


The getRowCount and getColumnCount methods simply return the number of rows and columns.

```
public int getRowCount() { return years; }
public int getColumnCount() {  return maxRate - minRate + 1; }
```


If you don't supply column names, the getColumnName method of the AbstractTableModel names the columns A, B, C, and so on. To change column names, override the getColumnName method. You will usually want to override that default behavior. In this example, we simply label each column with the interest rate.

```
public String getColumnName(int c) { return (c + minRate) + "%"; }
```

You can find the complete source code in Listing 6-5.

Listing 6-5. **InvestmentTable.java**

Code View:

```
import java.awt.*;

import javax.swing.*;
import javax.swing.table.*;

/**
 * This program shows how to build a table from a table model.
 * @version 1.02 2007-08-01
 * @author Cay Horstmann
 */
public class InvestmentTable
{
   public static void main(String[] args)
   {
      EventQueue.invokeLater(new Runnable()
         {
            public void run()
            {
               JFrame frame = new InvestmentTableFrame();
               frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
               frame.setVisible(true);
            }
         });
   }
}

/**
 * This frame contains the investment table.
```

```
 */
class InvestmentTableFrame extends JFrame
{
   public InvestmentTableFrame()
   {
      setTitle("InvestmentTable");
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

      TableModel model = new InvestmentTableModel(30, 5, 10);
      JTable table = new JTable(model);
      add(new JScrollPane(table));
   }

   private static final int DEFAULT_WIDTH = 600;
   private static final int DEFAULT_HEIGHT = 300;
}

/**
 * This table model computes the cell entries each time they are requested. The table contents
 * shows the growth of an investment for a number of years under different interest rates.
 */
class InvestmentTableModel extends AbstractTableModel
{
   /**
    * Constructs an investment table model.
    * @param y the number of years
    * @param r1 the lowest interest rate to tabulate
    * @param r2 the highest interest rate to tabulate
    */
   public InvestmentTableModel(int y, int r1, int r2)
   {
      years = y;
      minRate = r1;
      maxRate = r2;
   }

   public int getRowCount()
   {
      return years;
   }

   public int getColumnCount()
   {
      return maxRate - minRate + 1;
   }

   public Object getValueAt(int r, int c)
   {
      double rate = (c + minRate) / 100.0;
      int nperiods = r;
      double futureBalance = INITIAL_BALANCE * Math.pow(1 + rate, nperiods);
      return String.format("%.2f", futureBalance);
   }

   public String getColumnName(int c)
   {
      return (c + minRate) + "%";
   }

```

```
   private int years;
   private int minRate;
   private int maxRate;

   private static double INITIAL_BALANCE = 100000.0;
}
```


API `javax.swing.table.TableModel` 1.2

- `int getRowCount()`
- `int getColumnCount()`

  gets the number of rows and columns in the table model.

- `Object getValueAt(int row, int column)`

  gets the value at the given row and column.

- `void setValueAt(Object newValue, int row, int column)`

  sets a new value at the given row and column.

- `boolean isCellEditable(int row, int column)`

  returns `true` if the cell at the given row and column is editable.

- `String getColumnName(int column)`

  gets the column title.

Working with Rows and Columns

In this subsection, you will see how to manipulate the rows and columns in a table. As you read through this material, keep in mind that a Swing table is quite asymmetric—there are different operations that you can carry out on rows and columns. The table component was optimized to display rows of information with the same structure, such as the result of a database query, not an arbitrary two-dimensional grid of objects. You will see this asymmetry throughout this subsection.

Column Classes

In the next example, we again display our planet data, but this time, we want to give the table more information about the column types. This is achieved by defining the method

```
Class<?> getColumnClass(int columnIndex)
```

of the table model to return the class that describes the column type.

The `JTable` class uses this information to pick an appropriate renderer for the class. Table 6-1 shows the default rendering actions.

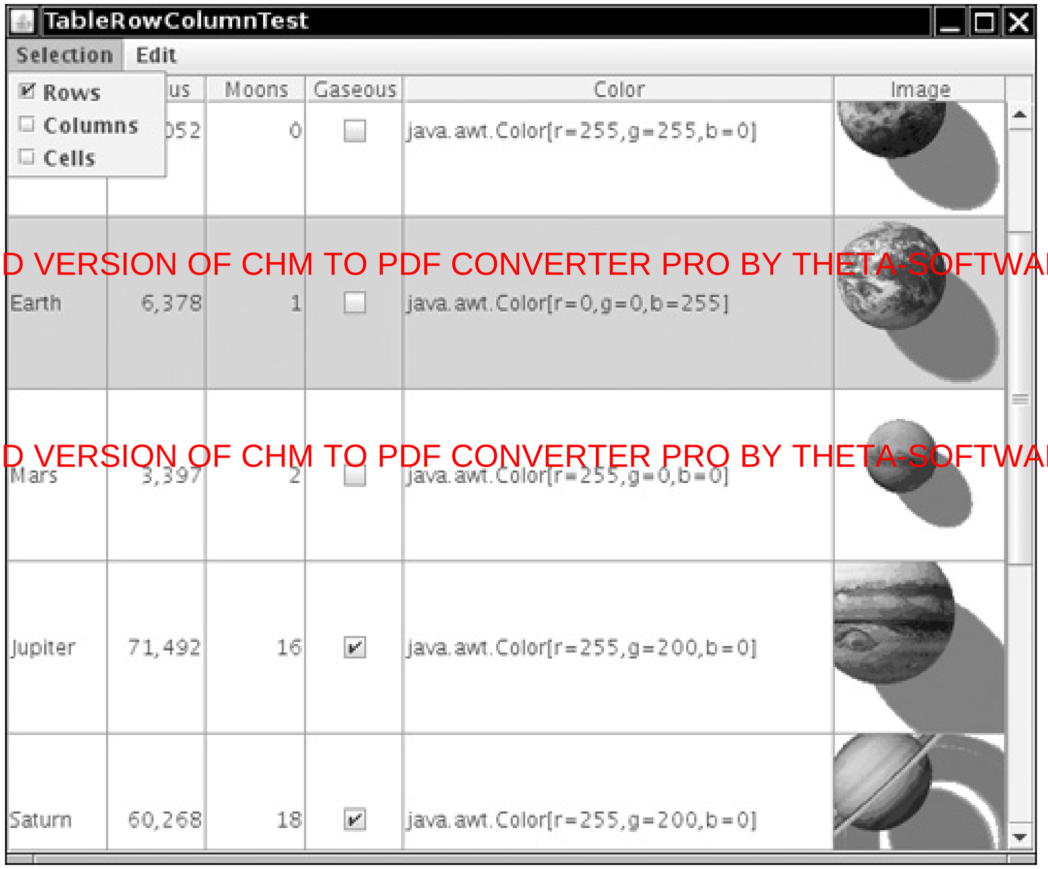
Table 6-1. Default Rendering Actions

Type	Rendered As
`Boolean`	Checkbox
`Icon`	Image
`Object`	String

You can see the checkboxes and images in Figure 6-9. (Thanks to Jim Evins, http://www.snaught.com/JimsCoolIcons/Planets, for providing the planet images!)

Figure 6-9. A table with planet data

[View full size image]



UNREGISTERED VERSION OF CHM TO PDF CONVERTER PRO BY THETA-SOFTWARE

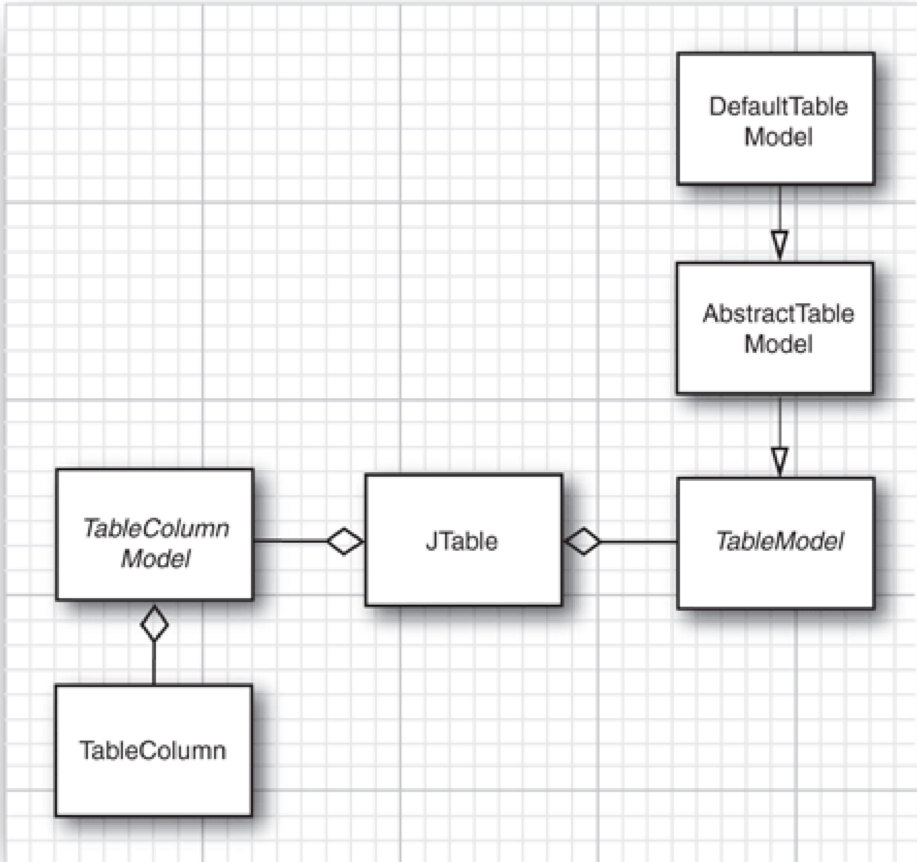
To render other types, you can install a custom renderer—see the "Cell Rendering and Editing" section beginning on page 392.

Accessing Table Columns

The JTable class stores information about table columns in objects of type TableColumn. A TableColumnModel object manages the columns. (Figure 6-10 shows the relationships among the most important table classes.) If you don't want to insert or remove columns dynamically, you won't use the column model much. The most common use for the column model is simply to get a TableColumn object:

```
int columnIndex = . . .;
TableColumn column = table.getColumnModel().getColumn(columnIndex);
```

Figure 6-10. Relationship between table classes



Resizing Columns

The `TableColumn` class gives you control over the resizing behavior of columns. You can set the preferred, minimum, and maximum width with the methods

```
void setPreferredWidth(int width)
void setMinWidth(int width)
void setMaxWidth(int width)
```

This information is used by the table component to lay out the columns.

Use the method

```
void setResizable(boolean resizable)
```

to control whether the user is allowed to resize the column.

You can programmatically resize a column with the method

```
void setWidth(int width)
```

When a column is resized, the default is to leave the total size of the table unchanged. Of course, the width increase or decrease of the resized column must then be distributed over other columns. The default behavior is to change the size of all columns to the right of the resized column. That's a good default because it allows a user to adjust all columns to a desired width, moving from left to right.

You can set another behavior from Table 6-2 by using the method


```
void setAutoResizeMode(int mode)
```

of the JTable class.

Table 6-2. Resize Modes

Mode	Behavior
AUTO_RESIZE_OFF	Don't resize other columns; change the table size.
AUTO_RESIZE_NEXT_COLUMN	Resize the next column only.
AUTO_RESIZE_SUBSEQUENT_COLUMNS	Resize all subsequent columns equally; this is the default behavior.
AUTO_RESIZE_LAST_COLUMN	Resize the last column only.
AUTO_RESIZE_ALL_COLUMNS	Resize all columns in the table; this is not a good choice because it prevents the user from adjusting multiple columns to a desired size.

Resizing Rows

Row heights are managed directly by the JTable class. If your cells are taller than the default, you want to set the row height:

```
table.setRowHeight(height);
```

By default, all rows of the table have the same height. You can set the heights of individual rows with the call

```
table.setRowHeight(row, height);
```

The actual row height equals the row height that has been set with these methods, reduced by the row margin. The default row margin is 1 pixel, but you can change it with the call

```
table.setRowMargin(margin);
```

Selecting Rows, Columns, and Cells

Depending on the selection mode, the user can select rows, columns, or individual cells in the table. By default, row selection is enabled. Clicking inside a cell selects the entire row (see Figure 6-9 on page 379). Call

```
table.setRowSelectionAllowed(false)
```

to disable row selection.

When row selection is enabled, you can control whether the user is allowed to select a single row, a contiguous set of rows, or any set of rows. You need to retrieve the *selection model* and use its setSelectionMode method:

```
table.getSelectionModel().setSelectionMode(mode);
```

Here, mode is one of the three values:

```
ListSelectionModel.SINGLE_SELECTION
ListSelectionModel.SINGLE_INTERVAL_SELECTION
ListSelectionModel.MULTIPLE_INTERVAL_SELECTION
```

Column selection is disabled by default. You turn it on with the call

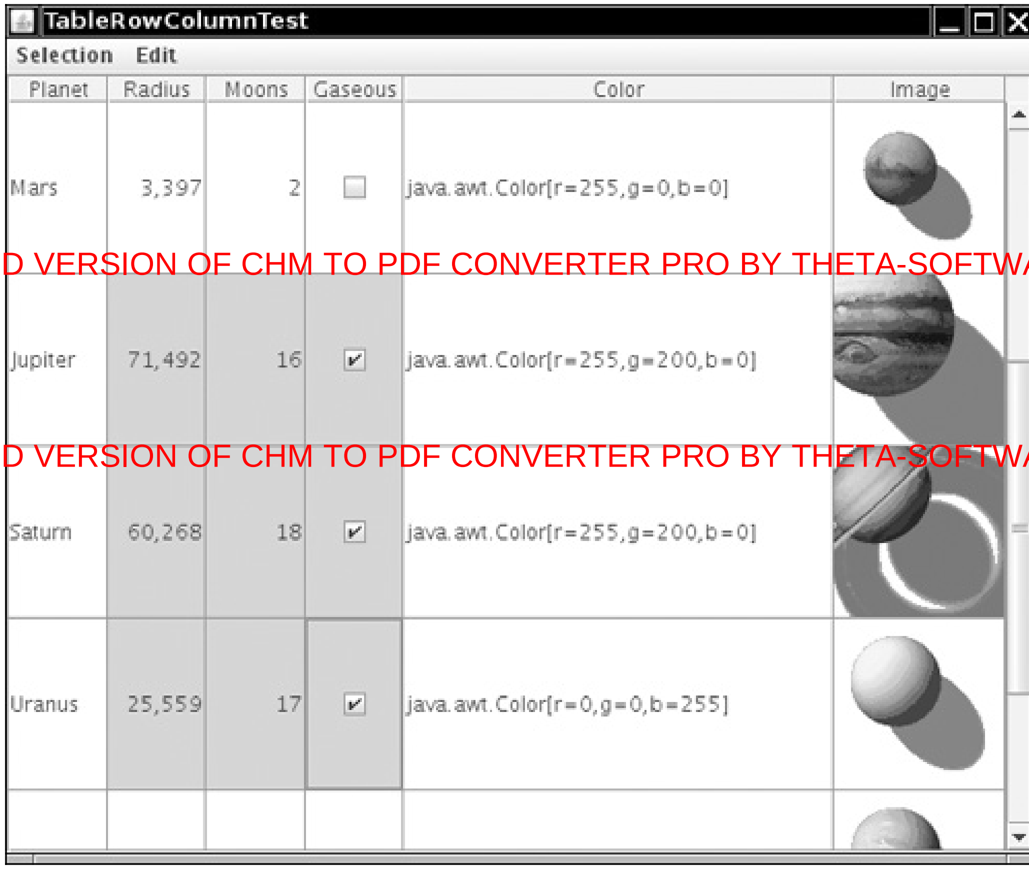
```
table.setColumnSelectionAllowed(true)
```

Enabling both row and column selection is equivalent to enabling cell selection. The user then selects ranges of cells (see Figure 6-11). You can also enable that setting with the call

```
table.setCellSelectionEnabled(true)
```

Figure 6-11. Selecting a range of cells

[View full size image]



Run the program in Listing 6-6 to watch cell selection in action. Enable row, column, or cell selection in the Selection menu and watch how the selection behavior changes.

You can find out which rows and columns are selected by calling the `getSelectedRows` and `getSelectedColumns` methods. Both return an `int[]` array of the indexes of the selected items. Note that the index values are those of the table view, not the underlying table model. Try selecting rows and columns, then drag columns to different places and sort the rows by clicking on column headers. Use the Print Selection menu item to see which rows and columns are reported as selected.

If you need to translate table index values to table model index values, use the `JTable` methods `convertRowIndexToModel` and `convertColumnIndexToModel`.

Sorting Rows

As you have seen in our first table example, it is easy to add row sorting to a `JTable`, simply by calling the `setAutoCreateRowSorter` method. However, to have finer-grained control over the sorting behavior, you install a `TableRowSorter<M>` object into a `JTable` and customize it. The type parameter `M` denotes the table model; it needs to be a subtype of the `TableModel` interface.

```
TableRowSorter<TableModel> sorter = new TableRowSorter<TableModel>(model);
table.setRowSorter(sorter);
```

Some columns should not be sorted, such as the image column in our planet data. Turn sorting off by calling

```
sorter.setSortable(IMAGE_COLUMN, false);
```

You can install a custom comparator for each column. In our example, we will sort the colors in the Color column so that we prefer blue and green over red. When you click on the Color column, you will see that the blue planets go to the bottom of the table. This is achieved with the following call:

```
sorter.setComparator(COLOR_COLUMN, new Comparator<Color>()
   {
      public int compare(Color c1, Color c2)
      {
         int d = c1.getBlue() - c2.getBlue();
         if (d != 0) return d;
         d = c1.getGreen() - c2.getGreen();
         if (d != 0) return d;
         return c1.getRed() - c2.getRed();
      }
   });
```

If you do not specify a comparator for a column, the sort order is determined as follows:

1. If the column class is `String`, use the default collator returned by `Collator.getInstance()`. It sorts strings in a way that is appropriate for the current locale. (See Chapter 5 for more information about locales and collators.)

2. If the column class implements `Comparable`, use its `compareTo` method.

3. If a `TableStringConverter` has been set for the comparator, sort the strings returned by the converter's `toString` method with the default collator. If you want to use this approach, define a converter as follows:

   ```
   sorter.setStringConverter(new TableStringConverter()
      {
         public String toString(TableModel model, int row, int column)
         {
            Object value = model.getValueAt(row, column);
            convert value to a string and return it
         }
      });
   ```

4. Otherwise, call the `toString` method on the cell values and sort them with the default collator.

Filtering Rows

In addition to sorting rows, the `TableRowSorter` can also selectively hide rows, a process called *filtering*. To activate filtering, set a `RowFilter`. For example, to include all rows that contain at least one moon, call

Code View:

```
sorter.setRowFilter(RowFilter.numberFilter(ComparisonType.NOT_EQUAL, 0, MOONS_COLUMN));
```

Here, we use a predefined filter, the number filter. To construct a number filter, supply

- The comparison type (one of EQUAL, NOT_EQUAL, AFTER, or BEFORE).
- An object of a subclass of Number (such as an Integer or Double). Only objects that have the same class as the given Number object are considered.

- Zero or more column index values. If no index values are supplied, all columns are searched.

The static RowFilter.dateFilter method constructs a date filter in the same way. You supply a Date object instead of the Number object.

Finally, the static RowFilter.regexFilter method constructs a filter that looks for strings matching a regular expression. For example,

```
sorter.setRowFilter(RowFilter.regexFilter(".*[^s]$", PLANET_COLUMN));
```

only displays those planets with a name that doesn't end with an "s". (See Chapter 1 for more information on regular expressions.)

You can also combine filters with the andFilter, orFilter, and notFilter methods. To filter for planets not ending in an "s" with at least one moon, you can use this filter combination:

```
sorter.setRowFilter(RowFilter.andFilter(Arrays.asList(
   RowFilter.regexFilter(".*[^s]$", PLANET_COLUMN),
   RowFilter.numberFilter(ComparisonType.NOT_EQUAL, 0, MOONS_COLUMN)));
```

Caution

Annoyingly, the andFilter and orFilter methods don't use variable arguments but a single parameter of type Iterable.

To implement your own filter, you provide a subclass of RowFilter and implement an include method to indicate which rows should be displayed. This is easy to do, but the glorious generality of the RowFilter class makes it a bit scary.

The RowFilter<M, I> class has two type parameters: the types for the model and for the row identifier. When dealing with tables, the model is always a subtype of TableModel and the identifier type is Integer. (At some point in the future, other components might also support row filtering. For example, to filter rows in a JTree, one might use a RowFilter<TreeModel, TreePath>.)

A row filter must implement the method

```
public boolean include(RowFilter.Entry<? extends M, ? extends I> entry)
```

The `RowFilter.Entry` class supplies methods to obtain the model, the row identifier, and the value at a given index. Therefore, you can filter both by row identifier and by the contents of the row.

For example, this filter displays every other row:

Code View:
```
RowFilter<TableModel, Integer> filter = new RowFilter<TableModel, Integer>()
   {
      public boolean include(Entry<? extends TableModel, ? extends Integer> entry)
      {
         return entry.getIdentifier() % 2 == 0;
      }
   };
```

If you wanted to include only those planets with an even number of moons, you would instead test for

```
((Integer) entry.getValue(MOONS_COLUMN)) % 2 == 0
```

In our sample program, we allow the user to hide arbitrary rows. We store the hidden row indexes in a set. The row filter includes all rows whose index is not in that set.

The filtering mechanism wasn't designed for filters with criteria that change over time. In our sample program, we keep calling

```
sorter.setRowFilter(filter);
```

whenever the set of hidden rows changes. Setting a filter causes it to be applied immediately.

Hiding and Displaying Columns

As you saw in the preceding section, you can filter table rows by either their contents or their row identifier. Hiding table columns uses a completely different mechanism.

The `removeColumn` method of the `JTable` class removes a column from the table view. The column data are not actually removed from the model—they are just hidden from view. The `removeColumn` method takes a `TableColumn` argument. If you have the column number (for example, from a call to `getSelectedColumns`), you need to ask the table model for the actual table column object:

```
TableColumnModel columnModel = table.getColumnModel();
TableColumn column = columnModel.getColumn(i);
table.removeColumn(column);
```

If you remember the column, you can later add it back in:

```
table.addColumn(column);
```

This method adds the column to the end. If you want it to appear elsewhere, you call the `moveColumn` method.

You can also add a new column that corresponds to a column index in the table model, by adding a new `TableColumn` object:

```
table.addColumn(new TableColumn(modelColumnIndex));
```

You can have multiple table columns that view the same column of the model.

The program in Listing 6-6 demonstrates selection and filtering of rows and columns.

Listing 6-6. **TableSelectionTest.java**

Code View:


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

/**
 * This program demonstrates selection, addition, and removal of rows and columns.
 * @version 1.03 2007-08-01
 * @author Cay Horstmann
 */
public class TableSelectionTest
{
   public static void main(String[] args)
   {
      EventQueue.invokeLater(new Runnable()
         {
            public void run()
            {
               JFrame frame = new TableSelectionFrame();
               frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
               frame.setVisible(true);
            }
         });
   }
}

/**
 * This frame shows a multiplication table and has menus for setting the row/column/cell
 * selection modes, and for adding and removing rows and columns.
 */
class TableSelectionFrame extends JFrame
{
   public TableSelectionFrame()
   {
      setTitle("TableSelectionTest");
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

      // set up multiplication table

      model = new DefaultTableModel(10, 10);

      for (int i = 0; i < model.getRowCount(); i++)
         for (int j = 0; j < model.getColumnCount(); j++)
            model.setValueAt((i + 1) * (j + 1), i, j);
```

```java

      table = new JTable(model);

      add(new JScrollPane(table), "Center");

      removedColumns = new ArrayList<TableColumn>();

      // create menu

      JMenuBar menuBar = new JMenuBar();
      setJMenuBar(menuBar);

      JMenu selectionMenu = new JMenu("Selection");
      menuBar.add(selectionMenu);

      final JCheckBoxMenuItem rowsItem = new JCheckBoxMenuItem("Rows");
      final JCheckBoxMenuItem columnsItem = new JCheckBoxMenuItem("Columns");
      final JCheckBoxMenuItem cellsItem = new JCheckBoxMenuItem("Cells");

      rowsItem.setSelected(table.getRowSelectionAllowed());
      columnsItem.setSelected(table.getColumnSelectionAllowed());
      cellsItem.setSelected(table.getCellSelectionEnabled());

      rowsItem.addActionListener(new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               table.clearSelection();
               table.setRowSelectionAllowed(rowsItem.isSelected());
               cellsItem.setSelected(table.getCellSelectionEnabled());
            }
         });
      selectionMenu.add(rowsItem);

      columnsItem.addActionListener(new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               table.clearSelection();
               table.setColumnSelectionAllowed(columnsItem.isSelected());
               cellsItem.setSelected(table.getCellSelectionEnabled());
            }
         });
      selectionMenu.add(columnsItem);

      cellsItem.addActionListener(new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               table.clearSelection();
               table.setCellSelectionEnabled(cellsItem.isSelected());
               rowsItem.setSelected(table.getRowSelectionAllowed());
               columnsItem.setSelected(table.getColumnSelectionAllowed());
            }
         });
      selectionMenu.add(cellsItem);

      JMenu tableMenu = new JMenu("Edit");
      menuBar.add(tableMenu);
```

```java

      JMenuItem hideColumnsItem = new JMenuItem("Hide Columns");
      hideColumnsItem.addActionListener(new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               int[] selected = table.getSelectedColumns();
               TableColumnModel columnModel = table.getColumnModel();

```



```java
               // remove columns from view, starting at the last
               // index so that column numbers aren't affected

               for (int i = selected.length - 1; i >= 0; i--)
               {
                  TableColumn column = columnModel.getColumn(selected[i]);
                  table.removeColumn(column);
```

UNREGISTERED VERSION OF CHM TO PDF CONVERTER PRO BY THETA-SOFTWARE

```java

                  // store removed columns for "show columns" command

                  removedColumns.add(column);
               }
            }
         });
      tableMenu.add(hideColumnsItem);

      JMenuItem showColumnsItem = new JMenuItem("Show Columns");
      showColumnsItem.addActionListener(new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               // restore all removed columns
               for (TableColumn tc : removedColumns)
                  table.addColumn(tc);
               removedColumns.clear();
            }
         });
      tableMenu.add(showColumnsItem);

      JMenuItem addRowItem = new JMenuItem("Add Row");
      addRowItem.addActionListener(new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               // add a new row to the multiplication table in
               // the model

               Integer[] newCells = new Integer[model.getColumnCount()];
               for (int i = 0; i < newCells.length; i++)
                  newCells[i] = (i + 1) * (model.getRowCount() + 1);
               model.addRow(newCells);
            }
         });
      tableMenu.add(addRowItem);

      JMenuItem removeRowsItem = new JMenuItem("Remove Rows");
      removeRowsItem.addActionListener(new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
```

```
               int[] selected = table.getSelectedRows();

               for (int i = selected.length - 1; i >= 0; i--)
                  model.removeRow(selected[i]);
            }
         });
      tableMenu.add(removeRowsItem);

      JMenuItem clearCellsItem = new JMenuItem("Clear Cells");
      clearCellsItem.addActionListener(new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               for (int i = 0; i < table.getRowCount(); i++)
                  for (int j = 0; j < table.getColumnCount(); j++)
                     if (table.isCellSelected(i, j)) table.setValueAt(0, i, j);
            }
         });
      tableMenu.add(clearCellsItem);
   }

   private DefaultTableModel model;
   private JTable table;
   private ArrayList<TableColumn> removedColumns;

   private static final int DEFAULT_WIDTH = 400;
   private static final int DEFAULT_HEIGHT = 300;
}
```

API `javax.swing.table.TableModel` 1.2

- `Class getColumnClass(int columnIndex)`

  gets the class for the values in this column. This information is used for sorting and rendering.

API `javax.swing.JTable` 1.2

- `TableColumnModel getColumnModel()`

  gets the "column model" that describes the arrangement of the table columns.

- `void setAutoResizeMode(int mode)`

sets the mode for automatic resizing of table columns.

Parameters:	`mode`	One of `AUTO_RESIZE_OFF`, `AUTO_RESIZE_NEXT_COLUMN`, `AUTO_RESIZE_SUBSEQUENT_COLUMNS`, `AUTO_RESIZE_LAST_COLUMN`, `AUTO_RESIZE_ALL_COLUMNS`

- `int getRowMargin()`
- `void setRowMargin(int margin)`

  gets or sets the amount of empty space between cells in adjacent rows.

- `int getRowHeight()`
- `void setRowHeight(int height)`

  gets or sets the default height of all rows of the table.

- `int getRowHeight(int row)`
- `void setRowHeight(int row, int height)`

  gets or sets the height of the given row of the table.

- `ListSelectionModel getSelectionModel()`

  returns the list selection model. You need that model to choose between row, column, and cell selection.

- `boolean getRowSelectionAllowed()`
- `void setRowSelectionAllowed(boolean b)`

  gets or sets the `rowSelectionAllowed` property. If `true`, then rows are selected when the user clicks cells.

- `boolean getColumnSelectionAllowed()`
- `void setColumnSelectionAllowed(boolean b)`

  gets or sets the `columnSelectionAllowed` property. If `true`, then columns are selected when the user clicks on cells.

- `boolean getCellSelectionEnabled()`

  returns `true` if both `rowSelectionAllowed` and `columnSelectionAllowed` are `true`.

- `void setCellSelectionEnabled(boolean b)`

  sets both `rowSelectionAllowed` and `columnSelectionAllowed` to `b`.

- `void addColumn(TableColumn column)`

  adds a column as the last column of the table view.

- `void moveColumn(int from, int to)`

  moves the column whose table index is `from` so that its index becomes `to`. Only the view is affected.

- `void removeColumn(TableColumn column)`

  removes the given column from the view.

- `int convertRowIndexToModel(int index)` **6**

- `int convertColumnIndexToModel(int index)`

  returns the model index of the row or column with the given index. This value is different from `index` when rows are sorted or filtered, or when columns are moved or removed.

- `void setRowSorter(RowSorter<? extends TableModel> sorter)`

  sets the row sorter.

API

`javax.swing.table.TableColumnModel` 1.2

- `TableColumn getColumn(int index)`

  gets the table column object that describes the column with the given view index.

API

`javax.swing.table.TableColumn` 1.2

- `TableColumn(int modelColumnIndex)`

  constructs a table column for viewing the model column with the given index.

- `void setPreferredWidth(int width)`

- `void setMinWidth(int width)`

- `void setMaxWidth(int width)`

  sets the preferred, minimum, and maximum width of this table column to `width`.

- `void setWidth(int width)`

  sets the actual width of this column to `width`.

- `void setResizable(boolean b)`

  If `b` is `true`, this column is resizable.

javax.swing.ListSelectionModel 1.2

- `void setSelectionMode(int mode)`

  Parameters: `mode` One of `SINGLE_SELECTION`, `SINGLE_INTERVAL_SELECTION`, and `MULTIPLE_INTERVAL_SELECTION`

javax.swing.DefaultRowSorter<M, I> 6

- `void setComparator(int column, Comparator<?> comparator)`

  sets the comparator to be used with the given column.

- `void setSortable(int column, boolean enabled)`

  enables or disables sorting for the given column.

- `void setRowFilter(RowFilter<? super M,? super I> filter)`

  sets the row filter.

javax.swing.table.TableRowSorter<M extends TableModel> 6

- `void setStringConverter(TableStringConverter stringConverter)`

  sets the string converter that is used for sorting and filtering.

API javax.swing.table.TableStringConverter<M extends TableModel> 6

- abstract String toString(TableModel model, int row, int column)

  override this method to convert the model value at the given location to a string.

API javax.swing.RowFilter<M, I> 6

- boolean include(RowFilter.Entry<? extends M,? extends I> entry)

  override this method to specify the rows that are retained.

- static <M,I> RowFilter<M,I> numberFilter(RowFilter.ComparisonType type, Number number, int... indices)

- static <M,I> RowFilter<M,I> dateFilter(RowFilter.ComparisonType type, Date date, int... indices)

  returns a filter that includes rows containing values that match the given comparison to the given number or date. The comparison type is one of EQUAL, NOT_EQUAL, AFTER, or BEFORE. If any column model indexes are given, then only those columns are searched. Otherwise, all columns are searched. For the number filter, the class of the cell value must match the class of number.

- static <M,I> RowFilter<M,I> regexFilter(String regex, int... indices)

  returns a filter that includes rows that have a string value matching the given regular expression. If any column model indexes are given, then only those columns are searched. Otherwise, all columns are searched. Note that the string returned by the getStringValue method of RowFilter.Entry is matched.

- static <M,I> RowFilter<M,I> andFilter(Iterable<? extends RowFilter<? super M,? super I>> filters)

- static <M,I> RowFilter<M,I> orFilter(Iterable<? extends RowFilter<? super M,? super I>> filters)

  returns a filter that includes the entries that are included by all filters or at least one of the filters.

- static <M,I> RowFilter<M,I> notFilter(RowFilter<M,I> filter)

  returns a filter that includes the entries that are not included by the given filter.

API javax.swing.RowFilter.Entry<M, I> 6

- I getIdentifier()

  returns the identifier of this row entry.

- M getModel()

  returns the model of this row entry.

- Object getValue(int index)

  returns the value stored at the given index of this row.

- int getValueCount()

  returns the number of values stored in this row.

- String getStringValue()

  returns the value stored at the given index of this row, converted to a string. The TableRowSorter produces entries whose getStringValue calls the sorter's string converter.

Cell Rendering and Editing

As you saw in the "Accessing Table Columns" section beginning on page 379, the column type determines how the cells are rendered. There are default renderers for the types Boolean and Icon that render a checkbox or icon. For all other types, you need to install a custom renderer.

Table cell renderers are similar to the list cell renderers that you saw earlier. They implement the TableCellRenderer interface, which has a single method:

Code View:

```
Component getTableCellRendererComponent(JTable table, Object value, boolean isSelected,
   boolean hasFocus, int row, int column)
```

That method is called when the table needs to draw a cell. You return a component whose `paint` method is then invokes to fill the cell area.

The table in Figure 6-12 contains cells of type `Color`. The renderer simply returns a panel with a background color that is the color object stored in the cell. The color is passed as the `value` parameter.

Code View:


```
class ColorTableCellRenderer extends JPanel implements TableCellRenderer
{
   public Component getTableCellRendererComponent(JTable table, Object value,
                                                  boolean isSelected,
      boolean hasFocus, int row, int column)
   {
      setBackground((Color) value);
      if (hasFocus)
         setBorder(UIManager.getBorder("Table.focusCellHighlightBorder"));
      else
         setBorder(null);
   }
   return this;
}
```

Figure 6-12. A table with cell renderers

[View full size image]

As you can see, the renderer installs a border when the cell has focus. (We ask the `UIManager` for the correct border. To find the lookup key, we peeked into the source code of the `DefaultTableCellRenderer` class.)

Generally, you will also want to set the background color of the cell to indicate whether it is currently selected. We skip this step because it would interfere with the displayed color. The `ListRenderingTest` example in Listing 6-3 shows how to indicate the selection status in a renderer.

Tip

If your renderer simply draws a text string or an icon, you can extend the `DefaultTableCellRenderer` class. It takes care of rendering the focus and selection status for you.

You need to tell the table to use this renderer with all objects of type `Color`. The `setDefaultRenderer` method of the `JTable` class lets you establish this association. You supply a `Class` object and the renderer:

```
table.setDefaultRenderer(Color.class, new ColorTableCellRenderer());
```

That renderer is now used for all objects of the given type in this table.

If you want to select a renderer based on some other criterion, you need to subclass the `JTable` class and override the `getCellRenderer` method.

Rendering the Header

To display an icon in the header, set the header value:

```
moonColumn.setHeaderValue(new ImageIcon("Moons.gif"));
```

However, the table header isn't smart enough to choose an appropriate renderer for the header value. You have to install the renderer manually. For example, to show an image icon in a column header, call


```
moonColumn.setHeaderRenderer(table.getDefaultRenderer(ImageIcon.class));
```

Cell Editing

To enable cell editing, the table model must indicate which cells are editable by defining the `isCellEditable` method. Most commonly, you will want to make certain columns editable. In the example program, we allow editing in four columns.

Code View:
```
public boolean isCellEditable(int r, int c)
{
   return c == PLANET_COLUMN || c == MOONS_COLUMN || c == GASEOUS_COLUMN || c == COLOR_COLUMN;
}
```

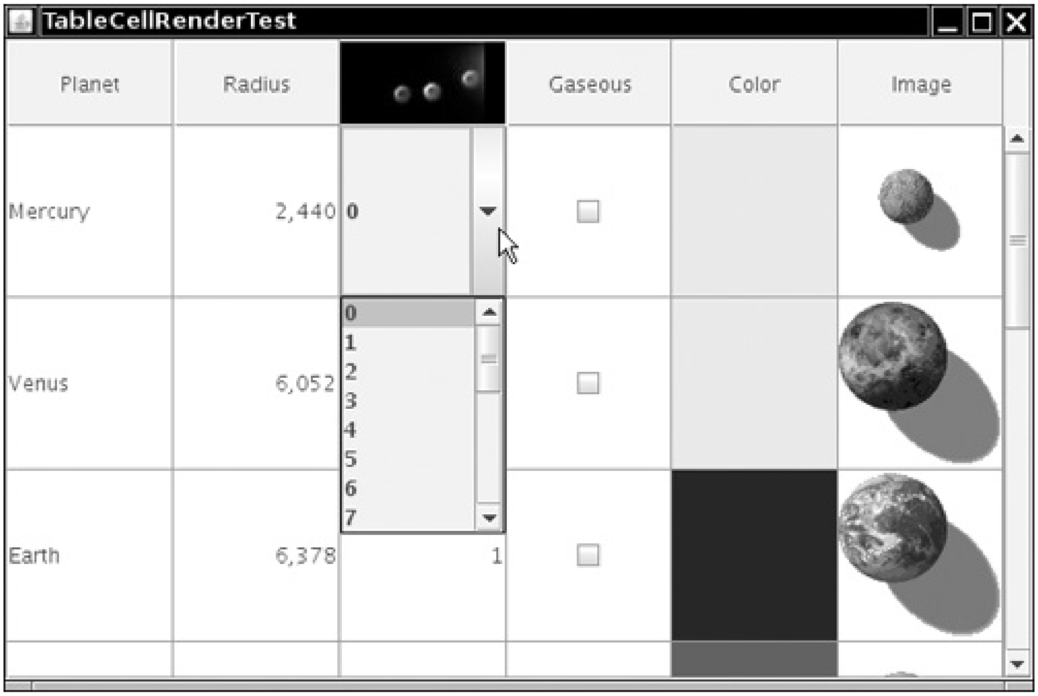
Note

The `AbstractTableModel` defines the `isCellEditable` method to always return `false`. The `DefaultTableModel` overrides the method to always return `true`.

If you run the program in Listing 6-7, note that you can click the checkboxes in the Gaseous column and turn the check marks on and off. If you click a cell in the Moons column, a combo box appears (see Figure 6-13). You will shortly see how to install such a combo box as a cell editor.

Figure 6-13. A cell editor

[View full size image]



Finally, click a cell in the first column. The cell gains focus. You can start typing and the cell contents change.

What you just saw in action are the three variations of the `DefaultCellEditor` class. A `DefaultCellEditor` can be constructed with a `JTextField`, a `JCheckBox`, or a `JComboBox`. The `JTable` class automatically installs a checkbox editor for `Boolean` cells and a text field editor for all editable cells that don't supply their own renderer. The text fields let the user edit the strings that result from applying `toString` to the return value of the `getValueAt` method of the table model.

When the edit is complete, the edited value is retrieved by calling the `getCellEditorValue` method of your editor. That method should return a value of the correct type (that is, the type returned by the `getColumnType` method of the model).

To get a combo box editor, you set a cell editor manually—the `JTable` component has no idea what values might be appropriate for a particular type. For the Moons column, we wanted to enable the user to pick any value between 0 and 20. Here is the code for initializing the combo box:

```
JComboBox moonCombo = new JComboBox();
for (int i = 0; i <= 20; i++)
   moonCombo.addItem(i);
```

To construct a `DefaultCellEditor`, supply the combo box in the constructor:

```
TableCellEditor moonEditor = new DefaultCellEditor(moonCombo);
```

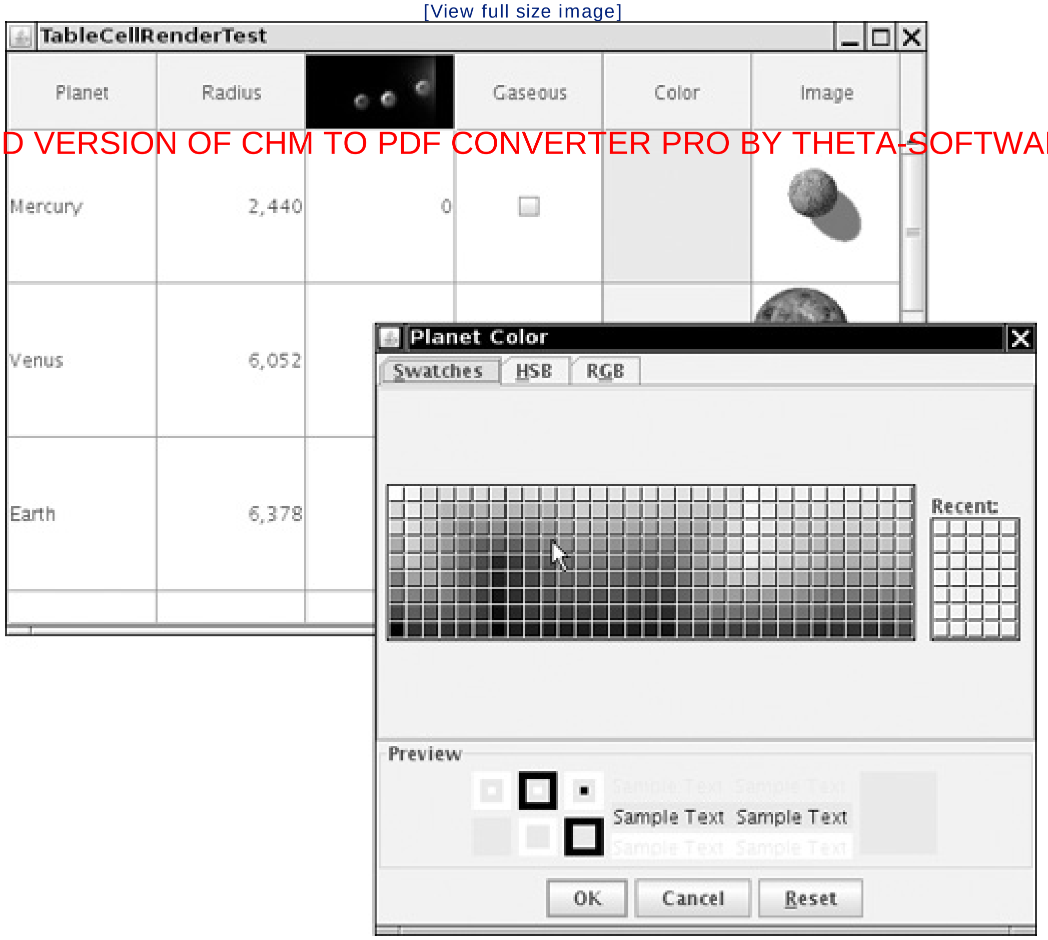
Next, we need to install the editor. Unlike the color cell renderer, this editor does not depend on the object *type*—we don't necessarily want to use it for all objects of type `Integer`. Instead, we need to install it into a particular column:

```
moonColumn.setCellEditor(moonEditor);
```

Run the example program again and click a color. A *color chooser* pops up to let you pick a new color for the planet. Select a color and click OK. The cell color is updated (see Figure 6-14).

Figure 6-14. Editing the cell color with a color chooser

[View full size image]



UNREGISTERED VERSION OF CHM TO PDF CONVERTER PRO BY THETA-SOFTWARE

The color cell editor is not a standard table cell editor but a custom implementation. To create a custom cell editor, you implement the `TableCellEditor` interface. That interface is a bit tedious, and as of Java SE 1.3, an `AbstractCellEditor` class is provided to take care of the event handling details.

The `getTableCellEditorComponent` method of the `TableCellEditor` interface requests a component to render the cell. It is exactly the same as the `getTableCellRendererComponent` method of the `TableCellRenderer` interface, except that there is no `focus` parameter. Because the cell is being edited, it is presumed to have focus. The editor component temporarily *replaces* the renderer when the editing is in progress. In our example, we return a blank panel that is not colored. This is an indication to the user that the cell is currently being edited.

Next, you want to have your editor pop up when the user clicks on the cell.

The `JTable` class calls your editor with an event (such as a mouse click) to find out if that event is acceptable to initiate the editing process. The AbstractCellEditor class defines the method to accept all events.

```
public boolean isCellEditable(EventObject anEvent)
{
   return true;
}
```

However, if you override this method to false, then the table would not go through the trouble of inserting the editor component.

Once the editor component is installed, the shouldSelectCell method is called, presumably with the same event. You should initiate editing in this method, for example, by popping up an external edit dialog box.

```
public boolean shouldSelectCell(EventObject anEvent)
{
   colorDialog.setVisible(true);
   return true;
}
```

If the user cancels the edit, the table calls the cancelCellEditing method. If the user has clicked on another table cell, the table calls the stopCellEditing method. In both cases, you should hide the dialog box. When your stopCellEditing method is called, the table would like to use the partially edited value. You should return true if the current value is valid. In the color chooser, any value is valid. But if you edit other data, you can ensure that only valid data is retrieved from the editor.

Also, you should call the superclass methods that take care of event firing—otherwise, the editing won't be properly canceled.

```
public void cancelCellEditing()
{
   colorDialog.setVisible(false);
   super.cancelCellEditing();
}
```

Finally, you need to supply a method that yields the value that the user supplied in the editing process:

```
public Object getCellEditorValue()
{
   return colorChooser.getColor();
}
```

To summarize, your custom editor should do the following:

1. Extend the AbstractCellEditor class and implement the TableCellEditor interface.

2. Define the getTableCellEditorComponent method to supply a component. This can either be a dummy component (if you pop up a dialog box) or a component for in-place editing such as a combo box or text field.

3. Define the shouldSelectCell, stopCellEditing, and cancelCellEditing methods to handle the start, completion, and cancellation of the editing process. The stopCellEditing and cancelCellEditing methods should call the superclass methods to ensure that listeners are notified.

4. Define the getCellEditorValue method to return the value that is the result of the editing process.

Finally, you indicate when the user is finished editing by calling the stopCellEditing and cancelCellEditing methods. When constructing the color dialog box, we install accept and cancel callbacks that fire these events.

Code View:

```
colorDialog = JColorChooser.createDialog(null, "Planet Color", false, colorChooser,
   new
      ActionListener() // OK button listener
      {
         public void actionPerformed(ActionEvent event)
         {
            stopCellEditing();
         }
      },
   new
      ActionListener() // Cancel button listener
      {
         public void actionPerformed(ActionEvent event)
         {
            cancelCellEditing();
         }
      });
```

Also, when the user closes the dialog box, editing should be canceled. This is achieved by installation of a window listener:

```
colorDialog.addWindowListener(new
   WindowAdapter()
   {
      public void windowClosing(WindowEvent event)
      {
         cancelCellEditing();
      }
   });
```

This completes the implementation of the custom editor.

You now know how to make a cell editable and how to install an editor. There is one remaining issue—how to update the model with the value that the user edited. When editing is complete, the JTable class calls the following method of the table model:

```
void setValueAt(Object value, int r, int c)
```

You need to override the method to store the new value. The value parameter is the object that was returned by the cell editor. If you implemented the cell editor, then you know the type of the object that you return from the getCellEditorValue method. In the case of the DefaultCellEditor, there are three possibilities for that value. It is a Boolean if the cell editor is a checkbox, a string if it is a text field. If the value comes from a combo box, then it is the object that the user selected.

If the value object does not have the appropriate type, you need to convert it. That happens most commonly when a number is edited in a text field. In our example, we populated the combo box with Integer objects so that no conversion is necessary.

Listing 6-7. **TableCellRenderTest.java**

Code View:
```
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import javax.swing.*;
import javax.swing.table.*;

/**
 * This program demonstrates cell rendering and editing in a table.
 * @version 1.02 2007-08-01
 * @author Cay Horstmann
 */
public class TableCellRenderTest
{
   public static void main(String[] args)
   {
      EventQueue.invokeLater(new Runnable()
         {
            public void run()
            {

               JFrame frame = new TableCellRenderFrame();
               frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
               frame.setVisible(true);
            }
         });
   }
}

/**
 * This frame contains a table of planet data.
 */
class TableCellRenderFrame extends JFrame
{
   public TableCellRenderFrame()
   {
      setTitle("TableCellRenderTest");
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

      TableModel model = new PlanetTableModel();
      JTable table = new JTable(model);
      table.setRowSelectionAllowed(false);

      // set up renderers and editors

      table.setDefaultRenderer(Color.class, new ColorTableCellRenderer());
```

```
      table.setDefaultEditor(Color.class, new ColorTableCellEditor());

      JComboBox moonCombo = new JComboBox();
      for (int i = 0; i <= 20; i++)
         moonCombo.addItem(i);

      TableColumnModel columnModel = table.getColumnModel();
      TableColumn moonColumn = columnModel.getColumn(PlanetTableModel.MOONS_COLUMN);
      moonColumn.setCellEditor(new DefaultCellEditor(moonCombo));
```

```
      moonColumn.setHeaderRenderer(table.getDefaultRenderer(ImageIcon.class));
      moonColumn.setHeaderValue(new ImageIcon("Moons.gif"));

      // show table

      table.setRowHeight(100);
      add(new JScrollPane(table), BorderLayout.CENTER);
```

```
   }

   private static final int DEFAULT_WIDTH = 600;
   private static final int DEFAULT_HEIGHT = 400;
}

/**
 * The planet table model specifies the values, rendering and editing properties for the
 * planet data.
 */
class PlanetTableModel extends AbstractTableModel
{
   public String getColumnName(int c)
   {
      return columnNames[c];
   }

   public Class<?> getColumnClass(int c)
   {
      return cells[0][c].getClass();
   }

   public int getColumnCount()
   {
      return cells[0].length;
   }

   public int getRowCount()
   {
      return cells.length;
   }

   public Object getValueAt(int r, int c)
   {
      return cells[r][c];
   }

   public void setValueAt(Object obj, int r, int c)
   {
      cells[r][c] = obj;
   }

   public boolean isCellEditable(int r, int c)
```

```java
   {
      return c == PLANET_COLUMN || c == MOONS_COLUMN || c == GASEOUS_COLUMN ||
         c == COLOR_COLUMN;
   }

   public static final int PLANET_COLUMN = 0;
   public static final int MOONS_COLUMN = 2;
   public static final int GASEOUS_COLUMN = 3;
   public static final int COLOR_COLUMN = 4;

   private Object[][] cells = {
         { "Mercury", 2440.0, 0, false, Color.YELLOW, new ImageIcon("Mercury.gif") },
         { "Venus", 6052.0, 0, false, Color.YELLOW, new ImageIcon("Venus.gif") },
         { "Earth", 6378.0, 1, false, Color.BLUE, new ImageIcon("Earth.gif") },
         { "Mars", 3397.0, 2, false, Color.RED, new ImageIcon("Mars.gif") },
         { "Jupiter", 71492.0, 16, true, Color.ORANGE, new ImageIcon("Jupiter.gif") },
         { "Saturn", 60268.0, 18, true, Color.ORANGE, new ImageIcon("Saturn.gif") },
         { "Uranus", 25559.0, 17, true, Color.BLUE, new ImageIcon("Uranus.gif") },
         { "Neptune", 24766.0, 8, true, Color.BLUE, new ImageIcon("Neptune.gif") },
         { "Pluto", 1137.0, 1, false, Color.BLACK, new ImageIcon("Pluto.gif") } };

   private String[] columnNames = { "Planet", "Radius", "Moons", "Gaseous", "Color",
      "Image" };
}

/**
 * This renderer renders a color value as a panel with the given color.
 */
class ColorTableCellRenderer extends JPanel implements TableCellRenderer
{
   public Component getTableCellRendererComponent(JTable table, Object value,
         boolean isSelected, boolean hasFocus, int row, int column)
   {
      setBackground((Color) value);
      if (hasFocus) setBorder(UIManager.getBorder("Table.focusCellHighlightBorder"));
      else setBorder(null);
      return this;
   }
}

/**
 * This editor pops up a color dialog to edit a cell value
 */
class ColorTableCellEditor extends AbstractCellEditor implements TableCellEditor
{
   public ColorTableCellEditor()
   {
      panel = new JPanel();
      // prepare color dialog

      colorChooser = new JColorChooser();
      colorDialog = JColorChooser.createDialog(null, "Planet Color", false, colorChooser,
            new ActionListener() // OK button listener
               {
                  public void actionPerformed(ActionEvent event)
                  {
                     stopCellEditing();
                  }
               }, new ActionListener() // Cancel button listener
```

```
               {
                  public void actionPerformed(ActionEvent event)
                  {
                     cancelCellEditing();
                  }
               });
      colorDialog.addWindowListener(new WindowAdapter()
         {
            public void windowClosing(WindowEvent event)
            {
               cancelCellEditing();
            }
         });
   }

   public Component getTableCellEditorComponent(JTable table, Object value,
      boolean isSelected, int row, int column)
   {
      // this is where we get the current Color value. We store it in the dialog in case
      // the user starts editing
      colorChooser.setColor((Color) value);
      return panel;
   }

   public boolean shouldSelectCell(EventObject anEvent)
   {
      // start editing
      colorDialog.setVisible(true);

      // tell caller it is ok to select this cell
      return true;
   }

   public void cancelCellEditing()
   {
      // editing is canceled--hide dialog
      colorDialog.setVisible(false);
      super.cancelCellEditing();
   }

   public boolean stopCellEditing()
   {
      // editing is complete--hide dialog
      colorDialog.setVisible(false);
      super.stopCellEditing();

      // tell caller is is ok to use color value
      return true;
   }

   public Object getCellEditorValue()
   {
      return colorChooser.getColor();
   }

   private JColorChooser colorChooser;
   private JDialog colorDialog;
   private JPanel panel;
}
```

javax.swing.JTable 1.2

- TableCellRenderer getDefaultRenderer(Class<?> type)

  gets the default renderer for the given type.

- TableCellEditor getDefaultEditor(Class<?> type)

  gets the default editor for the given type.

javax.swing.table.TableCellRenderer 1.2

- Component getTableCellRendererComponent(JTable table, Object value, boolean selected, boolean hasFocus, int row, int column)

  returns a component whose paint method is invoked to render a table cell.

Parameters:	table	The table containing the cell to be rendered
	value	The cell to be rendered
	selected	true if the cell is currently selected
	hasFocus	true if the cell currently has focus
	row, column	The row and column of the cell

`javax.swing.table.TableColumn` 1.2

- `void setCellEditor(TableCellEditor editor)`
- `void setCellRenderer(TableCellRenderer renderer)`

  sets the cell editor or renderer for all cells in this column.

- `void setHeaderRenderer(TableCellRenderer renderer)`

  sets the cell renderer for the header cell in this column.

- `void setHeaderValue(Object value)`

  sets the value to be displayed for the header in this column.

`javax.swing.DefaultCellEditor` 1.2

- `DefaultCellEditor(JComboBox comboBox)`

  constructs a cell editor that presents the combo box for selecting cell values.

`javax.swing.table.TableCellEditor` 1.2

- `Component getTableCellEditorComponent(JTable table, Object value, boolean selected, int row, int column)`

  returns a component whose `paint` method renders a table cell.

Parameters:	`table`	The table containing the cell to be rendered
	`value`	The cell to be rendered
	`selected`	`true` if the cell is currently selected
	`row, column`	The row and column of the cell

javax.swing.CellEditor 1.2

- `boolean isCellEditable(EventObject event)`

  returns `true` if the event is suitable for initiating the editing process for this cell.

- `boolean shouldSelectCell(EventObject anEvent)`

  starts the editing process. Returns `true` if the edited cell should be *selected*. Normally, you want to return `true`, but you can return `false` if you don't want the editing process to change the cell selection.

- `void cancelCellEditing()`

  cancels the editing process. You can abandon partial edits.

- `boolean stopCellEditing()`

  stops the editing process, with the intent of using the result. Returns `true` if the edited value is in a proper state for retrieval.

- `Object getCellEditorValue()`

  returns the edited result.

- `void addCellEditorListener(CellEditorListener l)`

- `void removeCellEditorListener(CellEditorListener l)`

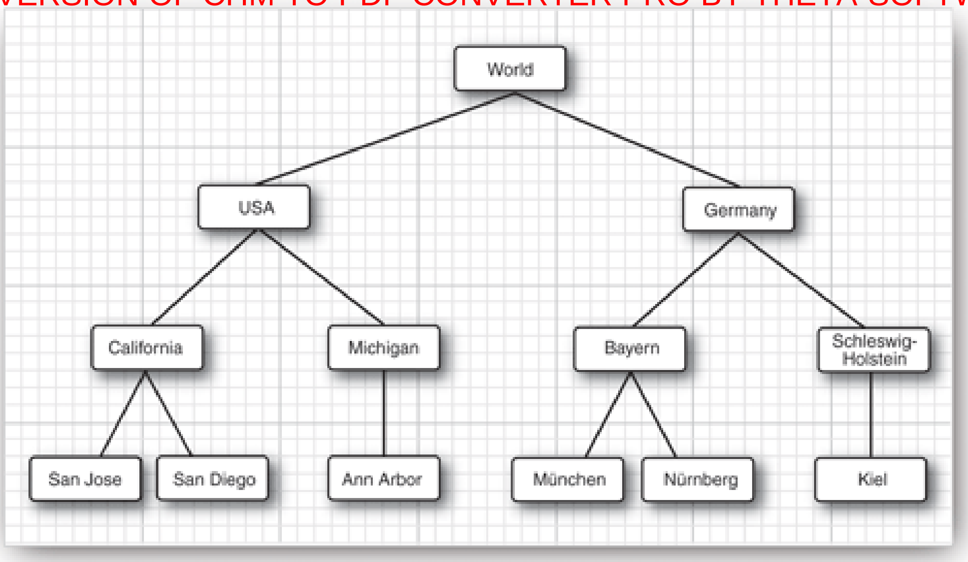
  adds or removes the obligatory cell editor listener.

Trees

Every computer user who uses a hierarchical file system has encountered tree displays. Of course, directories and files form only one of the many examples of treelike organizations. Many tree structures arise in everyday life, such as the hierarchy of countries, states, and cities shown in Figure 6-15.

Figure 6-15. A hierarchy of countries, states, and cities

[View full size image]

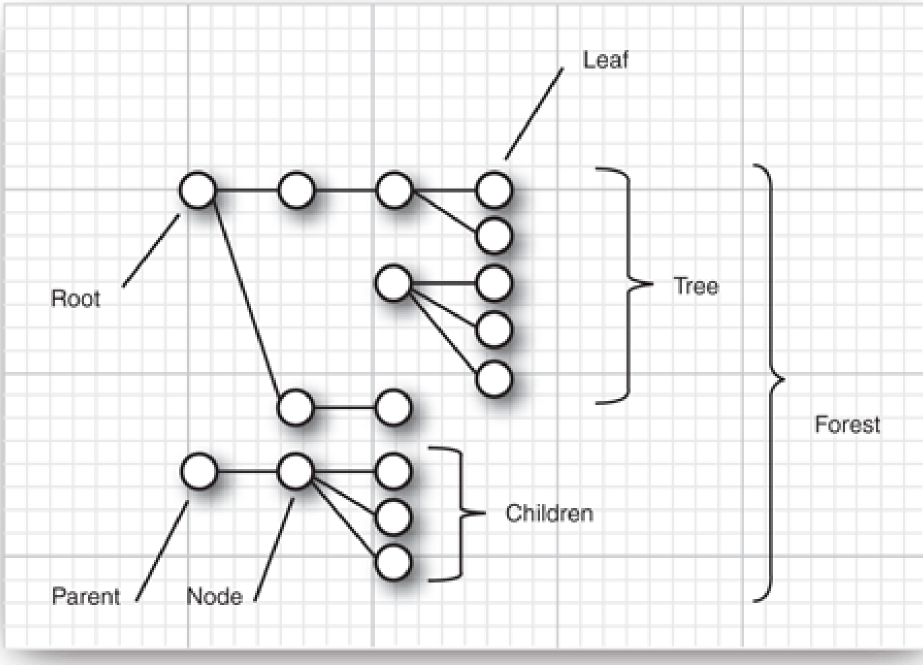


As programmers, we often have to display these tree structures. Fortunately, the Swing library has a `JTree` class for this purpose. The `JTree` class (together with its helper classes) takes care of laying out the tree and processing user requests for expanding and collapsing nodes. In this section, you will learn how to put the `JTree` class to use.

As with the other complex Swing components, we must focus on the common and useful cases and cannot cover every nuance. If you want to achieve an unusual effect, we recommend that you consult *Graphic Java 2: Mastering the JFC, Volume II: Swing*, 3rd ed., by David M. Geary, *Core Java Foundation Classes* by Kim Topley, or *Core Swing: Advanced Programming* by Kim Topley (Pearson Education 1999).

Before going any further, let's settle on some terminology (see Figure 6-16). A *tree* is composed of *nodes*. Every node is either a *leaf* or it has *child nodes*. Every node, with the exception of the root node, has exactly one *parent*. A tree has exactly one root node. Sometimes you have a collection of trees, each of which has its own root node. Such a collection is called a *forest*.

Figure 6-16. Tree terminology



Simple Trees

In our first example program, we simply display a tree with a few nodes (see Figure 6-18 on page 408). As with many other Swing components, you provide a model of the data, and the component displays it for you. To construct a JTree, you supply the tree model in the constructor:

```
TreeModel model = . . .;
JTree tree = new JTree(model);
```

Note

There are also constructors that construct trees out of a collection of elements:

```
JTree(Object[] nodes)
JTree(Vector<?> nodes)
JTree(Hashtable<?, ?> nodes) // the values become the nodes
```

These constructors are not very useful. They merely build a forest of trees, each with a single node. The third constructor seems particularly useless because the nodes appear in the seemingly random order given by the hash codes of the keys.

How do you obtain a tree model? You can construct your own model by creating a class that implements the

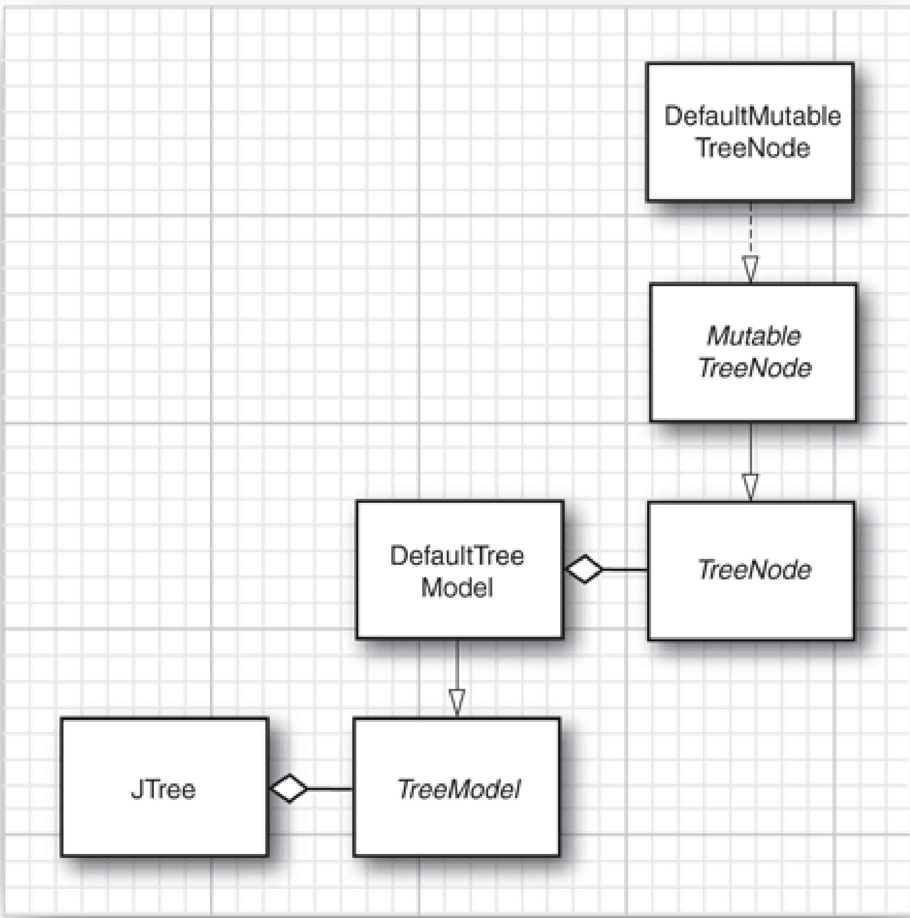
TreeModel interface. You see later in this chapter how to do that. For now, we stick with the DefaultTreeModel that the Swing library supplies.

To construct a default tree model, you must supply a root node.

```
TreeNode root = . . .;
DefaultTreeModel model = new DefaultTreeModel(root);
```


TreeNode is another interface. You populate the default tree model with objects of any class that implements the interface. For now, we use the concrete node class that Swing supplies, namely, DefaultMutableTreeNode. This class implements the MutableTreeNode interface, a subinterface of TreeNode (see Figure 6-17).

Figure 6-17. Tree classes



A default mutable tree node holds an object, the *user object*. The tree renders the user objects for all nodes. Unless you specify a renderer, the tree simply displays the string that is the result of the toString method.

In our first example, we use strings as user objects. In practice, you would usually populate a tree with more expressive user objects. For example, when displaying a directory tree, it makes sense to use File objects for the nodes.

You can specify the user object in the constructor, or you can set it later with the setUserObject method.

```
DefaultMutableTreeNode node = new DefaultMutableTreeNode("Texas");
. . .
node.setUserObject("California");
```

Next, you establish the parent/child relationships between the nodes. Start with the root node, and use the add method to add the children:

```
DefaultMutableTreeNode root = new DefaultMutableTreeNode("World");
DefaultMutableTreeNode country = new DefaultMutableTreeNode("USA");
root.add(country);
DefaultMutableTreeNode state = new DefaultMutableTreeNode("California");
country.add(state);
```

Figure 6-18 illustrates how the tree will look.

Figure 6-18. A simple tree

Link up all nodes in this fashion. Then, construct a DefaultTreeModel with the root node. Finally, construct a JTree with the tree model.

```
DefaultTreeModel treeModel = new DefaultTreeModel(root);
JTree tree = new JTree(treeModel);
```

Or, as a shortcut, you can simply pass the root node to the JTree constructor. Then the tree automatically constructs a default tree model:

```
JTree tree = new JTree(root);
```

Listing 6-8 contains the complete code.

Listing 6-8. **SimpleTree.java**

Code View:

```
import java.awt.*;

import javax.swing.*;
import javax.swing.tree.*;

/**
 * This program shows a simple tree.
 * @version 1.02 2007-08-01
 * @author Cay Horstmann
 */
public class SimpleTree
{
   public static void main(String[] args)
   {
      EventQueue.invokeLater(new Runnable()
         {
            public void run()
            {
               JFrame frame = new SimpleTreeFrame();
               frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
               frame.setVisible(true);
            }
         });
   }
}

/**
 * This frame contains a simple tree that displays a manually constructed tree model.
 */
class SimpleTreeFrame extends JFrame
{
   public SimpleTreeFrame()
   {
      setTitle("SimpleTree");
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

      // set up tree model data

      DefaultMutableTreeNode root = new DefaultMutableTreeNode("World");
      DefaultMutableTreeNode country = new DefaultMutableTreeNode("USA");
      root.add(country);
      DefaultMutableTreeNode state = new DefaultMutableTreeNode("California");
      country.add(state);
      DefaultMutableTreeNode city = new DefaultMutableTreeNode("San Jose");
      state.add(city);
      city = new DefaultMutableTreeNode("Cupertino");
      state.add(city);
      state = new DefaultMutableTreeNode("Michigan");
      country.add(state);
      city = new DefaultMutableTreeNode("Ann Arbor");
      state.add(city);
      country = new DefaultMutableTreeNode("Germany");
      root.add(country);
      state = new DefaultMutableTreeNode("Schleswig-Holstein");
```

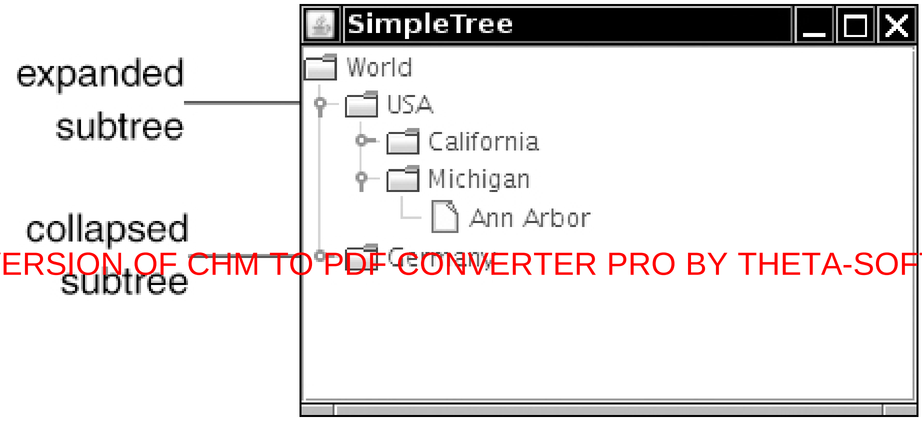

UNREGISTERED VERSION OF CHM TO PDF CONVERTER PRO BY THETA-SOFTWARE

```
      country.add(state);
      city = new DefaultMutableTreeNode("Kiel");
      state.add(city);

      // construct tree and put it in a scroll pane

      JTree tree = new JTree(root);
      add(new JScrollPane(tree));
   }

   private static final int DEFAULT_WIDTH = 300;
   private static final int DEFAULT_HEIGHT = 200;
}
```

When you run the program, the tree first looks as in Figure 6-19. Only the root node and its children are visible. Click on the circle icons (the *handles*) to open up the subtrees. The line sticking out from the handle icon points to the right when the subtree is collapsed, and it points down when the subtree is expanded (see Figure 6-20). We don't know what the designers of the Metal look and feel had in mind, but we think of the icon as a door handle. You push down on the handle to open the subtree.

Figure 6-19. The initial tree display

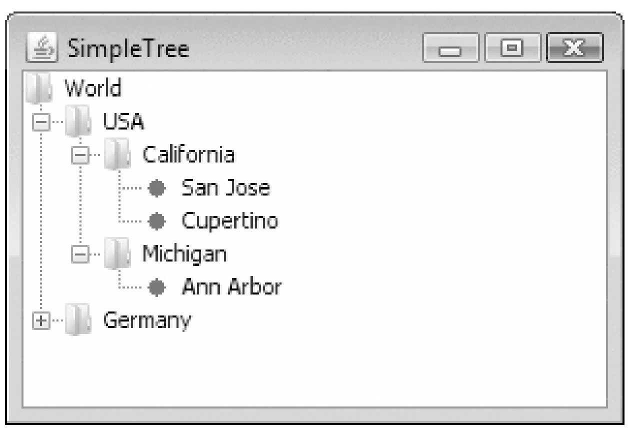
Figure 6-20. Collapsed and expanded subtrees



Note

Of course, the display of the tree depends on the selected look and feel. We just described the Metal look and feel. In the Windows look and feel, the handles have the more familiar look—a "-" or "+" in a box (see Figure 6-21).

Figure 6-21. A tree with the Windows look and feel



You can use the following magic incantation to turn off the lines joining parents and children (see Figure 6-22):

```
tree.putClientProperty("JTree.lineStyle", "None");
```

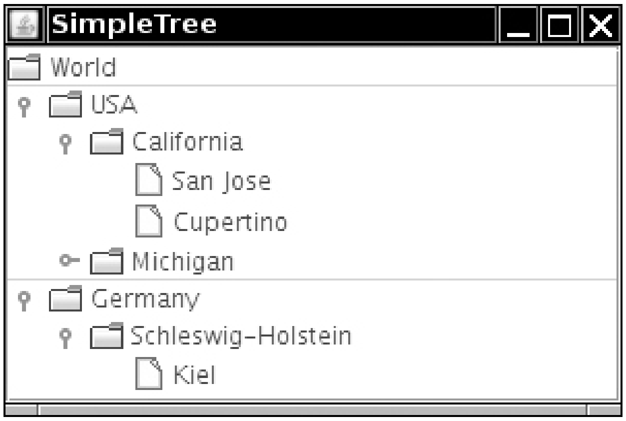
Figure 6-22. A tree with no connecting lines

Conversely, to make sure that the lines are shown, use

```
tree.putClientProperty("JTree.lineStyle", "Angled");
```

Another line style, "Horizontal", is shown in Figure 6-23. The tree is displayed with horizontal lines separating only the children of the root. We aren't quite sure what it is good for.

Figure 6-23. A tree with the horizontal line style

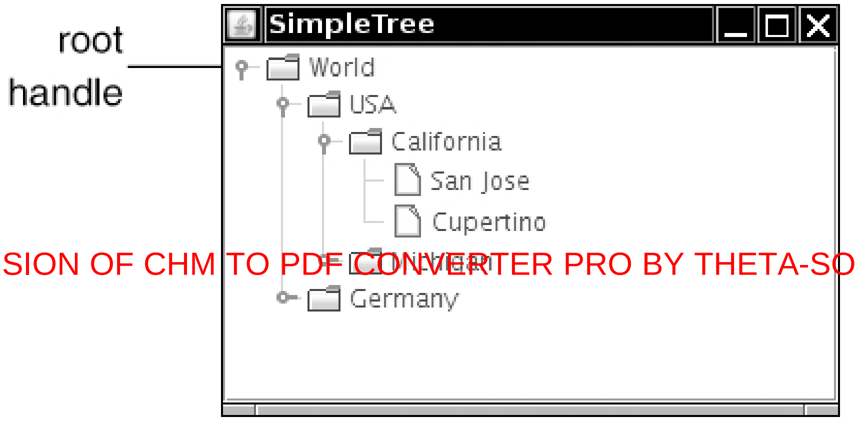


By default, there is no handle for collapsing the root of the tree. If you like, you can add one with the call

```
tree.setShowsRootHandles(true);
```

Figure 6-24 shows the result. Now you can collapse the entire tree into the root node.

Figure 6-24. A tree with a root handle



Conversely, you can hide the root altogether. You do that to display a *forest*, a set of trees, each of which has its own root. You still must join all trees in the forest to a common root. Then, you hide the root with the instruction

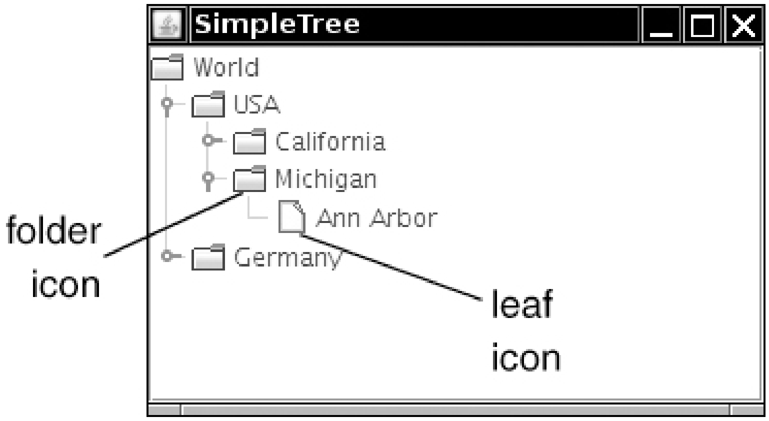
```
tree.setRootVisible(false);
```

Look at Figure 6-25. There appear to be two roots, labeled "USA" and "Germany." The actual root that joins the two is made invisible.

Figure 6-25. A forest

Let's turn from the root to the leaves of the tree. Note that the leaves have a different icon from the other nodes (see Figure 6-26).

Figure 6-26. Leaf and folder icons



When the tree is displayed, each node is drawn with an icon. There are actually three kinds of icons: a leaf icon, an opened nonleaf icon, and a closed nonleaf icon. For simplicity, we refer to the last two as folder icons.

The node renderer needs to know which icon to use for each node. By default, the decision process works like this: If the `isLeaf` method of a node returns `true`, then the leaf icon is used. Otherwise, a folder icon is used.

The `isLeaf` method of the `DefaultMutableTreeNode` class returns `true` if the node has no children. Thus, nodes with children get folder icons, and nodes without children get leaf icons.

Sometimes, that behavior is not appropriate. Suppose we added a node "Montana" to our sample tree, but we're at a loss as to what cities to add. We would not want the state node to get a leaf icon because conceptually only the cities are leaves.

The `JTree` class has no idea which nodes should be leaves. It asks the tree model. If a childless node isn't automatically a conceptual leaf, you can ask the tree model to use a different criterion for leafiness, namely, to query the "allows children" node property.

For those nodes that should not have children, call

```
node.setAllowsChildren(false);
```

Then, tell the tree model to ask the value of the "allows children" property to determine whether a node should be displayed with a leaf icon. You use the `setAsksAllowsChildren` method of the `DefaultTreeModel` class to set this behavior:

```
model.setAsksAllowsChildren(true);
```

With this decision criterion, nodes that allow children get folder icons, and nodes that don't allow children get leaf icons.

Alternatively, if you construct the tree by supplying the root node, supply the setting for the "asks allows children" property in the constructor.

Code View:
```
JTree tree = new JTree(root, true); // nodes that don't allow children get leaf icons
```

javax.swing.JTree 1.2

- `JTree(TreeModel model)`

  constructs a tree from a tree model.

- `JTree(TreeNode root)`
- `JTree(TreeNode root, boolean asksAllowChildren)`

  constructs a tree with a default tree model that displays the root and its children.

Parameters:	`root`	The root node
	`asksAllowsChildren`	`true` to use the "allows children" node property for determining whether a node is a leaf

- `void setShowsRootHandles(boolean b)`

  If `b` is `true`, then the root node has a handle for collapsing or expanding its children.

- `void setRootVisible(boolean b)`

  If `b` is true, then the root node is displayed. Otherwise, it is hidden.

javax.swing.tree.TreeNode 1.2

- `boolean isLeaf()`

  returns `true` if this node is conceptually a leaf.

- `boolean getAllowsChildren()`

  returns `true` if this node can have child nodes.

`javax.swing.tree.MutableTreeNode` 1.2

- `void setUserObject(Object userObject)`

  sets the "user object" that the tree node uses for rendering.

`javax.swing.tree.TreeModel` 1.2

- `boolean isLeaf(Object node)`

  returns `true` if `node` should be displayed as a leaf node.

`javax.swing.tree.DefaultTreeModel` 1.2

- `void setAsksAllowsChildren(boolean b)`

  If `b` is `true`, then nodes are displayed as leaves when their `getAllowsChildren` method returns `false`. Otherwise, they are displayed as leaves when their `isLeaf` method returns `true`.

`javax.swing.tree.DefaultMutableTreeNode` 1.2

- `DefaultMutableTreeNode(Object userObject)`

  constructs a mutable tree node with the given user object.

- `void add(MutableTreeNode child)`

  adds a node as the last child of this node.

- `void setAllowsChildren(boolean b)`

  If `b` is `true`, then children can be added to this node.

API `javax.swing.JComponent` 1.2

- `void putClientProperty(Object key, Object value)`

  adds a key/value pair to a small table that each component manages. This is an "escape hatch" mechanism that some Swing components use for storing look-and-feel–specific properties.

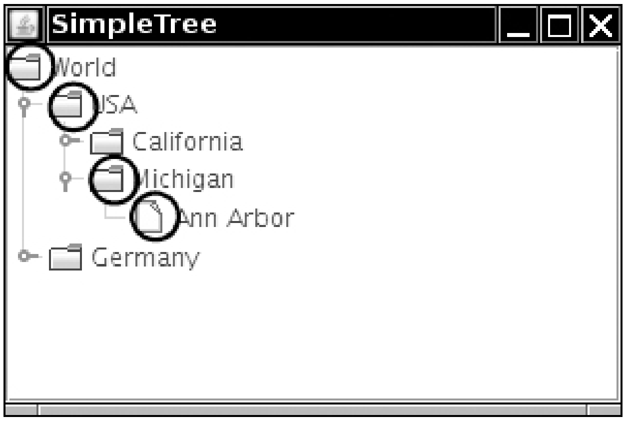
Editing Trees and Tree Paths

In the next example program, you see how to edit a tree. Figure 6-27 shows the user interface. If you click the Add Sibling or Add Child button, the program adds a new node (with title New) to the tree. If you click the Delete button, the program deletes the currently selected node.

Figure 6-27. Editing a tree

To implement this behavior, you need to find out which tree node is currently selected. The `JTree` class has a surprising way of identifying nodes in a tree. It does not deal with tree nodes, but with *paths of objects*, called *tree paths*. A tree path starts at the root and consists of a sequence of child nodes—see Figure 6-28.

Figure 6-28. A tree path



You might wonder why the JTree class needs the whole path. Couldn't it just get a TreeNode and keep calling the getParent method? In fact, the JTree class knows nothing about the TreeNode interface. That interface is never used by the TreeModel interface; it is only used by the DefaultTreeModel implementation. You can have other tree models in which the nodes do not implement the TreeNode interface at all. If you use a tree model that manages other types of objects, then those objects might not have getParent and getChild methods. They would of course need to have some other connection to each other. It is the job of the tree model to link nodes together. The JTree class itself has no clue about the nature of their linkage. For that reason, the JTree class always needs to work with complete paths.

The TreePath class manages a sequence of Object (not TreeNode!) references. A number of JTree methods return TreePath objects. When you have a tree path, you usually just need to know the terminal node, which you get with the getLastPathComponent method. For example, to find out the currently selected node in a tree, you use the getSelectionPath method of the JTree class. You get a TreePath object back, from which you can retrieve the actual node.

```
TreePath selectionPath = tree.getSelectionPath();
DefaultMutableTreeNode selectedNode
   = (DefaultMutableTreeNode) selectionPath.getLastPathComponent();
```

Actually, because this particular query is so common, there is a convenience method that gives the selected node immediately.

```
DefaultMutableTreeNode selectedNode
   = (DefaultMutableTreeNode) tree.getLastSelectedPathComponent();
```

This method is not called getSelectedNode because the tree does not know that it contains nodes—its tree model deals only with paths of objects.

Note

Tree paths are one of two ways in which the `JTree` class describes nodes. Quite a few `JTree` methods take or return an integer index, the *row position*. A row position is simply the row number (starting with 0) of the node in the tree display. Only visible nodes have row numbers, and the row number of a node changes if other nodes before it are expanded, collapsed, or modified. For that reason, you should avoid row positions. All `JTree` methods that use rows have equivalents that use tree paths instead.

Once you have the selected node, you can edit it. However, do not simply add children to a tree node:


```
selectedNode.add(newNode); // NO
```

If you change the structure of the nodes, you change the model but the associated view is not notified. You could send out a notification yourself, but if you use the `insertNodeInto` method of the `DefaultTreeModel` class, the model class takes care of that. For example, the following call appends a new node as the last child of the selected node and notifies the tree view.

```
model.insertNodeInto(newNode, selectedNode, selectedNode.getChildCount());
```

The analogous call `removeNodeFromParent` removes a node and notifies the view:

```
model.removeNodeFromParent(selectedNode);
```

If you keep the node structure in place but you changed the user object, you should call the following method:

```
model.nodeChanged(changedNode);
```

The automatic notification is a major advantage of using the `DefaultTreeModel`. If you supply your own tree model, you have to implement automatic notification by hand. (See *Core Java Foundation Classes* by Kim Topley for details.)

Caution

The `DefaultTreeModel` class has a `reload` method that reloads the entire model. However, don't call `reload` simply to update the tree after making a few changes. When the tree is regenerated, all nodes beyond the root's children are collapsed again. It is quite disconcerting to your users if they have to keep expanding the tree after every change.

When the view is notified of a change in the node structure, it updates the display but it does not automatically expand a node to show newly added children. In particular, if a user in our sample program adds a new child node to a node for which children are currently collapsed, then the new node is silently added to the collapsed subtree. This gives the user no feedback that the command was actually carried out. In such a case, you should make a special effort to expand all parent nodes so that the newly added node becomes visible. You use the makeVisible method of the JTree class for this purpose. The makeVisible method expects a tree path leading to the node that should become visible.

Thus, you need to construct a tree path from the root to the newly inserted node. To get a tree path, you first call the getPathToRoot method of the DefaultTreeModel class. It returns a TreeNode[] array of all nodes from a node to the root node. You pass that array to a TreePath constructor.

For example, here is how you make the new node visible:

```
TreeNode[] nodes = model.getPathToRoot(newNode);
TreePath path = new TreePath(nodes);
tree.makeVisible(path);
```

Note

It is curious that the DefaultTreeModel class feigns almost complete ignorance about the TreePath class, even though its job is to communicate with a JTree. The JTree class uses tree paths a lot, and it never uses arrays of node objects.

But now suppose your tree is contained inside a scroll pane. After the tree node expansion, the new node might still not be visible because it falls outside the viewport. To overcome that problem, call

```
tree.scrollPathToVisible(path);
```

instead of calling makeVisible. This call expands all nodes along the path, and it tells the ambient scroll pane to scroll the node at the end of the path into view (see Figure 6-29).

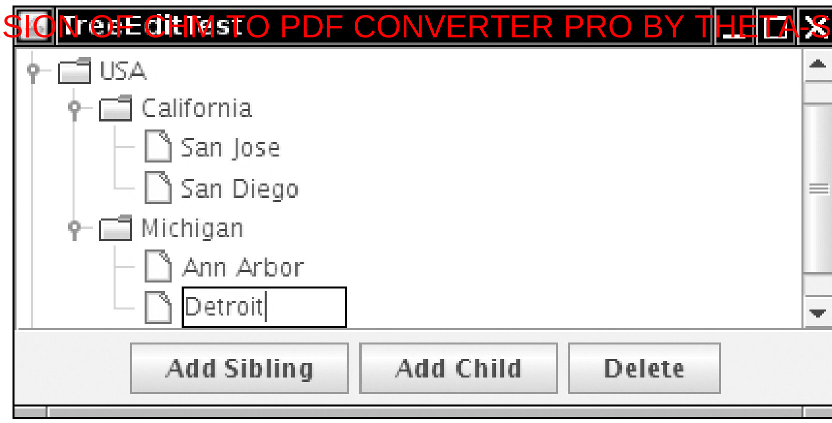
Figure 6-29. The scroll pane scrolls to display a new node

By default, tree nodes cannot be edited. However, if you call

```
tree.setEditable(true);
```

then the user can edit a node simply by double-clicking, editing the string, and pressing the ENTER key. Double-clicking invokes the *default cell editor*, which is implemented by the `DefaultCellEditor` class (see Figure 6-30). It is possible to install other cell editors, using the same process that you have seen in our discussion of table cell editors.

Figure 6-30. The default cell editor



Listing 6-9 shows the complete source code of the tree editing program. Run the program, add a few nodes, and edit them by double-clicking them. Observe how collapsed nodes expand to show added children and how the scroll pane keeps added nodes in the viewport.

Listing 6-9. **`TreeEditTest.java`**

Code View:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.tree.*;

/**
 * This program demonstrates tree editing.
 * @version 1.03 2007-08-01
 * @author Cay Horstmann
 */
public class TreeEditTest
{
   public static void main(String[] args)
   {
      EventQueue.invokeLater(new Runnable()
         {
            public void run()
            {
               JFrame frame = new TreeEditFrame();
```

```
               frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
               frame.setVisible(true);
            }
         });
   }
}

/**
 * A frame with a tree and buttons to edit the tree.
 */
class TreeEditFrame extends JFrame
{
   public TreeEditFrame()
   {
      setTitle("TreeEditTest");
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

      // construct tree

      TreeNode root = makeSampleTree();
      model = new DefaultTreeModel(root);
      tree = new JTree(model);
      tree.setEditable(true);

      // add scroll pane with tree

      JScrollPane scrollPane = new JScrollPane(tree);
      add(scrollPane, BorderLayout.CENTER);

      makeButtons();
   }

   public TreeNode makeSampleTree()
   {
      DefaultMutableTreeNode root = new DefaultMutableTreeNode("World");
      DefaultMutableTreeNode country = new DefaultMutableTreeNode("USA");
      root.add(country);
      DefaultMutableTreeNode state = new DefaultMutableTreeNode("California");
      country.add(state);
      DefaultMutableTreeNode city = new DefaultMutableTreeNode("San Jose");
      state.add(city);
      city = new DefaultMutableTreeNode("San Diego");
      state.add(city);
      state = new DefaultMutableTreeNode("Michigan");
      country.add(state);
      city = new DefaultMutableTreeNode("Ann Arbor");
      state.add(city);
      country = new DefaultMutableTreeNode("Germany");
      root.add(country);
      state = new DefaultMutableTreeNode("Schleswig-Holstein");
      country.add(state);
      city = new DefaultMutableTreeNode("Kiel");
      state.add(city);
      return root;
   }

   /**
    * Makes the buttons to add a sibling, add a child, and delete a node.
    */
```

```
public void makeButtons()
{
   JPanel panel = new JPanel();
   JButton addSiblingButton = new JButton("Add Sibling");
   addSiblingButton.addActionListener(new ActionListener()
      {
         public void actionPerformed(ActionEvent event)
         {
            DefaultMutableTreeNode selectedNode = (DefaultMutableTreeNode) tree
                  .getLastSelectedPathComponent();

            if (selectedNode == null) return;

            DefaultMutableTreeNode parent = (DefaultMutableTreeNode)
               selectedNode.getParent();

            if (parent == null) return;

            DefaultMutableTreeNode newNode = new DefaultMutableTreeNode("New");

            int selectedIndex = parent.getIndex(selectedNode);
            model.insertNodeInto(newNode, parent, selectedIndex + 1);

            // now display new node

            TreeNode[] nodes = model.getPathToRoot(newNode);
            TreePath path = new TreePath(nodes);
            tree.scrollPathToVisible(path);
         }
      });
   panel.add(addSiblingButton);

   JButton addChildButton = new JButton("Add Child");
   addChildButton.addActionListener(new ActionListener()
      {
         public void actionPerformed(ActionEvent event)
         {
            DefaultMutableTreeNode selectedNode = (DefaultMutableTreeNode) tree
                  .getLastSelectedPathComponent();

            if (selectedNode == null) return;

            DefaultMutableTreeNode newNode = new DefaultMutableTreeNode("New");
            model.insertNodeInto(newNode, selectedNode, selectedNode.getChildCount());

            // now display new node

            TreeNode[] nodes = model.getPathToRoot(newNode);
            TreePath path = new TreePath(nodes);
            tree.scrollPathToVisible(path);
         }
      });
   panel.add(addChildButton);

   JButton deleteButton = new JButton("Delete");
   deleteButton.addActionListener(new ActionListener()
      {
         public void actionPerformed(ActionEvent event)
         {
```

```
                DefaultMutableTreeNode selectedNode = (DefaultMutableTreeNode) tree
                      .getLastSelectedPathComponent();

                if (selectedNode != null && selectedNode.getParent() != null) model
                      .removeNodeFromParent(selectedNode);
             }
          });
       panel.add(deleteButton);
       add(panel, BorderLayout.SOUTH);
    }

    private DefaultTreeModel model;
    private JTree tree;
    private static final int DEFAULT_WIDTH = 400;
    private static final int DEFAULT_HEIGHT = 200;
 }
```

API

`javax.swing.JTree` 1.2

- `TreePath getSelectionPath()`

  gets the path to the currently selected node, or the path to the first selected node if multiple nodes are selected. Returns `null` if no node is selected.

- `Object getLastSelectedPathComponent()`

  gets the node object that represents the currently selected node, or the first node if multiple nodes are selected. Returns `null` if no node is selected.

- `void makeVisible(TreePath path)`

  expands all nodes along the path.

- `void scrollPathToVisible(TreePath path)`

  expands all nodes along the path and, if the tree is contained in a scroll pane, scrolls to ensure that the last node on the path is visible.

`javax.swing.tree.TreePath` 1.2

- `Object getLastPathComponent()`

  gets the last object on this path, that is, the node object that the path represents.

`javax.swing.tree.TreeNode` 1.2

- `TreeNode getParent()`

  returns the parent node of this node.

- `TreeNode getChildAt(int index)`

  looks up the child node at the given index. The index must be between 0 and `getChildCount()` - `1`.

- `int getChildCount()`

  returns the number of children of this node.

- `Enumeration children()`

  returns an enumeration object that iterates through all children of this node.

API javax.swing.tree.DefaultTreeModel 1.2

- `void insertNodeInto(MutableTreeNode newChild, MutableTreeNode parent, int index)`

  inserts `newChild` as a new child node of `parent` at the given index and notifies the tree model listeners.

- `void removeNodeFromParent(MutableTreeNode node)`

  removes `node` from this model and notifies the tree model listeners.

- `void nodeChanged(TreeNode node)`

  notifies the tree model listeners that `node` has changed.

- `void nodesChanged(TreeNode parent, int[] changedChildIndexes)`

  notifies the tree model listeners that all child nodes of `parent` with the given indexes have changed.

- `void reload()`

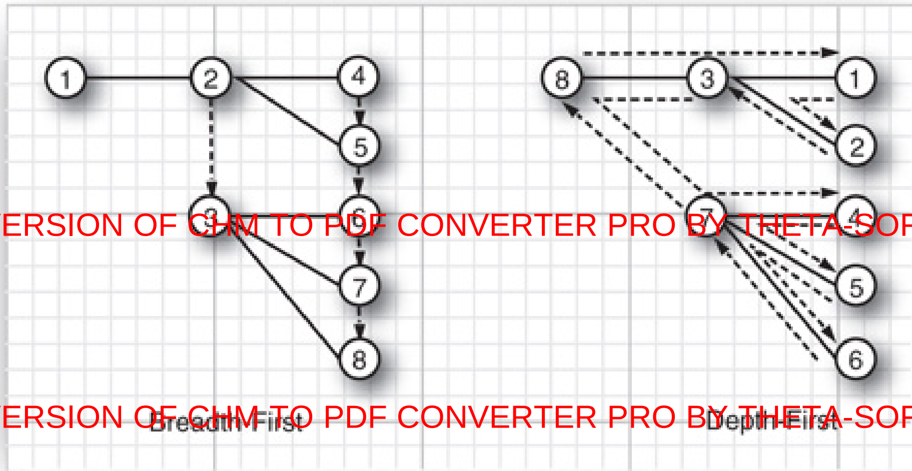
  reloads all nodes into the model. This is a drastic operation that you should use only if the nodes have changed completely because of some outside influence.

Node Enumeration

Sometimes you need to find a node in a tree by starting at the root and visiting all children until you have found a match. The `DefaultMutableTreeNode` class has several convenience methods for iterating through nodes.

The `breadthFirstEnumeration` and `depthFirstEnumeration` methods return enumeration objects whose `nextElement` method visits all children of the current node, using either a breadth-first or depth-first traversal. Figure 6-31 shows the traversals for a sample tree—the node labels indicate the order in which the nodes are traversed.

Figure 6-31. Tree traversal orders



Breadth-first enumeration is the easiest to visualize. The tree is traversed in layers. The root is visited first, followed by all of its children, then followed by the grandchildren, and so on.

To visualize depth-first enumeration, imagine a rat trapped in a tree-shaped maze. It rushes along the first path until it comes to a leaf. Then, it backtracks and turns around to the next path, and so on.

Computer scientists also call this *postorder traversal* because the search process visits the children before visiting the parents. The `postOrderTraversal` method is a synonym for `depthFirstTraversal`. For completeness, there is also a `preOrderTraversal`, a depth-first search that enumerates parents before the children.

Here is the typical usage pattern:

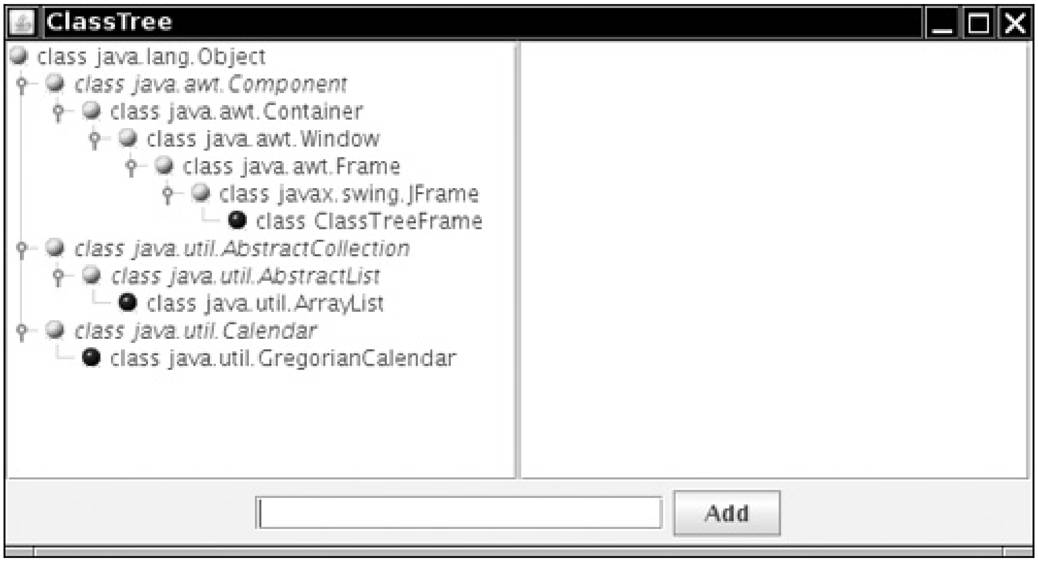
```
Enumeration breadthFirst = node.breadthFirstEnumeration();
while (breadthFirst.hasMoreElements())
   do something with breadthFirst.nextElement();
```

Finally, a related method, `pathFromAncestorEnumeration`, finds a path from an ancestor to a given node and then enumerates the nodes along that path. That's no big deal—it just keeps calling `getParent` until the ancestor is found and then presents the path in reverse order.

In our next example program, we put node enumeration to work. The program displays inheritance trees of classes. Type the name of a class into the text field on the bottom of the frame. The class and all of its superclasses are added to the tree (see Figure 6-32).

Figure 6-32. An inheritance tree

[View full size image]



In this example, we take advantage of the fact that the user objects of the tree nodes can be objects of any type. Because our nodes describe classes, we store `Class` objects in the nodes.

Of course, we don't want to add the same class object twice, so we need to check whether a class already exists in the tree. The following method finds the node with a given user object if it exists in the tree.

Code View:
```
public DefaultMutableTreeNode findUserObject(Object obj)
{
   Enumeration e = root.breadthFirstEnumeration();
   while (e.hasMoreElements())
   {
      DefaultMutableTreeNode node = (DefaultMutableTreeNode) e.nextElement();
      if (node.getUserObject().equals(obj))
         return node;
   }
   return null;
}
```

Rendering Nodes

In your applications, you will often need to change the way in which a tree component draws the nodes. The most common change is, of course, to choose different icons for nodes and leaves. Other changes might involve changing the font of the node labels or drawing images at the nodes. All these changes are made possible by installing a new *tree cell renderer* into the tree. By default, the `JTree` class uses `DefaultTreeCellRenderer` objects to draw each node. The `DefaultTreeCellRenderer` class extends the `JLabel` class. The label contains the node icon and the node label.

Note

The cell renderer does not draw the "handles" for expanding and collapsing subtrees. The handles are part of the look and feel, and it is recommended that you not change them.

You can customize the display in three ways.

- You can change the icons, font, and background color used by a `DefaultTreeCellRenderer`. These settings are used for all nodes in the tree.

- You can install a renderer that extends the `DefaultTreeCellRenderer` class and vary the icons, fonts, and background color for each node.

- You can install a renderer that implements the `TreeCellRenderer` interface, to draw a custom image for each node.

Let us look at these possibilities one by one. The easiest customization is to construct a `DefaultTreeCellRenderer` object, change the icons, and install it into the tree:

Code View:
```
DefaultTreeCellRenderer renderer = new DefaultTreeCellRenderer();
renderer.setLeafIcon(new ImageIcon("blue-ball.gif")); // used for leaf nodes
renderer.setClosedIcon(new ImageIcon("red-ball.gif")); // used for collapsed nodes
renderer.setOpenIcon(new ImageIcon("yellow-ball.gif")); // used for expanded nodes
tree.setCellRenderer(renderer);
```

You can see the effect in Figure 6-32. We just use the "ball" icons as placeholders—presumably your user interface designer would supply you with appropriate icons to use for your applications.

We don't recommend that you change the font or background color for an entire tree—that is really the job of the look and feel.

However, it can be useful to change the font for individual nodes in a tree to highlight some of them. If you look carefully at Figure 6-32, you will notice that the *abstract* classes are set in italics.

To change the appearance of individual nodes, you install a tree cell renderer. Tree cell renderers are very similar to the list cell renderers we discussed earlier in this chapter. The `TreeCellRenderer` interface has a single method:

Code View:
```
Component getTreeCellRendererComponent(JTree tree, Object value, boolean selected,
   boolean expanded, boolean leaf, int row, boolean hasFocus)
```

The getTreeCellRendererComponent method of the DefaultTreeCellRenderer class returns this—in other words, a label. (The DefaultTreeCellRenderer class extends the JLabel class.) To customize the component, extend the DefaultTreeCellRenderer class. Override the getTreeCellRendererComponent method as follows: Call the superclass method, so that it can prepare the label data. Customize the label properties, and finally return this.

Code View:

```
class MyTreeCellRenderer extends DefaultTreeCellRenderer
{
   public Component getTreeCellRendererComponent(JTree tree, Object value, boolean selected,
      boolean expanded, boolean leaf, int row, boolean hasFocus)
   {
      super.getTreeCellRendererComponent(tree, value, selected, expanded, leaf, row, hasFocus);
      DefaultMutableTreeNode node = (DefaultMutableTreeNode) value;
      look at node.getUserObject();
      Font font = appropriate font;
      setFont(font);
      return this;
   }
};
```

Caution

The value parameter of the getTreeCellRendererComponent method is the *node* object, *not* the user object! Recall that the user object is a feature of the DefaultMutableTreeNode, and that a JTree can contain nodes of an arbitrary type. If your tree uses DefaultMutableTreeNode nodes, then you must retrieve the user object in a second step, as we did in the preceding code sample.

Caution

The DefaultTreeCellRenderer uses the *same* label object for all nodes, only changing the label text for each node. If you change the font for a particular node, you must set it back to its default value when the method is called again. Otherwise, all subsequent nodes will be drawn in the changed font! Look at the code in Listing 6-10 to see how to restore the font to the default.

We do not show an example for a tree cell renderer that draws arbitrary graphics. If you need this capability, you can adapt the list cell renderer in Listing 6-3; the technique is entirely analogous.

The ClassNameTreeCellRenderer in Listing 6-10 sets the class name in either the normal or italic font, depending on the ABSTRACT modifier of the Class object. We don't want to set a particular font because we don't want to change whatever font the look and feel normally uses for labels. For that reason, we use the font from the label and *derive* an italic font from it. Recall that only a single shared JLabel object is returned by all calls. We need to hang on to the original font and restore it in the next call to the getTreeCellRendererComponent method.

Also, note how we change the node icons in the ClassTreeFrame constructor.

API javax.swing.tree.DefaultMutableTreeNode 1.2

- Enumeration breadthFirstEnumeration()

- Enumeration depthFirstEnumeration()
- Enumeration preOrderEnumeration()
- Enumeration postOrderEnumeration()

  returns enumeration objects for visiting all nodes of the tree model in a particular order. In breadth-first traversal, children that are closer to the root are visited before those that are farther away. In depth-first traversal, all children of a node are completely enumerated before its siblings are visited. The postOrderEnumeration method is a synonym for depthFirstEnumeration. The preorder traversal is identical to the postorder traversal except that parents are enumerated before their children.

API javax.swing.tree.TreeCellRenderer 1.2

- Component getTreeCellRendererComponent(JTree tree, Object value, boolean selected, boolean expanded, boolean leaf, int row, boolean hasFocus)

  returns a component whose paint method is invoked to render a tree cell.

Parameters:	`tree`	The tree containing the node to be rendered
	`value`	The node to be rendered
	`selected`	`true` if the node is currently selected
	`expanded`	`true` if the children of the node are visible
	`leaf`	`true` if the node needs to be displayed as a leaf
	`row`	The display row containing the node
	`hasFocus`	`true` if the node currently has input focus

API `javax.swing.tree.DefaultTreeCellRenderer` 1.2

- `void setLeafIcon(Icon icon)`
- `void setOpenIcon(Icon icon)`
- `void setClosedIcon(Icon icon)`

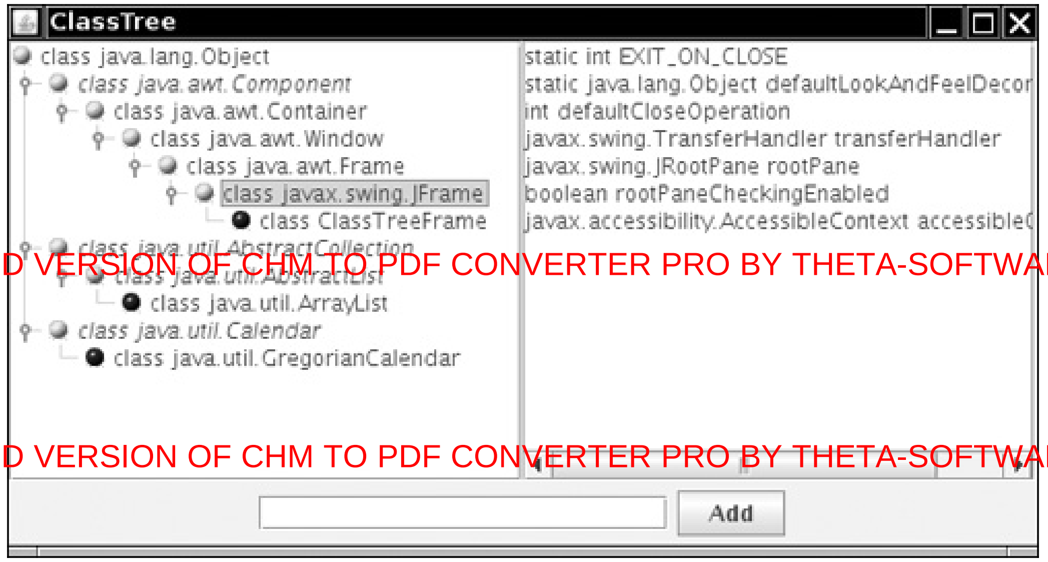
  sets the icon to show for a leaf node, an expanded node, and a collapsed node.

Listening to Tree Events

Most commonly, a tree component is paired with some other component. When the user selects tree nodes, some information shows up in another window. See Figure 6-33 for an example. When the user selects a class, the instance and static variables of that class are displayed in the text area to the right.

Figure 6-33. A class browser

[View full size image]



To obtain this behavior, you install a *tree selection listener*. The listener must implement the `TreeSelectionListener` interface, an interface with a single method:

```
void valueChanged(TreeSelectionEvent event)
```

That method is called whenever the user selects or deselects tree nodes.

You add the listener to the tree in the normal way:

```
tree.addTreeSelectionListener(listener);
```

You can specify whether the user is allowed to select a single node, a contiguous range of nodes, or an arbitrary, potentially discontiguous, set of nodes. The `JTree` class uses a `TreeSelectionModel` to manage node selection. You need to retrieve the model to set the selection state to one of `SINGLE_TREE_SELECTION`, `CONTIGUOUS_TREE_SELECTION`, or `DISCONTIGUOUS_TREE_SELECTION`. (Discontiguous selection mode is the default.) For example, in our class browser, we want to allow selection of only a single class:

```
int mode = TreeSelectionModel.SINGLE_TREE_SELECTION;
tree.getSelectionModel().setSelectionMode(mode);
```

Apart from setting the selection mode, you need not worry about the tree selection model.

Note

How the user selects multiple items depends on the look and feel. In the Metal look and feel, hold down the CTRL key while clicking an item to add the item to the selection, or to remove it if it was currently selected. Hold down the SHIFT key while clicking an item to select a *range* of items, extending from the previously selected item to the new item.

To find out the current selection, you query the tree with the `getSelectionPaths` method:

```
TreePath[] selectedPaths = tree.getSelectionPaths();
```

If you restricted the user to a single selection, you can use the convenience method `getSelectionPath`, which returns the first selected path, or `null` if no path was selected.

Caution

The `TreeSelectionEvent` class has a `getPaths` method that returns an array of `TreePath` objects, but that array describes *selection changes*, not the current selection.

Listing 6-10 shows the complete source code for the class tree program. The program displays inheritance hierarchies, and it customizes the display to show abstract classes in italics. You can type the name of any class into the text field at the bottom of the frame. Press the ENTER key or click the Add button to add the class and its superclasses to the tree. You must enter the full package name, such as `java.util.ArrayList`.

This program is a bit tricky because it uses reflection to construct the class tree. This work is contained inside the `addClass` method. (The details are not that important. We use the class tree in this example because inheritance trees yield a nice supply of trees without laborious coding. If you display trees in your own applications, you will have your own source of hierarchical data.) The method uses the breadth-first search algorithm to find whether the current class is already in the tree by calling the `findUserObject` method that we implemented in the preceding section. If the class is not already in the tree, we add the superclasses to the tree, then make the new class node a child and make that node visible.

When you select a tree node, the text area to the right is filled with the fields of the selected class. In the frame constructor, we restrict the user to single item selection and add a tree selection listener. When the `valueChanged` method is called, we ignore its event parameter and simply ask the tree for the current selection path. As always, we need to get the last node of the path and look up its user object. We then call the `getFieldDescription` method, which uses reflection to assemble a string with all fields of the selected class.

Listing 6-10. **`ClassTree.java`**

Code View:
```
import java.awt.*;
import java.awt.event.*;
import java.lang.reflect.*;
import java.util.*;
import javax.swing.*;
import javax.swing.event.*;
import javax.swing.tree.*;

/**
 * This program demonstrates cell rendering and listening to tree selection events.
 * @version 1.03 2007-08-01
```

```
 * @author Cay Horstmann
 */
public class ClassTree
{
   public static void main(String[] args)
   {
      EventQueue.invokeLater(new Runnable()
         {
            public void run()
```

```
            {
               JFrame frame = new ClassTreeFrame();
               frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
               frame.setVisible(true);
            }
         });
   }
```

```
}

/**
 * This frame displays the class tree, a text field and add button to add more classes
 * into the tree.
 */
class ClassTreeFrame extends JFrame
{
   public ClassTreeFrame()
   {
      setTitle("ClassTree");
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

      // the root of the class tree is Object
      root = new DefaultMutableTreeNode(java.lang.Object.class);
      model = new DefaultTreeModel(root);
      tree = new JTree(model);

      // add this class to populate the tree with some data
      addClass(getClass());

      // set up node icons
      ClassNameTreeCellRenderer renderer = new ClassNameTreeCellRenderer();
      renderer.setClosedIcon(new ImageIcon("red-ball.gif"));
      renderer.setOpenIcon(new ImageIcon("yellow-ball.gif"));
      renderer.setLeafIcon(new ImageIcon("blue-ball.gif"));
      tree.setCellRenderer(renderer);

      // set up selection mode
      tree.addTreeSelectionListener(new TreeSelectionListener()
         {
            public void valueChanged(TreeSelectionEvent event)
            {
               // the user selected a different node--update description
               TreePath path = tree.getSelectionPath();
               if (path == null) return;
               DefaultMutableTreeNode selectedNode = (DefaultMutableTreeNode) path
                     .getLastPathComponent();
               Class<?> c = (Class<?>) selectedNode.getUserObject();
               String description = getFieldDescription(c);
               textArea.setText(description);
            }
         });
```

```
      int mode = TreeSelectionModel.SINGLE_TREE_SELECTION;
      tree.getSelectionModel().setSelectionMode(mode);

      // this text area holds the class description
      textArea = new JTextArea();

      // add tree and text area
      JPanel panel = new JPanel();
      panel.setLayout(new GridLayout(1, 2));
      panel.add(new JScrollPane(tree));
      panel.add(new JScrollPane(textArea));

      add(panel, BorderLayout.CENTER);

      addTextField();
   }

   /**
    * Add the text field and "Add" button to add a new class.
    */
   public void addTextField()
   {
      JPanel panel = new JPanel();

      ActionListener addListener = new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               // add the class whose name is in the text field
               try
               {
                  String text = textField.getText();
                  addClass(Class.forName(text)); // clear text field to indicate success
                  textField.setText("");
               }
               catch (ClassNotFoundException e)
               {
                  JOptionPane.showMessageDialog(null, "Class not found");
               }
            }
         };

      // new class names are typed into this text field
      textField = new JTextField(20);
      textField.addActionListener(addListener);
      panel.add(textField);

      JButton addButton = new JButton("Add");
      addButton.addActionListener(addListener);
      panel.add(addButton);

      add(panel, BorderLayout.SOUTH);
   }

   /**
    * Finds an object in the tree.
    * @param obj the object to find
    * @return the node containing the object or null if the object is not present in the tree
    */
```

```
   @SuppressWarnings("unchecked")
   public DefaultMutableTreeNode findUserObject(Object obj)
   {
      // find the node containing a user object
      Enumeration<TreeNode> e = (Enumeration<TreeNode>) root.breadthFirstEnumeration();
      while (e.hasMoreElements())
      {
         DefaultMutableTreeNode node = (DefaultMutableTreeNode) e.nextElement();
         if (node.getUserObject().equals(obj)) return node;
```



```
      }
      return null;
   }

   /**
    * Adds a new class and any parent classes that aren't yet part of the tree
    * @param c the class to add
```



```
    * @return the newly added node.
    */
   public DefaultMutableTreeNode addClass(Class<?> c)
   {
      // add a new class to the tree

      // skip non-class types
      if (c.isInterface() || c.isPrimitive()) return null;

      // if the class is already in the tree, return its node
      DefaultMutableTreeNode node = findUserObject(c);
      if (node != null) return node;

      // class isn't present--first add class parent recursively

      Class<?> s = c.getSuperclass();

      DefaultMutableTreeNode parent;
      if (s == null) parent = root;
      else parent = addClass(s);

      // add the class as a child to the parent
      DefaultMutableTreeNode newNode = new DefaultMutableTreeNode(c);
      model.insertNodeInto(newNode, parent, parent.getChildCount());

      // make node visible
      TreePath path = new TreePath(model.getPathToRoot(newNode));
      tree.makeVisible(path);

      return newNode;
   }

   /**
    * Returns a description of the fields of a class.
    * @param the class to be described
    * @return a string containing all field types and names
    */
   public static String getFieldDescription(Class<?> c)
   {
      // use reflection to find types and names of fields
      StringBuilder r = new StringBuilder();
      Field[] fields = c.getDeclaredFields();
      for (int i = 0; i < fields.length; i++)
```

```
      {
         Field f = fields[i];
         if ((f.getModifiers() & Modifier.STATIC) != 0) r.append("static ");
         r.append(f.getType().getName());
         r.append(" ");
         r.append(f.getName());
         r.append("\n");
      }
      return r.toString();
   }

   private DefaultMutableTreeNode root;
   private DefaultTreeModel model;
   private JTree tree;
   private JTextField textField;
   private JTextArea textArea;
   private static final int DEFAULT_WIDTH = 400;
   private static final int DEFAULT_HEIGHT = 300;
}

/**
 * This class renders a class name either in plain or italic. Abstract classes are italic.
 */
class ClassNameTreeCellRenderer extends DefaultTreeCellRenderer
{
   public Component getTreeCellRendererComponent(JTree tree, Object value, boolean selected,
         boolean expanded, boolean leaf, int row, boolean hasFocus)
   {
      super.getTreeCellRendererComponent(tree, value, selected, expanded, leaf,
                                         row, hasFocus);
      // get the user object
      DefaultMutableTreeNode node = (DefaultMutableTreeNode) value;
      Class<?> c = (Class<?>) node.getUserObject();

      // the first time, derive italic font from plain font
      if (plainFont == null)
      {
         plainFont = getFont();
         // the tree cell renderer is sometimes called with a label that has a null font
         if (plainFont != null) italicFont = plainFont.deriveFont(Font.ITALIC);
      }

      // set font to italic if the class is abstract, plain otherwise
      if ((c.getModifiers() & Modifier.ABSTRACT) == 0) setFont(plainFont);
      else setFont(italicFont);
      return this;
   }

   private Font plainFont = null;
   private Font italicFont = null;
}
```

API `javax.swing.JTree` 1.2

- `TreePath getSelectionPath()`
- `TreePath[] getSelectionPaths()`

  returns the first selected path, or an array of paths to all selected nodes. If no paths are selected, both methods return `null`.

API `javax.swing.event.TreeSelectionListener` 1.2

- `void valueChanged(TreeSelectionEvent event)`

  is called whenever nodes are selected or deselected.

API `javax.swing.event.TreeSelectionEvent` 1.2

- `TreePath getPath()`
- `TreePath[] getPaths()`

  gets the first path or all paths that have *changed* in this selection event. If you want to know the current selection, not the selection change, you should call `JTree.getSelectionPaths` instead.

Custom Tree Models

In the final example, we implement a program that inspects the contents of an object, just like a debugger does (see Figure 6-34).

Figure 6-34. An object inspection tree

Before going further, compile and run the example program. Each node corresponds to an instance field. If the field is an object, expand it to see *its* instance fields. The program inspects the contents of the frame window. If you poke around a few of the instance fields, you should be able to find some familiar classes. You'll also gain some respect for how complex the Swing user interface components are under the hood.

What's remarkable about the program is that the tree does not use the `DefaultTreeModel`. If you already have data that are hierarchically organized, you might not want to build a duplicate tree and worry about keeping both trees synchronized. That is the situation in our case—the inspected objects are already linked to each other through the object references, so there is no need to replicate the linking structure.

The `TreeModel` interface has only a handful of methods. The first group of methods enables the `JTree` to find the tree nodes by first getting the root, then the children. The `JTree` class calls these methods only when the user actually expands a node.

```
Object getRoot()
int getChildCount(Object parent)
Object getChild(Object parent, int index)
```

This example shows why the `TreeModel` interface, like the `JTree` class itself, does not need an explicit notion of nodes. The root and its children can be any objects. The `TreeModel` is responsible for telling the `JTree` how they are connected.

The next method of the `TreeModel` interface is the reverse of `getChild`:

```
int getIndexOfChild(Object parent, Object child)
```

Actually, this method can be implemented in terms of the first three—see the code in Listing 6-11.

The tree model tells the `JTree` which nodes should be displayed as leaves:

```
boolean isLeaf(Object node)
```

If your code changes the tree model, then the tree needs to be notified so that it can redraw itself. The tree adds itself as a `TreeModelListener` to the model. Thus, the model must support the usual listener management methods:

```
void addTreeModelListener(TreeModelListener l)
void removeTreeModelListener(TreeModelListener l)
```


You can see implementations for these methods in Listing 6-11.

When the model modifies the tree contents, it calls one of the four methods of the `TreeModelListener` interface:


```
void treeNodesChanged(TreeModelEvent e)
void treeNodesInserted(TreeModelEvent e)
void treeNodesRemoved(TreeModelEvent e)
void treeStructureChanged(TreeModelEvent e)
```

The `TreeModelEvent` object describes the location of the change. The details of assembling a tree model event that describes an insertion or removal event are quite technical. You only need to worry about firing these events if your tree can actually have nodes added and removed. In Listing 6-11, we show you how to fire one event: replacing the root with a new object.

Tip

To simplify the code for event firing, we use the `javax.swing.EventListenerList` convenience class that collects listeners. See Volume I, Chapter 8 for more information on this class.

Finally, if the user edits a tree node, your model is called with the change:

```
void valueForPathChanged(TreePath path, Object newValue)
```

If you don't allow editing, this method is never called.

If you don't need to support editing, then constructing a tree model is easily done. Implement the three methods

```
Object getRoot()
int getChildCount(Object parent)
Object getChild(Object parent, int index)
```

These methods describe the structure of the tree. Supply routine implementations of the other five methods, as in Listing 6-11. You are then ready to display your tree.

Now let's turn to the implementation of the example program. Our tree will contain objects of type `Variable`.

Note

Had we used the `DefaultTreeModel`, our nodes would have been objects of type `DefaultMutableTreeNode` with *user objects* of type `Variable`.

For example, suppose you inspect the variable

```
Employee joe;
```

That variable has a *type* `Employee.class`, a *name* `"joe"`, and a *value*, the value of the object reference `joe`. We define a class `Variable` that describes a variable in a program:

```
Variable v = new Variable(Employee.class, "joe", joe);
```

If the type of the variable is a primitive type, you must use an object wrapper for the value.

```
new Variable(double.class, "salary", new Double(salary));
```

If the type of the variable is a class, then the variable has *fields*. Using reflection, we enumerate all fields and collect them in an `ArrayList`. Because the `getFields` method of the `Class` class does not return fields of the superclass, we need to call `getFields` on all superclasses as well. You can find the code in the `Variable` constructor. The `getFields` method of our `Variable` class returns the array of fields. Finally, the `toString` method of the `Variable` class formats the node label. The label always contains the variable type and name. If the variable is not a class, the label also contains the value.

Note

If the type is an array, then we do not display the elements of the array. This would not be difficult to do; we leave it as the proverbial "exercise for the reader."

Let's move on to the tree model. The first two methods are simple.

```
public Object getRoot()
{
   return root;
}
```

```
public int getChildCount(Object parent)
{
   return ((Variable) parent).getFields().size();
}
```

The getChild method returns a new Variable object that describes the field with the given index. The getType and getName method of the Field class yield the field type and name. By using reflection, you can read the field value as f.get(parentValue). That method can throw an IllegalAccessException. However, we made all fields accessible in the Variable constructor, so this won't happen in practice.

Here is the complete code of the getChild method:

```
public Object getChild(Object parent, int index)
{
   ArrayList fields = ((Variable) parent).getFields();
   Field f = (Field) fields.get(index);
   Object parentValue = ((Variable) parent).getValue();
   try
   {
      return new Variable(f.getType(), f.getName(), f.get(parentValue));
   }
   catch (IllegalAccessException e)
   {
      return null;
   }
}
```


These three methods reveal the structure of the object tree to the JTree component. The remaining methods are routine—see the source code in Listing 6-11.

There is one remarkable fact about this tree model: It actually describes an *infinite* tree. You can verify this by following one of the WeakReference objects. Click on the variable named referent. It leads you right back to the original object. You get an identical subtree, and you can open its WeakReference object again, ad infinitum. Of course, you cannot *store* an infinite set of nodes. The tree model simply generates the nodes on demand as the user expands the parents.

This example concludes our discussion on trees. We move on to the table component, another complex Swing component. Superficially, trees and tables don't seem to have much in common, but you will find that they both use the same concepts for data models and cell rendering.

Listing 6-11. **ObjectInspectorTest.java**

Code View:

```
import java.awt.*;
import java.lang.reflect.*;
import java.util.*;
import javax.swing.*;
import javax.swing.event.*;
import javax.swing.tree.*;

/**
 * This program demonstrates how to use a custom tree model. It displays the fields of
 * an object.
 * @version 1.03 2007-08-01
```

```
 * @author Cay Horstmann
 */
public class ObjectInspectorTest
{
   public static void main(String[] args)
   {
      EventQueue.invokeLater(new Runnable()
         {
            public void run()
            {
               JFrame frame = new ObjectInspectorFrame();
               frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
               frame.setVisible(true);
            }
         });
   }
}

/**
 * This frame holds the object tree.
 */
class ObjectInspectorFrame extends JFrame
{
   public ObjectInspectorFrame()
   {
      setTitle("ObjectInspectorTest");
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

      // we inspect this frame object

      Variable v = new Variable(getClass(), "this", this);
      ObjectTreeModel model = new ObjectTreeModel();
      model.setRoot(v);

      // construct and show tree

      tree = new JTree(model);
      add(new JScrollPane(tree), BorderLayout.CENTER);
   }

   private JTree tree;
   private static final int DEFAULT_WIDTH = 400;
   private static final int DEFAULT_HEIGHT = 300;
}

/**
 * This tree model describes the tree structure of a Java object. Children are the objects
 * that are stored in instance variables.
 */
class ObjectTreeModel implements TreeModel
{
   /**
    * Constructs an empty tree.
    */
   public ObjectTreeModel()
   {
      root = null;
   }

```

```
   /**
    * Sets the root to a given variable.
    * @param v the variable that is being described by this tree
    */
   public void setRoot(Variable v)
   {
      Variable oldRoot = v;
      root = v;
      fireTreeStructureChanged(oldRoot);
```

```
   }

   public Object getRoot()
   {
      return root;
   }
```

```
   public int getChildCount(Object parent)
   {
      return ((Variable) parent).getFields().size();
   }

   public Object getChild(Object parent, int index)
   {
      ArrayList<Field> fields = ((Variable) parent).getFields();
      Field f = (Field) fields.get(index);
      Object parentValue = ((Variable) parent).getValue();
      try
      {
         return new Variable(f.getType(), f.getName(), f.get(parentValue));
      }
      catch (IllegalAccessException e)
      {
         return null;
      }
   }

   public int getIndexOfChild(Object parent, Object child)
   {
      int n = getChildCount(parent);
      for (int i = 0; i < n; i++)
         if (getChild(parent, i).equals(child)) return i;
      return -1;
   }

   public boolean isLeaf(Object node)
   {
      return getChildCount(node) == 0;
   }

   public void valueForPathChanged(TreePath path, Object newValue)
   {
   }

   public void addTreeModelListener(TreeModelListener l)
   {
      listenerList.add(TreeModelListener.class, l);
   }

   public void removeTreeModelListener(TreeModelListener l)
```

```
   {
      listenerList.remove(TreeModelListener.class, l);
   }

   protected void fireTreeStructureChanged(Object oldRoot)
   {
      TreeModelEvent event = new TreeModelEvent(this, new Object[] { oldRoot });
      EventListener[] listeners = listenerList.getListeners(TreeModelListener.class);
      for (int i = 0; i < listeners.length; i++)
         ((TreeModelListener) listeners[i]).treeStructureChanged(event);
   }

   private Variable root;
   private EventListenerList listenerList = new EventListenerList();
}

/**
 * A variable with a type, name, and value.
 */
class Variable
{
   /**
    * Construct a variable
    * @param aType the type
    * @param aName the name
    * @param aValue the value
    */
   public Variable(Class<?> aType, String aName, Object aValue)
   {
      type = aType;
      name = aName;
      value = aValue;
      fields = new ArrayList<Field>();

      // find all fields if we have a class type except we don't expand strings and null values

      if (!type.isPrimitive() && !type.isArray() && !type.equals(String.class) && value != null
      {
         // get fields from the class and all superclasses
         for (Class<?> c = value.getClass(); c != null; c = c.getSuperclass())
         {
            Field[] fs = c.getDeclaredFields();
            AccessibleObject.setAccessible(fs, true);

            // get all nonstatic fields
            for (Field f : fs)
               if ((f.getModifiers() & Modifier.STATIC) == 0) fields.add(f);
         }
      }
   }

   /**
    * Gets the value of this variable.
    * @return the value
    */
   public Object getValue()
   {
      return value;
   }
```

```

   /**
    * Gets all nonstatic fields of this variable.
    * @return an array list of variables describing the fields
    */
   public ArrayList<Field> getFields()
   {
      return fields;
   }
```



```

   public String toString()
   {
      String r = type + " " + name;
      if (type.isPrimitive()) r += "=" + value;
      else if (type.equals(String.class)) r += "=" + value;
      else if (value == null) r += "=null";
      return r;
   }

   private Class<?> type;
   private String name;
   private Object value;
   private ArrayList<Field> fields;
}
```


API

`javax.swing.tree.TreeModel` 1.2

- `Object getRoot()`

  returns the root node.

- `int getChildCount(Object parent)`

  gets the number of children of the `parent` node.

- `Object getChild(Object parent, int index)`

  gets the child node of the `parent` node at the given index.

- `int getIndexOfChild(Object parent, Object child)`

  gets the index of the `child` node in the `parent` node, or -1 if `child` is not a child of `parent` in this tree model.

- `boolean isLeaf(Object node)`

  returns `true` if `node` is conceptually a leaf of the tree.

- `void addTreeModelListener(TreeModelListener l)`

- `void removeTreeModelListener(TreeModelListener l)`

  adds or removes listeners that are notified when the information in the tree model changes.

- `void valueForPathChanged(TreePath path, Object newValue)`

  is called when a cell editor has modified the value of a node.

Parameters:	`path`	The path to the node that has been edited
	`newValue`	The replacement value returned by the editor

API `javax.swing.event.TreeModelListener` 1.2

- `void treeNodesChanged(TreeModelEvent e)`

- `void treeNodesInserted(TreeModelEvent e)`

- `void treeNodesRemoved(TreeModelEvent e)`

- `void treeStructureChanged(TreeModelEvent e)`

  is called by the tree model when the tree has been modified.

API javax.swing.event.TreeModelEvent 1.2

- `TreeModelEvent(Object eventSource, TreePath node)`

  constructs a tree model event.

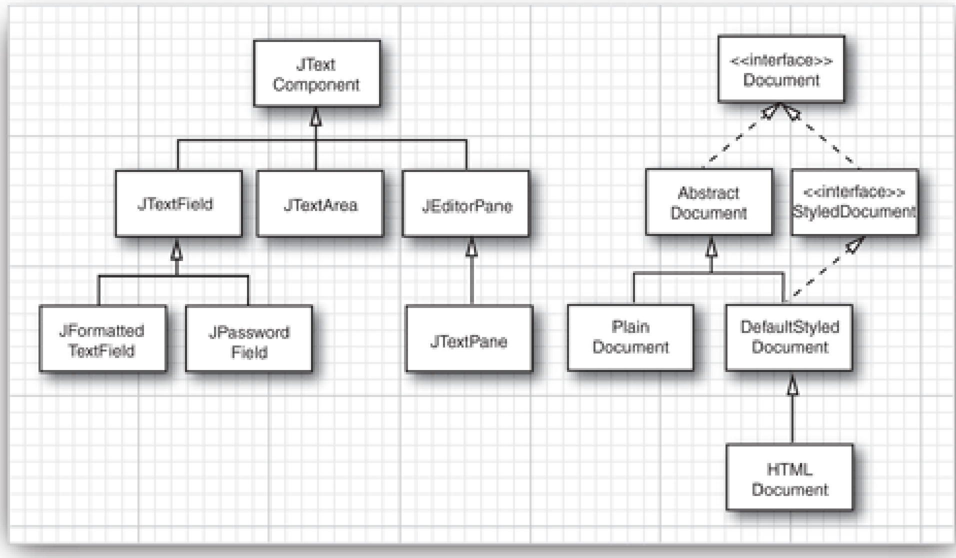
Parameters:	`eventSource`	The tree model generating this event
	`node`	The path to the node that is being changed

Text Components

Figure 6-35 shows all text components that are included in the Swing library. You already saw the three most commonly used components, `JTextField`, `JPasswordField`, and `JTextArea`, in Volume I, Chapter 9. In the following sections, we introduce the remaining text components. We also discuss the `JSpinner` component that contains a formatted text field together with tiny "up" and "down" buttons to change its contents.

Figure 6-35. The hierarchy of text components and documents

[View full size image]



All text components render and edit data that are stored in a model object of a class implementing the `Document` interface. The `JTextField` and `JTextArea` components use a `PlainDocument` that simply stores a sequence of lines of plain text without any formatting.

A `JEditorPane` can show and edit styled text (with fonts, colors, etc.) in a variety of formats, most notably HTML; see the "Displaying HTML with the JEditorPane" section beginning on page 472. The `StyledDocument` interface describes the additional requirements of styles, fonts, and colors. The `HTMLDocument` class implements this interface.

The subclass `JTextPane` of `JEditorPane` also holds styled text as well as embedded Swing components. We do no cover the very complex `JTextPane` in this book but instead refer you to the very detailed description in *Core Swing: Advanced Programming* by Kim Topley. For a typical use of the `JTextPane` class, have a look at the StylePad demo that is included in the JDK.

Change Tracking in Text Components

Most of the intricacies of the `Document` interface are of interest only if you implement your own text editor. There is, however, one common use of the interface: for tracking changes.

Sometimes, you want to update a part of your user interface whenever a user edits text, without waiting for the user to click a button. Here is a simple example. We show three text fields for the red, blue, and green component of a color. Whenever the content of the text fields changes, the color should be updated. Figure 6-36 shows the running application of Listing 6-12.

Figure 6-36. Tracking changes in a text field

First of all, note that it is not a good idea to monitor keystrokes. Some keystrokes (such as the arrow keys) don't change the text. More important, the text can be updated by mouse gestures (such as "middle mouse button pasting" in X11). Instead, you should ask the *document* (and not the text component) to notify you whenever the data have changed, by installing a *document listener:*

```
textField.getDocument().addDocumentListener(listener);
```

When the text has changed, one of the following `DocumentListener` methods is called:

```
void insertUpdate(DocumentEvent event)
void removeUpdate(DocumentEvent event)
void changedUpdate(DocumentEvent event)
```

The first two methods are called when characters have been inserted or removed. The third method is not called a all for text fields. For more complex document types, it would be called when some other change, such as a change in formatting, has occurred. Unfortunately, there is no single callback to tell you that the text has changed—usually you don't much care how it has changed. There is no adapter class, either. Thus, your document listener must implement all three methods. Here is what we do in our sample program:

```
DocumentListener listener = new DocumentListener()
   {
      public void insertUpdate(DocumentEvent event) { setColor(); }
      public void removeUpdate(DocumentEvent event) { setColor(); }
      public void changedUpdate(DocumentEvent event) {}
   }
```

The `setColor` method uses the `getText` method to obtain the current user input strings from the text fields and sets the color.

Our program has one limitation. Users can type malformed input, such as `"twenty"`, into the text field or leave a field blank. For now, we catch the `NumberFormatException` that the `parseInt` method throws, and we simply don' update the color when the text field entry is not a number. In the next section, you see how you can prevent the user from entering invalid input in the first place.

Note

Instead of listening to document events, you can also add an action event listener to a text field. The action listener is notified whenever the user presses the ENTER key. We don't recommend this approach, because users don't always remember to press ENTER when they are done entering data. If you use an action listener, you should also install a focus listener so that you can track when the user leaves the text field.

Listing 6-12. **ChangeTrackingTest.java**

Code View:
```
import java.awt.*;
import javax.swing.*;
import javax.swing.event.*;

/**
 * @version 1.40 2007-08-05
 * @author Cay Horstmann
 */
public class ChangeTrackingTest
{
   public static void main(String[] args)
   {
      EventQueue.invokeLater(new Runnable()
         {
            public void run()
            {
               ColorFrame frame = new ColorFrame();
               frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
               frame.setVisible(true);
            }
         });
   }
}

/**
 * A frame with three text fields to set the background color.
 */
class ColorFrame extends JFrame
{
   public ColorFrame()
   {
      setTitle("ChangeTrackingTest");

      DocumentListener listener = new DocumentListener()
         {
            public void insertUpdate(DocumentEvent event)
            {
               setColor();
            }

            public void removeUpdate(DocumentEvent event)
            {
               setColor();
            }
```

```

            public void changedUpdate(DocumentEvent event)
            {
            }
         };

      panel = new JPanel();

      panel.add(new JLabel("Red:"));
```

```
      redField = new JTextField("255", 3);
      panel.add(redField);
      redField.getDocument().addDocumentListener(listener);

      panel.add(new JLabel("Green:"));
      greenField = new JTextField("255", 3);
      panel.add(greenField);
```

```
      greenField.getDocument().addDocumentListener(listener);

      panel.add(new JLabel("Blue:"));
      blueField = new JTextField("255", 3);
      panel.add(blueField);
      blueField.getDocument().addDocumentListener(listener);

      add(panel);
      pack();
  }

  /**
   * Set the background color to the values stored in the text fields.
   */
  public void setColor()
  {
     try
     {
        int red = Integer.parseInt(redField.getText().trim());
        int green = Integer.parseInt(greenField.getText().trim());
        int blue = Integer.parseInt(blueField.getText().trim());
        panel.setBackground(new Color(red, green, blue));
     }
     catch (NumberFormatException e)
     {
        // don't set the color if the input can't be parsed
     }
  }

  private JPanel panel;
  private JTextField redField;
  private JTextField greenField;
  private JTextField blueField;
}
```

API `javax.swing.JComponent` 1.2

- `Dimension getPreferredSize()`
- `void setPreferredSize(Dimension d)`

  gets or sets the preferred size of this component.

API `javax.swing.text.Document` 1.2

- `int getLength()`

  returns the number of characters currently in the document.

- `String getText(int offset, int length)`

  returns the text contained within the given portion of the document.

Parameters:	`offset`	The start of the text
	`length`	The length of the desired string

- `void addDocumentListener(DocumentListener listener)`

  registers the listener to be notified when the document changes.

API `javax.swing.event.DocumentEvent` 1.2

- `Document getDocument()`

  gets the document that is the source of the event.

API javax.swing.event.DocumentListener 1.2

- `void changedUpdate(DocumentEvent event)`

  is called whenever an attribute or set of attributes changes.

- `void insertUpdate(DocumentEvent event)`

  is called whenever an insertion into the document occurs.

- `void removeUpdate(DocumentEvent event)`

  is called whenever a portion of the document has been removed.

Formatted Input Fields

In the last example program, we wanted the program user to type numbers, not arbitrary strings. That is, the use is allowed to enter only digits 0 through 9 and a hyphen (-). The hyphen, if present at all, must be the *first* symbo of the input string.

On the surface, this input validation task sounds simple. We can install a key listener to the text field and then consume all key events that aren't digits or a hyphen. Unfortunately, this simple approach, although commonly recommended as a method for input validation, does not work well in practice. First, not every combination of the valid input characters is a valid number. For example, `--3` and `3-3` aren't valid, even though they are made up from valid input characters. But, more important, there are other ways of changing the text that don't involve typing character keys. Depending on the look and feel, certain key combinations can be used to cut, copy, and paste text. For example, in the Metal look and feel, the CTRL+V key combination pastes the content of the paste buffer into the text field. That is, we also need to monitor that the user doesn't paste in an invalid character. Clearly, trying to filter keystrokes to ensure that the content of the text field is always valid begins to look like a real chore. This is certainly not something that an application programmer should have to worry about.

Perhaps surprisingly, before Java SE 1.4, there were no components for entering numeric values. Starting with the first edition of Core Java, we supplied an implementation for an `IntTextField`, a text field for entering a properly formatted integer. In every new edition, we changed the implementation to take whatever limited advantage we could from the various half-baked validation schemes that were added to each version of Java. Finally, in Java SE 1.4, the Swing designers faced the issues head-on and supplied a versatile `JFormattedTextField` class that can be used not just for numeric input, but also for dates and for even more esoteric formatted values such as IP addresses.

Integer Input

Let's get started with an easy case: a text field for integer input.

Code View:
```
JFormattedTextField intField = new JFormattedTextField(NumberFormat.getIntegerInstance());
```

The `NumberFormat.getIntegerInstance` returns a formatter object that formats integers, using the current locale. In the U.S. locale, commas are used as decimal separators, allowing users to enter values such as 1,729. Chapter 5 explains in detail how you can select other locales.

As with any text field, you can set the number of columns:

```
intField.setColumns(6);
```

You can set a default value with the `setValue` method. That method takes an `Object` parameter, so you'll need to wrap the default `int` value in an `Integer` object:

```
intField.setValue(new Integer(100));
```

Typically, users will supply inputs in multiple text fields and then click a button to read all values. When the button is clicked, you can get the user-supplied value with the `getValue` method. That method returns an `Object` result, and you need to cast it into the appropriate type. The `JFormattedTextField` returns an object of type `Long` if the user edited the value. However, if the user made no changes, the original `Integer` object is returned. Therefore, you should cast the return value to the common superclass `Number`:

```
Number value = (Number) intField.getValue();
int v = value.intValue();
```

The formatted text field is not very interesting until you consider what happens when a user provides illegal input. That is the topic of the next section.

Behavior on Loss of Focus

Consider what happens when a user supplies input to a text field. The user types input and eventually decides to leave the field, perhaps by clicking on another component with the mouse. Then the text field *loses focus*. The I-beam cursor is no longer visible in the text field, and keystrokes are directed toward a different component.

When the formatted text field loses focus, the formatter looks at the text string that the user produced. If the formatter knows how to convert the text string to an object, the text is valid. Otherwise it is invalid. You can use the `isEditValid` method to check whether the current content of the text field is valid.

The default behavior on loss of focus is called "commit or revert." If the text string is valid, it is *committed.* The formatter converts it to an object. That object becomes the current value of the field (that is, the return value of the `getValue` method that you saw in the preceding section). The value is then converted back to a string, which becomes the text string that is visible in the field. For example, the integer formatter recognizes the input `1729` as valid, sets the current value to `new Long(1729)`, and then converts it back into a string with a decimal comma: `1,729`.

Conversely, if the text string is invalid, then the current value is not changed and the text field *reverts* to the string that represents the old value. For example, if the user enters a bad value, such as `x1`, then the old value is restored when the text field loses focus.

Note

The integer formatter regards a text string as valid if it starts with an integer. For example, `1729x` is a valid string. It is converted to the number 1729, which is then formatted as the string `1,729`.

You can set other behaviors with the `setFocusLostBehavior` method. The "commit" behavior is subtly different from the default. If the text string is invalid, then both the text string and the field value stay unchanged—they are now out of sync. The "persist" behavior is even more conservative. Even if the text string is valid, neither the text field nor the current value are changed. You would need to call `commitEdit`, `setValue`, or `setText` to bring them back in sync. Finally, there is a "revert" behavior that doesn't ever seem to be useful. Whenever focus is lost, the user input is disregarded, and the text string reverts to the old value.

Note

Generally, the "commit or revert" default behavior is reasonable. There is just one potential problem. Suppose a dialog box contains a text field for an integer value. A user enters a string `" 1729"`, with a leading space and then clicks the OK button. The leading space makes the number invalid, and the field value reverts to the old value. The action listener of the OK button retrieves the field value and closes the dialog box. The user never knows that the new value has been rejected. In this situation, it is appropriate to select the "commit" behavior and have the OK button listener check that all field edits are valid before closing the dialog box.

Filters

The basic functionality of formatted text fields is straightforward and sufficient for most uses. However, you can add a couple of refinements. Perhaps you want to prevent the user from entering nondigits altogether. You achieve that behavior with a *document filter*. Recall that in the model-view-controller architecture, the controller translates input events into commands that modify the underlying document of the text field; that is, the text string that is stored in a `PlainDocument` object. For example, whenever the controller processes a command that causes text to be inserted into the document, it calls the "insert string" command. The string to be inserted can be either a single character or the content of the paste buffer. A document filter can intercept this command and modify the string or cancel the insertion altogether. Here is the code for the `insertString` method of a filter that analyzes the string to be inserted and inserts only the characters that are digits or a - sign. (The code handles supplementary Unicode characters, as explained in Chapter 3. See Chapter 12 for the `StringBuilder` class.)

Code View:
```
public void insertString(FilterBypass fb, int offset, String string, AttributeSet attr)
   throws BadLocationException
{
   StringBuilder builder = new StringBuilder(string);
   for (int i = builder.length() - 1; i >= 0; i--)
   {
      int cp = builder.codePointAt(i);
      if (!Character.isDigit(cp) && cp != '-')
      {
         builder.deleteCharAt(i);
```

```
         if (Character.isSupplementaryCodePoint(cp))
         {
            i--;
            builder.deleteCharAt(i);
         }
      }
   }
   super.insertString(fb, offset, builder.toString(), attr);
}
```

You should also override the replace method of the DocumentFilter class—it is called when text is selected and then replaced. The implementation of the replace method is straightforward—see Listing 6-13.

Now you need to install the document filter. Unfortunately, there is no straightforward method to do that. You need to override the getDocumentFilter method of a formatter class, and pass an object of that formatter class to the JFormattedTextField. The integer text field uses an InternationalFormatter that is initialized with NumberFormat.getIntegerInstance(). Here is how you install a formatter to yield the desired filter:

```
JFormattedTextField intField = new JFormattedTextField(new
      InternationalFormatter(NumberFormat.getIntegerInstance())
      {
         protected DocumentFilter getDocumentFilter()
         {
            return filter;
         }
         private DocumentFilter filter = new IntFilter();
      });
```

Note

The Java SE documentation states that the DocumentFilter class was invented to avoid subclassing. Until Java SE 1.3, filtering in a text field was achieved by extending the PlainDocument class and overriding the insertString and replace methods. Now the PlainDocument class has a pluggable filter instead. That is a splendid improvement. It would have been even more splendid if the filter had also been made pluggable in the formatter class. Alas, it was not, and we must subclass the formatter.

Try out the FormatTest example program at the end of this section. The third text field has a filter installed. You can insert only digits or the minus (-) character. Note that you can still enter invalid strings such as "1-2-3". In general, it is impossible to avoid all invalid strings through filtering. For example, the string "-" is invalid, but a filter can't reject it because it is a prefix of a legal string "-1". Even though filters can't give perfect protection, it makes sense to use them to reject inputs that are obviously invalid.

Tip

Another use for filtering is to turn all characters of a string to upper case. Such a filter is easy to write. In the `insertString` and `replace` methods of the filter, convert the string to be inserted to upper case and then invoke the superclass method.

Verifiers

There is another potentially useful mechanism to alert users to invalid inputs. You can attach a *verifier* to any `JComponent`. If the component loses focus, then the verifier is queried. If the verifier reports the content of the component to be invalid, the component immediately regains focus. The user is thus forced to fix the content before supplying other inputs.

A verifier must extend the abstract `InputVerifier` class and define a `verify` method. It is particularly easy to define a verifier that checks formatted text fields. The `isEditValid` method of the `JFormattedTextField` class calls the formatter and returns `true` if the formatter can turn the text string into an object. Here is the verifier:

```
class FormattedTextFieldVerifier extends InputVerifier
{
   public boolean verify(JComponent component)
   {
      JFormattedTextField field = (JFormattedTextField) component;
      return field.isEditValid();
   }
}
```

You can attach it to any `JFormattedTextField`:

```
intField.setInputVerifier(new FormattedTextFieldVerifier());
```

However, a verifier is not entirely foolproof. If you click on a button, then the button notifies its action listeners before an invalid component regains focus. The action listeners can then get an invalid result from the component that failed verification. There is a reason for this behavior: Users might want to click a Cancel button without first having to fix an invalid input.

The fourth text field in the example program has a verifier attached. Try entering an invalid number (such as `x1729`) and press the TAB key or click with the mouse on another text field. Note that the field immediately regains focus. However, if you click the OK button, the action listener calls `getValue`, which reports the last good value.

Other Standard Formatters

Besides the integer formatter, the `JFormattedTextField` supports several other formatters. The `NumberFormat` class has static methods

```
getNumberInstance
getCurrencyInstance
getPercentInstance
```

that yield formatters of floating-point numbers, currency values, and percentages. For example, you can obtain a text field for the input of currency values by calling

Code View:
```
JFormattedTextField currencyField = new JFormattedTextField(NumberFormat.getCurrencyInstance());
```

To edit dates and times, call one of the static methods of the `DateFormat` class:

```
getDateInstance
getTimeInstance
getDateTimeInstance
```

For example,

Code View:
```
JFormattedTextField dateField = new JFormattedTextField(DateFormat.getDateInstance());
```

This field edits a date in the default or "medium" format such as

```
Aug 5, 2007
```

You can instead choose a "short" format such as

```
8/5/07
```

by calling

```
DateFormat.getDateInstance(DateFormat.SHORT)
```

Note

By default, the date format is "lenient." That is, an invalid date such as February 31, 2002, is rolled over to the next valid date, March 3, 2002. That behavior might be surprising to your users. In that case, call `setLenient(false)` on the `DateFormat` object.

The `DefaultFormatter` can format objects of any class that has a constructor with a string parameter and a matching `toString` method. For example, the `URL` class has a `URL(String)` constructor that can be used to construct a URL from a string, such as

```
URL url = new URL("http://java.sun.com");
```

Therefore, you can use the `DefaultFormatter` to format `URL` objects. The formatter calls `toString` on the field value to initialize the field text. When the field loses focus, the formatter constructs a new object of the same class as the current value, using the constructor with a `String` parameter. If that constructor throws an exception, then the edit is not valid. You can try that out in the example program by entering a URL that does not start with a prefix such as `"http:"`.

Note

By default, the `DefaultFormatter` is in *overwrite mode.* That is different from the other formatters and not very useful. Call `setOverwriteMode(false)` to turn off overwrite mode.

Finally, the `MaskFormatter` is useful for fixed-size patterns that contain some constant and some variable characters. For example, Social Security numbers (such as 078-05-1120) can be formatted with a

```
new MaskFormatter("###-##-####")
```

The # symbol denotes a single digit. Table 6-3 shows the symbols that you can use in a mask formatter.

Table 6-3. **MaskFormatter** Symbols

Symbol	Explanation
`#`	A digit
`?`	A letter
`U`	A letter, converted to upper case
`L`	A letter, converted to lower case
`A`	A letter or digit
`H`	A hexadecimal digit `[0-9A-Fa-f]`
`*`	Any character
`'`	Escape character to include a symbol in the pattern

You can restrict the characters that can be typed into the field by calling one of the methods of the `MaskFormatter` class:

```
setValidCharacters
setInvalidCharacters
```

For example, to read in a letter grade (such as A+ or F), you could use

```
MaskFormatter formatter = new MaskFormatter("U*");
formatter.setValidCharacters("ABCDF+- ");
```

However, there is no way of specifying that the second character cannot be a letter.

Note that the string that is formatted by the mask formatter has exactly the same length as the mask. If the user erases characters during editing, then they are replaced with the *placeholder character*. The default placeholder character is a space, but you can change it with the `setPlaceholderCharacter` method, for example,

```
formatter.setPlaceholderCharacter('0');
```

By default, a mask formatter is in overtype mode, which is quite intuitive—try it out in the example program. Also note that the caret position jumps over the fixed characters in the mask.

The mask formatter is very effective for rigid patterns such as Social Security numbers or American telephone numbers. However, note that no variation at all is permitted in the mask pattern. For example, you cannot use a mask formatter for international telephone numbers that have a variable number of digits.

Custom Formatters

If none of the standard formatters is appropriate, it is fairly easy to define your own formatter. Consider 4-byte IP addresses such as

```
130.65.86.66
```

You can't use a `MaskFormatter` because each byte might be represented by one, two, or three digits. Also, we want to check in the formatter that each byte's value is at most 255.

To define your own formatter, extend the `DefaultFormatter` class and override the methods

```
String valueToString(Object value)
Object stringToValue(String text)
```

The first method turns the field value into the string that is displayed in the text field. The second method parses the text that the user typed and turns it back into an object. If either method detects an error, it should throw a `ParseException`.

In our example program, we store an IP address in a `byte[]` array of length 4. The `valueToString` method forms a string that separates the bytes with periods. Note that `byte` values are signed quantities between -128 and 127. (For example, in an IP address 130.65.86.66, the first octet is actually the byte with value -126.) To turn negative byte values into unsigned integer values, you add 256.

```
public String valueToString(Object value) throws ParseException
{
   if (!(value instanceof byte[]))
      throw new ParseException("Not a byte[]", 0);
   byte[] a = (byte[]) value;
   if (a.length != 4)
       throw new ParseException("Length != 4", 0);
   StringBuilder builder = new StringBuilder();
   for (int i = 0; i < 4; i++)
   {
      int b = a[i];
      if (b < 0) b += 256;
```

```
      builder.append(String.valueOf(b));
      if (i < 3) builder.append('.');
   }
   return builder.toString();
}
```

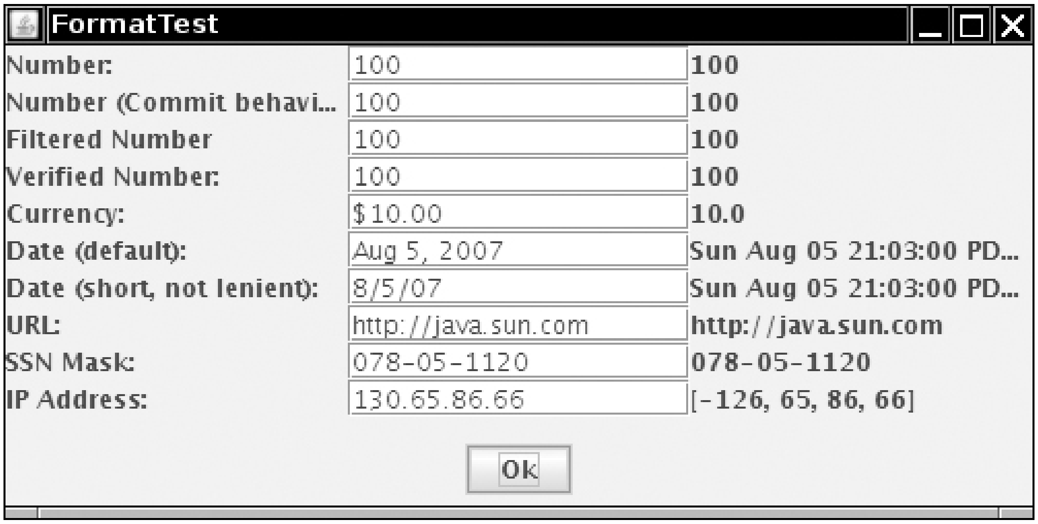
Conversely, the `stringToValue` method parses the string and produces a `byte[]` object if the string is valid. If not, it throws a `ParseException`.


```
public Object stringToValue(String text) throws ParseException
{
   StringTokenizer tokenizer = new StringTokenizer(text, ".");
   byte[] a = new byte[4];
   for (int i = 0; i < 4; i++)
   {
      int b = 0;
      try
      {
         b = Integer.parseInt(tokenizer.nextToken());
      }
      catch (NumberFormatException e)
      {
         throw new ParseException("Not an integer", 0);
      }
      if (b < 0 || b >= 256)
         throw new ParseException("Byte out of range", 0);
      a[i] = (byte) b;
   }
   return a;
}
```


Try out the IP address field in the sample program. If you enter an invalid address, the field reverts to the last valid address.

The program in Listing 6-13 shows various formatted text fields in action (see Figure 6-37). Click the Ok button to retrieve the current values from the fields.

Figure 6-37. The **FormatTest** program



Note

The "Swing Connection" online newsletter has a short article describing a formatter that matches any regular expression. See http://java.sun.com/products/jfc/tsc/articles/reftf/.

Listing 6-13. **FormatTest.java**

Code View:

```
import java.awt.*;
import java.awt.event.*;
import java.net.*;
import java.text.*;
import java.util.*;
import javax.swing.*;
import javax.swing.text.*;

/**
 * A program to test formatted text fields
 * @version 1.02 2007-06-12
 * @author Cay Horstmann
 */
public class FormatTest
{
   public static void main(String[] args)
   {
      EventQueue.invokeLater(new Runnable()
         {
            public void run()
            {
               FormatTestFrame frame = new FormatTestFrame();
               frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

```java
               frame.setVisible(true);
            }
         });
   }
}

/**
 * A frame with a collection of formatted text fields and a button that displays the
 * field values.
```

```java
 */
class FormatTestFrame extends JFrame
{
   public FormatTestFrame()
   {
      setTitle("FormatTest");
      setSize(WIDTH, HEIGHT);
```

```java

      JPanel buttonPanel = new JPanel();
      okButton = new JButton("Ok");
      buttonPanel.add(okButton);
      add(buttonPanel, BorderLayout.SOUTH);

      mainPanel = new JPanel();
      mainPanel.setLayout(new GridLayout(0, 3));
      add(mainPanel, BorderLayout.CENTER);

      JFormattedTextField intField =
         new JFormattedTextField(NumberFormat.getIntegerInstance());
      intField.setValue(new Integer(100));
      addRow("Number:", intField);

      JFormattedTextField intField2 =
         new JFormattedTextField(NumberFormat.getIntegerInstance());
      intField2.setValue(new Integer(100));
      intField2.setFocusLostBehavior(JFormattedTextField.COMMIT);
      addRow("Number (Commit behavior):", intField2);

      JFormattedTextField intField3 = new JFormattedTextField(new InternationalFormatter(
            NumberFormat.getIntegerInstance())
         {
            protected DocumentFilter getDocumentFilter()
            {
               return filter;
            }

            private DocumentFilter filter = new IntFilter();
         });
      intField3.setValue(new Integer(100));
      addRow("Filtered Number", intField3);

      JFormattedTextField intField4 =
         new JFormattedTextField(NumberFormat.getIntegerInstance());
      intField4.setValue(new Integer(100));
      intField4.setInputVerifier(new FormattedTextFieldVerifier());
      addRow("Verified Number:", intField4);

      JFormattedTextField currencyField = new JFormattedTextField(NumberFormat
            .getCurrencyInstance());
      currencyField.setValue(new Double(10));
```

```
      addRow("Currency:", currencyField);

      JFormattedTextField dateField = new JFormattedTextField(DateFormat.getDateInstance());
      dateField.setValue(new Date());
      addRow("Date (default):", dateField);

      DateFormat format = DateFormat.getDateInstance(DateFormat.SHORT);
      format.setLenient(false);
      JFormattedTextField dateField2 = new JFormattedTextField(format);
      dateField2.setValue(new Date());
      addRow("Date (short, not lenient):", dateField2);

      try
      {
         DefaultFormatter formatter = new DefaultFormatter();
         formatter.setOverwriteMode(false);
         JFormattedTextField urlField = new JFormattedTextField(formatter);
         urlField.setValue(new URL("http://java.sun.com"));
         addRow("URL:", urlField);
      }
      catch (MalformedURLException e)
      {
         e.printStackTrace();
      }

      try
      {
         MaskFormatter formatter = new MaskFormatter("###-##-####");
         formatter.setPlaceholderCharacter('0');
         JFormattedTextField ssnField = new JFormattedTextField(formatter);
         ssnField.setValue("078-05-1120");
         addRow("SSN Mask:", ssnField);
      }
      catch (ParseException exception)
      {
         exception.printStackTrace();
      }

      JFormattedTextField ipField = new JFormattedTextField(new IPAddressFormatter());
      ipField.setValue(new byte[] { (byte) 130, 65, 86, 66 });
      addRow("IP Address:", ipField);
   }

   /**
    * Adds a row to the main panel.
    * @param labelText the label of the field
    * @param field the sample field
    */
   public void addRow(String labelText, final JFormattedTextField field)
   {
      mainPanel.add(new JLabel(labelText));
      mainPanel.add(field);
      final JLabel valueLabel = new JLabel();
      mainPanel.add(valueLabel);
      okButton.addActionListener(new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               Object value = field.getValue();
```

UNREGISTERED VERSION OF CHM TO PDF CONVERTER PRO BY THETA-SOFTWARE

```
               Class<?> cl = value.getClass();
               String text = null;
               if (cl.isArray())
               {
                  if (cl.getComponentType().isPrimitive())
                  {
                     try
                     {
                        text = Arrays.class.getMethod("toString", cl).invoke(null, value)
                              .toString();
                     }
                     catch (Exception ex)
                     {
                        // ignore reflection exceptions
                     }
                  }
                  else text = Arrays.toString((Object[]) value);
               }
               else text = value.toString();
               valueLabel.setText(text);
            }
         });
   }

   public static final int WIDTH = 500;
   public static final int HEIGHT = 250;

   private JButton okButton;
   private JPanel mainPanel;
}

/**
 * A filter that restricts input to digits and a '-' sign.
 */
class IntFilter extends DocumentFilter
{
   public void insertString(FilterBypass fb, int offset, String string, AttributeSet attr)
         throws BadLocationException
   {
      StringBuilder builder = new StringBuilder(string);
      for (int i = builder.length() - 1; i >= 0; i--)
      {
         int cp = builder.codePointAt(i);
         if (!Character.isDigit(cp) && cp != '-')
         {
            builder.deleteCharAt(i);
            if (Character.isSupplementaryCodePoint(cp))
            {
               i--;
               builder.deleteCharAt(i);
            }
         }
      }
      super.insertString(fb, offset, builder.toString(), attr);
   }

   public void replace(FilterBypass fb, int offset, int length, String string,
                       AttributeSet attr)
         throws BadLocationException
```

```
   {
      if (string != null)
      {
         StringBuilder builder = new StringBuilder(string);
         for (int i = builder.length() - 1; i >= 0; i--)
         {
            int cp = builder.codePointAt(i);
            if (!Character.isDigit(cp) && cp != '-')
            {
               builder.deleteCharAt(i);
               if (Character.isSupplementaryCodePoint(cp))
               {
                  i--;
                  builder.deleteCharAt(i);
               }
            }
         }
         string = builder.toString();
      }
      super.replace(fb, offset, length, string, attr);
   }
}

/**
 * A verifier that checks whether the content of a formatted text field is valid.
 */
class FormattedTextFieldVerifier extends InputVerifier
{
   public boolean verify(JComponent component)
   {
      JFormattedTextField field = (JFormattedTextField) component;
      return field.isEditValid();
   }
}

/**
 * A formatter for 4-byte IP addresses of the form a.b.c.d
 */
class IPAddressFormatter extends DefaultFormatter
{
   public String valueToString(Object value) throws ParseException
   {
      if (!(value instanceof byte[])) throw new ParseException("Not a byte[]", 0);
      byte[] a = (byte[]) value;
      if (a.length != 4) throw new ParseException("Length != 4", 0);
      StringBuilder builder = new StringBuilder();
      for (int i = 0; i < 4; i++)
      {
         int b = a[i];
         if (b < 0) b += 256;
         builder.append(String.valueOf(b));
         if (i < 3) builder.append('.');
      }
      return builder.toString();
   }

   public Object stringToValue(String text) throws ParseException
   {
      StringTokenizer tokenizer = new StringTokenizer(text, ".");
```

```
      byte[] a = new byte[4];
      for (int i = 0; i < 4; i++)
      {
         int b = 0;
         if (!tokenizer.hasMoreTokens()) throw new ParseException("Too few bytes", 0);
         try
         {
            b = Integer.parseInt(tokenizer.nextToken());
         }
```



```
         catch (NumberFormatException e)
         {
            throw new ParseException("Not an integer", 0);
         }
         if (b < 0 || b >= 256) throw new ParseException("Byte out of range", 0);
         a[i] = (byte) b;
      }
```



```
      if (tokenizer.hasMoreTokens()) throw new ParseException("Too many bytes", 0);
      return a;
   }
}
```

API

`javax.swing.JFormattedTextField` 1.4

- `JFormattedTextField(Format fmt)`

  constructs a text field that uses the specified format.

- `JFormattedTextField(JFormattedTextField.AbstractFormatter formatter)`

  constructs a text field that uses the specified formatter. Note that `DefaultFormatter` and `InternationalFormatter` are subclasses of `JFormattedTextField.AbstractFormatter`.

- `Object getValue()`

  returns the current valid value of the field. Note that this might not correspond to the string that is being edited.

- `void setValue(Object value)`

  attempts to set the value of the given object. The attempt fails if the formatter cannot convert the object to a string.

- `void commitEdit()`

  attempts to set the valid value of the field from the edited string. The attempt might fail if the formatter cannot convert the string.

- `boolean isEditValid()`

  checks whether the edited string represents a valid value.

- `int getFocusLostBehavior()`

- `void setFocusLostBehavior(int behavior)`

  gets or sets the "focus lost" behavior. Legal values for `behavior` are the constants `COMMIT_OR_REVERT`, `REVERT`, `COMMIT`, and `PERSIST` of the `JFormattedTextField` class.

API `javax.swing.JFormattedTextField.AbstractFormatter` 1.4

- `abstract String valueToString(Object value)`

  converts a value to an editable string. Throws a `ParseException` if `value` is not appropriate for this formatter.

- `abstract Object stringToValue(String s)`

  converts a string to a value. Throws a `ParseException` if `s` is not in the appropriate format.

- `DocumentFilter getDocumentFilter()`

  override this method to provide a document filter that restricts inputs into the text field. A return value of `null` indicates that no filtering is needed.

javax.swing.text.DefaultFormatter 1.3

- `boolean getOverwriteMode()`
- `void setOverwriteMode(boolean mode)`

  gets or sets the overwrite mode. If `mode` is `true`, then new characters overwrite existing characters when editing text.

javax.swing.text.DocumentFilter 1.4

- `void insertString(DocumentFilter.FilterBypass bypass, int offset, String text, AttributeSet attrib)`

  is invoked before a string is inserted into a document. You can override the method and modify the string. You can disable insertion by not calling `super.insertString` or by calling `bypass` methods to modify the document without filtering.

Parameters:	`bypass`	An object that allows you to execute edit commands that bypass the filter
	`offset`	The offset at which to insert the text
	`text`	The characters to insert
	`attrib`	The formatting attributes of the inserted text

- `void replace(DocumentFilter.FilterBypass bypass, int offset, int length, String text, AttributeSet attrib)`

  is invoked before a part of a document is replaced with a new string. You can override the method and modify the string. You can disable replacement by not calling `super.replace` or by calling `bypass` methods to modify the document without filtering.

Parameters:	`bypass`	An object that allows you to execute edit commands that bypass the filter
	`offset`	The offset at which to insert the text
	`length`	The length of the part to be replaced
	`text`	The characters to insert
	`attrib`	The formatting attributes of the inserted text

- `void remove(DocumentFilter.FilterBypass bypass, int offset, int length)` is invoked before a part of a document is removed. Get the document by calling `bypass.getDocument()` if you need to analyze the effect of the removal.

Parameters:	`bypass`	An object that allows you to execute edit commands that bypass the filter
	`offset`	The offset of the part to be removed
	`length`	The length of the part to be removed

API `javax.swing.text.MaskFormatter` 1.4

- `MaskFormatter(String mask)`

  constructs a mask formatter with the given mask. See Table 6-3 on page 453 for the symbols in a mask.

- `String getValidCharacters()`

- `void setValidCharacters(String characters)`

  gets or sets the valid editing characters. Only the characters in the given string are accepted for the variable parts of the mask.

- `String getInvalidCharacters()`

- `void setInvalidCharacters(String characters)`

  gets or sets the invalid editing characters. None of the characters in the given string are accepted as input.

- `char getPlaceholderCharacter()`

- `void setPlaceholderCharacter(char ch)`

  gets or sets the placeholder character that is used for variable characters in the mask that the user has not yet supplied. The default placeholder character is a space.

- `String getPlaceholder()`

- `void setPlaceholder(String s)`

gets or sets the placeholder string. Its tail end is used if the user has not supplied all variable characters in the mask. If it is `null` or shorter than the mask, then the placeholder character fills remaining inputs.

- `boolean getValueContainsLiteralCharacters()`

- `void setValueContainsLiteralCharacters(boolean b)`

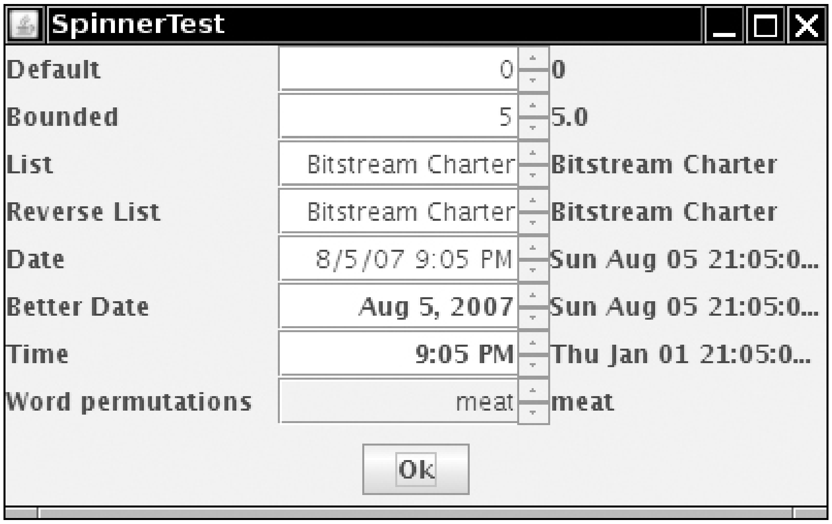
gets or sets the "value contains literal characters" flag. If this flag is `true`, then the field value contains the literal (nonvariable) parts of the mask. If it is `false`, then the literal characters are removed. The default is `true`.

UNREGISTERED VERSION OF CHM TO PDF CONVERTER PRO BY THETA-SOFTWARE

The **JSpinner** Component

A `JSpinner` is a component that contains a text field and two small buttons on the side. When the buttons are clicked, the text field value is incremented or decremented (see Figure 6-38).

Figure 6-38. Several variations of the **JSpinner** component



The values in the spinner can be numbers, dates, values from a list, or, in the most general case, any sequence of values for which predecessors and successors can be determined. The `JSpinner` class defines standard data models for the first three cases. You can define your own data model to describe arbitrary sequences.

By default, a spinner manages an integer, and the buttons increment or decrement it by 1. You can get the curren value by calling the `getValue` method. That method returns an `Object`. Cast it to an `Integer` and retrieve the wrapped value.

```
JSpinner defaultSpinner = new JSpinner();
. . .
int value = (Integer) defaultSpinner.getValue();
```

You can change the increment to a value other than 1, and you can also supply lower and upper bounds. Here is a spinner with starting value 5, bounded between 0 and 10, and an increment of 0.5:

Code View:
```
JSpinner boundedSpinner = new JSpinner(new SpinnerNumberModel(5, 0, 10, 0.5));
```

There are two `SpinnerNumberModel` constructors, one with only `int` parameters and one with `double` parameters. If any of the parameters is a floating-point number, then the second constructor is used. It sets the spinner value to a `Double` object.

Spinners aren't restricted to numeric values. You can have a spinner iterate through any collection of values. Simply pass a `SpinnerListModel` to the `JSpinner` constructor. You can construct a `SpinnerListModel` from an array or a class implementing the `List` interface (such as an `ArrayList`). In our sample program, we display a spinner control with all available font names.

Code View:
```
String[] fonts = GraphicsEnvironment.getLocalGraphicsEnvironment().getAvailableFontFamilyNames()
JSpinner listSpinner = new JSpinner(new SpinnerListModel(fonts));
```

However, we found that the direction of the iteration was mildly confusing because it is opposite from the user experience with a combo box. In a combo box, higher values are *below* lower values, so you would expect the downward arrow to navigate toward higher values. But the spinner increments the array index so that the upward arrow yields higher values. There is no provision for reversing the traversal order in the `SpinnerListModel`, but ar impromptu anonymous subclass yields the desired result:

```
JSpinner reverseListSpinner = new JSpinner(
   new SpinnerListModel(fonts)
   {
      public Object getNextValue()
      {
         return super.getPreviousValue();

      }
      public Object getPreviousValue()
      {
         return super.getNextValue();
      }
   });
```

Try both versions and see which you find more intuitive.

Another good use for a spinner is for a date that the user can increment or decrement. You get such a spinner, initialized with today's date, with the call

```
JSpinner dateSpinner = new JSpinner(new SpinnerDateModel());
```

However, if you look carefully at Figure 6-38, you will see that the spinner text shows both date and time, such as

```
8/05/07 7:23 PM
```

The time doesn't make any sense for a date picker. It turns out to be somewhat difficult to make the spinner show just the date. Here is the magic incantation:

Code View:
```
JSpinner betterDateSpinner = new JSpinner(new SpinnerDateModel());
String pattern = ((SimpleDateFormat) DateFormat.getDateInstance()).toPattern();
betterDateSpinner.setEditor(new JSpinner.DateEditor(betterDateSpinner, pattern));
```


Using the same approach, you can also make a time picker.

Code View:
```
JSpinner timeSpinner = new JSpinner(new SpinnerDateModel());
pattern = ((SimpleDateFormat) DateFormat.getTimeInstance(DateFormat.SHORT)).toPattern();
timeSpinner.setEditor(new JSpinner.DateEditor(timeSpinner, pattern));
```

You can display arbitrary sequences in a spinner by defining your own spinner model. In our sample program, we have a spinner that iterates through all permutations of the string "meat". You can get to "mate", "meta", "team", and another 20 permutations by clicking the spinner buttons.

When you define your own model, you should extend the `AbstractSpinnerModel` class and define the following four methods:

```
Object getValue()
void setValue(Object value)
Object getNextValue()
Object getPreviousValue()
```

The `getValue` method returns the value stored by the model. The `setValue` method sets a new value. It should throw an `IllegalArgumentException` if the new value is not appropriate.

Caution

The `setValue` method must call the `fireStateChanged` method after setting the new value. Otherwise, the spinner field won't be updated.

The `getNextValue` and `getPreviousValue` methods return the values that should come after or before the current value, or `null` if the end of the traversal has been reached.

Caution

The `getNextValue` and `getPreviousValue` methods should *not* change the current value. When a user clicks on the upward arrow of the spinner, the `getNextValue` method is called. If the return value is not `null`, it is set by a call to `setValue`.

In the sample program, we use a standard algorithm to determine the next and previous permutations. The details of the algorithm are not important.

Listing 6-14 shows how to generate the various spinner types. Click the Ok button to see the spinner values.

Listing 6-14. **SpinnerTest.java**

Code View:
```
import java.awt.*;
import java.awt.event.*;
import java.text.*;
import java.util.*;
import javax.swing.*;

/**
   A program to test spinners.
*/
public class SpinnerTest
{
   public static void main(String[] args)
   {
      SpinnerFrame frame = new SpinnerFrame();
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.setVisible(true);
   }
}

/**
   A frame with a panel that contains several spinners and
   a button that displays the spinner values.
*/
class SpinnerFrame extends JFrame
{
   public SpinnerFrame()
   {
      setTitle("SpinnerTest");
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
      JPanel buttonPanel = new JPanel();
      okButton = new JButton("Ok");
      buttonPanel.add(okButton);
      add(buttonPanel, BorderLayout.SOUTH);

      mainPanel = new JPanel();
      mainPanel.setLayout(new GridLayout(0, 3));
      add(mainPanel, BorderLayout.CENTER);

```

```
    JSpinner defaultSpinner = new JSpinner();
    addRow("Default", defaultSpinner);

    JSpinner boundedSpinner = new JSpinner(new SpinnerNumberModel(5, 0, 10, 0.5));
    addRow("Bounded", boundedSpinner);

    String[] fonts = GraphicsEnvironment
       .getLocalGraphicsEnvironment()
        .getAvailableFontFamilyNames();
```

```

    JSpinner listSpinner = new JSpinner(new SpinnerListModel(fonts));
    addRow("List", listSpinner);

    JSpinner reverseListSpinner = new JSpinner(
       new
          SpinnerListModel(fonts)
```

```
          {
             public Object getNextValue()
             {
                return super.getPreviousValue();
             }
             public Object getPreviousValue()
             {
                return super.getNextValue();
             }
          });
    addRow("Reverse List", reverseListSpinner);

    JSpinner dateSpinner = new JSpinner(new SpinnerDateModel());
    addRow("Date", dateSpinner);

    JSpinner betterDateSpinner = new JSpinner(new SpinnerDateModel());
    String pattern = ((SimpleDateFormat) DateFormat.getDateInstance()).toPattern();
    betterDateSpinner.setEditor(new JSpinner.DateEditor(betterDateSpinner, pattern));
    addRow("Better Date", betterDateSpinner);

    JSpinner timeSpinner = new JSpinner(
       new SpinnerDateModel(
          new GregorianCalendar(2000, Calendar.JANUARY, 1, 12, 0, 0).getTime(),
             null, null, Calendar.HOUR));
    addRow("Time", timeSpinner);

    JSpinner permSpinner = new JSpinner(new PermutationSpinnerModel("meat"));
    addRow("Word permutations", permSpinner);
 }

 /**
    Adds a row to the main panel.
    @param labelText the label of the spinner
    @param spinner the sample spinner
*/
public void addRow(String labelText, final JSpinner spinner)
{
   mainPanel.add(new JLabel(labelText));
   mainPanel.add(spinner);
   final JLabel valueLabel = new JLabel();
   mainPanel.add(valueLabel);
   okButton.addActionListener(new
      ActionListener()
```

```
         {
            public void actionPerformed(ActionEvent event)
            {
               Object value = spinner.getValue();
               valueLabel.setText(value.toString());
            }
         });
   }

   public static final int DEFAULT_WIDTH = 400;
   public static final int DEFAULT_HEIGHT = 250;

   private JPanel mainPanel;
   private JButton okButton;
}

/**
   A model that dynamically generates word permutations
*/
class PermutationSpinnerModel extends AbstractSpinnerModel
{
   /**
      Constructs the model.
      @param w the word to permute
   */
   public PermutationSpinnerModel(String w)
   {
      word = w;
   }

   public Object getValue()
   {
      return word;
   }

   public void setValue(Object value)
   {
      if (!(value instanceof String))
         throw new IllegalArgumentException();
      word = (String) value;
      fireStateChanged();
   }

   public Object getNextValue()
   {
      int[] codePoints = toCodePointArray(word);
      for (int i = codePoints.length - 1; i > 0; i--)
      {
         if (codePoints[i - 1] < codePoints[i])
         {
            int j = codePoints.length - 1;
            while (codePoints[i - 1] > codePoints[j]) j--;
            swap(codePoints, i - 1, j);
            reverse(codePoints, i, codePoints.length - 1);
            return new String(codePoints, 0, codePoints.length);
         }
      }
      reverse(codePoints, 0, codePoints.length - 1);
      return new String(codePoints, 0, codePoints.length);
```

```
   }

   public Object getPreviousValue()
   {
      int[] codePoints = toCodePointArray(word);
      for (int i = codePoints.length - 1; i > 0; i--)
      {
         if (codePoints[i - 1] > codePoints[i])
         {
```

```
            int j = codePoints.length - 1;
            while (codePoints[i - 1] < codePoints[j]) j--;
            swap(codePoints, i - 1, j);
            reverse(codePoints, i, codePoints.length - 1);
            return new String(codePoints, 0, codePoints.length);
         }
      }
```

```
      reverse(codePoints, 0, codePoints.length - 1);
      return new String(codePoints, 0, codePoints.length);
   }

   private static int[] toCodePointArray(String str)
   {
      int[] codePoints = new int[str.codePointCount(0, str.length())];
      for (int i = 0, j = 0; i < str.length(); i++, j++)
      {
         int cp = str.codePointAt(i);
         if (Character.isSupplementaryCodePoint(cp)) i++;
         codePoints[j] = cp;
      }
      return codePoints;
   }

   private static void swap(int[] a, int i, int j)
   {
      int temp = a[i];
      a[i] = a[j];
      a[j] = temp;
   }

   private static void reverse(int[] a, int i, int j)
   {
      while (i < j) { swap(a, i, j); i++; j--; }
   }

   private String word;
}
```

API `javax.swing.JSpinner` 1.4

- `JSpinner()`

  constructs a spinner that edits an integer with starting value 0, increment 1, and no bounds.

- `JSpinner(SpinnerModel model)`

  constructs a spinner that uses the given data model.

- `Object getValue()`

  gets the current value of the spinner.

- `void setValue(Object value)`

  attempts to set the value of the spinner. Throws an `IllegalArgumentException` if the model does not accept the value.

- `void setEditor(JComponent editor)`

  sets the component that is used for editing the spinner value.

API `javax.swing.SpinnerNumberModel` 1.4

- `SpinnerNumberModel(int initval, int minimum, int maximum, int stepSize)`

- `SpinnerNumberModel(double initval, double minimum, double maximum, double stepSize)`

  these constructors yield number models that manage an `Integer` or `Double` value. Use the `MIN_VALUE` and `MAX_VALUE` constants of the `Integer` and `Double` classes for unbounded values.

Parameters:	`initval`	The initial value
	`minimum`	The minimum valid value
	`maximum`	The maximum valid value
	`stepSize`	The increment or decrement of each spin

`javax.swing.SpinnerListModel` 1.4

- `SpinnerListModel(Object[] values)`

- `SpinnerListModel(List values)`

  these constructors yield models that select a value from among the given values.

`javax.swing.SpinnerDateModel` 1.4

- `SpinnerDateModel()`

  constructs a date model with today's date as the initial value, no lower or upper bounds, and an increment of `Calendar.DAY_OF_MONTH`.

- `SpinnerDateModel(Date initval, Comparable minimum, Comparable maximum, int step)`

Parameters:	`initval`	The initial value
	`minimum`	The minimum valid value, or `null` if no lower bound is desired
	`maximum`	The maximum valid value, or `null` if no upper bound is desired
	`step`	The date field to increment or decrement of each spin. One of the constants ERA, YEAR, MONTH, WEEK_OF_YEAR, WEEK_OF_MONTH, DAY_OF_MONTH, DAY_OF_YEAR, DAY_OF_WEEK, DAY_OF_WEEK_IN_MONTH, AM_PM, HOUR, HOUR_OF_DAY, MINUTE, SECOND, or `MILLISECOND` of the `Calendar` class

API

`java.text.SimpleDateFormat` 1.1

- `String toPattern()` **1.2**

  gets the editing pattern for this date formatter. A typical pattern is `"yyyy-MM-dd"`. See the Java SE documentation for more details about the pattern.

API

`javax.swing.JSpinner.DateEditor` 1.4

- `DateEditor(JSpinner spinner, String pattern)`

  constructs a date editor for a spinner.

Parameters:	`spinner`	The spinner to which this editor belongs
	`pattern`	The format pattern for the associated `SimpleDateFormat`

API

`javax.swing.AbstractSpinnerModel` 1.4

- `Object getValue()`

  gets the current value of the model.

- `void setValue(Object value)`

  attempts to set a new value for the model. Throws an `IllegalArgumentException` if the value is not acceptable. When overriding this method, you should call `fireStateChanged` after setting the new value.

- `Object getNextValue()`

- `Object getPreviousValue()`

  computes (but does not set) the next or previous value in the sequence that this model defines.

Displaying HTML with the **JEditorPane**

Unlike the text components that we discussed up to this point, the `JEditorPane` can display and edit styled text, in particular HTML and RTF. (RTF is the "rich text format" that is used by a number of Microsoft applications for document interchange. It is a poorly documented format that doesn't work well even between Microsoft's own applications. We do not cover RTF capabilities in this book.)

Frankly, the `JEditorPane` is not as functional as one would like it to be. The HTML renderer can display simple files, but it chokes at many complex pages that you typically find on the Web. The HTML editor is limited and unstable.

A plausible application for the `JEditorPane` is to display program help in HTML format. Because you have control over the help files that you provide, you can stay away from features that the `JEditorPane` does not display well.

Note

For more information on an industrial-strength help system, check out JavaHelp at http://java.sun.com/products/javahelp/index.html.

The program in Listing 6-15 contains an editor pane that shows the contents of an HTML page. Type a URL into the text field. The URL must start with `http:` or `file:`. Then, click the Load button. The selected HTML page is displayed in the editor pane (see Figure 6-39).

Figure 6-39. The editor pane displaying an HTML page

[View full size image]

The hyperlinks are active: If you click a link, the application loads it. The Back button returns to the previous page

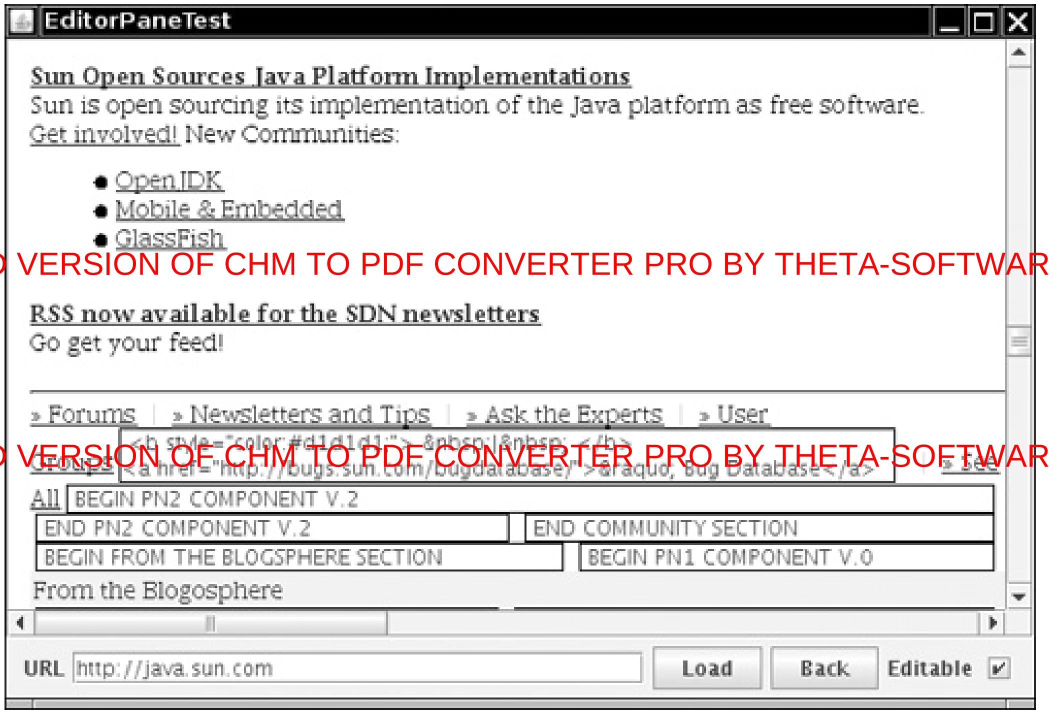
This program is in fact a very simple browser. Of course, it does not have any of the comfort features, such as page caching or bookmark lists, that you expect from a commercial browser. The editor pane does not even display applets!

If you click the Editable checkbox, then the editor pane becomes editable. You can type in text and use the BACKSPACE key to delete text. The component also understands the CTRL+X, CTRL+C, and CTRL+V shortcuts for cut, copy and paste. However, you would have to do quite a bit of programming to add support for fonts and formatting.

When the component is editable, hyperlinks are not active. Also, with some web pages you can see JavaScript commands, comments, and other tags when edit mode is turned on (see Figure 6-40). The example program lets you investigate the editing feature, but we recommend that you omit that feature in your programs.

Figure 6-40. The editor pane in edit mode

[View full size image]



Tip

By default, the `JEditorPane` is in edit mode. You should call `editorPane.setEditable(false)` to turn it off.

The features of the editor pane that you saw in the example program are easy to use. You use the `setPage` method to load a new document. For example,

```
JEditorPane editorPane = new JEditorPane();
editorPane.setPage(url);
```

The parameter is either a string or a `URL` object. The `JEditorPane` class extends the `JTextComponent` class. Therefore, you can call the `setText` method as well—it simply displays plain text.

Tip

The API documentation is unclear about whether setPage loads the new document in a separate thread (which is generally what you want—the JEditorPane is no speed demon). However, you can force loading in a separate thread with the following incantation:

```
AbstractDocument doc = (AbstractDocument) editorPane.getDocument();
doc.setAsynchronousLoadPriority(0);
```

To listen to hyperlink clicks, you add a HyperlinkListener. The HyperlinkListener interface has a single method, hyperlinkUpdate, that is called when the user moves over or clicks on a link. The method has a parameter of type HyperlinkEvent.

You need to call the getEventType method to find out what kind of event occurred. There are three possible return values:

```
HyperlinkEvent.EventType.ACTIVATED
HyperlinkEvent.EventType.ENTERED
HyperlinkEvent.EventType.EXITED
```

The first value indicates that the user clicked on the hyperlink. In that case, you typically want to open the new link. You can use the second and third values to give some visual feedback, such as a tooltip, when the mouse hovers over the link.

Note

It is a complete mystery why there aren't three separate methods to handle activation, entry, and exit in the HyperlinkListener interface.

The getURL method of the HyperlinkEvent class returns the URL of the hyperlink. For example, here is how you can install a hyperlink listener that follows the links that a user activates:

```
editorPane.addHyperlinkListener(new
   HyperlinkListener()
   {
      public void hyperlinkUpdate(HyperlinkEvent event)
      {
         if (event.getEventType() == HyperlinkEvent.EventType.ACTIVATED)
         {
            try
            {
               editorPane.setPage(event.getURL());
            }
            catch (IOException e)
            {
```

```
               editorPane.setText("Exception: " + e);
            }
         }
      }
   });
```

The event handler simply gets the URL and updates the editor pane. The setPage method can throw an IOException. In that case, we display an error message as plain text.

The program in Listing 6-15 shows all the features that you need to put together an HTML help system. Under the hood, the JEditorPane is even more complex than the tree and table components. However, if you don't need to write a text editor or a renderer of a custom text format, that complexity is hidden from you.

Listing 6-15. EditorPaneTest.java

Code View:

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

/**
 * This program demonstrates how to display HTML documents in an editor pane.
 * @version 1.03 2007-08-01
 * @author Cay Horstmann
 */
public class EditorPaneTest
{
   public static void main(String[] args)
   {
      EventQueue.invokeLater(new Runnable()
         {
            public void run()
            {
               JFrame frame = new EditorPaneFrame();
               frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
               frame.setVisible(true);
            }
         });
   }
}

/**
 * This frame contains an editor pane, a text field and button to enter a URL and load
 * a document, and a Back button to return to a previously loaded document.
 */
class EditorPaneFrame extends JFrame
{
   public EditorPaneFrame()
   {
      setTitle("EditorPaneTest");
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

      final Stack<String> urlStack = new Stack<String>();
      final JEditorPane editorPane = new JEditorPane();
```

```
      final JTextField url = new JTextField(30);

      // set up hyperlink listener

      editorPane.setEditable(false);
      editorPane.addHyperlinkListener(new HyperlinkListener()
         {
            public void hyperlinkUpdate(HyperlinkEvent event)
            {
               if (event.getEventType() == HyperlinkEvent.EventType.ACTIVATED)
               {
                  try
                  {
                     // remember URL for back button
                     urlStack.push(event.getURL().toString());
                     // show URL in text field
                     url.setText(event.getURL().toString());
                     editorPane.setPage(event.getURL());
                  }
                  catch (IOException e)
                  {
                     editorPane.setText("Exception: " + e);
                  }
               }
            }
         });
      // set up checkbox for toggling edit mode

      final JCheckBox editable = new JCheckBox();
      editable.addActionListener(new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               editorPane.setEditable(editable.isSelected());
            }
         });

      // set up load button for loading URL

      ActionListener listener = new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               try
               {
                  // remember URL for back button
                  urlStack.push(url.getText());
                  editorPane.setPage(url.getText());
               }
               catch (IOException e)
               {
                  editorPane.setText("Exception: " + e);
               }
            }
         };

      JButton loadButton = new JButton("Load");
      loadButton.addActionListener(listener);
      url.addActionListener(listener);
```

```java

      // set up back button and button action

      JButton backButton = new JButton("Back");
      backButton.addActionListener(new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               if (urlStack.size() <= 1) return;
```



```java
               try
               {
                  // get URL from back button
                  urlStack.pop();
                  // show URL in text field
                  String urlString = urlStack.peek();
                  url.setText(urlString);
```



```java
                  editorPane.setPage(urlString);
               }
               catch (IOException e)
               {
                  editorPane.setText("Exception: " + e);
               }
            }
         });

      add(new JScrollPane(editorPane), BorderLayout.CENTER);

      // put all control components in a panel

      JPanel panel = new JPanel();
      panel.add(new JLabel("URL"));
      panel.add(url);
      panel.add(loadButton);
      panel.add(backButton);
      panel.add(new JLabel("Editable"));
      panel.add(editable);

      add(panel, BorderLayout.SOUTH);
  }

  private static final int DEFAULT_WIDTH = 600;
  private static final int DEFAULT_HEIGHT = 400;
}
```

API

`javax.swing.JEditorPane` 1.2

- `void setPage(URL url)`

  loads the page from `url` into the editor pane.

- `void addHyperlinkListener(HyperLinkListener listener)`

  adds a hyperlink listener to this editor pane.

API

`javax.swing.event.HyperlinkListener` 1.2

- `void hyperlinkUpdate(HyperlinkEvent event)`

  is called whenever a hyperlink was selected.

API

`javax.swing.HyperlinkEvent` 1.2

- `URL getURL()`

  returns the URL of the selected hyperlink.

Progress Indicators

In the following sections, we discuss three classes for indicating the progress of a slow activity. A `JProgressBar` is a Swing component that indicates progress. A `ProgressMonitor` is a dialog box that contains a progress bar. A `ProgressMonitorInputStream` displays a progress monitor dialog box while the stream is read.

Progress Bars

A *progress bar* is a simple component—just a rectangle that is partially filled with color to indicate the progress of an operation. By default, progress is indicated by a string *"n%"*. You can see a progress bar in the bottom right of Figure 6-41.

UNREGISTERED VERSION OF CHM TO PDF CONVERTER PRO BY THETA-SOFTWARE

Figure 6-41. A progress bar

You construct a progress bar much as you construct a slider, by supplying the minimum and maximum value and an optional orientation:

```
progressBar = new JProgressBar(0, 1000);
progressBar = new JProgressBar(SwingConstants.VERTICAL, 0, 1000);
```

You can also set the minimum and maximum with the `setMinimum` and `setMaximum` methods.

Unlike a slider, the progress bar cannot be adjusted by the user. Your program needs to call `setValue` to update it.

If you call

```
progressBar.setStringPainted(true);
```

the progress bar computes the completion percentage and displays a string *"n%"*. If you want to show a different string, you can supply it with the `setString` method:

```
if (progressBar.getValue() > 900)
```

```
progressBar.setString("Almost Done");
```

The program in Listing 6-16 shows a progress bar that monitors a simulated time-consuming activity.

The `SimulatedActivity` class increments a value `current` ten times per second. When it reaches a target value, the activity finishes. We use the `SwingWorker` class to implement the activity and update the progress bar in the `process` method. The `SwingWorker` invokes the method from the event dispatch thread, so that it is safe to update the progress bar. (See Volume I, Chapter 14 for more information about thread safety in Swing.)

Java SE 1.4 added support for an *indeterminate* progress bar that shows an animation indicating some kind of progress, without giving an indication of the percentage of completion. That is the kind of progress bar that you see in your browser—it indicates that the browser is waiting for the server and has no idea how long the wait might be. To display the "indeterminate wait" animation, call the `setIndeterminate` method.

Listing 6-16 shows the full program code.

Listing 6-16. **ProgressBarTest.java**

Code View:

```
import java.awt.*;
import java.awt.event.*;
import java.util.List;

import javax.swing.*;

/**
 * This program demonstrates the use of a progress bar to monitor the progress of a thread.
 * @version 1.04 2007-08-01
 * @author Cay Horstmann
 */
public class ProgressBarTest
{
   public static void main(String[] args)
   {
      EventQueue.invokeLater(new Runnable()
         {
            public void run()
            {
               JFrame frame = new ProgressBarFrame();
               frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
               frame.setVisible(true);
            }
         });
   }
}

/**
 * A frame that contains a button to launch a simulated activity, a progress bar, and a
 * text area for the activity output.
 */
class ProgressBarFrame extends JFrame
{
   public ProgressBarFrame()
   {
      setTitle("ProgressBarTest");
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

```

```
      // this text area holds the activity output
      textArea = new JTextArea();

      // set up panel with button and progress bar

      final int MAX = 1000;
      JPanel panel = new JPanel();
      startButton = new JButton("Start");
      progressBar = new JProgressBar(0, MAX);
```

```
      progressBar.setStringPainted(true);
      panel.add(startButton);
      panel.add(progressBar);

      checkBox = new JCheckBox("indeterminate");
      checkBox.addActionListener(new ActionListener()
         {
```

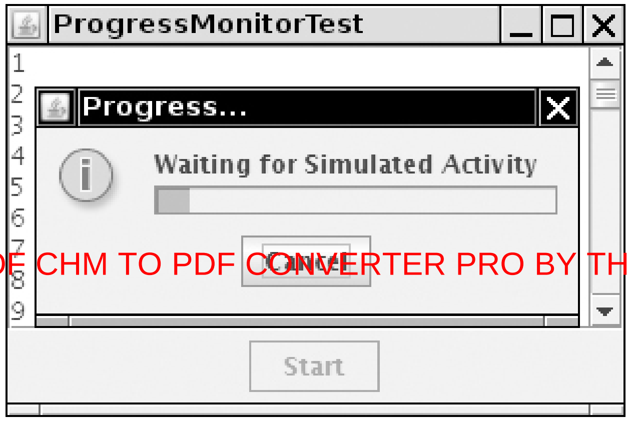
```
            public void actionPerformed(ActionEvent event)
            {
               progressBar.setIndeterminate(checkBox.isSelected());
               progressBar.setStringPainted(!progressBar.isIndeterminate());
            }
         });
      panel.add(checkBox);
      add(new JScrollPane(textArea), BorderLayout.CENTER);
      add(panel, BorderLayout.SOUTH);

      // set up the button action

      startButton.addActionListener(new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               startButton.setEnabled(false);
               activity = new SimulatedActivity(MAX);
               activity.execute();
            }
         });
   }

   private JButton startButton;
   private JProgressBar progressBar;
   private JCheckBox checkBox;
   private JTextArea textArea;
   private SimulatedActivity activity;

   public static final int DEFAULT_WIDTH = 400;
   public static final int DEFAULT_HEIGHT = 200;

   class SimulatedActivity extends SwingWorker<Void, Integer>
   {
      /**
       * Constructs the simulated activity that increments a counter from 0 to a
       * given target.
       * @param t the target value of the counter.
       */
      public SimulatedActivity(int t)
      {
         current = 0;
         target = t;
```

```
      }

      protected Void doInBackground() throws Exception
      {
         try
         {
            while (current < target)
            {
               Thread.sleep(100);
               current++;
               publish(current);
            }
         }
         catch (InterruptedException e)
         {
         }
         return null;
      }

      protected void process(List<Integer> chunks)
      {
         for (Integer chunk : chunks)
         {
            textArea.append(chunk + "\n");
            progressBar.setValue(chunk);
         }
      }

      protected void done()
      {
         startButton.setEnabled(true);
      }

      private int current;
      private int target;
   }
}
```

Progress Monitors

A progress bar is a simple component that can be placed inside a window. In contrast, a `ProgressMonitor` is a complete dialog box that contains a progress bar (see Figure 6-42). The dialog box contains a Cancel button. If you click it, the monitor dialog box is closed. In addition, your program can query whether the user has canceled the dialog box and terminate the monitored action. (Note that the class name does not start with a "`J`".)

Figure 6-42. A progress monitor dialog box



You construct a progress monitor by supplying the following:

- The parent component over which the dialog box should pop up.
- An object (which should be a string, icon, or component) that is displayed on the dialog box.
- An optional note to display below the object.
- The minimum and maximum values.

However, the progress monitor cannot measure progress or cancel an activity by itself. You still need to periodically set the progress value by calling the `setProgress` method. (This is the equivalent of the `setValue` method of the `JProgressBar` class.) When the monitored activity has concluded, call the `close` method to dismiss the dialog box. You can reuse the same dialog box by calling `start` again.

The biggest problem with using a progress monitor dialog box is the handling of cancellation requests. You cannot attach an event handler to the Cancel button. Instead, you need to periodically call the `isCanceled` method to see if the program user has clicked the Cancel button.

If your worker thread can block indefinitely (for example, when reading input from a network connection), then it cannot monitor the Cancel button. In our sample program, we show you how to use a timer for that purpose. We also make the timer responsible for updating the progress measurement.

If you run the program in Listing 6-17, you can observe an interesting feature of the progress monitor dialog box. The dialog box doesn't come up immediately. Instead, it waits for a short interval to see if the activity has already been completed or is likely to complete in less time than it would take for the dialog box to appear.

You control the timing as follows. Use the `setMillisToDecideToPopup` method to set the number of milliseconds to wait between the construction of the dialog object and the decision whether to show the pop-up at all. The default value is 500 milliseconds. The `setMillisToPopup` is your estimation of the time the dialog box needs to pop up. The Swing designers set this value to a default of 2 seconds. Clearly they were mindful of the fact that Swing dialogs don't always come up as snappily as we all would like. You should probably not touch this value.

Listing 6-17. **`ProgressMonitorTest.java`**

Code View:
```
import java.awt.*;
import java.awt.event.*;

import javax.swing.*;

/**
 * A program to test a progress monitor dialog.
 * @version 1.04 2007-08-01
 * @author Cay Horstmann
 */
public class ProgressMonitorTest
{
   public static void main(String[] args)
   {
      EventQueue.invokeLater(new Runnable()
         {
            public void run()
            {
               JFrame frame = new ProgressMonitorFrame();
               frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
               frame.setVisible(true);
            }
         });
   }
}

/**
 * A frame that contains a button to launch a simulated activity and a text area for the
 * activity output.
 */
class ProgressMonitorFrame extends JFrame
{
   public ProgressMonitorFrame()
   {
      setTitle("ProgressMonitorTest");
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

      // this text area holds the activity output
      textArea = new JTextArea();

      // set up a button panel
      JPanel panel = new JPanel();
      startButton = new JButton("Start");
      panel.add(startButton);

      add(new JScrollPane(textArea), BorderLayout.CENTER);
      add(panel, BorderLayout.SOUTH);

      // set up the button action

      startButton.addActionListener(new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               startButton.setEnabled(false);
               final int MAX = 1000;

               // start activity
```

```
               activity = new SimulatedActivity(MAX);
               activity.execute();

               // launch progress dialog
               progressDialog = new ProgressMonitor(ProgressMonitorFrame.this,
                     "Waiting for Simulated Activity", null, 0, MAX);
               cancelMonitor.start();
            }
         });
```

```

      // set up the timer action

      cancelMonitor = new Timer(500, new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
```

```
               if (progressDialog.isCanceled())
               {
                  activity.cancel(true);
                  startButton.setEnabled(true);
               }
               else if (activity.isDone())
               {
                  progressDialog.close();
                  startButton.setEnabled(true);
               }
               else
               {
                  progressDialog.setProgress(activity.getProgress());
               }
            }
         });
   }

   private Timer cancelMonitor;
   private JButton startButton;
   private ProgressMonitor progressDialog;
   private JTextArea textArea;
   private SimulatedActivity activity;

   public static final int DEFAULT_WIDTH = 300;
   public static final int DEFAULT_HEIGHT = 200;

   class SimulatedActivity extends SwingWorker<Void, Integer>
   {
      /**
       * Constructs the simulated activity that increments a counter from 0 to a
       * given target.
       * @param t the target value of the counter.
       */
      public SimulatedActivity(int t)
      {
         current = 0;
         target = t;
      }

      protected Void doInBackground() throws Exception
      {
         try
```

```
         {
            while (current < target)
            {
               Thread.sleep(100);
               current++;
               textArea.append(current + "\n");
               setProgress(current);
            }
         }
         catch (InterruptedException e)
         {
         }
         return null;
      }

      private int current;
      private int target;
   }
}
```

Monitoring the Progress of Input Streams

The Swing package contains a useful stream filter, `ProgressMonitorInputStream`, that automatically pops up a dialog box that monitors how much of the stream has been read.

This filter is extremely easy to use. You sandwich in a `ProgressMonitorInputStream` between your usual sequence of filtered streams. (See Volume I, Chapter 12 for more information on streams.)

For example, suppose you read text from a file. You start out with a `FileInputStream`:

```
FileInputStream in = new FileInputStream(f);
```

Normally, you would convert `in` to an `InputStreamReader`:

```
InputStreamReader reader = new InputStreamReader(in);
```

However, to monitor the stream, first turn the file input stream into a stream with a progress monitor:

Code View:

```
ProgressMonitorInputStream progressIn = new ProgressMonitorInputStream(parent, caption, in);
```

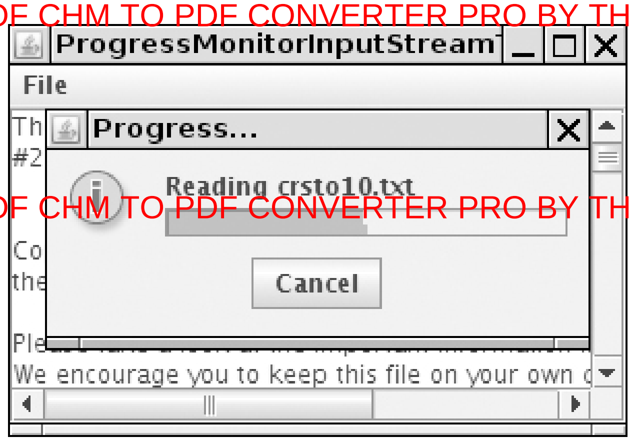
You supply the parent component, a caption, and, of course, the stream to monitor. The `read` method of the progress monitor stream simply passes along the bytes and updates the progress dialog box.

You now go on building your filter sequence:

```
InputStreamReader reader = new InputStreamReader(progressIn);
```

That's all there is to it. When the file is read, the progress monitor automatically pops up (see Figure 6-43). This is a very nice application of stream filtering.

Figure 6-43. A progress monitor for an input stream



Caution

The progress monitor stream uses the `available` method of the `InputStream` class to determine the total number of bytes in the stream. However, the `available` method only reports the number of bytes in the stream that are available *without blocking.* Progress monitors work well for files and HTTP URLs because their length is known in advance, but they don't work with all streams.

The program in Listing 6-18 counts the lines in a file. If you read in a large file (such as "The Count of Monte Cristo" in the `gutenberg` directory of the companion code), then the progress dialog box pops up.

If the user clicks the Cancel button, the input stream closes. Because the code that processes the input already knows how to deal with the end of input, no change to the programming logic is required to handle cancellation.

Note that the program doesn't use a very efficient way of filling up the text area. It would be faster to first read the file into a `StringBuilder` and then set the text of the text area to the string builder contents. However, in this example program, we actually like this slow approach—it gives you more time to admire the progress dialog box.

To avoid flicker, we do not display the text area while it is filling up.

Listing 6-18. **`ProgressMonitorInputStreamTest.java`**

Code View:
```
import java.awt.*;
```

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

/**
 * A program to test a progress monitor input stream.
 * @version 1.04 2007-08-01
 * @author Cay Horstmann
 */
public class ProgressMonitorInputStreamTest
{
   public static void main(String[] args)
   {
      EventQueue.invokeLater(new Runnable()
         {
            public void run()
            {
               JFrame frame = new TextFrame();
               frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
               frame.setVisible(true);
            }
         });
   }
}

/**
 * A frame with a menu to load a text file and a text area to display its contents. The text
 * area is constructed when the file is loaded and set as the content pane of the frame when
 * the loading is complete. That avoids flicker during loading.
 */
class TextFrame extends JFrame
{
   public TextFrame()
   {
      setTitle("ProgressMonitorInputStreamTest");
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

      textArea = new JTextArea();
      add(new JScrollPane(textArea));

      chooser = new JFileChooser();
      chooser.setCurrentDirectory(new File("."));

      JMenuBar menuBar = new JMenuBar();
      setJMenuBar(menuBar);
      JMenu fileMenu = new JMenu("File");
      menuBar.add(fileMenu);
      openItem = new JMenuItem("Open");
      openItem.addActionListener(new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               try
               {
                  openFile();
               }
               catch (IOException exception)
               {
```

```
                  exception.printStackTrace();
               }
            }
         });

      fileMenu.add(openItem);
      exitItem = new JMenuItem("Exit");
      exitItem.addActionListener(new ActionListener()
```

```
         {
            public void actionPerformed(ActionEvent event)
            {
               System.exit(0);
            }
         });
      fileMenu.add(exitItem);
```

```
   }

   /**
    * Prompts the user to select a file, loads the file into a text area, and sets it as
    * the content pane of the frame.
    */
   public void openFile() throws IOException
   {
      int r = chooser.showOpenDialog(this);
      if (r != JFileChooser.APPROVE_OPTION) return;
      final File f = chooser.getSelectedFile();

      // set up stream and reader filter sequence

      FileInputStream fileIn = new FileInputStream(f);
      ProgressMonitorInputStream progressIn = new ProgressMonitorInputStream(this,
            "Reading " + f.getName(), fileIn);
      final Scanner in = new Scanner(progressIn);

      textArea.setText("");

      SwingWorker<Void, Void> worker = new SwingWorker<Void, Void>()
         {
            protected Void doInBackground() throws Exception
            {
               while (in.hasNextLine())
               {
                  String line = in.nextLine();
                  textArea.append(line);
                  textArea.append("\n");
               }
               in.close();
               return null;
            }
         };
      worker.execute();
   }

   private JMenuItem openItem;
   private JMenuItem exitItem;
   private JTextArea textArea;
   private JFileChooser chooser;

   public static final int DEFAULT_WIDTH = 300;
```

```
    public static final int DEFAULT_HEIGHT = 200;
}
```

javax.swing.JProgressBar 1.2

- `JProgressBar()`
- `JProgressBar(int direction)`
- `JProgressBar(int min, int max)`
- `JProgressBar(int direction, int min, int max)`

  constructs a slider with the given direction, minimum, and maximum.

Parameters:	`direction`	One of `SwingConstants.HORIZONTAL` or `SwingConstants.VERTICAL`. The default is horizontal
	`min`, `max`	The minimum and maximum for the progress bar values. Defaults are 0 and 100

- `int getMinimum()`
- `int getMaximum()`
- `void setMinimum(int value)`
- `void setMaximum(int value)`

  gets or sets the minimum and maximum values.

- `int getValue()`
- `void setValue(int value)`

  gets or sets the current value.

- `String getString()`
- `void setString(String s)`

  gets or sets the string to be displayed in the progress bar. If the string is `null`, then a default string *"n%"* is displayed.

- `boolean isStringPainted()`
- `void setStringPainted(boolean b)`

  gets or sets the "string painted" property. If this property is `true`, then a string is painted on top of the progress bar. The default is `false`; no string is painted.

- `boolean isIndeterminate()` **1.4**
- `void setIndeterminate(boolean b)` **1.4**

  gets or sets the "indeterminate" property. If this property is `true`, then the progress bar becomes a block that moves backward and forward, indicating a wait of unknown duration. The default is `false`.

API

`javax.swing.ProgressMonitor` **1.2**

- `ProgressMonitor(Component parent, Object message, String note, int min, int max)`

  constructs a progress monitor dialog box.

Parameters:	`parent`	The parent component over which this dialog box pops up
	`message`	The message object to display in the dialog box
	`note`	The optional string to display under the message. If this value is `null`, then no space is set aside for the note, and a later call to `setNote` has no effect
	`min`, `max`	The minimum and maximum values of the progress bar

- `void setNote(String note)`

  changes the note text.

- `void setProgress(int value)`

  sets the progress bar value to the given value.

- `void close()`

  closes this dialog box.

- `boolean isCanceled()`

  returns `true` if the user canceled this dialog box.

API `javax.swing.ProgressMonitorInputStream` 1.2

- `ProgressMonitorInputStream(Component parent, Object message, InputStream in)`

  constructs an input stream filter with an associated progress monitor dialog box.

Parameters:	`parent`	The parent component over which this dialog box pops up
	`message`	The message object to display in the dialog box
	`in`	The input stream that is being monitored

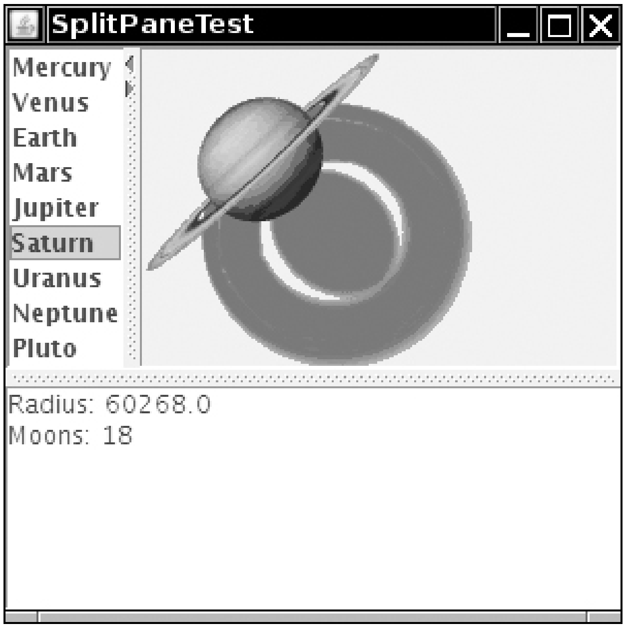
Component Organizers

We conclude the discussion of advanced Swing features with a presentation of components that help organize other components. These include the *split pane*, a mechanism for splitting an area into multiple parts with boundaries that can be adjusted, the *tabbed pane*, which uses tab dividers to allow a user to flip through multiple panels, and the *desktop pane*, which can be used to implement applications that display multiple *internal frames*.

Split Panes

Split panes split a component into two parts, with an adjustable boundary in between. Figure 6-44 shows a frame with two split panes. The components in the outer split pane are arranged vertically, with a text area on the bottom and another split pane on the top. That split pane's components are arranged horizontally, with a list on the left and a label containing an image on the right.

Figure 6-44. A frame with two nested split panes



You construct a split pane by specifying the orientation, one of `JSplitPane.HORIZONTAL_SPLIT` or `JSplitPane.VERTICAL_SPLIT`, followed by the two components. For example,

Code View:
```
JSplitPane innerPane = new JSplitPane(JSplitPane.HORIZONTAL_SPLIT, planetList, planetImage);
```

That's all you have to do. If you like, you can add "one-touch expand" icons to the splitter bar. You see those icons in the top pane in Figure 6-44. In the Metal look and feel, they are small triangles. If you click one of them, the splitter moves all the way in the direction to which the triangle is pointing, expanding one of the panes completely.

To add this capability, call

```
innerPane.setOneTouchExpandable(true);
```

The "continuous layout" feature continuously repaints the contents of both components as the user adjusts the splitter. That looks classier, but it can be slow. You turn on that feature with the call

```
innerPane.setContinuousLayout(true);
```

In the example program, we left the bottom splitter at the default (no continuous layout). When you drag it, you only move a black outline. When you release the mouse, the components are repainted.

The straightforward program in Listing 6-19 populates a list box with planets. When the user makes a selection, the planet image is displayed to the right and a description is placed in the text area on the bottom. When you run the program, adjust the splitters and try out the one-touch expansion and continuous layout features.

Listing 6-19. **SplitPaneTest.java**

Code View:

```
import java.awt.*;

import javax.swing.*;
import javax.swing.event.*;

/**
 * This program demonstrates the split pane component organizer.
 * @version 1.03 2007-08-01
 * @author Cay Horstmann
 */
public class SplitPaneTest
{
   public static void main(String[] args)
   {
      EventQueue.invokeLater(new Runnable()
         {
            public void run()
            {
               JFrame frame = new SplitPaneFrame();
               frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
               frame.setVisible(true);
            }
         });
   }
}

/**
 * This frame consists of two nested split panes to demonstrate planet images and data.
 */
class SplitPaneFrame extends JFrame
{
   public SplitPaneFrame()
   {
      setTitle("SplitPaneTest");
```

```
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

      // set up components for planet names, images, descriptions

      final JList planetList = new JList(planets);
      final JLabel planetImage = new JLabel();
      final JTextArea planetDescription = new JTextArea();

      planetList.addListSelectionListener(new ListSelectionListener()
```

```
         {
            public void valueChanged(ListSelectionEvent event)
            {
               Planet value = (Planet) planetList.getSelectedValue();

               // update image and description
```

```
               planetImage.setIcon(value.getImage());
               planetDescription.setText(value.getDescription());
            }
         });

      // set up split panes

      JSplitPane innerPane = new JSplitPane(JSplitPane.HORIZONTAL_SPLIT, planetList,
            planetImage);

      innerPane.setContinuousLayout(true);
      innerPane.setOneTouchExpandable(true);

      JSplitPane outerPane = new JSplitPane(JSplitPane.VERTICAL_SPLIT, innerPane,
            planetDescription);

      add(outerPane, BorderLayout.CENTER);
   }

   private Planet[] planets = { new Planet("Mercury", 2440, 0), new Planet("Venus", 6052, 0),
         new Planet("Earth", 6378, 1), new Planet("Mars", 3397, 2),
         new Planet("Jupiter", 71492, 16), new Planet("Saturn", 60268, 18),
         new Planet("Uranus", 25559, 17), new Planet("Neptune", 24766, 8),
         new Planet("Pluto", 1137, 1), };
   private static final int DEFAULT_WIDTH = 300;
   private static final int DEFAULT_HEIGHT = 300;
}

/**
 * Describes a planet.
 */
class Planet
{
   /**
    * Constructs a planet.
    * @param n the planet name
    * @param r the planet radius
    * @param m the number of moons
    */
   public Planet(String n, double r, int m)
   {
      name = n;
      radius = r;
```

```
      moons = m;
      image = new ImageIcon(name + ".gif");
   }

   public String toString()
   {
      return name;
   }

   /**
    * Gets a description of the planet.
    * @return the description
    */
   public String getDescription()
   {
      return "Radius: " + radius + "\nMoons: " + moons + "\n";
   }

   /**
    * Gets an image of the planet.
    * @return the image
    */
   public ImageIcon getImage()
   {
      return image;
   }

   private String name;
   private double radius;
   private int moons;
   private ImageIcon image;
}
```

API

javax.swing.JSplitPane 1.2

- `JSplitPane()`
- `JSplitPane(int direction)`
- `JSplitPane(int direction, boolean continuousLayout)`
- `JSplitPane(int direction, Component first, Component second)`
- `JSplitPane(int direction, boolean continuousLayout, Component first, Component second)`

constructs a new split pane.

Parameters:	`direction`	One of `HORIZONTAL_SPLIT` or `VERTICAL_SPLIT`
	`continousLayout`	`true` if the components are continuously updated when the splitter is moved
	`first, second`	The components to add

- `boolean isOneTouchExpandable()`

- `void setOneTouchExpandable(boolean b)`

  gets or sets the "one-touch expandable" property. When this property is set, the splitter has two icons to completely expand one or the other component.

- `boolean isContinuousLayout()`

- `void setContinuousLayout(boolean b)`

  gets or sets the "continuous layout" property. When this property is set, then the components are continuously updated when the splitter is moved.

- `void setLeftComponent(Component c)`

- `void setTopComponent(Component c)`

  These operations have the same effect, to set `c` as the first component in the split pane.

- `void setRightComponent(Component c)`

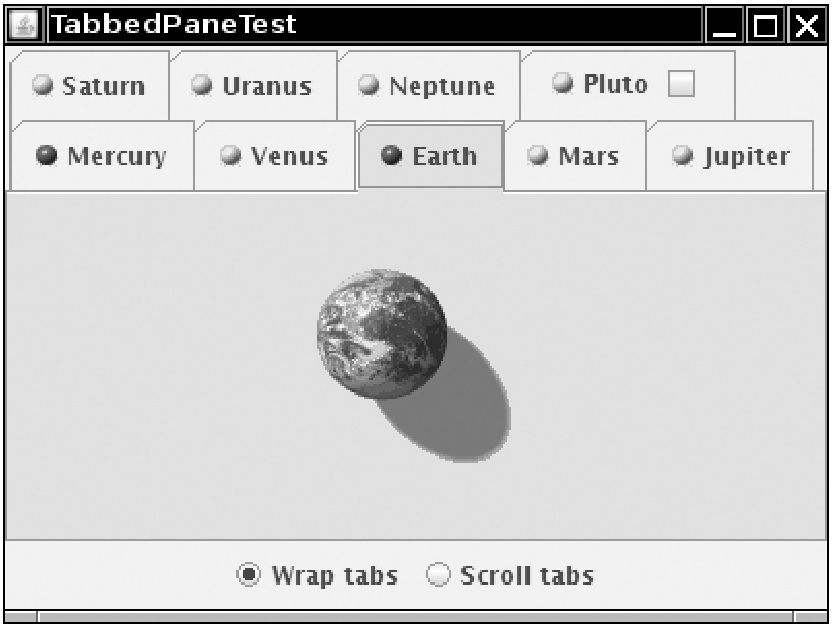
- `void setBottomComponent(Component c)`

  These operations have the same effect, to set `c` as the second component in the split pane.

Tabbed Panes

Tabbed panes are a familiar user interface device to break up a complex dialog box into subsets of related options. You can also use tabs to let a user flip through a set of documents or images (see Figure 6-45). That is what we do in our sample program.

Figure 6-45. A tabbed pane



To create a tabbed pane, you first construct a JTabbedPane object, then you add tabs to it.

```
JTabbedPane tabbedPane = new JTabbedPane();
tabbedPane.addTab(title, icon, component);
```

The last parameter of the addTab method has type Component. To add multiple components into the same tab, you first pack them up in a container, such as a JPanel.

The icon is optional; for example, the addTab method does not require an icon:

```
tabbedPane.addTab(title, component);
```

You can also add a tab in the middle of the tab collection with the insertTab method:

```
tabbedPane.insertTab(title, icon, component, tooltip, index);
```

To remove a tab from the tab collection, use

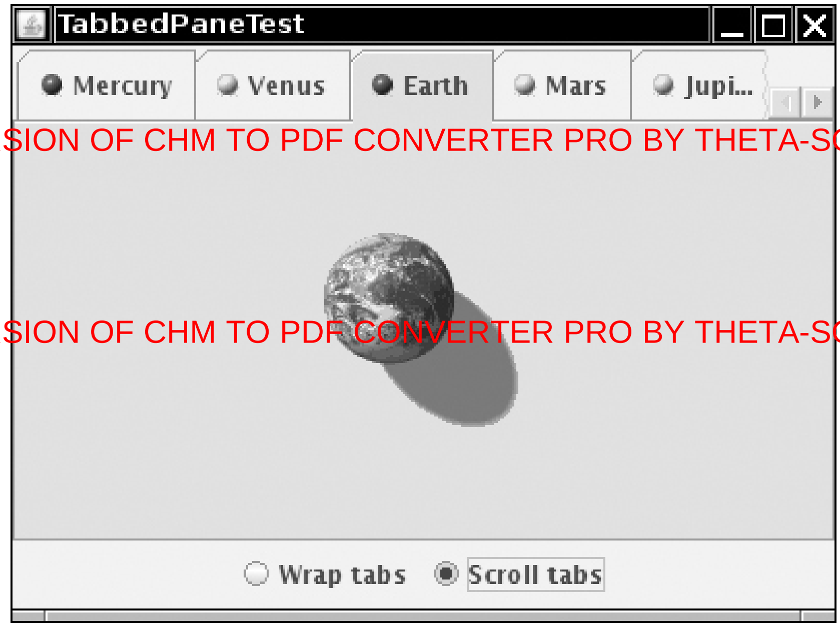
```
tabPane.removeTabAt(index);
```

When you add a new tab to the tab collection, it is not automatically displayed. You must select it with the setSelectedIndex method. For example, here is how you show a tab that you just added to the end:

```
tabbedPane.setSelectedIndex(tabbedPane.getTabCount() - 1);
```

If you have a lot of tabs, then they can take up quite a bit of space. Starting with Java SE 1.4, you can display the tabs in scrolling mode, in which only one row of tabs is displayed, together with a set of arrow buttons that allow the user to scroll through the tab set (see Figure 6-46).

Figure 6-46. A tabbed pane with scrolling tabs



You set the tab layout to wrapped or scrolling mode by calling

```
tabbedPane.setTabLayoutPolicy(JTabbedPane.WRAP_TAB_LAYOUT);
```

or

```
tabbedPane.setTabLayoutPolicy(JTabbedPane.SCROLL_TAB_LAYOUT);
```

The tab labels can have mnemonics, just like menu items. For example,

```
int marsIndex = tabbedPane.indexOfTab("Mars");
tabbedPane.setMnemonicAt(marsIndex, KeyEvent.VK_M);
```

Then the M is underlined, and program users can select the tab by pressing ALT+M.

As of Java SE 6, you can add arbitrary components into the tab titles. First add the tab, then call

```
tabbedPane.setTabComponentAt(index, component);
```

In our sample program, we add a "close box" to the Pluto tab (because, after all, some astronomers do not consider Pluto a real planet). This is achieved by setting the tab component to a panel containing two components: a label with the icon and tab text, and a checkbox with an action listener that removes the tab.

The example program shows a useful technique with tabbed panes. Sometimes, you want to update a component just before it is displayed. In our example program, we load the planet image only when the user actually clicks a tab.

To be notified whenever the user clicks on a new tab, you install a `ChangeListener` with the tabbed pane. Note that you must install the listener with the tabbed pane itself, not with any of the components.

```
tabbedPane.addChangeListener(listener);
```

When the user selects a tab, the `stateChanged` method of the change listener is called. You retrieve the tabbed pane as the source of the event. Call the `getSelectedIndex` method to find out which pane is about to be displayed.

```
public void stateChanged(ChangeEvent event)
{
   int n = tabbedPane.getSelectedIndex();
   loadTab(n);
}
```

In Listing 6-20, we first set all tab components to `null`. When a new tab is selected, we test whether its component is still `null`. If so, we replace it with the image. (This happens instantaneously when you click on the tab. You will not see an empty pane.) Just for fun, we also change the icon from a yellow ball to a red ball to indicate which panes have been visited.

Listing 6-20. **`TabbedPaneTest.java`**

Code View:

```
import java.awt.*;
import java.awt.event.*;

import javax.swing.*;
import javax.swing.event.*;

/**
 * This program demonstrates the tabbed pane component organizer.
 * @version 1.03 2007-08-01
 * @author Cay Horstmann
 */
public class TabbedPaneTest
{
   public static void main(String[] args)
   {
      EventQueue.invokeLater(new Runnable()
         {
            public void run()
            {

               JFrame frame = new TabbedPaneFrame();
               frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
               frame.setVisible(true);
            }
         });
   }
}

/**
 * This frame shows a tabbed pane and radio buttons to switch between wrapped and scrolling
 * tab layout.
 */
```

```
class TabbedPaneFrame extends JFrame
{
   public TabbedPaneFrame()
   {
      setTitle("TabbedPaneTest");
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

      tabbedPane = new JTabbedPane();
      // we set the components to null and delay their loading until the tab is shown
```

```
      // for the first time

      ImageIcon icon = new ImageIcon("yellow-ball.gif");

      tabbedPane.addTab("Mercury", icon, null);
      tabbedPane.addTab("Venus", icon, null);
      tabbedPane.addTab("Earth", icon, null);
```

```
      tabbedPane.addTab("Mars", icon, null);
      tabbedPane.addTab("Jupiter", icon, null);
      tabbedPane.addTab("Saturn", icon, null);
      tabbedPane.addTab("Uranus", icon, null);
      tabbedPane.addTab("Neptune", icon, null);
      tabbedPane.addTab("Pluto", null, null);

      final int plutoIndex = tabbedPane.indexOfTab("Pluto");
      JPanel plutoPanel = new JPanel();
      plutoPanel.add(new JLabel("Pluto", icon, SwingConstants.LEADING));
      JToggleButton plutoCheckBox = new JCheckBox();
      plutoCheckBox.addActionListener(new ActionListener()
      {
         public void actionPerformed(ActionEvent e)
         {
            tabbedPane.remove(plutoIndex);
         }
      });
      plutoPanel.add(plutoCheckBox);
      tabbedPane.setTabComponentAt(plutoIndex, plutoPanel);

      add(tabbedPane, "Center");

      tabbedPane.addChangeListener(new ChangeListener()
         {
            public void stateChanged(ChangeEvent event)
            {

               // check if this tab still has a null component

               if (tabbedPane.getSelectedComponent() == null)
               {
                  // set the component to the image icon

                  int n = tabbedPane.getSelectedIndex();
                  loadTab(n);
               }
            }
         });

      loadTab(0);

      JPanel buttonPanel = new JPanel();
```

```
      ButtonGroup buttonGroup = new ButtonGroup();
      JRadioButton wrapButton = new JRadioButton("Wrap tabs");
      wrapButton.addActionListener(new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               tabbedPane.setTabLayoutPolicy(JTabbedPane.WRAP_TAB_LAYOUT);
            }
         });
      buttonPanel.add(wrapButton);
      buttonGroup.add(wrapButton);
      wrapButton.setSelected(true);
      JRadioButton scrollButton = new JRadioButton("Scroll tabs");
      scrollButton.addActionListener(new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               tabbedPane.setTabLayoutPolicy(JTabbedPane.SCROLL_TAB_LAYOUT);
            }
         });
      buttonPanel.add(scrollButton);
      buttonGroup.add(scrollButton);
      add(buttonPanel, BorderLayout.SOUTH);
   }

   /**
    * Loads the tab with the given index.
    * @param n the index of the tab to load
    */
   private void loadTab(int n)
   {
      String title = tabbedPane.getTitleAt(n);
      ImageIcon planetIcon = new ImageIcon(title + ".gif");
      tabbedPane.setComponentAt(n, new JLabel(planetIcon));

      // indicate that this tab has been visited--just for fun

      tabbedPane.setIconAt(n, new ImageIcon("red-ball.gif"));
   }

   private JTabbedPane tabbedPane;

   private static final int DEFAULT_WIDTH = 400;
   private static final int DEFAULT_HEIGHT = 300;
}
```

API `javax.swing.JTabbedPane` 1.2

- `JTabbedPane()`

- `JTabbedPane(int placement)`

  constructs a tabbed pane.

Parameters:	`placement`	One of `SwingConstants.TOP`, `SwingConstants.LEFT`, `SwingConstants.RIGHT`, or `SwingConstants.BOTTOM`

- `void addTab(String title, Component c)`

- `void addTab(String title, Icon icon, Component c)`

- `void addTab(String title, Icon icon, Component c, String tooltip)`

  adds a tab to the end of the tabbed pane.

- `void insertTab(String title, Icon icon, Component c, String tooltip, int index)`

  inserts a tab to the tabbed pane at the given index.

- `void removeTabAt(int index)`

  removes the tab at the given index.

- `void setSelectedIndex(int index)`

  selects the tab at the given index.

- `int getSelectedIndex()`

  returns the index of the selected tab.

- `Component getSelectedComponent()`

  returns the component of the selected tab.

- `String getTitleAt(int index)`

- `void setTitleAt(int index, String title)`

- `Icon getIconAt(int index)`

- `void setIconAt(int index, Icon icon)`
- `Component getComponentAt(int index)`
- `void setComponentAt(int index, Component c)`

  gets or sets the title, icon, or component at the given index.

- `int indexOfTab(String title)`
- `int indexOfTab(Icon icon)`
- `int indexOfComponent(Component c)`

  returns the index of the tab with the given title, icon, or component.

- `int getTabCount()`

  returns the total number of tabs in this tabbed pane.

- `int getTabLayoutPolicy()`
- `void setTabLayoutPolicy(int policy)` **1.4**

  gets or sets the tab layout policy. `policy` is one of `JTabbedPane.WRAP_TAB_LAYOUT` or `JTabbedPane.SCROLL_TAB_LAYOUT`.

- `int getMnemonicAt(int index)` **1.4**
- `void setMnemonicAt(int index, int mnemonic)`

  gets or sets the mnemonic character at a given tab index. The character is specified as a VK_*X* constant from the `KeyEvent` class. -1 means that there is no mnemonic.

- `Component getTabComponentAt(int index)` **6**
- `void setTabComponentAt(int index, Component c)` **6**

  gets or sets the component that renders the title of the tab with the given index. If this component is null, the tab icon and title are rendered. Otherwise, only the given component is rendered in the tab.

- `int indexOfTabComponent(Component c)` **6**

  returns the index of the tab with the given title component.

- `void addChangeListener(ChangeListener listener)`

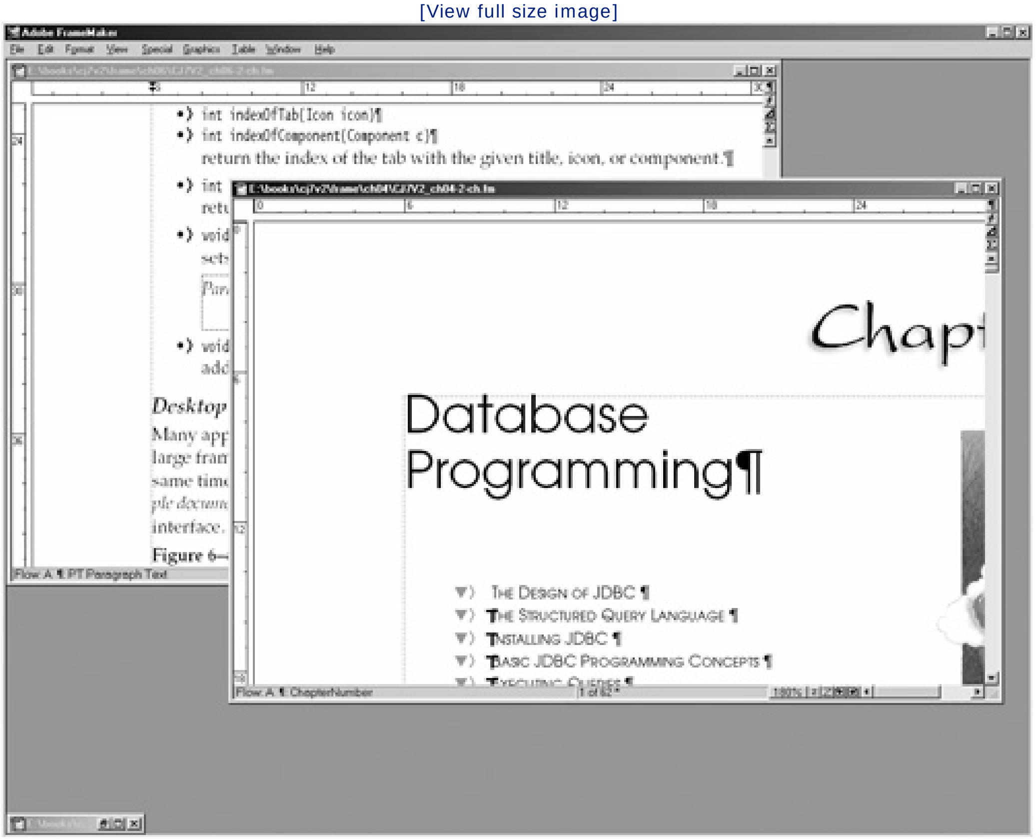
  adds a change listener that is notified when the user selects a different tab.

Desktop Panes and Internal Frames

Many applications present information in multiple windows that are all contained inside a large frame. If you minimize the application frame, then all of its windows are hidden at the same time. In the Windows environment, this user interface is sometimes called the *multiple document interface* (MDI). Figure 6-47 shows a typical application that uses this interface.

Figure 6-47. A multiple document interface application

[View full size image]



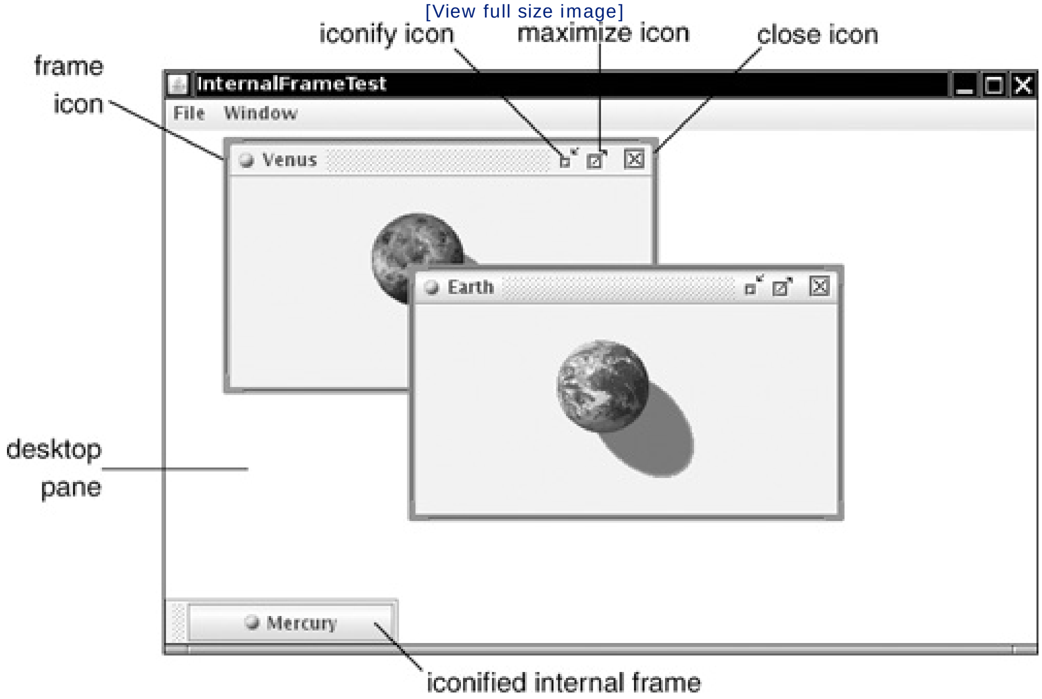
For some time, this user interface style was popular, but it has become less prevalent in recent years. Nowadays, many applications simply display a separate top-level frame for each document. Which is better? MDI reduces window clutter, but having separate top-level windows means that you can use the buttons and hotkeys of the host windowing system to flip through your windows.

In the world of Java, where you can't rely on a rich host windowing system, it makes a lot of sense to have your application manage its frames.

Figure 6-48 shows a Java application with three internal frames. Two of them have decorations on the border to maximize and iconify them. The third is in its iconified state.

Figure 6-48. A Java application with three internal frames

[View full size image]



In the Metal look and feel, the internal frames have distinctive "grabber" areas that you use to move the frames around. You can resize the windows by dragging the resize corners.

To achieve this capability, follow these steps:

1. Use a regular `JFrame` window for the application.

2. Add the `JDesktopPane` to the `JFrame`.

```
desktop = new JDesktopPane();
add(desktop, BorderLayout.CENTER);
```

3. Construct `JInternalFrame` windows. You can specify whether you want the icons for resizing or closing the frame. Normally, you want all icons.

```
JInternalFrame iframe = new JInternalFrame(title,
   true, // resizable
   true, // closable
   true, // maximizable
```

```
   true); // iconifiable
```

4. Add components to the frame.

```
iframe.add(c, BorderLayout.CENTER);
```

5. Set a frame icon. The icon is shown in the top-left corner of the frame.


```
iframe.setFrameIcon(icon);
```

Note

In the current version of the Metal look and feel, the frame icon is not displayed in iconized frames.

6. Set the size of the internal frame. As with regular frames, internal frames initially have a size of 0 by 0 pixels. Because you don't want internal frames to be displayed on top of each other, use a variable position for the next frame. Use the `reshape` method to set both the position and size of the frame:

```
iframe.reshape(nextFrameX, nextFrameY, width, height);
```

7. As with `JFrames`, you need to make the frame visible.

```
iframe.setVisible(true);
```

Note

In earlier versions of Swing, internal frames were automatically visible and this call was not necessary.

8. Add the frame to the `JDesktopPane`.

```
desktop.add(iframe);
```

9. You probably want to make the new frame the *selected frame*. Of the internal frames on the desktop, only the selected frame receives keyboard focus. In the Metal look and feel, the selected frame has a blue title bar, whereas the other frames have a gray title bar. You use the `setSelected` method to select a frame. However, the "selected" property can be *vetoed*—the currently selected frame can refuse to give up focus. In that case, the `setSelected` method throws a `PropertyVetoException` that you need to handle.

```
try
{
   iframe.setSelected(true);
}
```

```
catch (PropertyVetoException e)
{
   // attempt was vetoed
}
```

10. You probably want to move the position for the next internal frame down so that it won't overlay the existing frame. A good distance between frames is the height of the title bar, which you can obtain as

    Code View:
    ```
    int frameDistance = iframe.getHeight() - iframe.getContentPane().getHeight()
    ```

11. Use that distance to determine the next internal frame position.

    ```
    nextFrameX += frameDistance;
    nextFrameY += frameDistance;
    if (nextFrameX + width > desktop.getWidth())
       nextFrameX = 0;
    if (nextFrameY + height > desktop.getHeight())
       nextFrameY = 0;
    ```

Cascading and Tiling

In Windows, there are standard commands for *cascading* and *tiling* windows (see Figures 6-49 and 6-50). The Java JDesktopPane and JInternalFrame classes have no built-in support for these operations. In Listing 6-21, we show you how to implement these operations yourself.

Figure 6-49. Cascaded internal frames

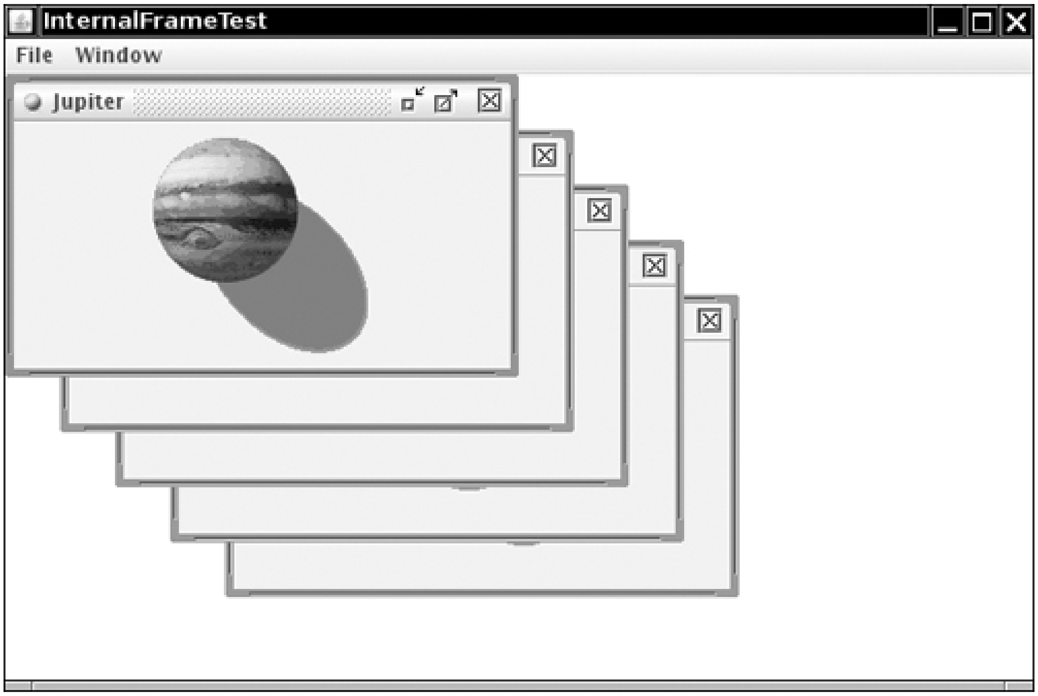
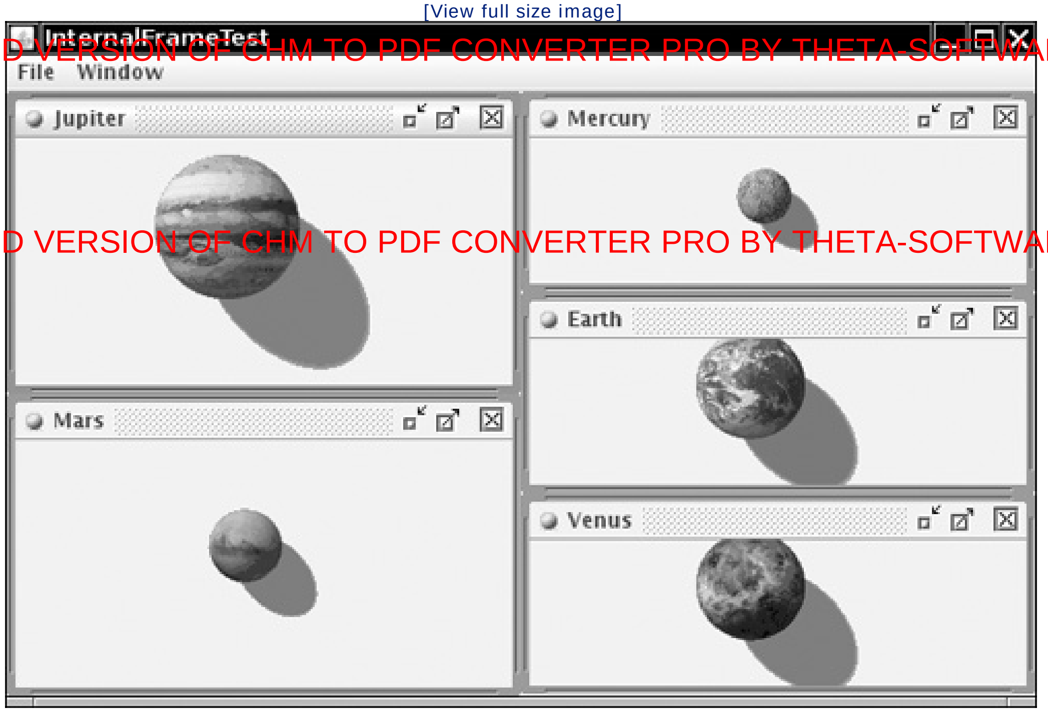


Figure 6-50. Tiled internal frames

[View full size image]



To cascade all windows, you reshape windows to the same size and stagger their positions. The `getAllFrames` method of the `JDesktopPane` class returns an array of all internal frames.

```
JInternalFrame[] frames = desktop.getAllFrames();
```

However, you need to pay attention to the frame state. An internal frame can be in one of three states:

- Icon
- Resizable
- Maximum

You use the `isIcon` method to find out which internal frames are currently icons and should be skipped. However, if a frame is in the maximum state, you first set it to be resizable by calling `setMaximum(false)`. This is another property that can be vetoed, so you must catch the `PropertyVetoException`.

The following loop cascades all internal frames on the desktop:

```
for (JInternalFrame frame : desktop.getAllFrames())
{
   if (!frame.isIcon())
   {
      try
      {
         // try to make maximized frames resizable; this might be vetoed
         frame.setMaximum(false);
         frame.reshape(x, y, width, height);
         x += frameDistance;
         y += frameDistance;
         // wrap around at the desktop edge
         if (x + width > desktop.getWidth()) x = 0;
         if (y + height > desktop.getHeight()) y = 0;
      }
      catch (PropertyVetoException e)
      {}
   }
}
```

Tiling frames is trickier, particularly if the number of frames is not a perfect square. First, count the number of frames that are not icons. Then, compute the number of rows as

```
int rows = (int) Math.sqrt(frameCount);
```

Then the number of columns is

```
int cols = frameCount / rows;
```

except that the last

```
int extra = frameCount % rows
```

columns have `rows + 1` rows.

Here is the loop for tiling all frames on the desktop:

Code View:

```
int width = desktop.getWidth() / cols;
int height = desktop.getHeight() / rows;
int r = 0;
int c = 0;
for (JInternalFrame frame : desktop.getAllFrames())
{
   if (!frame.isIcon())
   {
      try
      {
         frame.setMaximum(false);
         frame.reshape(c * width, r * height, width, height);
         r++;
         if (r == rows)
```

```
         {
            r = 0;
            c++;
            if (c == cols - extra)
            {
               // start adding an extra row
               rows++;
               height = desktop.getHeight() / rows;
            }
         }
      }
      catch (PropertyVetoException e)
      {}
   }
}
```


UNREGISTERED VERSION OF CHM TO PDF CONVERTER PRO BY THETA-SOFTWARE

The example program shows another common frame operation: moving the selection from the current frame to the next frame that isn't an icon. Traverse all frames and call `isSelected` until you find the currently selected frame. Then, look for the next frame in the sequence that isn't an icon, and try to select it by calling

```
frames[next].setSelected(true);
```

As before, that method can throw a `PropertyVetoException`, in which case you keep looking. If you come back to the original frame, then no other frame was selectable, and you give up. Here is the complete loop:

Code View:

```
JInternalFrame[] frames = desktop.getAllFrames();
for (int i = 0; i < frames.length; i++)
{
   if (frames[i].isSelected())
   {
      // find next frame that isn't an icon and can be selected
      int next = (i + 1) % frames.length;
      while (next != i)
      {
         if (!frames[next].isIcon())
         {
            try
            {
               // all other frames are icons or veto selection
               frames[next].setSelected(true);
               frames[next].toFront();
               frames[i].toBack();
               return;
            }
            catch (PropertyVetoException e)
            {}
         }
         next = (next + 1) % frames.length;
      }
   }
}
```

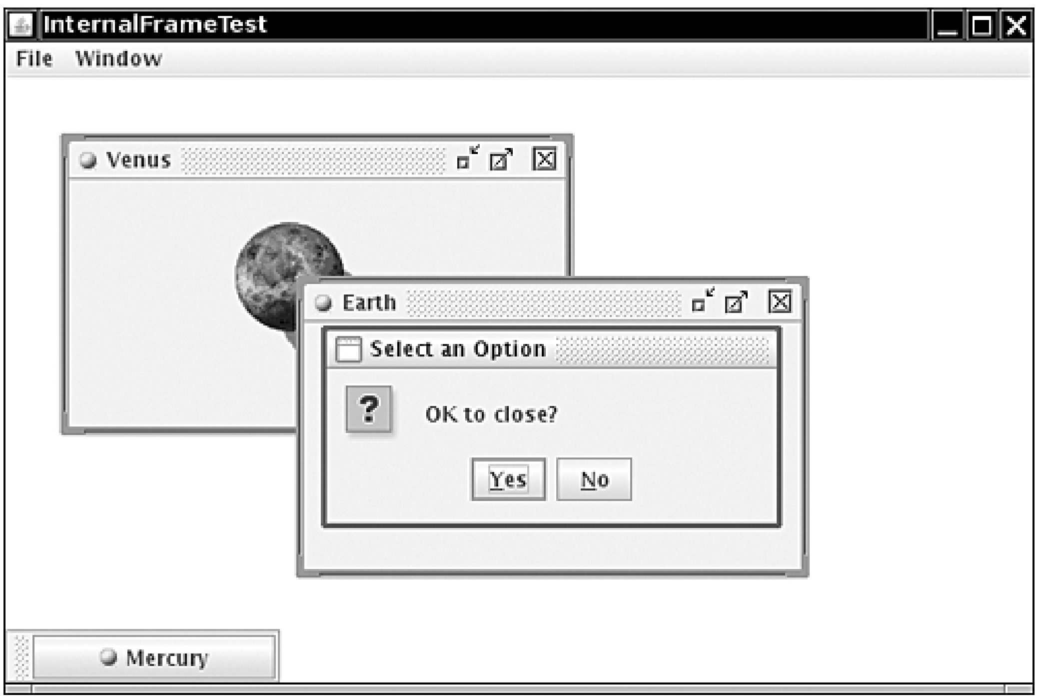
Vetoing Property Settings

Now that you have seen all these veto exceptions, you might wonder how your frames can issue a veto. The `JInternalFrame` class uses a general *JavaBeans* mechanism for monitoring the setting of properties. We discuss this mechanism in full detail in Chapter 8. For now, we just want to show you how your frames can veto requests for property changes.

Frames don't usually want to use a veto to protest iconization or loss of focus, but it is very common for frames to check whether it is okay to *close* them. You close a frame with the `setClosed` method of the `JInternalFrame` class. Because the method is vetoable, it calls all registered *vetoable change listeners* before proceeding to make the change. That gives each of the listeners the opportunity to throw a `PropertyVetoException` and thereby terminate the call to `setClosed` before it changed any settings.

In our example program, we put up a dialog box to ask the user whether it is okay to close the window (see Figure 6-51). If the user doesn't agree, the window stays open.

Figure 6-51. The user can veto the close property



Here is how you achieve such a notification.

1. Add a listener object to each frame. The object must belong to some class that implements the `VetoableChangeListener` interface. It is best to add the listener right after constructing the frame. In our example, we use the frame class that constructs the internal frames. Another option would be to use an anonymous inner class.

```
iframe.addVetoableChangeListener(listener);
```

2. Implement the vetoableChange method, the only method required by the VetoableChangeListener interface. The method receives a PropertyChangeEvent object. Use the getName method to find the name of the property that is about to be changed (such as "closed" if the method call to veto is setClosed(true)). As you see in Chapter 8, you obtain the property name by removing the "set" prefix from the method name and changing the next letter to lower case.

   Use the getNewValue method to get the proposed new value.

```
String name = event.getPropertyName();
Object value = event.getNewValue();
if (name.equals("closed") && value.equals(true))
{
   ask user for confirmation
}
```


3. Simply throw a PropertyVetoException to block the property change. Return normally if you don't want to veto the change.

```
class DesktopFrame extends JFrame
   implements VetoableChangeListener
{
   . . .
   public void vetoableChange(PropertyChangeEvent event)
      throws PropertyVetoException
   {
      . . .
      if (not ok)
         throw new PropertyVetoException(reason, event);
      // return normally if ok
   }
}
```

Dialogs in Internal Frames

If you use internal frames, you should not use the JDialog class for dialog boxes. Those dialog boxes have two disadvantages:

- They are heavyweight because they create a new frame in the windowing system.
- The windowing system does not know how to position them relative to the internal frame that spawned them.

Instead, for simple dialog boxes, use the showInternal*Xxx*Dialog methods of the JOptionPane class. They work exactly like the show*Xxx*Dialog methods, except they position a lightweight window over an internal frame.

As for more complex dialog boxes, construct them with a JInternalFrame. Unfortunately, you then have no built-in support for modal dialog boxes.

In our sample program, we use an internal dialog box to ask the user whether it is okay to close a frame.

```
int result = JOptionPane.showInternalConfirmDialog(
   iframe, "OK to close?", "Select an Option", JOptionPane.YES_NO_OPTION);;
```

Note

If you simply want to be *notified* when a frame is closed, then you should not use the veto mechanism. Instead, install an `InternalFrameListener`. An internal frame listener works just like a `WindowListener`. When the internal frame is closing, the `internalFrameClosing` method is called instead of the familiar `windowClosing` method. The other six internal frame notifications (opened/closed, iconified/deiconified, activated/deactivated) also correspond to the window listener methods.

Outline Dragging

One criticism that developers have leveled against internal frames is that performance has not been great. By far the slowest operation is to drag a frame with complex content across the desktop. The desktop manager keeps asking the frame to repaint itself as it is being dragged, which is quite slow.

Actually, if you use Windows or X Windows with a poorly written video driver, you'll experience the same problem. Window dragging appears to be fast on most systems because the video hardware supports the dragging operation by mapping the image inside the frame to a different screen location during the dragging process.

To improve performance without greatly degrading the user experience, you can set "outline dragging" on. When the user drags the frame, only the outline of the frame is continuously updated. The inside is repainted only when the user drops the frame to its final resting place.

To turn on outline dragging, call

```
desktop.setDragMode(JDesktopPane.OUTLINE_DRAG_MODE);
```

This setting is the equivalent of "continuous layout" in the `JSplitPane` class.

Note

In early versions of Swing, you had to use the magic incantation

```
desktop.putClientProperty("JDesktopPane.dragMode", "outline");
```

to turn on outline dragging.

In the sample program, you can use the Window -> Drag Outline checkbox menu selection to toggle outline dragging on or off.

Note

The internal frames on the desktop are managed by a DesktopManager class. You don't need to know about this class for normal programming. It is possible to implement different desktop behavior by installing a new desktop manager, but we don't cover that.

Listing 6-21 populates a desktop with internal frames that show HTML pages. The File -> Open menu option pops up a file dialog box for reading a local HTML file into a new internal frame. If you click on any link, the linked document is displayed in another internal frame. Try out the Window -> Cascade and Window -> Tile commands.

Listing 6-21. **InternalFrameTest.java**

Code View:

```
import java.awt.*;
import java.awt.event.*;
import java.beans.*;
import javax.swing.*;

/**
 * This program demonstrates the use of internal frames.
 * @version 1.11 2007-08-01
 * @author Cay Horstmann
 */
public class InternalFrameTest
{
   public static void main(String[] args)
   {
      EventQueue.invokeLater(new Runnable()
         {
            public void run()
            {
               JFrame frame = new DesktopFrame();
               frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
               frame.setVisible(true);
            }
         });
   }
}

/**
 * This desktop frame contains editor panes that show HTML documents.
 */
class DesktopFrame extends JFrame
{
```

```
   public DesktopFrame()
   {
      setTitle("InternalFrameTest");
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

      desktop = new JDesktopPane();
      add(desktop, BorderLayout.CENTER);

      // set up menus

      JMenuBar menuBar = new JMenuBar();
      setJMenuBar(menuBar);
      JMenu fileMenu = new JMenu("File");
      menuBar.add(fileMenu);
      JMenuItem openItem = new JMenuItem("New");
      openItem.addActionListener(new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               createInternalFrame(new JLabel(new ImageIcon(planets[counter] + ".gif")),
                     planets[counter]);
               counter = (counter + 1) % planets.length;
            }
         });
      fileMenu.add(openItem);
      JMenuItem exitItem = new JMenuItem("Exit");
      exitItem.addActionListener(new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               System.exit(0);
            }
         });
      fileMenu.add(exitItem);
      JMenu windowMenu = new JMenu("Window");
      menuBar.add(windowMenu);
      JMenuItem nextItem = new JMenuItem("Next");
      nextItem.addActionListener(new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               selectNextWindow();
            }
         });
      windowMenu.add(nextItem);
      JMenuItem cascadeItem = new JMenuItem("Cascade");
      cascadeItem.addActionListener(new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               cascadeWindows();
            }
         });
      windowMenu.add(cascadeItem);
      JMenuItem tileItem = new JMenuItem("Tile");
      tileItem.addActionListener(new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
```

```
               tileWindows();
            }
         });
      windowMenu.add(tileItem);
      final JCheckBoxMenuItem dragOutlineItem = new JCheckBoxMenuItem("Drag Outline");
      dragOutlineItem.addActionListener(new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
```

```
               desktop.setDragMode(dragOutlineItem.isSelected() ?
                     JDesktopPane.OUTLINE_DRAG_MODE : JDesktopPane.LIVE_DRAG_MODE);
            }
         });
      windowMenu.add(dragOutlineItem);
   }
```

```
   /**
    * Creates an internal frame on the desktop.
    * @param c the component to display in the internal frame
    * @param t the title of the internal frame.
    */
   public void createInternalFrame(Component c, String t)
   {
      final JInternalFrame iframe = new JInternalFrame(t, true, // resizable
            true, // closable
            true, // maximizable
            true); // iconifiable

      iframe.add(c, BorderLayout.CENTER);
      desktop.add(iframe);

      iframe.setFrameIcon(new ImageIcon("document.gif"));

      // add listener to confirm frame closing
      iframe.addVetoableChangeListener(new VetoableChangeListener()
         {
            public void vetoableChange(PropertyChangeEvent event) throws PropertyVetoException
            {
               String name = event.getPropertyName();
               Object value = event.getNewValue();

               // we only want to check attempts to close a frame
               if (name.equals("closed") && value.equals(true))
               {
                  // ask user if it is ok to close
                  int result = JOptionPane.showInternalConfirmDialog(iframe, "OK to close?",
                        "Select an Option", JOptionPane.YES_NO_OPTION);

                  // if the user doesn't agree, veto the close
                  if (result != JOptionPane.YES_OPTION) throw new PropertyVetoException(
                        "User canceled close", event);
               }
            }
         });

      // position frame
      int width = desktop.getWidth() / 2;
      int height = desktop.getHeight() / 2;
      iframe.reshape(nextFrameX, nextFrameY, width, height);
```

```java

      iframe.show();

      // select the frame--might be vetoed
      try
      {
         iframe.setSelected(true);
      }
      catch (PropertyVetoException e)
      {
      }

      frameDistance = iframe.getHeight() - iframe.getContentPane().getHeight();

      // compute placement for next frame

      nextFrameX += frameDistance;
      nextFrameY += frameDistance;
      if (nextFrameX + width > desktop.getWidth()) nextFrameX = 0;
      if (nextFrameY + height > desktop.getHeight()) nextFrameY = 0;
   }

   /**
    * Cascades the non-iconified internal frames of the desktop.
    */
   public void cascadeWindows()
   {
      int x = 0;
      int y = 0;
      int width = desktop.getWidth() / 2;
      int height = desktop.getHeight() / 2;

      for (JInternalFrame frame : desktop.getAllFrames())
      {
         if (!frame.isIcon())
         {
            try
            {
               // try to make maximized frames resizable; this might be vetoed
               frame.setMaximum(false);
               frame.reshape(x, y, width, height);

               x += frameDistance;
               y += frameDistance;
               // wrap around at the desktop edge
               if (x + width > desktop.getWidth()) x = 0;
               if (y + height > desktop.getHeight()) y = 0;
            }
            catch (PropertyVetoException e)
            {
            }
         }
      }
   }

   /**
    * Tiles the non-iconified internal frames of the desktop.
    */
   public void tileWindows()
```

```
   {
      // count frames that aren't iconized
      int frameCount = 0;
      for (JInternalFrame frame : desktop.getAllFrames())
         if (!frame.isIcon()) frameCount++;
      if (frameCount == 0) return;

      int rows = (int) Math.sqrt(frameCount);
      int cols = frameCount / rows;
```

```
      int extra = frameCount % rows;
      // number of columns with an extra row

      int width = desktop.getWidth() / cols;
      int height = desktop.getHeight() / rows;
      int r = 0;
```
UNREGISTERED VERSION OF CHM TO PDF CONVERTER PRO BY THETA-SOFTWARE
```
      int c = 0;
      for (JInternalFrame frame : desktop.getAllFrames())
      {
         if (!frame.isIcon())
         {
            try
            {
               frame.setMaximum(false);
               frame.reshape(c * width, r * height, width, height);
               r++;
               if (r == rows)
               {
                  r = 0;
                  c++;
                  if (c == cols - extra)
                  {
                     // start adding an extra row
                     rows++;
                     height = desktop.getHeight() / rows;
                  }
               }
            }
            catch (PropertyVetoException e)
            {
            }
         }
      }
   }

   /**
    * Brings the next non-iconified internal frame to the front.
    */
   public void selectNextWindow()
   {
      JInternalFrame[] frames = desktop.getAllFrames();
      for (int i = 0; i < frames.length; i++)
      {
         if (frames[i].isSelected())
         {
            // find next frame that isn't an icon and can be selected
            int next = (i + 1) % frames.length;
            while (next != i)
            {
               if (!frames[next].isIcon())
```

```
                {
                   try
                   {
                      // all other frames are icons or veto selection
                      frames[next].setSelected(true);
                      frames[next].toFront();
                      frames[i].toBack();
                      return;
                   }
                   catch (PropertyVetoException e)
                   {
                   }
                }
                next = (next + 1) % frames.length;
             }
          }
       }
    }

    private JDesktopPane desktop;
    private int nextFrameX;
    private int nextFrameY;
    private int frameDistance;
    private int counter;
    private static final String[] planets = { "Mercury", "Venus", "Earth", "Mars", "Jupiter",
          "Saturn", "Uranus", "Neptune", "Pluto", };

    private static final int DEFAULT_WIDTH = 600;
    private static final int DEFAULT_HEIGHT = 400;
 }
```

API

javax.swing.JDesktopPane 1.2

- `JInternalFrame[] getAllFrames()`

  gets all internal frames in this desktop pane.

- `void setDragMode(int mode)`

  sets the drag mode to live or outline drag mode.

Parameters:	`mode`	One of `JDesktopPane.LIVE_DRAG_MODE` or `JDesktopPane.OUTLINE_DRAG_MODE`

API javax.swing.JInternalFrame 1.2

- `JInternalFrame()`
- `JInternalFrame(String title)`

- `JInternalFrame(String title, boolean resizable)`
- `JInternalFrame(String title, boolean resizable, boolean closable)`

- `JInternalFrame(String title, boolean resizable, boolean closable, boolean maximizable)`
- `JInternalFrame(String title, boolean resizable, boolean closable, boolean maximizable, boolean iconifiable)`

  constructs a new internal frame.

Parameters:	`title`	The string to display in the title bar
	`resizable`	`true` if the frame can be resized
	`closable`	`true` if the frame can be closed
	`maximizable`	`true` if the frame can be maximized
	`iconifiable`	`true` if the frame can be iconified

- `boolean isResizable()`
- `void setResizable(boolean b)`
- `boolean isClosable()`
- `void setClosable(boolean b)`
- `boolean isMaximizable()`
- `void setMaximizable(boolean b)`
- `boolean isIconifiable()`

- `void setIconifiable(boolean b)`

  gets or sets the `resizable`, `closable`, `maximizable`, and `iconifiable` properties. When the property is `true`, an icon appears in the frame title to resize, close, maximize, or iconify the internal frame.

- `boolean isIcon()`

- `void setIcon(boolean b)`

- `boolean isMaximum()`

- `void setMaximum(boolean b)`

- `boolean isClosed()`

- `void setClosed(boolean b)`

  gets or sets the `icon`, `maximum`, or `closed` property. When this property is `true`, the internal frame is iconified, maximized, or closed.

- `boolean isSelected()`

- `void setSelected(boolean b)`

  gets or sets the `selected` property. When this property is `true`, the current internal frame becomes the selected frame on the desktop.

- `void moveToFront()`

- `void moveToBack()`

  moves this internal frame to the front or the back of the desktop.

- `void reshape(int x, int y, int width, int height)`

  moves and resizes this internal frame.

Parameters:	`x, y`	The top-left corner of the frame
	`width, height`	The width and height of the frame

- `Container getContentPane()`

- `void setContentPane(Container c)`

  gets or sets the content pane of this internal frame.

- `JDesktopPane getDesktopPane()`

  gets the desktop pane of this internal frame.

- `Icon getFrameIcon()`

- `void setFrameIcon(Icon anIcon)`

  gets or sets the frame icon that is displayed in the title bar.

- `boolean isVisible()`

- `void setVisible(boolean b)`

  gets or sets the "visible" property.

- `void show()`

  makes this internal frame visible and brings it to the front.

API `javax.swing.JComponent` 1.2

- `void addVetoableChangeListener(VetoableChangeListener listener)`

  adds a vetoable change listener that is notified when an attempt is made to change a constrained property.

API java.beans.VetoableChangeListener 1.1

- `void vetoableChange(PropertyChangeEvent event)`

  is called when the `set` method of a constrained property notifies the vetoable change listeners.

API java.beans.PropertyChangeEvent 1.1

- `String getPropertyName()`

  returns the name of the property that is about to be changed.

- `Object getNewValue()`

  returns the proposed new value for the property.

API java.beans.PropertyVetoException 1.1

- `PropertyVetoException(String reason, PropertyChangeEvent event)`

  constructs a property veto exception.

Parameters:	`reason`	The reason for the veto
	`event`	The vetoed event

You have now seen how to use the complex components that the Swing framework offers. In the next chapter, we turn to advanced AWT issues: complex drawing operations, image manipulation, printing, and interfacing with the native windowing system.

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

You can use the methods of the `Graphics` class to create simple drawings. Those methods are sufficient for simple applets and applications, but they fall short when you create complex shapes or when you require complete control over the appearance of the graphics. The Java 2D API is a more sophisticated class library that you can use to produce high-quality drawings. In this chapter, we give you an overview of that API.

We then turn to the topic of printing and show how you can implement printing capabilities into your programs.

Finally, we cover two techniques for transferring data between programs: the system clipboard and the drag-and-drop mechanism. You can use these techniques to transfer data between two Java applications or between a Java application and a native program.

The Rendering Pipeline

The original JDK 1.0 had a very simple mechanism for drawing shapes. You selected color and paint mode, and called methods of the `Graphics` class such as `drawRect` or `fillOval`. The Java 2D API supports many more options.

- You can easily produce a wide variety of *shapes.*
- You have control over the *stroke,* the pen that traces shape boundaries.
- You can *fill* shapes with solid colors, varying hues, and repeating patterns.
- You can use *transformations* to move, scale, rotate, or stretch shapes.
- You can *clip* shapes to restrict them to arbitrary areas.
- You can select *composition rules* to describe how to combine the pixels of a new shape with existing pixels.
- You can give *rendering hints* to make trade-offs between speed and drawing quality.

To draw a shape, you go through the following steps:

1. Obtain an object of the `Graphics2D` class. This class is a subclass of the `Graphics` class. Ever since Java SE 1.2, methods such as `paint` and `paintComponent` automatically receive an object of the `Graphics2D` class. Simply use a cast, as follows:

   ```
   public void paintComponent(Graphics g)
   ```

```
{
   Graphics2D g2 = (Graphics2D) g;
   . . .
}
```

2. Use the `setRenderingHints` method to set *rendering hints*: trade-offs between speed and drawing quality.


   ```
   RenderingHints hints = . . .;
   g2.setRenderingHints(hints);
   ```

3. Use the `setStroke` method to set the *stroke.* The stroke draws the outline of the shape. You can select the thickness and choose among solid and dotted lines.


   ```
   Stroke stroke = . . .;
   g2.setStroke(stroke);
   ```

4. Use the `setPaint` method to set the *paint.* The paint fills areas such as the stroke path or the interior of a shape. You can create solid color paint, paint with changing hues, or tiled fill patterns.

   ```
   Paint paint = . . .;
   g2.setPaint(paint);
   ```

5. Use the `clip` method to set the *clipping region*.

   ```
   Shape clip = . . .;
   g2.clip(clip);
   ```

6. Use the `transform` method to set a *transformation* from user space to device space. You use transformations if it is easier for you to define your shapes in a custom coordinate system than by using pixel coordinates.

   ```
   AffineTransform transform = . . .;
   g2.transform(transform);
   ```

7. Use the `setComposite` method to set a *composition rule* that describes how to combine the new pixels with the existing pixels.

   ```
   Composite composite = . . .;
   g2.setComposite(composite);
   ```

8. Create a shape. The Java 2D API supplies many shape objects and methods to combine shapes.

   ```
   Shape shape = . . .;
   ```

9. Draw or fill the shape. If you draw the shape, its outline is stroked. If you fill the shape, the interior is painted.

   ```
   g2.draw(shape);
   g2.fill(shape);
   ```

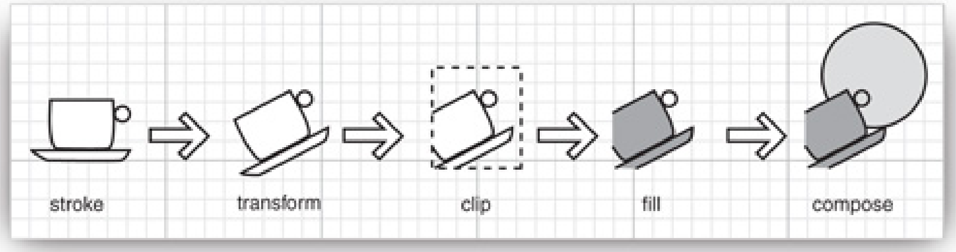
Of course, in many practical circumstances, you don't need all these steps. There are reasonable defaults for the settings of the 2D graphics context. You would change the settings only if you want to change the defaults.

In the following sections, you will see how to describe shapes, strokes, paints, transformations, and composition rules.

The various `set` methods simply set the state of the 2D graphics context. They don't cause any drawing. Similarly, when you construct `Shape` objects, no drawing takes place. A shape is only rendered when you call `draw` or `fill`. At that time, the new shape is computed in a *rendering pipeline* (see Figure 7-1).

Figure 7-1. The rendering pipeline

[View full size image]



In the rendering pipeline, the following steps take place to render a shape:

1. The path of the shape is stroked.
2. The shape is transformed.
3. The shape is clipped. If there is no intersection between the shape and the clipping area, then the process stops.
4. The remainder of the shape after clipping is filled.
5. The pixels of the filled shape are composed with the existing pixels. (In Figure 7-1, the circle is part of the existing pixels, and the cup shape is superimposed over it.)

In the next section, you will see how to define shapes. Then, we turn to the 2D graphics context settings.

API

`java.awt.Graphics2D` 1.2

- `void draw(Shape s)`

  draws the outline of the given shape with the current stroke.

- `void fill(Shape s)`

  fills the interior of the given shape with the current paint.

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

You can use the methods of the `Graphics` class to create simple drawings. Those methods are sufficient for simple applets and applications, but they fall short when you create complex shapes or when you require complete control over the appearance of the graphics. The Java 2D API is a more sophisticated class library that you can use to produce high-quality drawings. In this chapter, we give you an overview of that API.

We then turn to the topic of printing and show how you can implement printing capabilities into your programs.

Finally, we cover two techniques for transferring data between programs: the system clipboard and the drag-and-drop mechanism. You can use these techniques to transfer data between two Java applications or between a Java application and a native program.

The Rendering Pipeline

The original JDK 1.0 had a very simple mechanism for drawing shapes. You selected color and paint mode, and called methods of the `Graphics` class such as `drawRect` or `fillOval`. The Java 2D API supports many more options.

- You can easily produce a wide variety of *shapes.*
- You have control over the *stroke,* the pen that traces shape boundaries.
- You can *fill* shapes with solid colors, varying hues, and repeating patterns.
- You can use *transformations* to move, scale, rotate, or stretch shapes.
- You can *clip* shapes to restrict them to arbitrary areas.
- You can select *composition rules* to describe how to combine the pixels of a new shape with existing pixels.
- You can give *rendering hints* to make trade-offs between speed and drawing quality.

To draw a shape, you go through the following steps:

1. Obtain an object of the `Graphics2D` class. This class is a subclass of the `Graphics` class. Ever since Java SE 1.2, methods such as `paint` and `paintComponent` automatically receive an object of the `Graphics2D` class. Simply use a cast, as follows:

```
public void paintComponent(Graphics g)
```

```
{
   Graphics2D g2 = (Graphics2D) g;
   . . .
}
```

2. Use the `setRenderingHints` method to set *rendering hints*: trade-offs between speed and drawing quality.

   ```
   RenderingHints hints = . . .;
   g2.setRenderingHints(hints);
   ```

3. Use the `setStroke` method to set the *stroke.* The stroke draws the outline of the shape. You can select the thickness and choose among solid and dotted lines.

   ```
   Stroke stroke = . . .;
   g2.setStroke(stroke);
   ```

4. Use the `setPaint` method to set the *paint.* The paint fills areas such as the stroke path or the interior of a shape. You can create solid color paint, paint with changing hues, or tiled fill patterns.

   ```
   Paint paint = . . .;
   g2.setPaint(paint);
   ```

5. Use the `clip` method to set the *clipping region*.

   ```
   Shape clip = . . .;
   g2.clip(clip);
   ```

6. Use the `transform` method to set a *transformation* from user space to device space. You use transformations if it is easier for you to define your shapes in a custom coordinate system than by using pixel coordinates.

   ```
   AffineTransform transform = . . .;
   g2.transform(transform);
   ```

7. Use the `setComposite` method to set a *composition rule* that describes how to combine the new pixels with the existing pixels.

   ```
   Composite composite = . . .;
   g2.setComposite(composite);
   ```

8. Create a shape. The Java 2D API supplies many shape objects and methods to combine shapes.

   ```
   Shape shape = . . .;
   ```

9. Draw or fill the shape. If you draw the shape, its outline is stroked. If you fill the shape, the interior is painted.

   ```
   g2.draw(shape);
   g2.fill(shape);
   ```

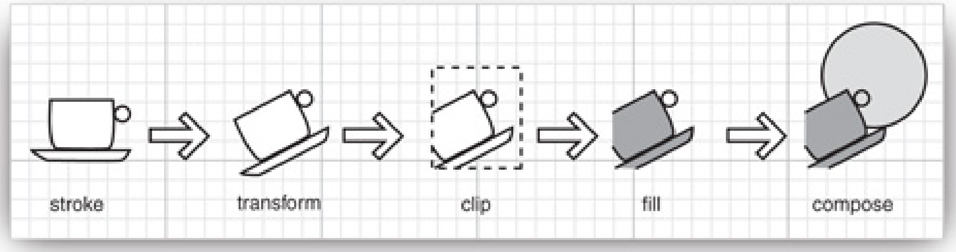
Of course, in many practical circumstances, you don't need all these steps. There are reasonable defaults for the settings of the 2D graphics context. You would change the settings only if you want to change the defaults.

In the following sections, you will see how to describe shapes, strokes, paints, transformations, and composition rules.

The various `set` methods simply set the state of the 2D graphics context. They don't cause any drawing. Similarly, when you construct `Shape` objects, no drawing takes place. A shape is only rendered when you call `draw` or `fill`. At that time, the new shape is computed in a *rendering pipeline* (see Figure 7-1).

Figure 7-1. The rendering pipeline

[View full size image]



In the rendering pipeline, the following steps take place to render a shape:

1. The path of the shape is stroked.
2. The shape is transformed.
3. The shape is clipped. If there is no intersection between the shape and the clipping area, then the process stops.
4. The remainder of the shape after clipping is filled.
5. The pixels of the filled shape are composed with the existing pixels. (In Figure 7-1, the circle is part of the existing pixels, and the cup shape is superimposed over it.)

In the next section, you will see how to define shapes. Then, we turn to the 2D graphics context settings.

API

`java.awt.Graphics2D` 1.2

- `void draw(Shape s)`

  draws the outline of the given shape with the current stroke.

- `void fill(Shape s)`

  fills the interior of the given shape with the current paint.

Shapes

Here are some of the methods in the `Graphics` class to draw shapes:

```
drawLine
drawRectangle
drawRoundRect
draw3DRect
drawPolygon
drawPolyline
drawOval
drawArc
```


There are also corresponding `fill` methods. These methods have been in the `Graphics` class ever since JDK 1.0. The Java 2D API uses a completely different, object-oriented approach. Instead of methods, there are classes:

```
Line2D
Rectangle2D
RoundRectangle2D
Ellipse2D
Arc2D
QuadCurve2D
CubicCurve2D
GeneralPath
```

These classes all implement the `Shape` interface.

Finally, the `Point2D` class describes a point with an *x*- and a *y*- coordinate. Points are useful to define shapes, but they aren't themselves shapes.

To draw a shape, you first create an object of a class that implements the `Shape` interface and then call the `draw` method of the `Graphics2D` class.

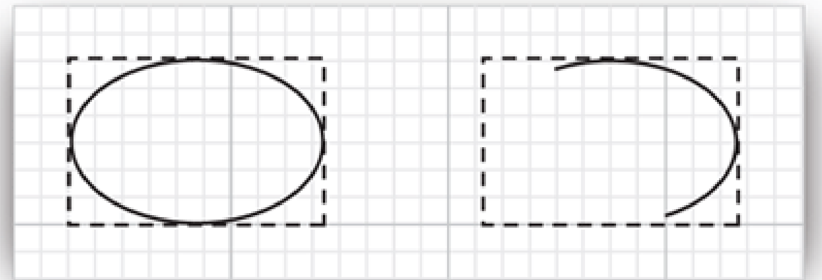
The `Line2D`, `Rectangle2D`, `RoundRectangle2D`, `Ellipse2D`, and `Arc2D` classes correspond to the `drawLine`, `drawRectangle`, `drawRoundRect`, `drawOval`, and `drawArc` methods. (The concept of a "3D rectangle" has died the death that it so richly deserved—there is no analog to the `draw3DRect` method.) The Java 2D API supplies two additional classes: quadratic and cubic curves. We discuss these shapes later in this section. There is no `Polygon2D` class. Instead, the `GeneralPath` class describes paths that are made up from lines, quadratic and cubic curves. You can use a `GeneralPath` to describe a polygon; we show you how later in this section.

The classes

```
Rectangle2D
RoundRectangle2D
Ellipse2D
Arc2D
```

all inherit from a common superclass `RectangularShape`. Admittedly, ellipses and arcs are not rectangular, but they have a *bounding rectangle* (see Figure 7-2).

Figure 7-2. The bounding rectangle of an ellipse and an arc



Each of the classes with a name ending in "2D" has two subclasses for specifying coordinates as `float` or `double` quantities. In Volume I, you already encountered `Rectangle2D.Float` and `Rectangle2D.Double`.

The same scheme is used for the other classes, such as `Arc2D.Float` and `Arc2D.Double`.

Internally, all graphics classes use `float` coordinates because `float` numbers use less storage space and they have sufficient precision for geometric computations. However, the Java programming language makes it a bit more tedious to manipulate `float` numbers. For that reason, most methods of the graphics classes use `double` parameters and return values. Only when constructing a 2D object must you choose between a constructor with `float` or `double` coordinates. For example,

```
Rectangle2D floatRect = new Rectangle2D.Float(5F, 10F, 7.5F, 15F);
Rectangle2D doubleRect = new Rectangle2D.Double(5, 10, 7.5, 15);
```

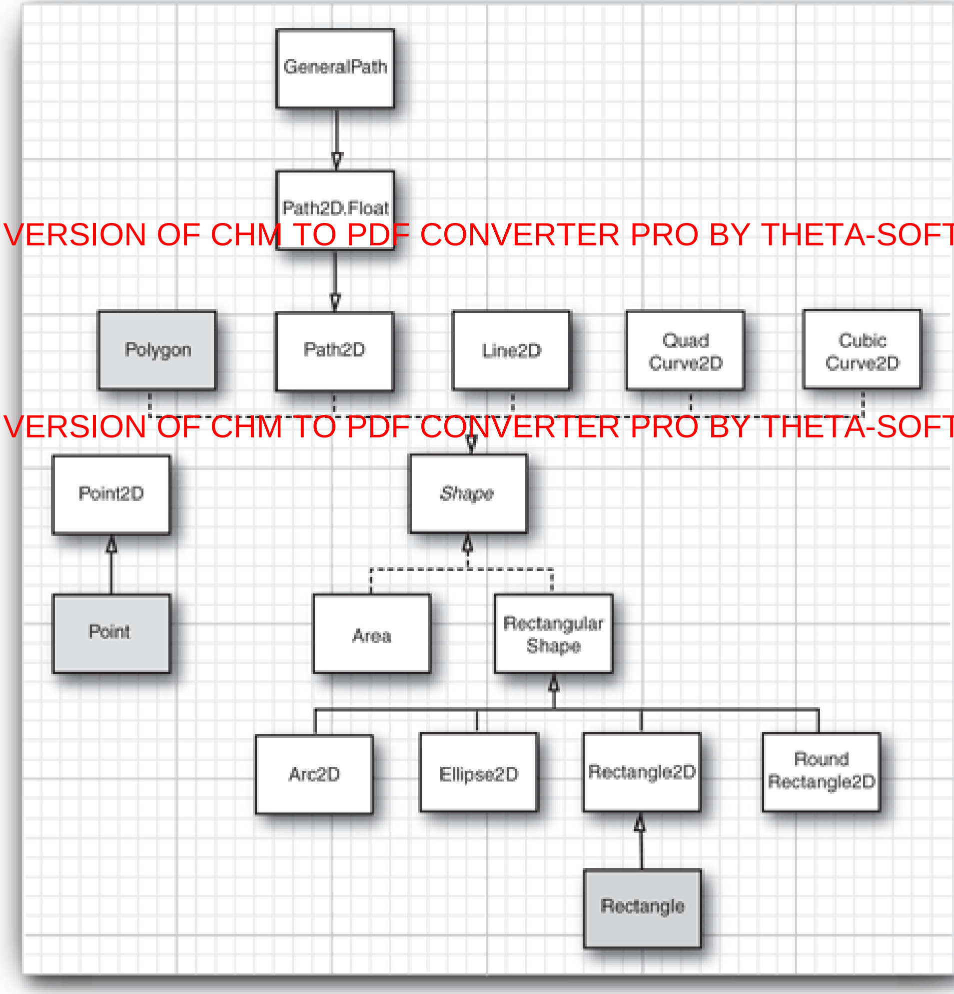
The *Xxx*`2D.Float` and *Xxx*`2D.Double` classes are subclasses of the *Xxx*`2D` classes. After object construction, essentially no benefit accrues from remembering the subclass, and you can just store the constructed object in a superclass variable, just as in the code example.

As you can see from the curious names, the *Xxx*`2D.Float` and *Xxx*`2D.Double` classes are also inner classes of the *Xxx*`2D` classes. That is just a minor syntactical convenience, to avoid an inflation of outer class names.

Figure 7-3 shows the relationships between the shape classes. However, the `Double` and `Float` subclasses are omitted. Legacy classes from the pre-2D library are marked with a gray fill.

Figure 7-3. Relationships between the shape classes

[View full size image]



UNREGISTERED VERSION OF CHM TO PDF CONVERTER PRO BY THETA-SOFTWARE

Using the Shape Classes

You already saw how to use the `Rectangle2D`, `Ellipse2D`, and `Line2D` classes in Volume I, Chapter 7. In this section, you will learn how to work with the remaining 2D shapes.

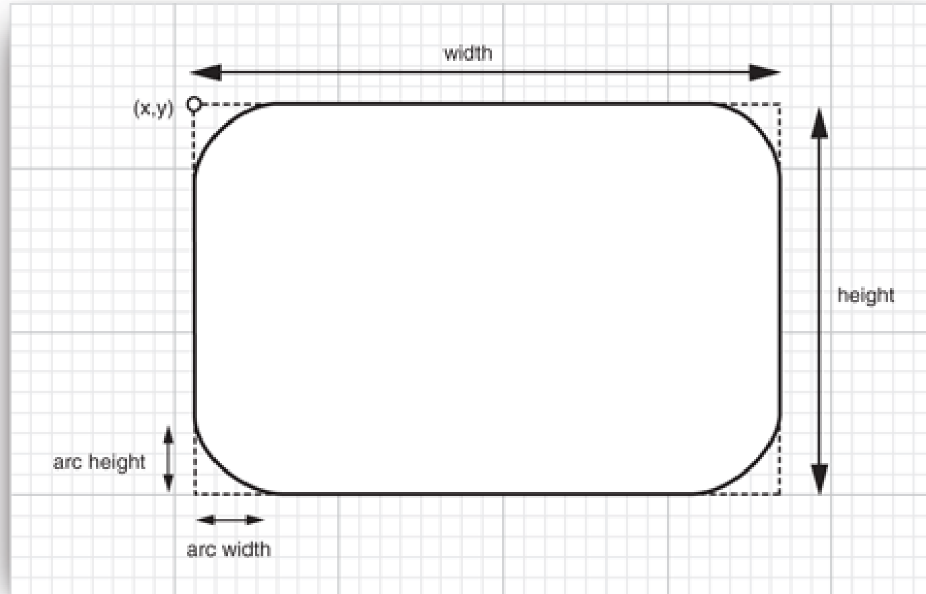
For the `RoundRectangle2D` shape, you specify the top-left corner, width and height, and the *x*- and *y*-dimension of the corner area that should be rounded (see Figure 7-4). For example, the call

Code View:
```
RoundRectangle2D r = new RoundRectangle2D.Double(150, 200, 100, 50, 20, 20);
```

produces a rounded rectangle with circles of radius 20 at each of the corners.

Figure 7-4. Constructing a **`RoundRectangle2D`**

[View full size image]



To construct an arc, you specify the bounding box, the start angle, the angle swept out by the arc (see Figure 7-5), and the closure type, one of `Arc2D.OPEN`, `Arc2D.PIE`, or `Arc2D.CHORD`.

Code View:

```
Arc2D a = new Arc2D(x, y, width, height, startAngle, arcAngle, closureType);
```

Figure 7-5. Constructing an elliptical arc

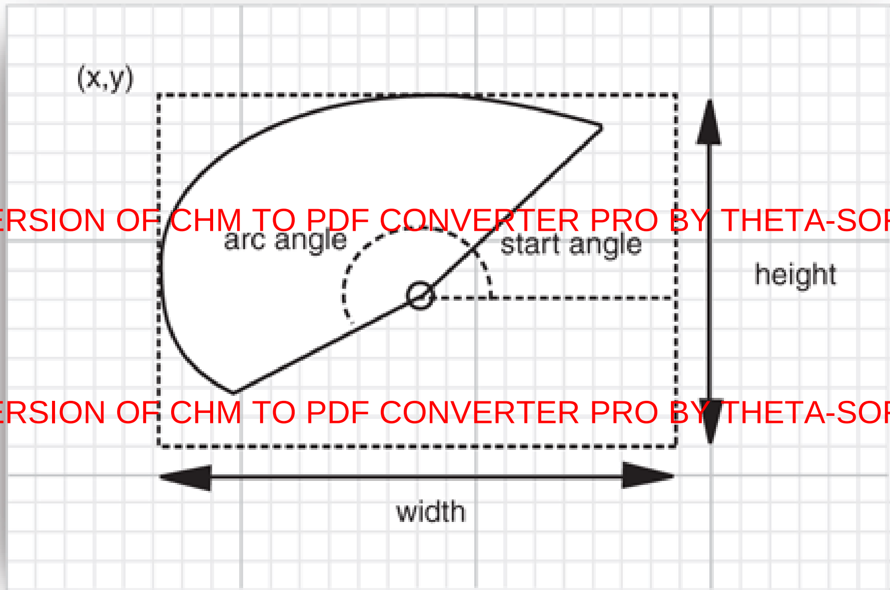
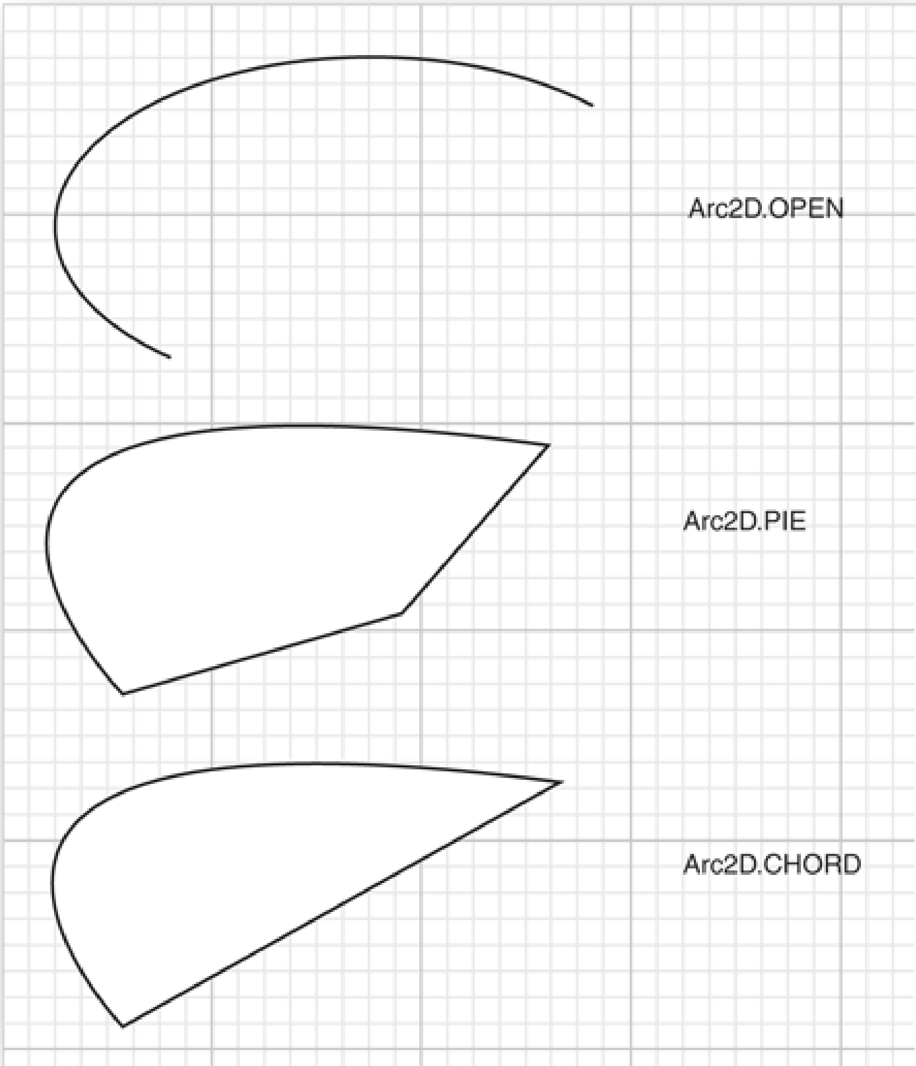


Figure 7-6 illustrates the arc types.

Figure 7-6. Arc types



Caution

If the arc is elliptical, the computation of the arc angles is not at all straightforward. The API documentation states: "The angles are specified relative to the non-square framing rectangle such that 45 degrees always falls on the line from the center of the ellipse to the upper right corner of the framing rectangle. As a result, if the framing rectangle is noticeably longer along one axis than the other, the angles to the start and end of the arc segment will be skewed farther along the longer axis of the frame." Unfortunately, the documentation is silent on how to compute this "skew." Here are the details:

Suppose the center of the arc is the origin and the point (*x*, *y*) lies on the arc. You get a skewed angle with the following formula:

```
skewedAngle = Math.toDegrees(Math.atan2(x * width, y * height));
```


The result is a value between -180 and 180. Compute the skewed start and end angles in this way. Then, compute the difference between the two skewed angles. If the start angle or the angle difference is negative, add 360. Then, supply the start angle and the angle difference to the arc constructor.

If you run the example program at the end of this section, then you can visually check that this calculation yields the correct values for the arc constructor (see Figure 7-9 on page 531).

The Java 2D API supports *quadratic* and *cubic* curves. In this chapter, we do not get into the mathematics of these curves. We suggest you get a feel for how the curves look by running the program in Listing 7-1. As you can see in Figures 7-7 and 7-8, quadratic and cubic curves are specified by two *end points* and one or two *control points.* Moving the control points changes the shape of the curves.

Figure 7-7. A quadratic curve

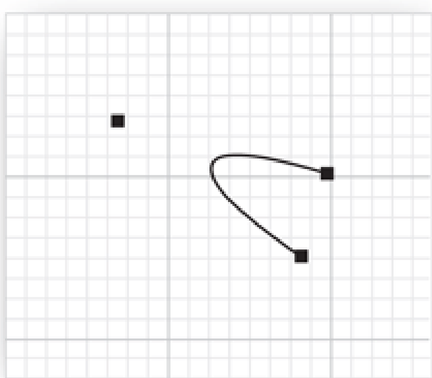
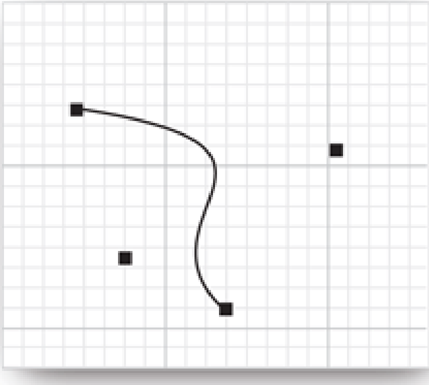


Figure 7-8. A cubic curve



To construct quadratic and cubic curves, you give the coordinates of the end points and the control points. For example,

Code View:
```
QuadCurve2D q = new QuadCurve2D.Double(startX, startY, controlX, controlY, endX, endY);
CubicCurve2D c = new CubicCurve2D.Double(startX, startY, control1X, control1Y,
   control2X, control2Y, endX, endY);
```

Quadratic curves are not very flexible, and they are not commonly used in practice. Cubic curves (such as the Bezier curves drawn by the `CubicCurve2D` class) are, however, very common. By combining many cubic curves so that the slopes at the connection points match, you can create complex, smooth-looking curved shapes. For more information, we refer you to *Computer Graphics: Principles and Practice, Second Edition in C* by James D. Foley, Andries van Dam, Steven K. Feiner, et al. (Addison-Wesley 1995).

You can build arbitrary sequences of line segments, quadratic curves, and cubic curves, and store them in a `GeneralPath` object. You specify the first coordinate of the path with the `moveTo` method. For example,

```
GeneralPath path = new GeneralPath();
path.moveTo(10, 20);
```

You then extend the path by calling one of the methods `lineTo`, `quadTo`, or `curveTo`. These methods extend the path by a line, a quadratic curve, or a cubic curve. To call `lineTo`, supply the end point. For the two curve methods, supply the control points, then the end point. For example,

```
path.lineTo(20, 30);
path.curveTo(control1X, control1Y, control2X, control2Y, endX, endY);
```

You close the path by calling the `closePath` method. It draws a line back to the starting point of the path.

To make a polygon, simply call `moveTo` to go to the first corner point, followed by repeated calls to `lineTo` to visit the other corner points. Finally, call `closePath` to close the polygon. The program in Listing 7-1 shows this in more detail.

A general path does not have to be connected. You can call moveTo at any time to start a new path segment.

Finally, you can use the append method to add arbitrary Shape objects to a general path. The outline of the shape is added to the end to the path. The second parameter of the append method is true if the new shape should be connected to the last point on the path, false if it should not be connected. For example, the call

```
Rectangle2D r = . . .;
path.append(r, false);
```


appends the outline of a rectangle to the path without connecting it to the existing path. But

```
path.append(r, true);
```


adds a straight line from the end point of the path to the starting point of the rectangle, and then adds the rectangle outline to the path.

The program in Listing 7-1 lets you create sample paths. Figures 7-7 and 7-8 show sample runs of the program. You pick a shape maker from the combo box. The program contains shape makers for

- Straight lines.
- Rectangles, round rectangles, and ellipses.
- Arcs (showing lines for the bounding rectangle and the start and end angles, in addition to the arc itself).
- Polygons (using a GeneralPath).
- Quadratic and cubic curves.

Use the mouse to adjust the control points. As you move them, the shape continuously repaints itself.

The program is a bit complex because it handles a multiplicity of shapes and supports dragging of the control points.

An abstract superclass ShapeMaker encapsulates the commonality of the shape maker classes. Each shape has a fixed number of control points that the user can move around. The getPointCount method returns that value. The abstract method

```
Shape makeShape(Point2D[] points)
```

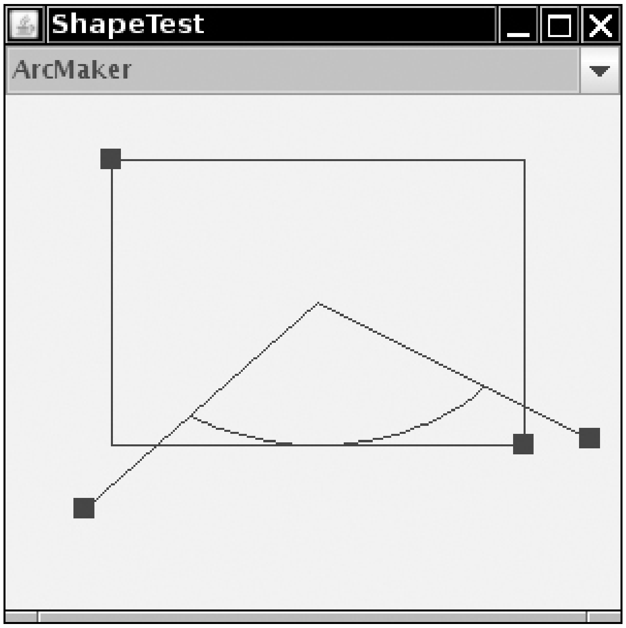
computes the actual shape, given the current positions of the control points. The toString method returns the class name so that the ShapeMaker objects can simply be dumped into a JComboBox.

To enable dragging of the control points, the ShapePanel class handles both mouse and mouse motion events. If the mouse is pressed on top of a rectangle, subsequent mouse drags move the rectangle.

The majority of the shape maker classes are simple—their makeShape methods just construct and return the requested shape. However, the ArcMaker class needs to compute the distorted start and end angles.

Furthermore, to demonstrate that the computation is indeed correct, the returned shape is a GeneralPath containing the arc itself, the bounding rectangle, and the lines from the center of the arc to the angle control points (see Figure 7-9).

Figure 7-9. The ShapeTest program



Listing 7-1. **ShapeTest.java**

Code View:
```
import java.awt.*;
import java.awt.event.*;
import java.awt.geom.*;
import java.util.*;
import javax.swing.*;

/**
 * This program demonstrates the various 2D shapes.
 * @version 1.02 2007-08-16
 * @author Cay Horstmann
 */
public class ShapeTest
{
   public static void main(String[] args)
   {
      EventQueue.invokeLater(new Runnable()
         {
            public void run()
            {
               JFrame frame = new ShapeTestFrame();
               frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
               frame.setVisible(true);
```

```
            }
         });
   }
}

/**
 * This frame contains a combo box to select a shape and a component to draw it.
 */
class ShapeTestFrame extends JFrame
```

```
{
   public ShapeTestFrame()
   {
      setTitle("ShapeTest");
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

      final ShapeComponent comp = new ShapeComponent();
```

```
      add(comp, BorderLayout.CENTER);
      final JComboBox comboBox = new JComboBox();
      comboBox.addItem(new LineMaker());
      comboBox.addItem(new RectangleMaker());
      comboBox.addItem(new RoundRectangleMaker());
      comboBox.addItem(new EllipseMaker());
      comboBox.addItem(new ArcMaker());
      comboBox.addItem(new PolygonMaker());
      comboBox.addItem(new QuadCurveMaker());
      comboBox.addItem(new CubicCurveMaker());
      comboBox.addActionListener(new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               ShapeMaker shapeMaker = (ShapeMaker) comboBox.getSelectedItem();
               comp.setShapeMaker(shapeMaker);
            }
         });
      add(comboBox, BorderLayout.NORTH);
      comp.setShapeMaker((ShapeMaker) comboBox.getItemAt(0));
   }

   private static final int DEFAULT_WIDTH = 300;
   private static final int DEFAULT_HEIGHT = 300;
}

/**
 * This component draws a shape and allows the user to move the points that define it.
 */
class ShapeComponent extends JComponent
{
   public ShapeComponent()
   {
      addMouseListener(new MouseAdapter()
         {
            public void mousePressed(MouseEvent event)
            {
               Point p = event.getPoint();
               for (int i = 0; i < points.length; i++)
               {
                  double x = points[i].getX() - SIZE / 2;
                  double y = points[i].getY() - SIZE / 2;
                  Rectangle2D r = new Rectangle2D.Double(x, y, SIZE, SIZE);
```

```
                  if (r.contains(p))
                  {
                     current = i;
                     return;
                  }
               }
            }

            public void mouseReleased(MouseEvent event)
            {
               current = -1;
            }
         });
      addMouseMotionListener(new MouseMotionAdapter()
         {
            public void mouseDragged(MouseEvent event)
            {
               if (current == -1) return;
               points[current] = event.getPoint();
               repaint();
            }
         });
      current = -1;
   }

   /**
    * Set a shape maker and initialize it with a random point set.
    * @param aShapeMaker a shape maker that defines a shape from a point set
    */
   public void setShapeMaker(ShapeMaker aShapeMaker)
   {
      shapeMaker = aShapeMaker;
      int n = shapeMaker.getPointCount();
      points = new Point2D[n];
      for (int i = 0; i < n; i++)
      {
         double x = generator.nextDouble() * getWidth();
         double y = generator.nextDouble() * getHeight();
         points[i] = new Point2D.Double(x, y);
      }
      repaint();
   }

   public void paintComponent(Graphics g)
   {
      if (points == null) return;
      Graphics2D g2 = (Graphics2D) g;
      for (int i = 0; i < points.length; i++)
      {
         double x = points[i].getX() - SIZE / 2;
         double y = points[i].getY() - SIZE / 2;
         g2.fill(new Rectangle2D.Double(x, y, SIZE, SIZE));
      }

      g2.draw(shapeMaker.makeShape(points));
   }

   private Point2D[] points;
   private static Random generator = new Random();
```

```
   private static int SIZE = 10;
   private int current;
   private ShapeMaker shapeMaker;
}

/**
 * A shape maker can make a shape from a point set. Concrete subclasses must return a shape
 *  in the makeShape method.
 */
```

```
abstract class ShapeMaker
{
   /**
    * Constructs a shape maker.
    * @param aPointCount the number of points needed to define this shape.
    */
   public ShapeMaker(int aPointCount)
```

```
   {
      pointCount = aPointCount;
   }

   /**
    * Gets the number of points needed to define this shape.
    * @return the point count
    */
   public int getPointCount()
   {
      return pointCount;
   }

   /**
    * Makes a shape out of the given point set.
    * @param p the points that define the shape
    * @return the shape defined by the points
    */
   public abstract Shape makeShape(Point2D[] p);

   public String toString()
   {
      return getClass().getName();
   }

   private int pointCount;
}

/**
 * Makes a line that joins two given points.
 */
class LineMaker extends ShapeMaker
{
   public LineMaker()
   {
      super(2);
   }

   public Shape makeShape(Point2D[] p)
   {
      return new Line2D.Double(p[0], p[1]);
   }
}
```

```
/**
 * Makes a rectangle that joins two given corner points.
 */
class RectangleMaker extends ShapeMaker
{
   public RectangleMaker()
   {
      super(2);
   }

   public Shape makeShape(Point2D[] p)
   {
      Rectangle2D s = new Rectangle2D.Double();
      s.setFrameFromDiagonal(p[0], p[1]);
      return s;
   }
}

/**
 * Makes a round rectangle that joins two given corner points.
 */
class RoundRectangleMaker extends ShapeMaker
{
   public RoundRectangleMaker()
   {
      super(2);
   }

   public Shape makeShape(Point2D[] p)
   {
      RoundRectangle2D s = new RoundRectangle2D.Double(0, 0, 0, 0, 20, 20);
      s.setFrameFromDiagonal(p[0], p[1]);
      return s;
   }
}

/**
 * Makes an ellipse contained in a bounding box with two given corner points.
 */
class EllipseMaker extends ShapeMaker
{
   public EllipseMaker()
   {
      super(2);
   }

   public Shape makeShape(Point2D[] p)
   {
      Ellipse2D s = new Ellipse2D.Double();
      s.setFrameFromDiagonal(p[0], p[1]);
      return s;
   }
}

/**
 * Makes an arc contained in a bounding box with two given corner points, and with starting
 * and ending angles given by lines emanating from the center of the bounding box and ending
 *  in two given points. To show the correctness of the angle computation, the returned shape
```

```
 * contains the arc, the bounding box, and the lines.
 */
class ArcMaker extends ShapeMaker
{
   public ArcMaker()
   {
      super(4);
   }

```

```
   public Shape makeShape(Point2D[] p)
   {
      double centerX = (p[0].getX() + p[1].getX()) / 2;
      double centerY = (p[0].getY() + p[1].getY()) / 2;
      double width = Math.abs(p[1].getX() - p[0].getX());
      double height = Math.abs(p[1].getY() - p[0].getY());

```

```
      double skewedStartAngle = Math.toDegrees(Math.atan2(-(p[2].getY() - centerY)
            * width, (p[2].getX() - centerX)
            * height));
      double skewedEndAngle = Math.toDegrees(Math.atan2(-(p[3].getY() - centerY)
            * width, (p[3].getX() - centerX)
            * height));
      double skewedAngleDifference = skewedEndAngle - skewedStartAngle;
      if (skewedStartAngle < 0) skewedStartAngle += 360;
      if (skewedAngleDifference < 0) skewedAngleDifference += 360;

      Arc2D s = new Arc2D.Double(0, 0, 0, 0, skewedStartAngle, skewedAngleDifference,
                           Arc2D.OPEN);
      s.setFrameFromDiagonal(p[0], p[1]);

      GeneralPath g = new GeneralPath();
      g.append(s, false);
      Rectangle2D r = new Rectangle2D.Double();
      r.setFrameFromDiagonal(p[0], p[1]);
      g.append(r, false);
      Point2D center = new Point2D.Double(centerX, centerY);
      g.append(new Line2D.Double(center, p[2]), false);
      g.append(new Line2D.Double(center, p[3]), false);
      return g;
   }
}

/**
 * Makes a polygon defined by six corner points.
 */
class PolygonMaker extends ShapeMaker
{
   public PolygonMaker()
   {
      super(6);
   }

   public Shape makeShape(Point2D[] p)
   {
      GeneralPath s = new GeneralPath();
      s.moveTo((float) p[0].getX(), (float) p[0].getY());
      for (int i = 1; i < p.length; i++)
         s.lineTo((float) p[i].getX(), (float) p[i].getY());
      s.closePath();
```

```
      return s;
   }
}

/**
 * Makes a quad curve defined by two end points and a control point.
 */
class QuadCurveMaker extends ShapeMaker
{
   public QuadCurveMaker()
   {
      super(3);
   }

   public Shape makeShape(Point2D[] p)
   {
      return new QuadCurve2D.Double(p[0].getX(), p[0].getY(), p[1].getX(), p[1].getY(), p[2]
            .getX(), p[2].getY());
   }
}

/**
 * Makes a cubic curve defined by two end points and two control points.
 */
class CubicCurveMaker extends ShapeMaker
{
   public CubicCurveMaker()
   {
      super(4);
   }

   public Shape makeShape(Point2D[] p)
   {
      return new CubicCurve2D.Double(p[0].getX(), p[0].getY(), p[1].getX(), p[1].getY(), p[2]
            .getX(), p[2].getY(), p[3].getX(), p[3].getY());
   }
}
```

API

`java.awt.geom.RoundRectangle2D.Double` 1.2

- `RoundRectangle2D.Double(double x, double y, double width, double height, double arcWidth, double arcHeight)`

  constructs a round rectangle with the given bounding rectangle and arc dimensions. See Figure 7-4 for an explanation of the `arcWidth` and `arcHeight` parameters.

java.awt.geom.Arc2D.Double 1.2

- `Arc2D.Double(double x, double y, double w, double h, double startAngle, double arcAngle, int type)`

  constructs an arc with the given bounding rectangle, start, and arc angle and arc type. The `startAngle` and `arcAngle` are explained on page 528. The type is one of `Arc2D.OPEN`, `Arc2D.PIE`, and `Arc2D.CHORD`.

java.awt.geom.QuadCurve2D.Double 1.2

- `QuadCurve2D.Double(double x1, double y1, double ctrlx, double ctrly, double x2, double y2)`

  constructs a quadratic curve from a start point, a control point, and an end point.

java.awt.geom.CubicCurve2D.Double 1.2

- `CubicCurve2D.Double(double x1, double y1, double ctrlx1, double ctrly1, double ctrlx2, double ctrly2, double x2, double y2)`

  constructs a cubic curve from a start point, two control points, and an end point.

java.awt.geom.GeneralPath 1.2

- `GeneralPath()`

  constructs an empty general path.

API

java.awt.geom.Path2D.Float 6

- `void moveTo(float x, float y)`

  makes (x, y) the *current point*, that is, the starting point of the next segment.

- `void lineTo(float x, float y)`

- `void quadTo(float ctrlx, float ctrly, float x, float y)`

- `void curveTo(float ctrl1x, float ctrl1y, float ctrl2x, float ctrl2y, float x, float y)`

  draws a line, quadratic curve, or cubic curve from the current point to the end point (x, y), and makes that end point the current point.

API

java.awt.geom.Path2D 6

- `void append(Shape s, boolean connect)`

  adds the outline of the given shape to the general path. If `connect` is `true`, the current point of the general path is connected to the starting point of the added shape by a straight line.

- `void closePath()`

  closes the path by drawing a straight line from the current point to the first point in the path.

Areas

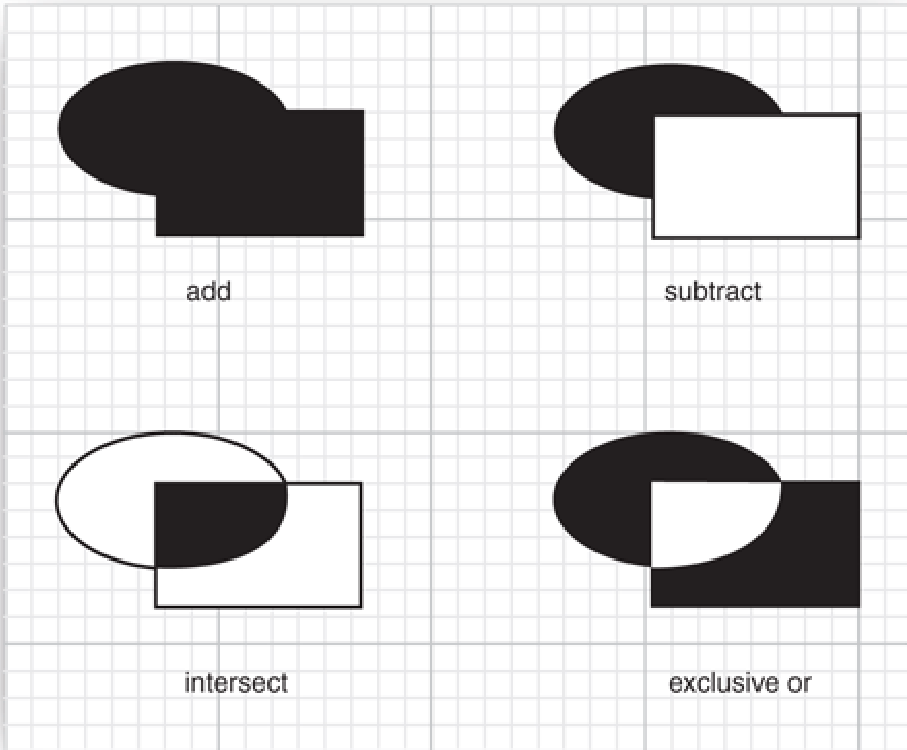
In the preceding section, you saw how you can specify complex shapes by constructing general paths that are composed of lines and curves. By using a sufficient number of lines and curves, you can draw essentially any shape. For example, the shapes of characters in the fonts that you see on the screen and on your printouts are all made up of lines and cubic curves.

Occasionally, it is easier to describe a shape by composing it from *areas,* such as rectangles, polygons, or ellipses. The Java 2D API supports four *constructive area geometry* operations that combine two areas into a new area:

- `add`— The combined area contains all points that are in the first or the second area.
- `subtract`— The combined area contains all points that are in the first but not the second area.
- `intersect`— The combined area contains all points that are in the first and the second area.
- `exclusiveOr`— The combined area contains all points that are in either the first or the second area, but not in both.

Figure 7-10 shows these operations.

Figure 7-10. Constructive area geometry operations



To construct a complex area, you start with a default area object.

```
Area a = new Area();
```

Then, you combine the area with any shape.

```
a.add(new Rectangle2D.Double(. . .));
a.subtract(path);
. . .
```

The `Area` class implements the `Shape` interface. You can stroke the boundary of the area with the `draw` method or paint the interior with the `fill` method of the `Graphics2D` class.

java.awt.geom.Area

- `void add(Area other)`

- `void subtract(Area other)`
- `void intersect(Area other)`

- `void exclusiveOr(Area other)`

  carries out the constructive area geometry operation with this area and the other area and sets this area to the result.

Strokes

The `draw` operation of the `Graphics2D` class draws the boundary of a shape by using the currently selected *stroke.* By default, the stroke is a solid line that is 1 pixel wide. You can select a different stroke by calling the `setStroke` method. You supply an object of a class that implements the `Stroke` interface. The Java 2D API defines only one such class, called `BasicStroke`. In this section, we look at the capabilities of the `BasicStroke` class.

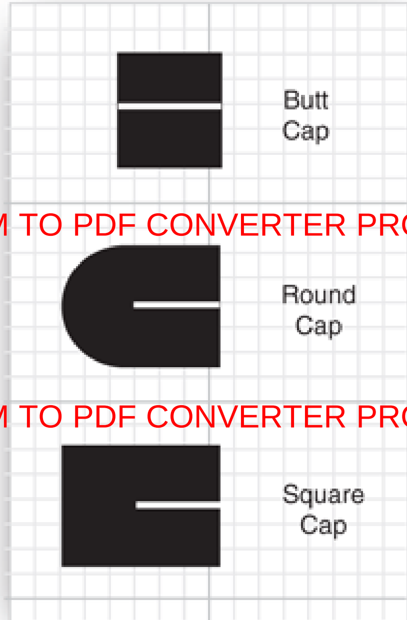
You can construct strokes of arbitrary thickness. For example, here is how you draw lines that are 10 pixels wide.

```
g2.setStroke(new BasicStroke(10.0F));
g2.draw(new Line2D.Double(. . .));
```

When a stroke is more than a pixel thick, then the *end* of the stroke can have different styles. Figure 7-11 shows these so-called *end cap styles.* You have three choices:

- A *butt cap* simply ends the stroke at its end point.
- A *round cap* adds a half-circle to the end of the stroke.
- A *square cap* adds a half-square to the end of the stroke.

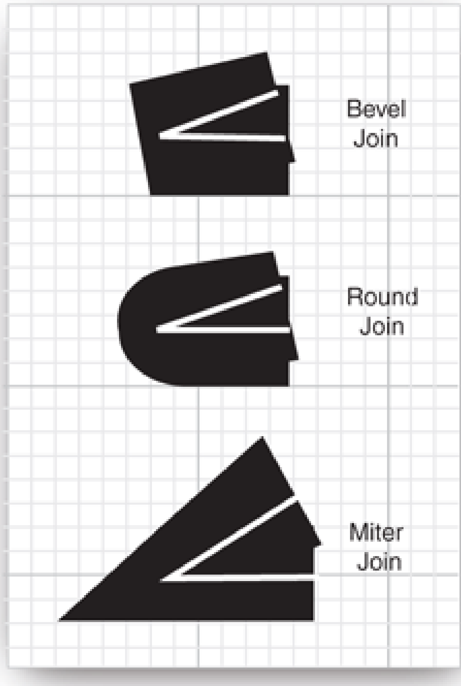
Figure 7-11. End cap styles



When two thick strokes meet, there are three choices for the *join style* (see Figure 7-12).

- A *bevel join* joins the strokes with a straight line that is perpendicular to the bisector of the angle between the two strokes.
- A *round join* extends each stroke to have a round cap.
- A *miter join* extends both strokes by adding a "spike."

Figure 7-12. Join styles



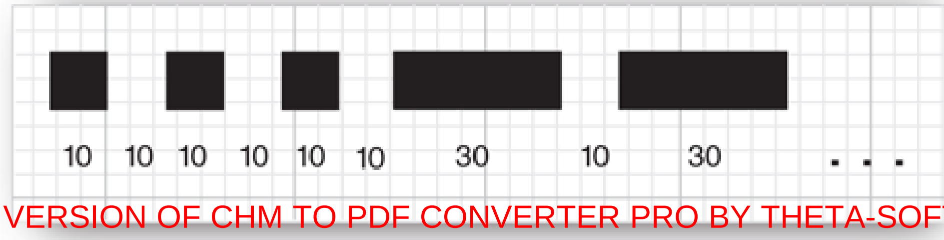
The miter join is not suitable for lines that meet at small angles. If two lines join with an angle that is less than the *miter limit*, then a bevel join is used instead. That usage prevents extremely long spikes. By default, the miter limit is 10 degrees.

You specify these choices in the BasicStroke constructor, for example:

Code View:
```
g2.setStroke(new BasicStroke(10.0F, BasicStroke.CAP_ROUND, BasicStroke.JOIN_ROUND));
g2.setStroke(new BasicStroke(10.0F, BasicStroke.CAP_BUTT, BasicStroke.JOIN_MITER,
   15.0F /* miter limit */));
```

Finally, you can specify dashed lines by setting a *dash pattern.* In the program in Listing 7-2, you can select a dash pattern that spells out SOS in Morse code. The dash pattern is a `float[]` array of numbers that contains the lengths of the "on" and "off" strokes (see Figure 7-13).

Figure 7-13. A dash pattern



You specify the dash pattern and a *dash phase* when constructing the `BasicStroke`. The dash phase indicates where in the dash pattern each line should start. Normally, you set this value to 0.

Code View:

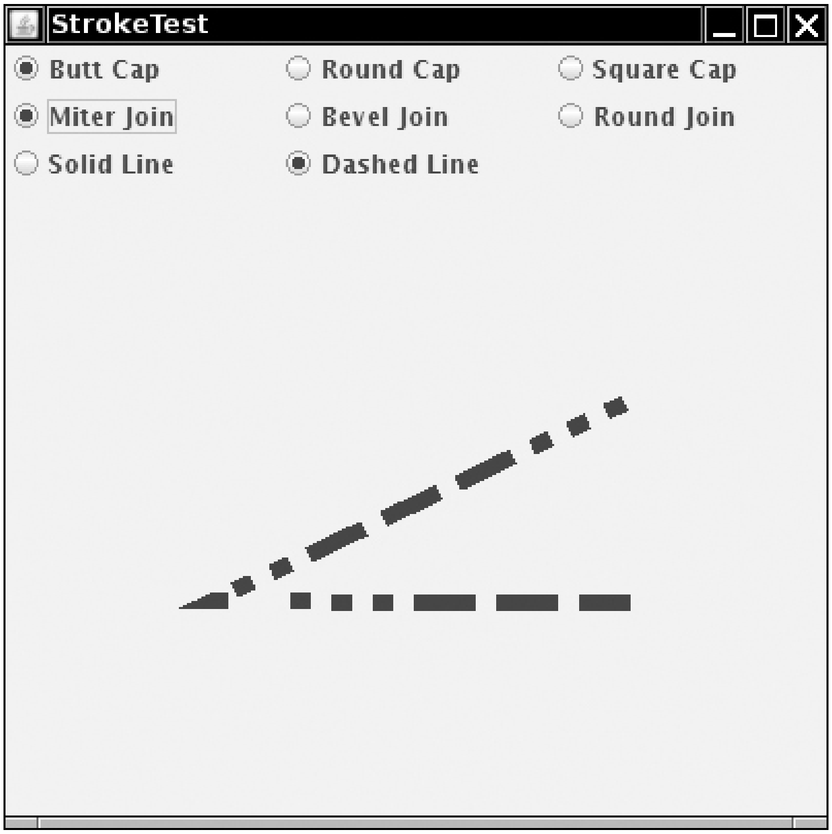
```
float[] dashPattern = { 10, 10, 10, 10, 10, 10, 30, 10, 30, ... };
g2.setStroke(new BasicStroke(10.0F, BasicStroke.CAP_BUTT, BasicStroke.JOIN_MITER,
   10.0F /* miter limit */, dashPattern, 0 /* dash phase */));
```

Note

End cap styles are applied to the ends of *each dash* in a dash pattern.

The program in Listing 7-2 lets you specify end cap styles, join styles, and dashed lines (see Figure 7-14). You can move the ends of the line segments to test the miter limit: Select the miter join, then move the line segment to form a very acute angle. You will see the miter join turn into a bevel join.

Figure 7-14. The StrokeTest program



The program is similar to the program in Listing 7-1. The mouse listener remembers if you click on the end point of a line segment, and the mouse motion listener monitors the dragging of the end point. A set of radio buttons signal the user choices for the end cap style, join style, and solid or dashed line. The `paintComponent` method of the `StrokePanel` class constructs a `GeneralPath` consisting of the two line segments that join the three points that the user can move with the mouse. It then constructs a `BasicStroke`, according to the selections that the user made, and finally draws the path.

Listing 7-2. **`StrokeTest.java`**

Code View:

```
import java.awt.*;
import java.awt.event.*;
import java.awt.geom.*;
import javax.swing.*;

/**
 * This program demonstrates different stroke types.
 * @version 1.03 2007-08-16
 * @author Cay Horstmann
 */
public class StrokeTest
{
   public static void main(String[] args)
   {
      EventQueue.invokeLater(new Runnable()
         {
```

```
            public void run()
            {
               JFrame frame = new StrokeTestFrame();
               frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
               frame.setVisible(true);
            }
         });
   }
}
```



```
/**
 * This frame lets the user choose the cap, join, and line style, and shows the resulting
 * stroke.
 */
class StrokeTestFrame extends JFrame
{
```



```
   public StrokeTestFrame()
   {
      setTitle("StrokeTest");
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

      canvas = new StrokeComponent();
      add(canvas, BorderLayout.CENTER);

      buttonPanel = new JPanel();
      buttonPanel.setLayout(new GridLayout(3, 3));
      add(buttonPanel, BorderLayout.NORTH);

      ButtonGroup group1 = new ButtonGroup();
      makeCapButton("Butt Cap", BasicStroke.CAP_BUTT, group1);
      makeCapButton("Round Cap", BasicStroke.CAP_ROUND, group1);
      makeCapButton("Square Cap", BasicStroke.CAP_SQUARE, group1);

      ButtonGroup group2 = new ButtonGroup();
      makeJoinButton("Miter Join", BasicStroke.JOIN_MITER, group2);
      makeJoinButton("Bevel Join", BasicStroke.JOIN_BEVEL, group2);
      makeJoinButton("Round Join", BasicStroke.JOIN_ROUND, group2);

      ButtonGroup group3 = new ButtonGroup();
      makeDashButton("Solid Line", false, group3);
      makeDashButton("Dashed Line", true, group3);
   }

   /**
    * Makes a radio button to change the cap style.
    * @param label the button label
    * @param style the cap style
    * @param group the radio button group
    */
   private void makeCapButton(String label, final int style, ButtonGroup group)
   {
      // select first button in group
      boolean selected = group.getButtonCount() == 0;
      JRadioButton button = new JRadioButton(label, selected);
      buttonPanel.add(button);
      group.add(button);
      button.addActionListener(new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
```

```
            {
               canvas.setCap(style);
            }
         });
   }

   /**
    * Makes a radio button to change the join style.
    * @param label the button label
    * @param style the join style
    * @param group the radio button group
    */
   private void makeJoinButton(String label, final int style, ButtonGroup group)
   {
      // select first button in group
      boolean selected = group.getButtonCount() == 0;
      JRadioButton button = new JRadioButton(label, selected);
      buttonPanel.add(button);
      group.add(button);
      button.addActionListener(new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               canvas.setJoin(style);
            }
         });
   }

   /**
    * Makes a radio button to set solid or dashed lines
    * @param label the button label
    * @param style false for solid, true for dashed lines
    * @param group the radio button group
    */
   private void makeDashButton(String label, final boolean style, ButtonGroup group)
   {
      // select first button in group
      boolean selected = group.getButtonCount() == 0;
      JRadioButton button = new JRadioButton(label, selected);
      buttonPanel.add(button);
      group.add(button);
      button.addActionListener(new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               canvas.setDash(style);
            }
         });
   }

   private StrokeComponent canvas;
   private JPanel buttonPanel;

   private static final int DEFAULT_WIDTH = 400;
   private static final int DEFAULT_HEIGHT = 400;
}

/**
 * This component draws two joined lines, using different stroke objects, and allows the
```

```
 * user to drag the three points defining the lines.
 */
class StrokeComponent extends JComponent
{
   public StrokeComponent()
   {
      addMouseListener(new MouseAdapter()
         {
            public void mousePressed(MouseEvent event)
            {
```


```
               Point p = event.getPoint();
               for (int i = 0; i < points.length; i++)
               {
                  double x = points[i].getX() - SIZE / 2;
                  double y = points[i].getY() - SIZE / 2;
                  Rectangle2D r = new Rectangle2D.Double(x, y, SIZE, SIZE);
```


```
                  if (r.contains(p))
                  {
                     current = i;
                     return;
                  }
               }
            }

            public void mouseReleased(MouseEvent event)
            {
               current = -1;
            }
         });

      addMouseMotionListener(new MouseMotionAdapter()
         {
            public void mouseDragged(MouseEvent event)
            {
               if (current == -1) return;
               points[current] = event.getPoint();
               repaint();
            }
         });

      points = new Point2D[3];
      points[0] = new Point2D.Double(200, 100);
      points[1] = new Point2D.Double(100, 200);
      points[2] = new Point2D.Double(200, 200);
      current = -1;
      width = 8.0F;
   }

   public void paintComponent(Graphics g)
   {
      Graphics2D g2 = (Graphics2D) g;
      GeneralPath path = new GeneralPath();
      path.moveTo((float) points[0].getX(), (float) points[0].getY());
      for (int i = 1; i < points.length; i++)
         path.lineTo((float) points[i].getX(), (float) points[i].getY());
      BasicStroke stroke;
      if (dash)
      {
         float miterLimit = 10.0F;
```

```
         float[] dashPattern = { 10F, 10F, 10F, 10F, 10F, 10F, 30F, 10F, 30F, 10F, 30F, 10F,
               10F, 10F, 10F, 10F, 10F, 30F };
         float dashPhase = 0;
         stroke = new BasicStroke(width, cap, join, miterLimit, dashPattern, dashPhase);
      }
      else stroke = new BasicStroke(width, cap, join);
      g2.setStroke(stroke);
      g2.draw(path);
   }

   /**
    * Sets the join style.
    * @param j the join style
    */
   public void setJoin(int j)
   {
      join = j;
      repaint();
   }

   /**
    * Sets the cap style.
    * @param c the cap style
    */
   public void setCap(int c)
   {
      cap = c;
      repaint();
   }

   /**
    * Sets solid or dashed lines
    * @param d false for solid, true for dashed lines
    */
   public void setDash(boolean d)
   {
      dash = d;
      repaint();
   }

   private Point2D[] points;
   private static int SIZE = 10;
   private int current;
   private float width;
   private int cap;
   private int join;
   private boolean dash;
}
```

API java.awt.Graphics2D 1.2

- `void setStroke(Stroke s)`

  sets the stroke of this graphics context to the given object that implements the `Stroke` interface.

API java.awt.BasicStroke 1.2

- `BasicStroke(float width)`
- `BasicStroke(float width, int cap, int join)`
- `BasicStroke(float width, int cap, int join, float miterlimit)`
- `BasicStroke(float width, int cap, int join, float miterlimit, float[] dash, float dashPhase)`

  constructs a stroke object with the given attributes.

Parameters:	`width`	The width of the pen
	`cap`	The end cap style, one of `CAP_BUTT`, `CAP_ROUND`, and `CAP_SQUARE`
	`join`	The join style, one of `JOIN_BEVEL`, `JOIN_MITER`, and `JOIN_ROUND`
	`miterlimit`	The angle, in degrees, below which a miter join is rendered as a bevel join
	`dash`	An array of the lengths of the alternating filled and blank portions of a dashed stroke
	`dashPhase`	The "phase" of the dash pattern; a segment of this length, preceding the starting point of the stroke, is assumed to have the dash pattern already applied

Paint

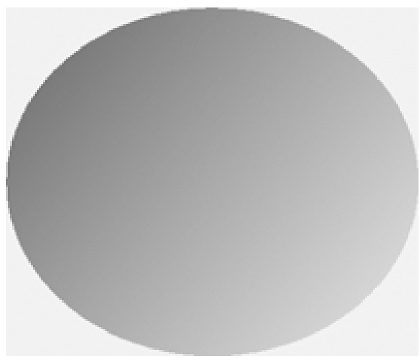
When you fill a shape, its inside is covered with *paint.* You use the setPaint method to set the paint style to an object with a class that implements the Paint interface. The Java 2D API provides three such classes:

- The Color class implements the Paint interface. To fill shapes with a solid color, simply call setPaint with a Color object, such as

```
g2.setPaint(Color.red);
```

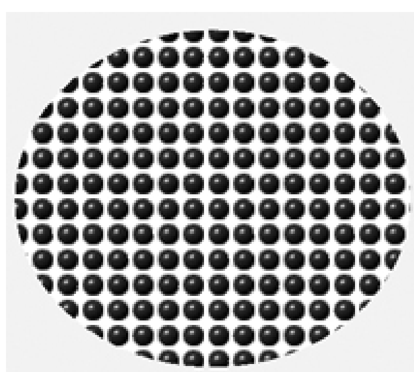
- The GradientPaint class varies colors by interpolating between two given color values (see Figure 7-15).

Figure 7-15. Gradient paint



- The TexturePaint class fills an area with repetitions of an image (see Figure 7-16).

Figure 7-16. Texture paint



You construct a GradientPaint object by specifying two points and the colors that you want at these two points.

```
g2.setPaint(new GradientPaint(p1, Color.RED, p2, Color.YELLOW));
```

Colors are interpolated along the line joining the two points. Colors are constant along lines that are perpendicular to that joining line. Points beyond an end point of the line are given the color at the end point.

Alternatively, if you call the `GradientPaint` constructor with `true` for the `cyclic` parameter,


```
g2.setPaint(new GradientPaint(p1, Color.RED, p2, Color.YELLOW, true));
```

then the color variation *cycles* and keeps varying beyond the end points.

To construct a `TexturePaint` object, you specify a `BufferedImage` and an *anchor* rectangle.


```
g2.setPaint(new TexturePaint(bufferedImage, anchorRectangle));
```

We introduce the `BufferedImage` class later in this chapter when we discuss images in detail. The simplest way of obtaining a buffered image is to read an image file:

```
bufferedImage = ImageIO.read(new File("blue-ball.gif"));
```

The anchor rectangle is extended indefinitely in *x*- and *y*-directions to tile the entire coordinate plane. The image is scaled to fit into the anchor and then replicated into each tile.

API `java.awt.Graphics2D` 1.2

- `void setPaint(Paint s)`

  sets the paint of this graphics context to the given object that implements the `Paint` interface.

`java.awt.GradientPaint` 1.2

- `GradientPaint(float x1, float y1, Color color1, float x2, float y2, Color color2)`
- `GradientPaint(float x1, float y1, Color color1, float x2, float y2, Color color2, boolean cyclic)`
- `GradientPaint(Point2D p1, Color color1, Point2D p2, Color color2)`
- `GradientPaint(Point2D p1, Color color1, Point2D p2, Color color2, boolean cyclic)`

  constructs a gradient paint object that fills shapes with color such that the start point is colored with `color1`, the end point is colored with `color2`, and the colors in between are linearly interpolated. Colors are constant along lines that are perpendicular to the line joining the start and the end point. By default, the gradient paint is not cyclic; that is, points beyond the start and end points are colored with the same color as the start and end point. If the gradient paint is *cyclic,* then colors continue to be interpolated, first returning to the starting point color and then repeating indefinitely in both directions.

`java.awt.TexturePaint` 1.2

- `TexturePaint(BufferedImage texture, Rectangle2D anchor)`

  creates a texture paint object. The anchor rectangle defines the tiling of the space to be painted; it is repeated indefinitely in *x*- and *y*-directions, and the texture image is scaled to fill each tile.

Coordinate Transformations

Suppose you need to draw an object such as an automobile. You know, from the manufacturer's specifications, the height, wheelbase, and total length. You could, of course, figure out all pixel positions, assuming some number of pixels per meter. However, there is an easier way: You can ask the graphics context to carry out the conversion for you.

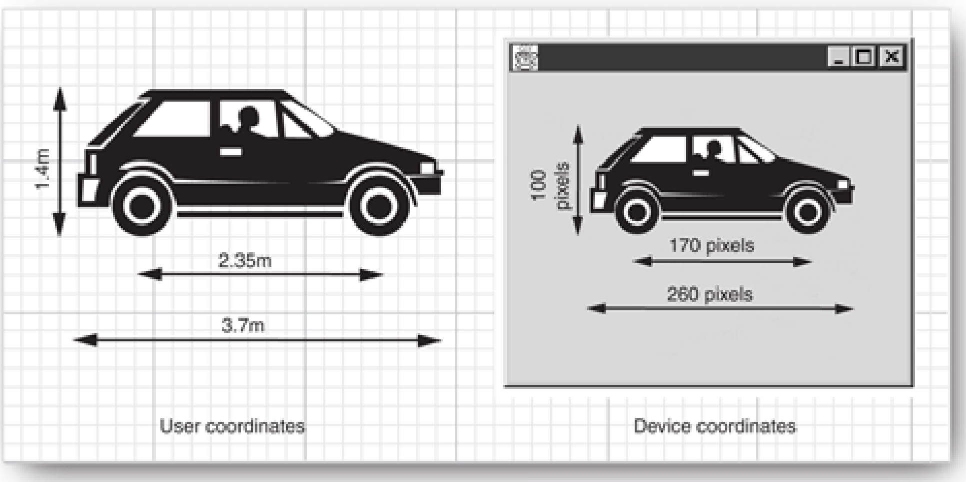
Code View:
```
g2.scale(pixelsPerMeter, pixelsPerMeter);
g2.draw(new Line2D.Double(coordinates in meters)); // converts to pixels and draws scaled line
```

UNREGISTERED VERSION OF CHM TO PDF CONVERTER PRO BY THETA-SOFTWARE

The `scale` method of the `Graphics2D` class sets the *coordinate transformation* of the graphics context to a scaling transformation. That transformation changes *user coordinates* (user-specified units) to *device coordinates* (pixels). Figure 7-17 shows how the transformation works.

Figure 7-17. User and device coordinates

[View full size image]



Coordinate transformations are very useful in practice. They allow you to work with convenient coordinate values. The graphics context takes care of the dirty work of transforming them to pixels.

There are four fundamental transformations.

- Scaling: blowing up, or shrinking, all distances from a fixed point.
- Rotation: rotating all points around a fixed center.

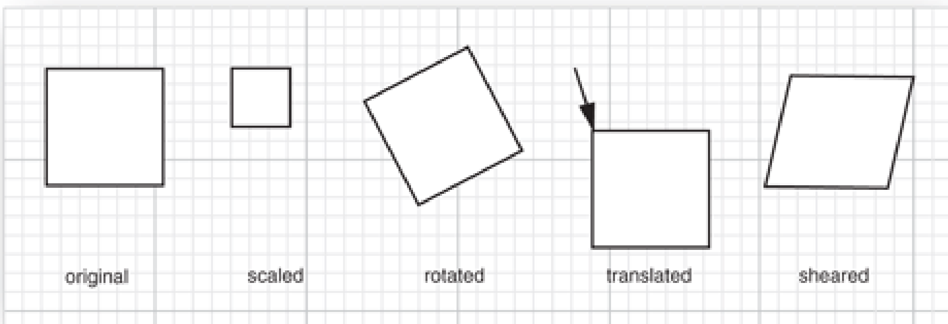
- Translation: moving all points by a fixed amount.

- Shear: leaving one line fixed and "sliding" the lines parallel to it by an amount that is proportional to the distance from the fixed line.

Figure 7-18 shows how these four fundamental transformations act on a unit square.

Figure 7-18. The fundamental transformations

[View full size image]



The `scale`, `rotate`, `translate`, and `shear` methods of the `Graphics2D` class set the coordinate transformation of the graphics context to one of these fundamental transformations.

You can *compose* the transformations. For example, you might want to rotate shapes *and* double their size. Then, you supply both a rotation and a scaling transformation.

```
g2.rotate(angle);
g2.scale(2, 2);
g2.draw(. . .);
```

In this case, it does not matter in which order you supply the transformations. However, with most transformations, order does matter. For example, if you want to rotate and shear, then it makes a difference which of the transformations you supply first. You need to figure out what your intention is. The graphics context will apply the transformations in the opposite order in which you supplied them. That is, the last transformation that you supply is applied first.

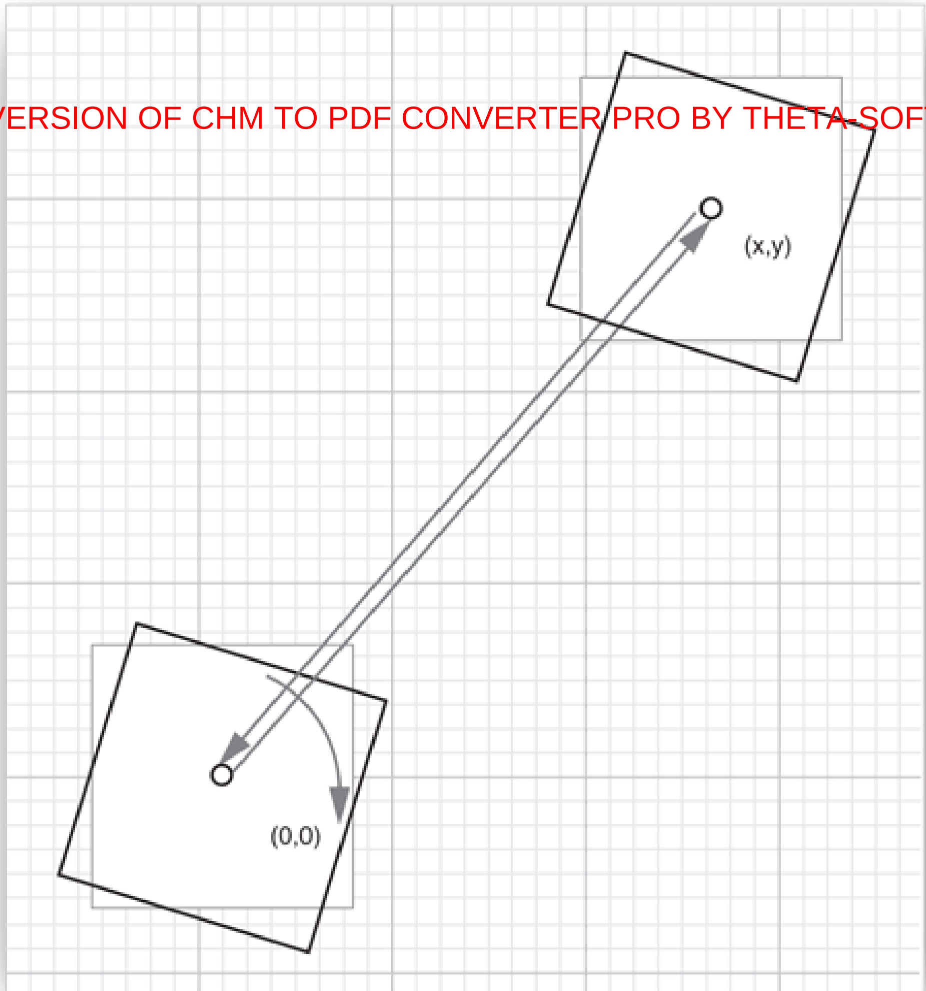
You can supply as many transformations as you like. For example, consider the following sequence of transformations:

```
g2.translate(x, y);
g2.rotate(a);
g2.translate(-x, -y);
```

The last transformation (which is applied first) moves the point (`x`, `y`) to the origin. The second transformation rotates with an angle a around the origin. The final transformation moves the origin back to (x, y). The overall effect is a rotation with center point (x, y)—see Figure 7-19. Because rotating about a point other than the origin is such a common operation, there is a shortcut:

```
g2.rotate(a, x, y);
```


Figure 7-19. Composing transformations



If you know some matrix theory, you are probably aware that all rotations, translations, scalings, shears, and their compositions can be expressed by matrix transformations of the form:

$$\begin{bmatrix} x_{\text{new}} \\ y_{\text{new}} \\ 1 \end{bmatrix} = \begin{bmatrix} a & c & e \\ b & d & f \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Such a transformation is called an *affine transformation.* In the Java 2D API, the `AffineTransform` class describes such a transformation. If you know the components of a particular transformation matrix, you can construct it directly as

```
AffineTransform t = new AffineTransform(a, b, c, d, e, f);
```

Additionally, the factory methods `getRotateInstance`, `getScaleInstance`, `getTranslateInstance`, and `getShearInstance` construct the matrices that represent these transformation types. For example, the call

```
t = AffineTransform.getScaleInstance(2.0F, 0.5F);
```

returns a transformation that corresponds to the matrix

$$\begin{bmatrix} 2 & 0 & 0 \\ 0 & 0.5 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Finally, the instance methods `setToRotation`, `setToScale`, `setToTranslation`, and `setToShear` set a transformation object to a new type. Here is an example:

```
t.setToRotation(angle); // sets t to a rotation
```

You can set the coordinate transformation of the graphics context to an `AffineTransform` object.

```
g2.setTransform(t); // replaces current transformation
```

However, in practice, you shouldn't call the `setTransform` operation, as it replaces any existing transformation that the graphics context may have. For example, a graphics context for printing in landscape mode already contains a 90-degree rotation transformation. If you call `setTransform`, you obliterate that rotation. Instead, call the `transform` method.

```
g2.transform(t); // composes current transformation with t
```

It composes the existing transformation with the new `AffineTransform` object.

If you just want to apply a transformation temporarily, then you first get the old transformation, compose with your new transformation, and finally restore the old transformation when you are done.

```
AffineTransform oldTransform = g2.getTransform(); // save old transform
g2.transform(t); // apply temporary transform // now draw on g2
g2.setTransform(oldTransform); // restore old transform
```

API java.awt.geom.AffineTransform 1.2

- `AffineTransform(double a, double b, double c, double d, double e, double f)`

- `AffineTransform(float a, float b, float c, float d, float e, float f)`

  constructs the affine transform with matrix

$$\begin{bmatrix} a & c & e \\ b & d & f \\ 0 & 0 & 1 \end{bmatrix}$$

- `AffineTransform(double[] m)`

- `AffineTransform(float[] m)`

  constructs the affine transform with matrix

$$\begin{bmatrix} \texttt{m[0]} & \texttt{m[2]} & \texttt{m[4]} \\ \texttt{m[1]} & \texttt{m[3]} & \texttt{m[5]} \\ 0 & 0 & 1 \end{bmatrix}$$

- `static AffineTransform getRotateInstance(double a)`

  creates a rotation around the origin by the angle `a` (in radians). The transformation matrix is

$$\begin{bmatrix} \cos(\mathrm{a}) & -\sin(\mathrm{a}) & 0 \\ \sin(\mathrm{a}) & \cos(\mathrm{a}) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

  If `a` is between 0 and π / 2, the rotation moves the positive *x*-axis toward the positive *y*-axis.

- `static AffineTransform getRotateInstance(double a, double x, double y)`

  creates a rotation around the point `(x,y)` by the angle `a` (in radians).

- `static AffineTransform getScaleInstance(double sx, double sy)`

  creates a scaling transformation that scales the *x*-axis by `sx` and the *y*-axis by `sy`. The transformation matrix is

$$\begin{bmatrix} sx & 0 & 0 \\ 0 & sy & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

- `static AffineTransform getShearInstance(double shx, double shy)`

  creates a shear transformation that shears the *x*-axis by `shx` and the *y*-axis by `shy`. The transformation matrix is

$$\begin{bmatrix} 1 & shx & 0 \\ shy & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

- `static AffineTransform getTranslateInstance(double tx, double ty)`

  creates a translation that moves the *x*-axis by `tx` and the *y*-axis by `ty`. The transformation matrix is

$$\begin{bmatrix} 1 & 0 & tx \\ 0 & 1 & ty \\ 0 & 0 & 1 \end{bmatrix}$$

- `void setToRotation(double a)`
- `void setToRotation(double a, double x, double y)`
- `void setToScale(double sx, double sy)`
- `void setToShear(double sx, double sy)`
- `void setToTranslation(double tx, double ty)`

  sets this affine transformation to a basic transformation with the given parameters. See the `getXxxInstance` method for an explanation of the basic transformations and their parameters.

API `java.awt.Graphics2D` 1.2

- `void setTransform(AffineTransform t)`

  replaces the existing coordinate transformation of this graphics context with `t`.

- `void transform(AffineTransform t)`

  composes the existing coordinate transformation of this graphics context with `t`.

- `void rotate(double a)`
- `void rotate(double a, double x, double y)`
- `void scale(double sx, double sy)`
- `void shear(double sx, double sy)`
- `void translate(double tx, double ty)`

  composes the existing coordinate transformation of this graphics context with a basic transformation with the given parameters. See the `AffineTransform.getXxxInstance` method for an explanation of the basic transformations and their parameters.

Clipping

By setting a *clipping shape* in the graphics context, you constrain all drawing operations to the interior of that clipping shape.

```
g2.setClip(clipShape); // but see below
g2.draw(shape); // draws only the part that falls inside the clipping shape
```

However, in practice, you don't want to call the `setClip` operation, because it replaces any existing clipping shape that the graphics context might have. For example, as you will see later in this chapter, a graphics context for printing comes with a clip rectangle that ensures that you don't draw on the margins. Instead, call the `clip` method.

```
g2.clip(clipShape); // better
```

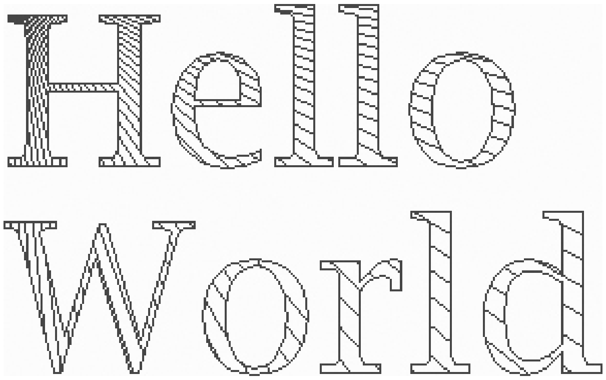
The `clip` method intersects the existing clipping shape with the new one that you supply.

If you just want to apply a clipping area temporarily, then you should first get the old clip, then add your new clip, and finally restore the old clip when you are done:

```
Shape oldClip = g2.getClip(); // save old clip
g2.clip(clipShape); // apply temporary clip
draw on g2
g2.setClip(oldClip); // restore old clip
```

In Figure 7-20, we show off the clipping capability with a rather dramatic drawing of a line pattern that is clipped by a complex shape, namely, the outline of a set of letters.

Figure 7-20. Using letter shapes to clip a line pattern



To obtain character outlines, you need a *font render context.* Use the `getFontRenderContext` method of the `Graphics2D` class.

```
FontRenderContext context = g2.getFontRenderContext();
```

Next, using a string, a font, and the font render context, create a TextLayout object:

```
TextLayout layout = new TextLayout("Hello", font, context);
```

This text layout object describes the layout of a sequence of characters, as rendered by a particular font render context. The layout depends on the font render context—the same characters will look different on a screen or a printer.

More important for our application, the getOutline method returns a Shape object that describes the shape of the outline of the characters in the text layout. The outline shape starts at the origin (0, 0), which might not be what you want. In that case, supply an affine transform to the getOutline operation that specifies where you would like the outline to appear.


```
AffineTransform transform = AffineTransform.getTranslateInstance(0, 100);
Shape outline = layout.getOutline(transform);
```

Then, append the outline to the clipping shape.

```
GeneralPath clipShape = new GeneralPath();
clipShape.append(outline, false);
```

Finally, set the clipping shape and draw a set of lines. The lines appear only inside the character boundaries.

```
g2.setClip(clipShape);
Point2D p = new Point2D.Double(0, 0);
for (int i = 0; i < NLINES; i++)
{
   double x = . . .;
   double y = . . .;
   Point2D q = new Point2D.Double(x, y);
   g2.draw(new Line2D.Double(p, q)); // lines are clipped
}
```

You can see the complete code in Listing 7-8 on page 607.

API

java.awt.Graphics 1.0

- void setClip(Shape s) **1.2**

  sets the current clipping shape to the shape s.

- Shape getClip() **1.2**

  returns the current clipping shape.

java.awt.Graphics2D 1.2

- `void clip(Shape s)`

  intersects the current clipping shape with the shape `s`.

- `FontRenderContext getFontRenderContext()`

  returns a font render context that is necessary for constructing `TextLayout` objects.

java.awt.font.TextLayout 1.2

- `TextLayout(String s, Font f, FontRenderContext context)`

  constructs a text layout object from a given string and font, using the font render context to obtain font properties for a particular device.

- `float getAdvance()`

  returns the width of this text layout.

- `float getAscent()`

- `float getDescent()`

  returns the height of this text layout above and below the baseline.

- `float getLeading()`

  returns the distance between successive lines in the font used by this text layout.

Transparency and Composition

In the standard RGB color model, every color is described by its red, green, and blue components. However, it is also convenient to describe areas of an image that are *transparent* or partially transparent. When you superimpose an image onto an existing drawing, the transparent pixels do not obscure the pixels under them at all, whereas partially transparent pixels are mixed with the pixels under them. Figure 7-21 shows the effect of overlaying a partially transparent rectangle on an image. You can still see the details of the image shine through from under the rectangle.

Figure 7-21. Overlaying a partially transparent rectangle on an image

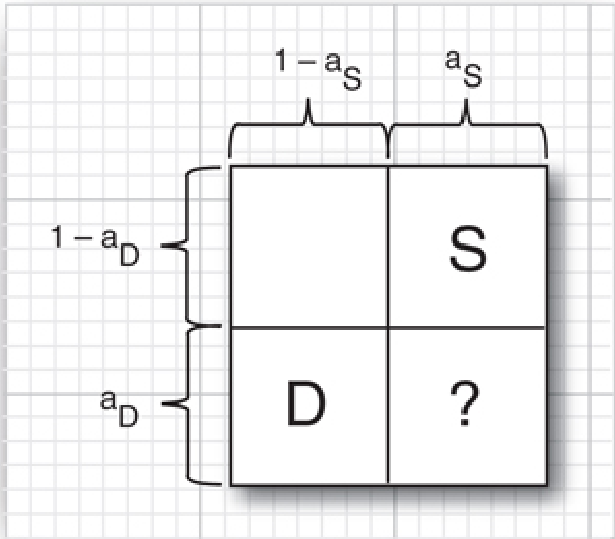
In the Java 2D API, transparency is described by an *alpha channel.* Each pixel has, in addition to its red, green, and blue color components, an alpha value between 0 (fully transparent) and 1 (fully opaque). For example, the rectangle in Figure 7-21 was filled with a pale yellow color with 50% transparency:

```
new Color(0.7F, 0.7F, 0.0F, 0.5F);
```

Now let us look at what happens if you superimpose two shapes. You need to blend or *compose* the colors and alpha values of the source and destination pixels. Porter and Duff, two researchers in the field of computer graphics, have formulated 12 possible *composition rules* for this blending process. The Java 2D API implements all of these rules. Before we go any further, we want to point out that only two of these rules have practical significance. If you find the rules arcane or confusing, just use the `SRC_OVER` rule. It is the default rule for a `Graphics2D` object, and it gives the most intuitive results.

Here is the theory behind the rules. Suppose you have a *source pixel* with alpha value a_S. In the image, there is already a *destination pixel* with alpha value a_D. You want to compose the two. The diagram in Figure 7-22 shows how to design a composition rule.

Figure 7-22. Designing a composition rule



Porter and Duff consider the alpha value as the probability that the pixel color should be used. From the perspective of the source, there is a probability a_S that it wants to use the source color and a probability of 1 - a_S that it doesn't care. The same holds for the destination. When composing the colors, let us assume that the probabilities are independent. Then there are four cases, as shown in Figure 7-22. If the source wants to use the source color and the destination doesn't care, then it seems reasonable to let the source have its way. That's why the upper-right corner of the diagram is labeled "S." The probability for that event is $a_S \cdot (1 - a_D)$. Similarly, the lower-left corner is labeled "D." What should one do if both destination and source would like to select their color? That's where the Porter–Duff rules come in. If we decide that the source is more important, then we label the lower-right corner with an "S" as well. That rule is called `SRC_OVER`. In that rule, you combine the source colors with a weight of a_S and the destination colors with a weight of $(1 - a_S) \cdot a_D$.

The visual effect is a blending of the source and destination, with preference given to the source. In particular, if a_S is 1, then the destination color is not taken into account at all. If a_S is 0, then the source pixel is completely transparent and the destination color is unchanged.

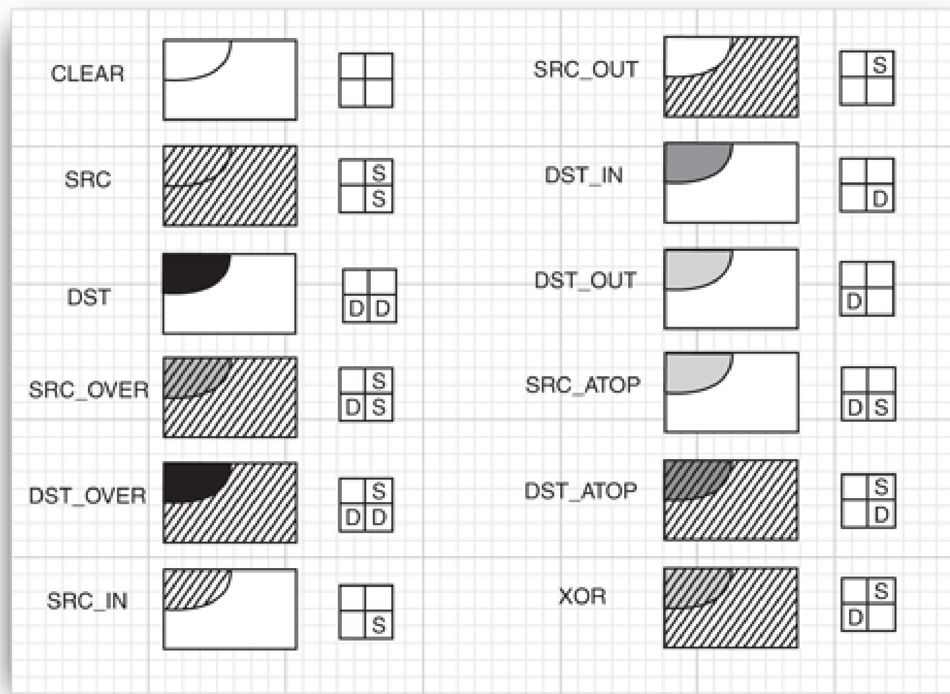
The other rules depend on what letters you put in the boxes of the probability diagram. Table 7-1 and Figure 7-23 show all rules that are supported by the Java 2D API. The images in the figure show the results of the rules when a rectangular source region with an alpha of 0.75 is combined with an elliptical destination region with an alpha of 1.0.

Table 7-1. The Porter–Duff Composition Rules

Rule	Explanation
`CLEAR`	Source clears destination.
`SRC`	Source overwrites destination and empty pixels.
`DST`	Source does not affect destination.
`SRC_OVER`	Source blends with destination and overwrites empty pixels.
`DST_OVER`	Source does not affect destination and overwrites empty pixels.

Rule	Explanation
SRC_IN	Source overwrites destination.
SRC_OUT	Source clears destination and overwrites empty pixels.
DST_IN	Source alpha modifies destination.
DST_OUT	Source alpha complement modifies destination.
SRC_ATOP	Source blends with destination.
DST_ATOP	Source alpha modifies destination. Source overwrites empty pixels.
XOR	Source alpha complement modifies destination. Source overwrites empty pixels.

Figure 7-23. Porter–Duff composition rules



As you can see, most of the rules aren't very useful. Consider, as an extreme case, the DST_IN rule. It doesn't take the source color into account at all, but it uses the alpha of the source to affect the destination. The SRC rule is potentially useful—it forces the source color to be used, turning off blending with the destination.

For more information on the Porter–Duff rules, see, for example, *Computer Graphics: Principles and Practice, Second Edition in C* by James D. Foley, Andries van Dam, Steven K. Feiner, et al.

You use the setComposite method of the Graphics2D class to install an object of a class that implements the Composite interface. The Java 2D API supplies one such class, AlphaComposite, that implements all the

Porter–Duff rules in Figure 7-23.

The factory method getInstance of the AlphaComposite class yields an AlphaComposite object. You supply the rule and the alpha value to be used for source pixels. For example, consider the following code:

```
int rule = AlphaComposite.SRC_OVER;
float alpha = 0.5f;
g2.setComposite(AlphaComposite.getInstance(rule, alpha));
g2.setPaint(Color.blue);
g2.fill(rectangle);
```

The rectangle is then painted with blue color and an alpha value of 0.5. Because the composition rule is SRC_OVER, it is transparently overlaid on the existing image.

The program in Listing 7-3 lets you explore these composition rules. Pick a rule from the combo box and use the slider to set the alpha value of the AlphaComposite object.

Furthermore, the program displays a verbal description of each rule. Note that the descriptions are computed from the composition rule diagrams. For example, a "DS" in the second row stands for "blends with destination."

The program has one important twist. There is no guarantee that the graphics context that corresponds to the screen has an alpha channel. (In fact, it generally does not.) When pixels are deposited to a destination without an alpha channel, then the pixel colors are multiplied with the alpha value and the alpha value is discarded. Because several of the Porter–Duff rules use the alpha values of the destination, a destination alpha channel is important. For that reason, we use a buffered image with the ARGB color model to compose the shapes. After the images have been composed, we draw the resulting image to the screen.

Code View:
```
BufferedImage image = new BufferedImage(getWidth(), getHeight(), BufferedImage.TYPE_INT_ARGB);
Graphics2D gImage = image.createGraphics();
// now draw to gImage
g2.drawImage(image, null, 0, 0);
```

The complete code for the program is shown in Listing 7-3. Figure 7-24 shows the screen display. As you run the program, move the alpha slider from left to right to see the effect on the composed shapes. In particular, note that the only difference between the DST_IN and DST_OUT rules is how the destination (!) color changes when you change the source alpha.

Figure 7-24. The CompositeTest program

UNREGISTERED VERSION OF CHM TO PDF CONVERTER PRO BY THETA-SOFTWARE

Listing 7-3. **CompositeTest.java**

Code View:

```
import java.awt.*;
import java.awt.event.*;
import java.awt.image.*;
import java.awt.geom.*;
import javax.swing.*;
import javax.swing.event.*;

/**
 * This program demonstrates the Porter-Duff composition rules.
 * @version 1.03 2007-08-16
 * @author Cay Horstmann
 */
public class CompositeTest
{
   public static void main(String[] args)
   {
      EventQueue.invokeLater(new Runnable()
         {
            public void run()
            {
               JFrame frame = new CompositeTestFrame();
               frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
               frame.setVisible(true);
```

```
            }
         });
   }
}

/**
 * This frame contains a combo box to choose a composition rule, a slider to change the
 *  source alpha channel, and a component that shows the composition.
 */
class CompositeTestFrame extends JFrame
{
   public CompositeTestFrame()
   {
      setTitle("CompositeTest");
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

      canvas = new CompositeComponent();
      add(canvas, BorderLayout.CENTER);

      ruleCombo = new JComboBox(new Object[] { new Rule("CLEAR", "  ", "  "),
            new Rule("SRC", " S", " S"), new Rule("DST", "  ", "DD"),
            new Rule("SRC_OVER", " S", "DS"), new Rule("DST_OVER", " S", "DD"),
            new Rule("SRC_IN", "  ", " S"), new Rule("SRC_OUT", " S", "  "),
            new Rule("DST_IN", "  ", " D"), new Rule("DST_OUT", "  ", "D "),
            new Rule("SRC_ATOP", "  ", "DS"), new Rule("DST_ATOP", " S", " D"),
            new Rule("XOR", " S", "D "), });
      ruleCombo.addActionListener(new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               Rule r = (Rule) ruleCombo.getSelectedItem();
               canvas.setRule(r.getValue());
               explanation.setText(r.getExplanation());
            }
         });

      alphaSlider = new JSlider(0, 100, 75);
      alphaSlider.addChangeListener(new ChangeListener()
         {
            public void stateChanged(ChangeEvent event)
            {
               canvas.setAlpha(alphaSlider.getValue());
            }
         });
      JPanel panel = new JPanel();
      panel.add(ruleCombo);
      panel.add(new JLabel("Alpha"));
      panel.add(alphaSlider);
      add(panel, BorderLayout.NORTH);

      explanation = new JTextField();
      add(explanation, BorderLayout.SOUTH);

      canvas.setAlpha(alphaSlider.getValue());
      Rule r = (Rule) ruleCombo.getSelectedItem();
      canvas.setRule(r.getValue());
      explanation.setText(r.getExplanation());
   }

```

```
   private CompositeComponent canvas;
   private JComboBox ruleCombo;
   private JSlider alphaSlider;
   private JTextField explanation;
   private static final int DEFAULT_WIDTH = 400;
   private static final int DEFAULT_HEIGHT = 400;
}

/**
 * This class describes a Porter-Duff rule.
 */
class Rule
{
   /**
    * Constructs a Porter-Duff rule
    * @param n the rule name
    * @param pd1 the first row of the Porter-Duff square
    * @param pd2 the second row of the Porter-Duff square
    */
   public Rule(String n, String pd1, String pd2)
   {
      name = n;
      porterDuff1 = pd1;
      porterDuff2 = pd2;
   }

   /**
    * Gets an explanation of the behavior of this rule.
    * @return the explanation
    */
   public String getExplanation()
   {
      StringBuilder r = new StringBuilder("Source ");
      if (porterDuff2.equals("  ")) r.append("clears");
      if (porterDuff2.equals(" S")) r.append("overwrites");
      if (porterDuff2.equals("DS")) r.append("blends with");
      if (porterDuff2.equals(" D")) r.append("alpha modifies");
      if (porterDuff2.equals("D ")) r.append("alpha complement modifies");
      if (porterDuff2.equals("DD")) r.append("does not affect");
      r.append(" destination");
      if (porterDuff1.equals(" S")) r.append(" and overwrites empty pixels");
      r.append(".");
      return r.toString();
   }

   public String toString()
   {
      return name;
   }

   /**
    * Gets the value of this rule in the AlphaComposite class
    * @return the AlphaComposite constant value, or -1 if there is no matching constant.
    */
   public int getValue()
   {
      try
      {
         return (Integer) AlphaComposite.class.getField(name).get(null);
```

```
      }
      catch (Exception e)
      {
         return -1;
      }
   }

   private String name;
   private String porterDuff1;
   private String porterDuff2;
}

/**
 * This component draws two shapes, composed with a composition rule.
 */
class CompositeComponent extends JComponent
{
   public CompositeComponent()
   {
      shape1 = new Ellipse2D.Double(100, 100, 150, 100);
      shape2 = new Rectangle2D.Double(150, 150, 150, 100);
   }

   public void paintComponent(Graphics g)
   {
      Graphics2D g2 = (Graphics2D) g;

      BufferedImage image = new BufferedImage(getWidth(), getHeight(),
                                        BufferedImage.TYPE_INT_ARGB);
      Graphics2D gImage = image.createGraphics();
      gImage.setPaint(Color.red);
      gImage.fill(shape1);
      AlphaComposite composite = AlphaComposite.getInstance(rule, alpha);
      gImage.setComposite(composite);
      gImage.setPaint(Color.blue);
      gImage.fill(shape2);
      g2.drawImage(image, null, 0, 0);
   }

   /**
    * Sets the composition rule.
    * @param r the rule (as an AlphaComposite constant)
    */
   public void setRule(int r)
   {
      rule = r;
      repaint();
   }

   /**
    * Sets the alpha of the source
    * @param a the alpha value between 0 and 100
    */
   public void setAlpha(int a)
   {
      alpha = (float) a / 100.0F;
      repaint();
   }

```

```
    private int rule;
    private Shape shape1;
    private Shape shape2;
    private float alpha;
}
```


API java.awt.Graphics2D 1.2

UNREGISTERED VERSION OF CHM TO PDF CONVERTER PRO BY THETA-SOFTWARE

- `void setComposite(Composite s)`

  sets the composite of this graphics context to the given object that implements the `Composite` interface.

API java.awt.AlphaComposite 1.2

- `static AlphaComposite getInstance(int rule)`

- `static AlphaComposite getInstance(int rule, float sourceAlpha)`

  constructs an alpha composite object. The rule is one of `CLEAR`, `SRC`, `SRC_OVER`, `DST_OVER`, `SRC_IN`, `SRC_OUT`, `DST_IN`, `DST_OUT`, `DST`, `DST_ATOP`, `SRC_ATOP`, `XOR`.

Rendering Hints

In the preceding sections you have seen that the rendering process is quite complex. Although the Java 2D API is
in most cases, there are cases when you would like to have control over trade-offs between speed and quality. You
setting *rendering hints.* The setRenderingHint method of the Graphics2D class lets you set a single hint. The hin
values are declared in the RenderingHints class. Table 7-2 summarizes the choices. The values that end in _DEFA
defaults that are chosen by a particular implementation as a good trade-off between performance and quality.

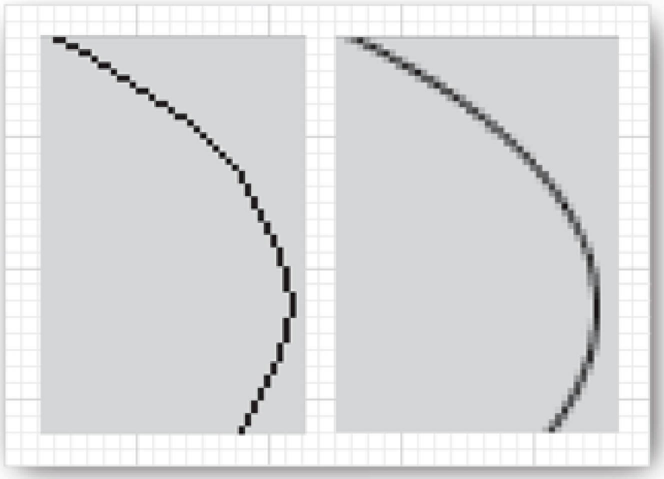
Table 7-2. Rendering Hints

Key	Value	Explanation
KEY_ANTIALIASING	VALUE_ANTIALIAS_ON VALUE_ANTIALIAS_OFF VALUE_ANTIALIAS_DEFAULT	Turn antialiasing for shapes on or off.
KEY_TEXT_ANTIALIASING	VALUE_TEXT_ANTIALIAS_ON VALUE_TEXT_ANTIALIAS_OFF VALUE_TEXT_ANTIALIAS_DEFAULT VALUE_TEXT_ANTIALIAS_GASP **6** VALUE_TEXT_ANTIALIAS_LCD_HRGB **6** VALUE_TEXT_ANTIALIAS_LCD_HBGR **6** VALUE_TEXT_ANTIALIAS_LCD_VRGB **6** VALUE_TEXT_ANTIALIAS_LCD_VBGR **6**	Turn antialiasing for fonts on or off. When VALUE_TEXT_ANTIALIAS_GASP , the "gasp font is consulted to decide whether a part font should be antialiased. The LCD values rendering for a particular display type.
KEY_FRACTIONALMETRICS	VALUE_FRACTIONALMETRICS_ON VALUE_FRACTIONALMETRICS_OFF VALUE_FRACTIONALMETRICS_DEFAULT	Turn the computation of fractional charact on or off. Fractional character dimensions placement of characters.
KEY_RENDERING	VALUE_RENDER_QUALITY VALUE_RENDER_SPEED VALUE_RENDER_DEFAULT	When available, select rendering algorithn quality or speed.
KEY_STROKE_CONTROL **1.3**	VALUE_STROKE_NORMALIZE VALUE_STROKE_PURE VALUE_STROKE_DEFAULT	Select whether the placement of strokes is the graphics accelerator (which may move a pixel) or is computed by the "pure" rule that strokes run through the centers of pi
KEY_DITHERING	VALUE_DITHER_ENABLE VALUE_DITHER_DISABLE VALUE_DITHER_DEFAULT	Turn dithering for colors on or off. Ditheri approximates color values by drawing gro similar colors. (Note that antialiasing can dithering.)
KEY_ALPHA_INTERPOLATION	VALUE_ALPHA_INTERPOLATION_QUALITY VALUE_ALPHA_INTERPOLATION_SPEED VALUE_ALPHA_INTERPOLATION_DEFAULT	Turn precise computation of alpha compos
KEY_COLOR_RENDERING	VALUE_COLOR_RENDER_QUALITY VALUE_COLOR_RENDER_SPEED VALUE_COLOR_RENDER_DEFAULT	Select quality or speed for color rendering issue when you use different color spaces

Key	Value	Explanation
KEY_INTERPOLATION	VALUE_INTERPOLATION_NEAREST_NEIGHBOR VALUE_INTERPOLATION_BILINEAR VALUE_INTERPOLATION_BICUBIC	Select a rule for interpolating pixels when rotating images.

The most useful of these settings involves *antialiasing*. This technique removes the "jaggies" from slanted lines an can see in Figure 7-25 , a slanted line must be drawn as a "staircase" of pixels. Especially on low-resolution screen look ugly. But if, rather than drawing each pixel completely on or off, you color in the pixels that are partially cove color value proportional to the area of the pixel that the line covers, then the result looks much smoother. This tec antialiasing. Of course, antialiasing takes a bit longer because it takes time to compute all those color values.

Figure 7-25. Antialiasing



For example, here is how you can request the use of antialiasing:

Code View:
```
g2.setRenderingHint(RenderingHints.KEY_ANTIALIASING, RenderingHints.VALUE_ANTIALIAS_ON);
```

It also makes sense to use antialiasing for fonts.

```
g2.setRenderingHint(RenderingHints.KEY_TEXT_ANTIALIASING, Rendering-
Hints.VALUE_TEXT_ANTIALIAS_ON);
```

The other rendering hints are not as commonly used.

You can also put a bunch of key/value hint pairs into a map and set them all at once by calling the setRenderingH
Any collection class implementing the map interface will do, but you might as well use the RenderingHints class i
implements the Map interface and supplies a default map implementation if you pass null to the constructor. For e

Code View:
```
RenderingHints hints = new RenderingHints(null);
hints.put(RenderingHints.KEY_ANTIALIASING, RenderingHints.VALUE_ANTIALIAS_ON);
hints.put(RenderingHints.KEY_TEXT_ANTIALIASING, RenderingHints.VALUE_TEXT_ANTIALIAS_ON);
g2.setRenderingHints(hints);
```

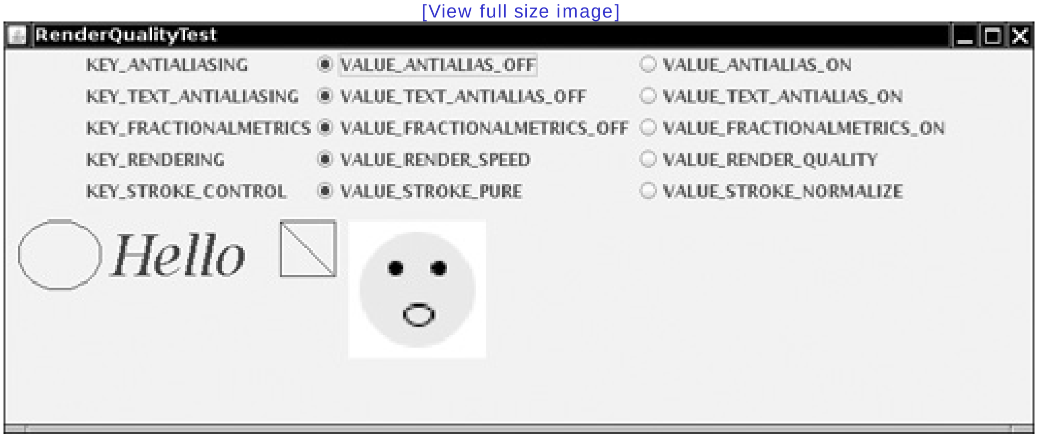
That is the technique we use in Listing 7-4 . The program shows several rendering hints that we found beneficial. N
following:

- Antialiasing smooths the ellipse.
- Text antialiasing smooths the text.
- On some platforms, fractional text metrics move the letters a bit closer together.
- Selecting VALUE_RENDER_QUALITY smooths the scaled image. (You would get the same effect by setting KEY_
  to VALUE_INTERPOLATION_BICUBIC).
- When antialiasing is turned off, selecting VALUE_STROKE_NORMALIZE changes the appearance of the ellipse an
  of the diagonal line in the square.

Figure 7-26 shows a screen capture of the program.

Figure 7-26. Testing the effect of rendering hints

[View full size image]



Listing 7-4. **RenderQualityTest.java**

Code View:

```
import java.awt.*;
import java.awt.event.*;
import java.awt.geom.*;
import java.io.*;
import javax.imageio.*;
import javax.swing.*;

```



```
/**
 * This program demonstrates the effect of the various rendering hints.
 * @version 1.10 2007-08-16
 * @author Cay Horstmann
 */
public class RenderQualityTest
```



```
{
   public static void main(String[] args)
   {
      EventQueue.invokeLater(new Runnable()
         {
            public void run()
            {
               JFrame frame = new RenderQualityTestFrame();
               frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
               frame.setVisible(true);
            }
         });
   }
}

/**
 * This frame contains buttons to set rendering hints and an image that is drawn with
 * the selected hints.
 */
class RenderQualityTestFrame extends JFrame
{
   public RenderQualityTestFrame()
   {
      setTitle("RenderQualityTest");
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

      buttonBox = new JPanel();
      buttonBox.setLayout(new GridBagLayout());
      hints = new RenderingHints(null);

      makeButtons("KEY_ANTIALIASING", "VALUE_ANTIALIAS_OFF", "VALUE_ANTIALIAS_ON");
      makeButtons("KEY_TEXT_ANTIALIASING", "VALUE_TEXT_ANTIALIAS_OFF",
                  "VALUE_TEXT_ANTIALIAS_ON");
      makeButtons("KEY_FRACTIONALMETRICS", "VALUE_FRACTIONALMETRICS_OFF",
            "VALUE_FRACTIONALMETRICS_ON");
      makeButtons("KEY_RENDERING", "VALUE_RENDER_SPEED", "VALUE_RENDER_QUALITY");
      makeButtons("KEY_STROKE_CONTROL", "VALUE_STROKE_PURE", "VALUE_STROKE_NORMALIZE");
      canvas = new RenderQualityComponent();
      canvas.setRenderingHints(hints);

      add(canvas, BorderLayout.CENTER);
      add(buttonBox, BorderLayout.NORTH);
   }

```

```
   /**
    * Makes a set of buttons for a rendering hint key and values
    * @param key the key name
    * @param value1 the name of the first value for the key
    * @param value2 the name of the second value for the key
    */
   void makeButtons(String key, String value1, String value2)
   {
      try
      {
         final RenderingHints.Key k =
            (RenderingHints.Key) RenderingHints.class.getField(key).get(null);
         final Object v1 = RenderingHints.class.getField(value1).get(null);
         final Object v2 = RenderingHints.class.getField(value2).get(null);
         JLabel label = new JLabel(key);

         buttonBox.add(label, new GBC(0, r).setAnchor(GBC.WEST));
         ButtonGroup group = new ButtonGroup();
         JRadioButton b1 = new JRadioButton(value1, true);

         buttonBox.add(b1, new GBC(1, r).setAnchor(GBC.WEST));
         group.add(b1);
         b1.addActionListener(new ActionListener()
            {
               public void actionPerformed(ActionEvent event)
               {
                  hints.put(k, v1);
                  canvas.setRenderingHints(hints);
               }
            });
         JRadioButton b2 = new JRadioButton(value2, false);

         buttonBox.add(b2, new GBC(2, r).setAnchor(GBC.WEST));
         group.add(b2);
         b2.addActionListener(new ActionListener()
            {
               public void actionPerformed(ActionEvent event)
               {
                  hints.put(k, v2);
                  canvas.setRenderingHints(hints);
               }
            });
         hints.put(k, v1);
         r++;
      }
      catch (Exception e)
      {
         e.printStackTrace();
      }
   }

   private RenderQualityComponent canvas;
   private JPanel buttonBox;
   private RenderingHints hints;
   private int r;
   private static final int DEFAULT_WIDTH = 750;
   private static final int DEFAULT_HEIGHT = 300;
}
```

```
/**
 * This component produces a drawing that shows the effect of rendering hints.
 */
class RenderQualityComponent extends JComponent
{
   public RenderQualityComponent()
   {
      try
      {
```



```
         image = ImageIO.read(new File("face.gif"));
      }
      catch (IOException e)
      {
         e.printStackTrace();
      }
   }
```



```

   public void paintComponent(Graphics g)
   {
      Graphics2D g2 = (Graphics2D) g;
      g2.setRenderingHints(hints);

      g2.draw(new Ellipse2D.Double(10, 10, 60, 50));
      g2.setFont(new Font("Serif", Font.ITALIC, 40));
      g2.drawString("Hello", 75, 50);

      g2.draw(new Rectangle2D.Double(200, 10, 40, 40));
      g2.draw(new Line2D.Double(201, 11, 239, 49));

      g2.drawImage(image, 250, 10, 100, 100, null);
   }

   /**
    * Sets the hints and repaints.
    * @param h the rendering hints
    */
   public void setRenderingHints(RenderingHints h)
   {
      hints = h;
      repaint();
   }

   private RenderingHints hints = new RenderingHints(null);
   private Image image;
}
```

`java.awt.Graphics2D` 1.2

- `void setRenderingHint(RenderingHints.Key key, Object value)`

  sets a rendering hint for this graphics context.

- `void setRenderingHints(Map m)`

  sets all rendering hints whose key/value pairs are stored in the map.

`java.awt.RenderingHints` 1.2

- `RenderingHints(Map<RenderingHints.Key, ?> m)`

  constructs a rendering hints map for storing rendering hints. If `m` is `null` , then a default map implementation is provided.

Readers and Writers for Images

Prior to version 1.4, Java SE had very limited capabilities for reading and writing image files. It was possible to read GIF and JPEG images, but there was no official support for writing images at all.

This situation is now much improved. Java SE 1.4 introduced the `javax.imageio` package that contains "out of the box" support for reading and writing several common file formats, as well as a framework that enables third parties to add readers and writers for other formats. As of Java SE 6, the GIF, JPEG, PNG, BMP (Windows bitmap), and WBMP (wireless bitmap) file formats are supported. In earlier versions, writing of GIF files was not supported because of patent issues.

The basics of the library are extremely straightforward. To load an image, use the static `read` method of the `ImageIO` class:


```
File f = . . .;
BufferedImage image = ImageIO.read(f);
```

The `ImageIO` class picks an appropriate reader, based on the file type. It may consult the file extension and the "magic number" at the beginning of the file for that purpose. If no suitable reader can be found or the reader can't decode the file contents, then the `read` method returns `null`.

Writing an image to a file is just as simple:

```
File f = . . .;
String format = . . .;
ImageIO.write(image, format, f);
```

Here the format string is a string identifying the image format, such as `"JPEG"` or `"PNG"`. The `ImageIO` class picks an appropriate writer and saves the file.

Obtaining Readers and Writers for Image File Types

For more advanced image reading and writing operations that go beyond the static `read` and `write` methods of the `ImageIO` class, you first need to get the appropriate `ImageReader` and `ImageWriter` objects. The `ImageIO` class enumerates readers and writers that match one of the following:

- An image format (such as "JPEG")
- A file suffix (such as "`jpg`")
- A MIME type (such as "image/jpeg")

Note

MIME is the Multipurpose Internet Mail Extensions standard. The MIME standard defines common data formats such as "image/jpeg" and "application/pdf". For an HTML version of the Request for Comments (RFC) that defines the MIME format, see http://www.oac.uci.edu/indiv/ehood/MIME.

For example, you can obtain a reader that reads JPEG files as follows:

```
ImageReader reader = null;
Iterator<ImageReader> iter = ImageIO.getImageReadersByFormatName("JPEG");
if (iter.hasNext()) reader = iter.next();
```

The `getImageReadersBySuffix` and `getImageReadersByMIMEType` method enumerate readers that match a file extension or MIME type.

It is possible that the `ImageIO` class can locate multiple readers that can all read a particular image type. In that case, you have to pick one of them, but it isn't clear how you can decide which one is the best. To find out more information about a reader, obtain its *service provider interface*:

```
ImageReaderSpi spi = reader.getOriginatingProvider();
```

Then you can get the vendor name and version number:

```
String vendor = spi.getVendor();
String version = spi.getVersion();
```

Perhaps that information can help you decide among the choices, or you might just present a list of readers to your program users and let them choose. However, for now, we assume that the first enumerated reader is adequate.

In the sample program in Listing 7-5, we want to find all file suffixes of all available readers so that we can use them in a file filter. As of Java SE 6, we can use the static `ImageIO.getReaderFileSuffixes` method for this purpose:

Code View:
```
String[] extensions = ImageIO.getWriterFileSuffixes();
chooser.setFileFilter(new FileNameExtensionFilter("Image files", extensions));
```

For saving files, we have to work harder. We'd like to present the user with a menu of all supported image types. Unfortunately, the `getWriterFormatNames` of the `IOImage` class returns a rather curious list with redundant names, such as

```
jpg, BMP, bmp, JPG, jpeg, wbmp, png, JPEG, PNG, WBMP, GIF, gif
```

That's not something one would want to present in a menu. What is needed is a list of "preferred" format names. We supply a helper method `getWriterFormats` for this purpose (see Listing 7-5). We look up the first writer associated with each format name. Then we ask it what its format names are, in the hope that it will list the most popular one first. Indeed, for the JPEG writer, this works fine: It lists "JPEG" before the other options. (The PNG writer, on the other hand, lists "png" in lower case before "PNG". We hope this behavior will be addressed at some time in the future. In the meantime, we force all-lowercase names to upper case.) Once we pick a preferred name, we remove all alternate names from the original set. We keep going until all format names are handled.

Reading and Writing Files with Multiple Images

Some files, in particular, animated GIF files, contain multiple images. The read method of the ImageIO class reads a single image. To read multiple images, turn the input source (for example, an input stream or file) into an ImageInputStream.

```
InputStream in = . . .;
ImageInputStream imageIn = ImageIO.createImageInputStream(in);
```


Then attach the image input stream to the reader:

```
reader.setInput(imageIn, true);
```

The second parameter indicates that the input is in "seek forward only" mode. Otherwise, random access is used, either by buffering stream input as it is read or by using random file access. Random access is required for certain operations. For example, to find out the number of images in a GIF file, you need to read the entire file. If you then want to fetch an image, the input must be read again.

This consideration is only important if you read from a stream, if the input contains multiple images, and if the image format doesn't have the information that you request (such as the image count) in the header. If you read from a file, simply use

```
File f = . . .;
ImageInputStream imageIn = ImageIO.createImageInputStream(f);
reader.setInput(imageIn);
```

Once you have a reader, you can read the images in the input by calling

```
BufferedImage image = reader.read(index);
```

where index is the image index, starting with 0.

If the input is in "seek forward only" mode, you keep reading images until the read method throws an IndexOutOfBoundsException. Otherwise, you can call the getNumImages method:

```
int n = reader.getNumImages(true);
```

Here, the parameter indicates that you allow a search of the input to determine the number of images. That method throws an IllegalStateException if the input is in "seek forward only" mode. Alternatively, you can set the "allow search" parameter to false. Then the getNumImages method returns -1 if it can't determine the number of images without a search. In that case, you'll have to switch to Plan B and keep reading images until you get an IndexOutOfBoundsException.

Some files contain thumbnails, smaller versions of an image for preview purposes. You can get the number of thumbnails of an image with the call

```
int count = reader.getNumThumbnails(index);
```

Then you get a particular index as

```
BufferedImage thumbnail = reader.getThumbnail(index, thumbnailIndex);
```

Another consideration is that you sometimes want to get the image size before actually getting the image, in particular, if the image is huge or comes from a slow network connection. Use the calls

```
int width = reader.getWidth(index);
int height = reader.getHeight(index);
```

to get the dimensions of an image with a given index.

To write a file with multiple images, you first need an `ImageWriter`. The `ImageIO` class can enumerate the writers that are capable of writing a particular image format:

Code View:

```
String format = . . .;
ImageWriter writer = null;
Iterator<ImageWriter> iter =  ImageIO.getImageWritersByFormatName( format );
if (iter.hasNext()) writer = iter.next();
```

Next, turn an output stream or file into an `ImageOutputStream` and attach it to the writer. For example,

```
File f = . . .;
ImageOutputStream imageOut = ImageIO.createImageOutputStream(f);
writer.setOutput(imageOut);
```

You must wrap each image into an `IIOImage` object. You can optionally supply a list of thumbnails and image metadata (such as compression algorithms and color information). In this example, we just use `null` for both; see the API documentation for additional information.

```
IIOImage iioImage = new IIOImage(images[i], null, null);
```

Write out the *first* image, using the `write` method:

```
writer.write(new IIOImage(images[0], null, null));
```

For subsequent images, use

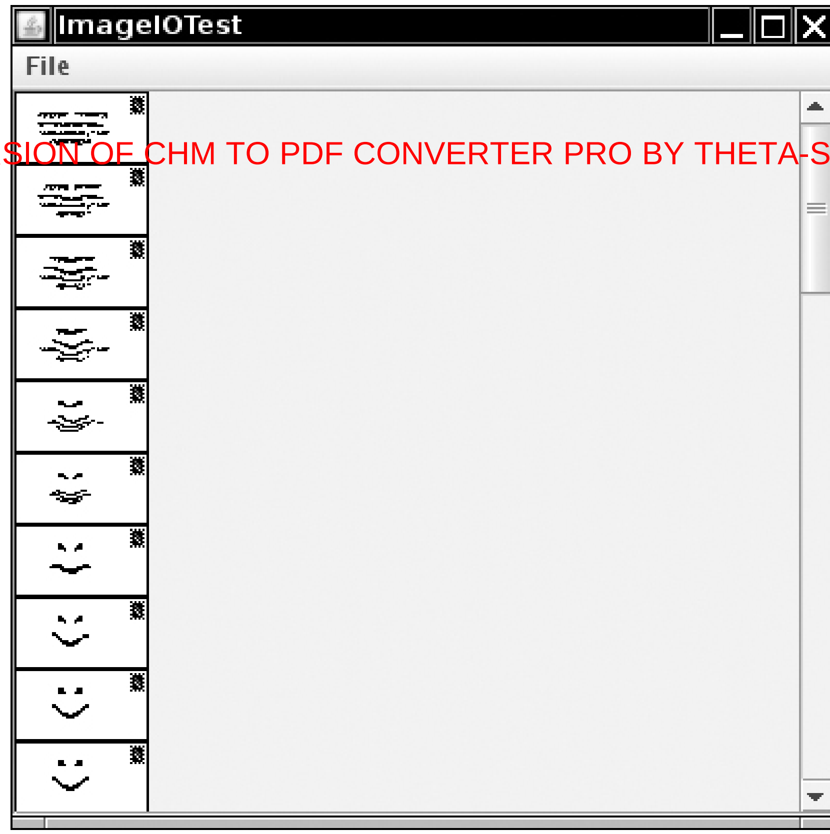
```
if (writer.canInsertImage(i))
   writer.writeInsert(i, iioImage, null);
```

The third parameter can contain an `ImageWriteParam` object to set image writing details such as tiling and compression; use `null` for default values.

Not all file formats can handle multiple images. In that case, the canInsertImage method returns false for i > 0, and only a single image is saved.

The program in Listing 7-5 lets you load and save files in the formats for which the Java library supplies readers and writers. The program displays multiple images (see Figure 7-27), but not thumbnails.

Figure 7-27. An animated GIF image



Listing 7-5. **ImageIOTest.java**

Code View:
```
import java.awt.*;
import java.awt.event.*;
import java.awt.image.*;
import java.io.*;
import java.util.*;
import javax.imageio.*;
import javax.imageio.stream.*;
import javax.swing.*;
import javax.swing.filechooser.*;

/**
 * This program lets you read and write image files in the formats that the JDK supports.
```

```
 * Multi-file images are supported.
 * @version 1.02 2007-08-16
 * @author Cay Horstmann
 */
public class ImageIOTest
{
   public static void main(String[] args)
   {
      EventQueue.invokeLater(new Runnable()
         {
            public void run()
            {
               JFrame frame = new ImageIOFrame();
               frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
               frame.setVisible(true);
            }
         });
   }
}

/**
 * This frame displays the loaded images. The menu has items for loading and saving files.
 */
class ImageIOFrame extends JFrame
{
   public ImageIOFrame()
   {
      setTitle("ImageIOTest");
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

      JMenu fileMenu = new JMenu("File");
      JMenuItem openItem = new JMenuItem("Open");
      openItem.addActionListener(new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               openFile();
            }
         });
      fileMenu.add(openItem);

      JMenu saveMenu = new JMenu("Save");
      fileMenu.add(saveMenu);
      Iterator<String> iter = writerFormats.iterator();
      while (iter.hasNext())
      {
         final String formatName = iter.next();
         JMenuItem formatItem = new JMenuItem(formatName);
         saveMenu.add(formatItem);
         formatItem.addActionListener(new ActionListener()
            {
               public void actionPerformed(ActionEvent event)
               {
                  saveFile(formatName);
               }
            });
      }

      JMenuItem exitItem = new JMenuItem("Exit");
```

```
      exitItem.addActionListener(new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               System.exit(0);
            }
         });
      fileMenu.add(exitItem);

```

```
      JMenuBar menuBar = new JMenuBar();
      menuBar.add(fileMenu);
      setJMenuBar(menuBar);
   }

   /**
    * Open a file and load the images.
```

```
    */
   public void openFile()
   {
      JFileChooser chooser = new JFileChooser();
      chooser.setCurrentDirectory(new File("."));
      String[] extensions = ImageIO.getReaderFileSuffixes();
      chooser.setFileFilter(new FileNameExtensionFilter("Image files", extensions));
      int r = chooser.showOpenDialog(this);
      if (r != JFileChooser.APPROVE_OPTION) return;
      File f = chooser.getSelectedFile();
      Box box = Box.createVerticalBox();
      try
      {
         String name = f.getName();
         String suffix = name.substring(name.lastIndexOf('.') + 1);
         Iterator<ImageReader> iter = ImageIO.getImageReadersBySuffix(suffix);
         ImageReader reader = iter.next();
         ImageInputStream imageIn = ImageIO.createImageInputStream(f);
         reader.setInput(imageIn);
         int count = reader.getNumImages(true);
         images = new BufferedImage[count];
         for (int i = 0; i < count; i++)
         {
            images[i] = reader.read(i);
            box.add(new JLabel(new ImageIcon(images[i])));
         }
      }
      catch (IOException e)
      {
         JOptionPane.showMessageDialog(this, e);
      }
      setContentPane(new JScrollPane(box));
      validate();
   }

   /**
    * Save the current image in a file
    * @param formatName the file format
    */
   public void saveFile(final String formatName)
   {
      if (images == null) return;
      Iterator<ImageWriter> iter = ImageIO.getImageWritersByFormatName(formatName);
```

```java
      ImageWriter writer = iter.next();
      JFileChooser chooser = new JFileChooser();
      chooser.setCurrentDirectory(new File("."));
      String[] extensions = writer.getOriginatingProvider().getFileSuffixes();
      chooser.setFileFilter(new FileNameExtensionFilter("Image files", extensions));

      int r = chooser.showSaveDialog(this);
      if (r != JFileChooser.APPROVE_OPTION) return;
      File f = chooser.getSelectedFile();
      try
      {
         ImageOutputStream imageOut = ImageIO.createImageOutputStream(f);
         writer.setOutput(imageOut);

         writer.write(new IIOImage(images[0], null, null));
         for (int i = 1; i < images.length; i++)
         {
            IIOImage iioImage = new IIOImage(images[i], null, null);
            if (writer.canInsertImage(i)) writer.writeInsert(i, iioImage, null);
         }
      }
      catch (IOException e)
      {
         JOptionPane.showMessageDialog(this, e);
      }
   }

   /**
    * Gets a set of "preferred" format names of all image writers. The preferred format name
    * is the first format name that a writer specifies.
    * @return the format name set
    */
   public static Set<String> getWriterFormats()
   {
      TreeSet<String> writerFormats = new TreeSet<String>();
      TreeSet<String> formatNames = new TreeSet<String>(Arrays.asList(ImageIO
            .getWriterFormatNames()));
      while (formatNames.size() > 0)
      {
         String name = formatNames.iterator().next();
         Iterator<ImageWriter> iter = ImageIO.getImageWritersByFormatName(name);
         ImageWriter writer = iter.next();
         String[] names = writer.getOriginatingProvider().getFormatNames();
         String format = names[0];
         if (format.equals(format.toLowerCase())) format = format.toUpperCase();
         writerFormats.add(format);
         formatNames.removeAll(Arrays.asList(names));
      }
      return writerFormats;
   }

   private BufferedImage[] images;
   private static Set<String> writerFormats = getWriterFormats();
   private static final int DEFAULT_WIDTH = 400;
   private static final int DEFAULT_HEIGHT = 400;
}
```

javax.imageio.ImageIO 1.4

- static BufferedImage read(File input)
- static BufferedImage read(InputStream input)

- static BufferedImage read(URL input)

  reads an image from input.

- static boolean write(RenderedImage image, String formatName, File output)
- static boolean write(RenderedImage image, String formatName, OutputStream output)

  writes an image in the given format to output. Returns false if no appropriate writer was found.

- static Iterator<ImageReader> getImageReadersByFormatName(String formatName)
- static Iterator<ImageReader> getImageReadersBySuffix(String fileSuffix)
- static Iterator<ImageReader> getImageReadersByMIMEType(String mimeType)
- static Iterator<ImageWriter> getImageWritersByFormatName(String formatName)
- static Iterator<ImageWriter> getImageWritersBySuffix(String fileSuffix)
- static Iterator<ImageWriter> getImageWritersByMIMEType(String mimeType)

  gets all readers and writers that are able to handle the given format (e.g., "JPEG"), file suffix (e.g., "jpg"), or MIME type (e.g., "image/jpeg").

- `static String[] getReaderFormatNames()`
- `static String[] getReaderMIMETypes()`
- `static String[] getWriterFormatNames()`
- `static String[] getWriterMIMETypes()`
- `static String[] getReaderFileSuffixes()` **6**
- `static String[] getWriterFileSuffixes()` **6**

  gets all format names, MIME type names, and file suffixes supported by readers and writers.

- `ImageInputStream createImageInputStream(Object input)`
- `ImageOutputStream createImageOutputStream(Object output)`

  creates an image input or image output stream from the given object. The object can be a file, a stream, a `RandomAccessFile`, or another object for which a service provider exists. Returns `null` if no registered service provider can handle the object.

API `javax.imageio.ImageReader` 1.4

- `void setInput(Object input)`
- `void setInput(Object input, boolean seekForwardOnly)`

  sets the input source of the reader.

Parameters:	`input`	An `ImageInputStream` object or another object that this reader can accept.
	`seekForwardOnly`	`true` if the reader should read forward only. By default, the reader uses random access and, if necessary, buffers image data.

- `BufferedImage read(int index)`

reads the image with the given image index (starting at 0). Throws an `IndexOutOfBoundsException` if no such image is available.

- `int getNumImages(boolean allowSearch)`

  gets the number of images in this reader. If `allowSearch` is `false` and the number of images cannot be determined without reading forward, then `-1` is returned. If `allowSearch` is `true` and the reader is in "seek forward only" mode, then an `IllegalStateException` is thrown.

- `int getNumThumbnails(int index)`

  gets the number of thumbnails of the image with the given index.

- `BufferedImage readThumbnail(int index, int thumbnailIndex)`

  gets the thumbnail with index `thumbnailIndex` of the image with the given index.

- `int getWidth(int index)`

- `int getHeight(int index)`

  gets the image width and height. Throw an `IndexOutOfBoundsException` if no such image is available.

- `ImageReaderSpi getOriginatingProvider()`

  gets the service provider that constructed this reader.

API `javax.imageio.spi.IIOServiceProvider` 1.4

- `String getVendorName()`

- `String getVersion()`

  gets the vendor name and version of this service provider.

API

javax.imageio.spi.ImageReaderWriterSpi 1.4

- `String[] getFormatNames()`
- `String[] getFileSuffixes()`
- `String[] getMIMETypes()`

  gets the format names, file suffixes, and MIME types supported by the readers or writers that this service provider creates.

API

javax.imageio.ImageWriter 1.4

- `void setOutput(Object output)`

  sets the output target of this writer.

Parameters:	`output`	An `ImageOutputStream` object or another object that this writer can accept

- `void write(IIOImage image)`
- `void write(RenderedImage image)`

  writes a single image to the output.

- `void writeInsert(int index, IIOImage image, ImageWriteParam param)`

  writes an image into a multi-image file.

- `boolean canInsertImage(int index)`

  returns `true` if it is possible to insert an image at the given index.

- `ImageWriterSpi getOriginatingProvider()`

  gets the service provider that constructed this writer.

API

`javax.imageio.IIOImage` 1.4

- `IIOImage(RenderedImage image, List thumbnails, IIOMetadata metadata)`

  constructs an `IIOImage` from an image, optional thumbnails, and optional metadata.

Image Manipulation

Suppose you have an image and you would like to improve its appearance. You then need to access the individual pixels of the image and replace them with other pixels. Or perhaps you want to compute the pixels of an image from scratch, for example, to show the result of physical measurements or a mathematical computation. The `BufferedImage` class gives you control over the pixels in an image, and classes that implement the `BufferedImageOp` interface let you transform images.

Note

JDK 1.0 had a completely different, and far more complex, imaging framework that was optimized for *incremental rendering* of images that are downloaded from the Web, a scan line at a time. However, it was difficult to manipulate those images. We do not discuss that framework in this book.

Constructing Raster Images

Most of the images that you manipulate are simply read in from an image file—they were either produced by a device such as a digital camera or scanner, or constructed by a drawing program. In this section, we show you a different technique for constructing an image, namely, to build up an image a pixel at a time.

To create an image, construct a `BufferedImage` object in the usual way.

```
image = new BufferedImage(width, height, BufferedImage.TYPE_INT_ARGB);
```

Now, call the `getRaster` method to obtain an object of type `WritableRaster`. You use this object to access and modify the pixels of the image.

```
WritableRaster raster = image.getRaster();
```

The `setPixel` method lets you set an individual pixel. The complexity here is that you can't simply set the pixel to a `Color` value. You must know how the buffered image specifies color values. That depends on the *type* of the image. If your image has a type of `TYPE_INT_ARGB`, then each pixel is described by four values, for red, green, blue, and alpha, each of which is between 0 and 255. You supply them in an array of four integers.

```
int[] black = { 0, 0, 0, 255 };
raster.setPixel(i, j, black);
```

In the lingo of the Java 2D API, these values are called the *sample values* of the pixel.

Caution

There are also `setPixel` methods that take array parameters of types `float[]` and `double[]`. However, the values that you need to place into these arrays are *not* normalized color values between 0.0 and 1.0.

```
float[] red = { 1.0F, 0.0F, 0.0F, 1.0F };
raster.setPixel(i, j, red); // ERROR
```


You need to supply values between 0 and 255, no matter what the type of the array is.

You can supply batches of pixels with the `setPixels` method. Specify the starting pixel position and the width and height of the rectangle that you want to set. Then, supply an array that contains the sample values for all pixels. For example, if your buffered image has a type of `TYPE_INT_ARGB`, then you supply the red, green, blue, and alpha value of the first pixel, then the red, green, blue, and alpha value for the second pixel, and so on.

```
int[] pixels = new int[4 * width * height];
pixels[0] = . . . // red value for first pixel
pixels[1] = . . . // green value for first pixel
pixels[2] = . . . // blue value for first pixel
pixels[3] = . . . // alpha value for first pixel
. . .
raster.setPixels(x, y, width, height, pixels);
```

Conversely, to read a pixel, you use the `getPixel` method. Supply an array of four integers to hold the sample values.

```
int[] sample = new int[4];
raster.getPixel(x, y, sample);
Color c = new Color(sample[0], sample[1], sample[2], sample[3]);
```

You can read multiple pixels with the `getPixels` method.

```
raster.getPixels(x, y, width, height, samples);
```

If you use an image type other than `TYPE_INT_ARGB` and you know how that type represents pixel values, then you can still use the `getPixel/setPixel` methods. However, you have to know the encoding of the sample values in the particular image type.

If you need to manipulate an image with an arbitrary, unknown image type, then you have to work a bit harder. Every image type has a *color model* that can translate between sample value arrays and the standard RGB color model.

Note

The RGB color model isn't as standard as you might think. The exact look of a color value depends on the characteristics of the imaging device. Digital cameras, scanners, monitors, and LCD displays all have their own idiosyncrasies. As a result, the same RGB value can look quite different on different devices. The International Color Consortium (http://www.color.org) recommends that all color data be accompanied by an *ICC profile* that specifies how the colors map to a standard form such as the 1931 CIE XYZ color specification. That specification was designed by the Commission Internationale de l'Eclairage or CIE (http://www.cie.co.at/cie), the international organization in charge of providing technical guidance in all matters of illumination and color. The specification is a standard method for representing all colors that the human eye can perceive as a triplet of coordinates called X, Y, Z. (See, for example, *Computer Graphics: Principles and Practice, Second Edition in C* by James D. Foley, Andries van Dam, Steven K. Feiner, et al., Chapter 13, for more information on the 1931 CIE XYZ specification.)

ICC profiles are complex, however. A simpler proposed standard, called sRGB (http://www.w3.org/Graphics/Color/sRGB.html), specifies an exact mapping between RGB values and the 1931 CIE XYZ values that was designed to work well with typical color monitors. The Java 2D API uses that mapping when converting between RGB and other color spaces.

The `getColorModel` method returns the color model:

```
ColorModel model = image.getColorModel();
```

To find the color value of a pixel, you call the `getDataElements` method of the `Raster` class. That call returns an `Object` that contains a color-model-specific description of the color value.

```
Object data = raster.getDataElements(x, y, null);
```

Note

The object that is returned by the `getDataElements` method is actually an array of sample values. You don't need to know this to process the object, but it explains why the method is called `getDataElements`.

The color model can translate the object to standard ARGB values. The `getRGB` method returns an `int` value that has the alpha, red, green, and blue values packed in four blocks of 8 bits each. You can construct a `Color` value out of that integer with the `Color(int argb, boolean hasAlpha)` constructor.

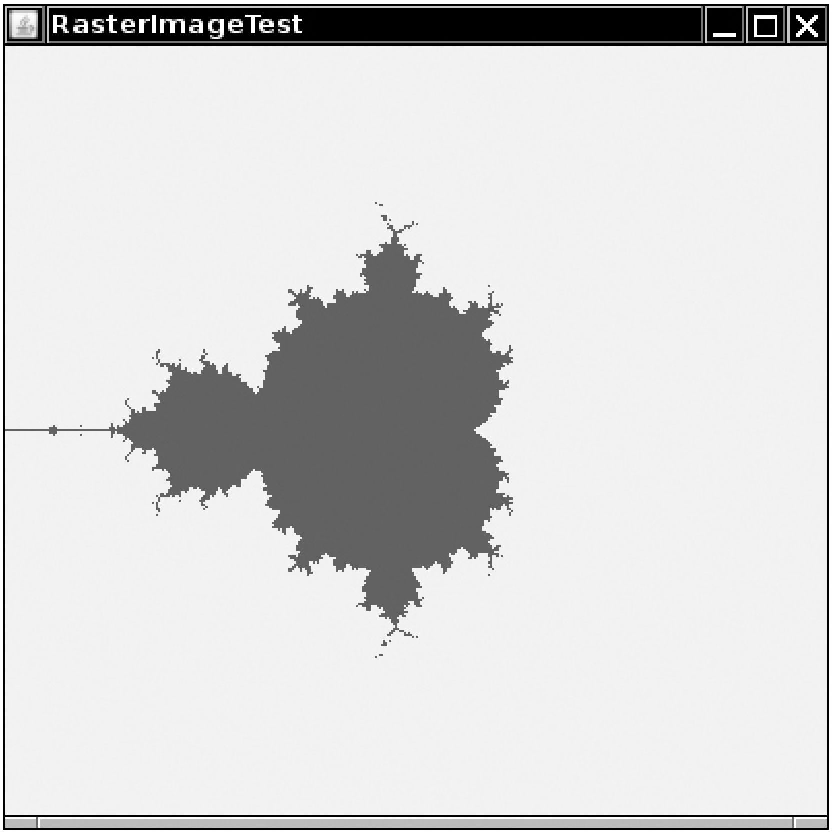
```
int argb = model.getRGB(data);
Color color = new Color(argb, true);
```

To set a pixel to a particular color, you reverse these steps. The `getRGB` method of the `Color` class yields an `int` value with the alpha, red, green, and blue values. Supply that value to the getDataElements method of the ColorModel class. The return value is an Object that contains the color-model-specific description of the color value. Pass the object to the setDataElements method of the WritableRaster class.

```
int argb = color.getRGB();
Object data = model.getDataElements(argb, null);
raster.setDataElements(x, y, data);
```


To illustrate how to use these methods to build an image from individual pixels, we bow to tradition and draw a Mandelbrot set, as shown in Figure 7-28.

Figure 7-28. A Mandelbrot set



The idea of the Mandelbrot set is that you associate with each point in the plane a sequence of numbers. If that sequence stays bounded, you color the point. If it "escapes to infinity," you leave it transparent.

Here is how you can construct the simplest Mandelbrot set. For each point (a, b), you look at sequences that start with (x, y) = (0, 0) and iterate:

$$x_{new} = x^2 - y^2 + a$$

$$y_{new} = 2 \cdot x \cdot y + b$$

It turns out that if *x* or *y* ever gets larger than 2, then the sequence escapes to infinity. Only the pixels that correspond to points (*a*, *b*) leading to a bounded sequence are colored. (The formulas for the number sequences come ultimately from the mathematics of complex numbers. We just take them for granted. For more on the mathematics of fractals, see, for example, http://classes.yale.edu/fractals/.)

Listing 7-6 shows the code. In this program, we demonstrate how to use the `ColorModel` class for translating `Color` values into pixel data. That process is independent of the image type. Just for fun, change the color type of the buffered image to TYPE_BYTE_GRAY. You don't need to change any other code—the color model of the image automatically takes care of the conversion from colors to sample values.

Listing 7-6. **RasterImageTest.java**

Code View:

```
import java.awt.*;
import java.awt.image.*;
import javax.swing.*;

/**
 * This program demonstrates how to build up an image from individual pixels.
 * @version 1.13 2007-08-16
 * @author Cay Horstmann
 */
public class RasterImageTest
{
   public static void main(String[] args)
   {
      EventQueue.invokeLater(new Runnable()
         {
            public void run()
            {
               JFrame frame = new RasterImageFrame();
               frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
               frame.setVisible(true);
            }
         });
   }
}

/**
 * This frame shows an image with a Mandelbrot set.
 */
class RasterImageFrame extends JFrame
{
   public RasterImageFrame()
   {
      setTitle("RasterImageTest");
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
      BufferedImage image = makeMandelbrot(DEFAULT_WIDTH, DEFAULT_HEIGHT);
      add(new JLabel(new ImageIcon(image)));
   }

   /**
    * Makes the Mandelbrot image.
    * @param width the width
    * @parah height the height
    * @return the image
```

```
    */
   public BufferedImage makeMandelbrot(int width, int height)
   {
      BufferedImage image = new BufferedImage(width, height, BufferedImage.TYPE_INT_ARGB);
      WritableRaster raster = image.getRaster();
      ColorModel model = image.getColorModel();

      Color fractalColor = Color.red;
      int argb = fractalColor.getRGB();
      Object colorData = model.getDataElements(argb, null);

      for (int i = 0; i < width; i++)
         for (int j = 0; j < height; j++)
         {
            double a = XMIN + i * (XMAX - XMIN) / width;
            double b = YMIN + j * (YMAX - YMIN) / height;
            if (!escapesToInfinity(a, b)) raster.setDataElements(i, j, colorData);
         }
      return image;
   }

   private boolean escapesToInfinity(double a, double b)
   {
      double x = 0.0;
      double y = 0.0;
      int iterations = 0;
      while (x <= 2 && y <= 2 && iterations < MAX_ITERATIONS)
      {
         double xnew = x * x - y * y + a;
         double ynew = 2 * x * y + b;
         x = xnew;
         y = ynew;
         iterations++;
      }
      return x > 2 || y > 2;
   }

   private static final double XMIN = -2;
   private static final double XMAX = 2;
   private static final double YMIN = -2;
   private static final double YMAX = 2;
   private static final int MAX_ITERATIONS = 16;
   private static final int DEFAULT_WIDTH = 400;
   private static final int DEFAULT_HEIGHT = 400;
}
```

API

java.awt.image.BufferedImage 1.2

- `BufferedImage(int width, int height, int imageType)`

  constructs a buffered image object.

Parameters:	`width, height`	The image dimensions
	`imageType`	The image type. The most common types are `TYPE_INT_RGB`, `TYPE_INT_ARGB`, `TYPE_BYTE_GRAY`, and `TYPE_BYTE_INDEXED`

- `ColorModel getColorModel()`

  returns the color model of this buffered image.

- `WritableRaster getRaster()`

  gets the raster for accessing and modifying pixels of this buffered image.

API

java.awt.image.Raster 1.2

- `Object getDataElements(int x, int y, Object data)`

  returns the sample data for a raster point, in an array whose element type and length depend on the color model. If `data` is not `null`, it is assumed to be an array that is appropriate for holding sample data and it is filled. If `data` is `null`, a new array is allocated. Its element type and length depend on the color model.

- `int[] getPixel(int x, int y, int[] sampleValues)`

- `float[] getPixel(int x, int y, float[] sampleValues)`

- `double[] getPixel(int x, int y, double[] sampleValues)`

- `int[] getPixels(int x, int y, int width, int height, int[] sampleValues)`

- `float[] getPixels(int x, int y, int width, int height,` float[] sampleValues)

- double[] getPixels(int x, int y, int width, int height, double[] sampleValues)

  returns the sample values for a raster point, or a rectangle of raster points, in an array whose length depends on the color model. If sampleValues is not null, it is assumed to be sufficiently long for holding the sample values and it is filled. If sampleValues is null, a new array is allocated. These methods are only useful if you know the meaning of the sample values for a color model.

UNREGISTERED VERSION OF CHM TO PDF CONVERTER PRO BY THETA-SOFTWARE

API java.awt.image.WritableRaster 1.2

- void setDataElements(int x, int y, Object data)

  sets the sample data for a raster point. data is an array filled with the sample data for a pixel. Its element type and length depend on the color model.

- void setPixel(int x, int y, int[] sampleValues)

- void setPixel(int x, int y, float[] sampleValues)

- void setPixel(int x, int y, double[] sampleValues)

- void setPixels(int x, int y, int width, int height, int[] sampleValues)

- void setPixels(int x, int y, int width, int height, float[] sampleValues)

- void setPixels(int x, int y, int width, int height, double[] sampleValues)

  sets the sample values for a raster point or a rectangle of raster points. These methods are only useful if you know the encoding of the sample values for a color model.

API

java.awt.image.ColorModel 1.2

- `int getRGB(Object data)`

  returns the ARGB value that corresponds to the sample data passed in the `data` array. Its element type and length depend on the color model.

- `Object getDataElements(int argb, Object data);`

  returns the sample data for a color value. If `data` is not `null`, it is assumed to be an array that is appropriate for holding sample data and it is filled. If `data` is `null`, a new array is allocated. `data` is an array filled with the sample data for a pixel. Its element type and length depend on the color model.

API

java.awt.Color 1.0

- `Color(int argb, boolean hasAlpha)` **1.2**

  creates a color with the specified combined ARGB value if `hasAlpha` is `true`, or the specified RGB value if `hasAlpha` is `false`.

- `int getRGB()`

  returns the ARGB color value corresponding to this color.

Filtering Images

In the preceding section, you saw how to build up an image from scratch. However, often you want to access image data for a different reason: You already have an image and you want to improve it in some way.

Of course, you can use the `getPixel/getDataElements` methods that you saw in the preceding section to read the image data, manipulate them, and then write them back. But fortunately, the Java 2D API already supplies a number of *filters* that carry out common image processing operations for you.

The image manipulations all implement the `BufferedImageOp` interface. After you construct the operation, you simply call the `filter` method to transform an image into another.

Code View:
```
BufferedImageOp op = . . .;
BufferedImage filteredImage
   = new BufferedImage(image.getWidth(), image.getHeight(), image.getType());
op.filter(image, filteredImage);
```

Some operations can transform an image in place (`op.filter(image, image)`), but most can't.

Five classes implement the `BufferedImageOp` interface:

```
AffineTransformOp
RescaleOp
LookupOp
ColorConvertOp
ConvolveOp
```

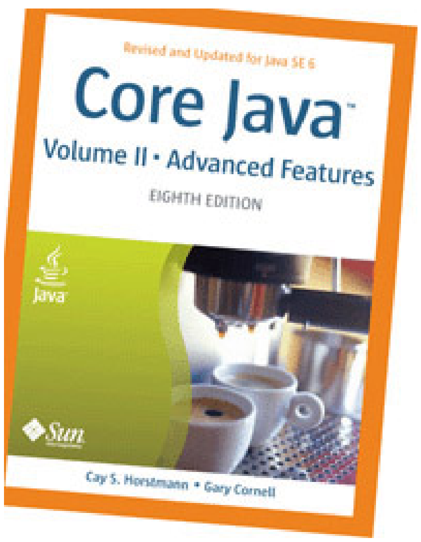

The `AffineTransformOp` carries out an affine transformation on the pixels. For example, here is how you can rotate an image about its center:

Code View:
```
AffineTransform transform = AffineTransform.getRotateInstance(Math.toRadians(angle),
   image.getWidth() / 2, image.getHeight() / 2);
AffineTransformOp op = new AffineTransformOp(transform, interpolation);
op.filter(image, filteredImage);
```

The `AffineTransformOp` constructor requires an affine transform and an *interpolation* strategy. Interpolation is necessary to determine pixels in the target image if the source pixels are transformed somewhere between target pixels. For example, if you rotate source pixels, then they will generally not fall exactly onto target pixels. There are two interpolation strategies: `AffineTransformOp.TYPE_BILINEAR` and `AffineTransformOp.TYPE_NEAREST_NEIGHBOR`. Bilinear interpolation takes a bit longer but looks better.

The program in Listing 7-7 lets you rotate an image by 5 degrees (see Figure 7-29).

Figure 7-29. A rotated image



The RescaleOp carries out a rescaling operation

$$x_{\text{new}} = a \cdot x + b$$

for each of the color components in the image. (Alpha components are not affected.) The effect of rescaling with $a > 1$ is to brighten the image. You construct the RescaleOp by specifying the scaling parameters and optional rendering hints. In Listing 7-7, we use:

```
float a = 1.1f;
float 20.0f;
RescaleOp op = new RescaleOp(a, b, null);
```

You can also supply separate scaling values for each color component—see the API notes.

The LookupOp operation lets you specify an arbitrary mapping of sample values. You supply a table that specifies how each value should be mapped. In the example program, we compute the *negative* of all colors, changing the color *c* to 255 - *c*.

The LookupOp constructor requires an object of type LookupTable and a map of optional hints. The LookupTable class is abstract, with two concrete subclasses: ByteLookupTable and ShortLookupTable. Because RGB color values are bytes, a ByteLookupTable should suffice. However, because of the bug described in http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=6183251, we will use a ShortLookupTable instead. Here is how we construct the LookupOp for the example program:

```
short negative[] = new short[256];
for (int i = 0; i < 256; i++) negative[i] = (short) (255 - i);
ShortLookupTable table = new ShortLookupTable(0, negative);
LookupOp op = new LookupOp(table, null);
```

The lookup is applied to each color component separately, but not to the alpha component. You can also supply different lookup tables for each color component—see the API notes.

Note

You cannot apply a LookupOp to an image with an indexed color model. (In those images, each sample value is an offset into a color palette.)

The ColorConvertOp is useful for color space conversions. We do not discuss it here.

The most powerful of the transformations is the ConvolveOp, which carries out a mathematical *convolution.* We do not want to get too deeply into the mathematical details of convolution, but the basic idea is simple. Consider, for example, the *blur filter* (see Figure 7-30).

Figure 7-30. Blurring an image

The blurring is achieved by replacement of each pixel with the *average* value from the pixel and its eight neighbors. Intuitively, it makes sense why this operation would blur out the picture. Mathematically, the averaging can be expressed as a convolution operation with the following *kernel:*

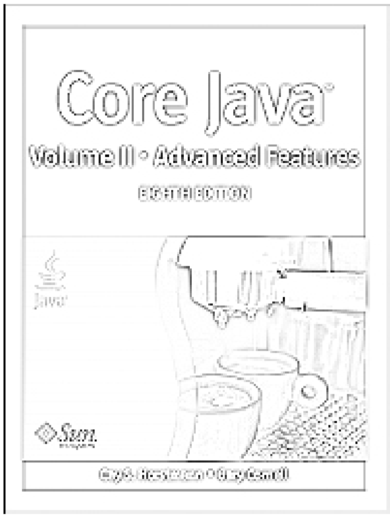
$$\begin{bmatrix} \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \\ \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \\ \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \end{bmatrix}$$

The kernel of a convolution is a matrix that tells what weights should be applied to the neighboring values. The kernel above leads to a blurred image. A different kernel carries out *edge detection,* locating areas of color changes:

$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

Edge detection is an important technique for analyzing photographic images (see Figure 7-31).

Figure 7-31. Edge detection and inversion



To construct a convolution operation, you first set up an array of the values for the kernel and construct a `Kernel` object. Then, construct a `ConvolveOp` object from the kernel and use it for filtering.

```
float[] elements =
   {
      0.0f, -1.0f, 0.0f,
      -1.0f,  4.f, -1.0f,
      0.0f, -1.0f, 0.0f
   };
Kernel kernel = new Kernel(3, 3, elements);
ConvolveOp op = new ConvolveOp(kernel);
op.filter(image, filteredImage);
```

The program in Listing 7-7 allows a user to load in a GIF or JPEG image and carry out the image manipulations that we discussed. Thanks to the power of the image operations that the Java 2D API provides, the program is very simple.

Listing 7-7. **ImageProcessingTest.java**

Code View:

```
import java.awt.*;
import java.awt.event.*;
import java.awt.geom.*;
import java.awt.image.*;
import java.io.*;
import javax.imageio.*;
import javax.swing.*;
import javax.swing.filechooser.*;

/**
 * This program demonstrates various image processing operations.
 * @version 1.03 2007-08-16
 * @author Cay Horstmann
 */
public class ImageProcessingTest
{
```

```java
   public static void main(String[] args)
   {
      EventQueue.invokeLater(new Runnable()
         {
            public void run()
            {
               JFrame frame = new ImageProcessingFrame();
               frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
               frame.setVisible(true);
            }
         });
   }
}

/**
 * This frame has a menu to load an image and to specify various transformations, and
 * a component to show the resulting image.
 */
class ImageProcessingFrame extends JFrame
{
   public ImageProcessingFrame()
   {
      setTitle("ImageProcessingTest");
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

      add(new JComponent()
         {
            public void paintComponent(Graphics g)
            {
               if (image != null) g.drawImage(image, 0, 0, null);
            }
         });

      JMenu fileMenu = new JMenu("File");
      JMenuItem openItem = new JMenuItem("Open");
      openItem.addActionListener(new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               openFile();
            }
         });
      fileMenu.add(openItem);

      JMenuItem exitItem = new JMenuItem("Exit");
      exitItem.addActionListener(new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               System.exit(0);
            }
         });
      fileMenu.add(exitItem);

      JMenu editMenu = new JMenu("Edit");
      JMenuItem blurItem = new JMenuItem("Blur");
      blurItem.addActionListener(new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
```

```
            {
               float weight = 1.0f / 9.0f;
               float[] elements = new float[9];
               for (int i = 0; i < 9; i++)
                  elements[i] = weight;
               convolve(elements);
            }
         });
      editMenu.add(blurItem);

      JMenuItem sharpenItem = new JMenuItem("Sharpen");
      sharpenItem.addActionListener(new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               float[] elements = { 0.0f, -1.0f, 0.0f, -1.0f, 5.f, -1.0f, 0.0f, -1.0f, 0.0f };
               convolve(elements);
            }
         });
      editMenu.add(sharpenItem);

      JMenuItem brightenItem = new JMenuItem("Brighten");
      brightenItem.addActionListener(new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               float a = 1.1f;
               // float b = 20.0f;
               float b = 0;
               RescaleOp op = new RescaleOp(a, b, null);
               filter(op);
            }
         });
      editMenu.add(brightenItem);

      JMenuItem edgeDetectItem = new JMenuItem("Edge detect");
      edgeDetectItem.addActionListener(new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               float[] elements = { 0.0f, -1.0f, 0.0f, -1.0f, 4.f, -1.0f, 0.0f, -1.0f, 0.0f };
               convolve(elements);
            }
         });
      editMenu.add(edgeDetectItem);

      JMenuItem negativeItem = new JMenuItem("Negative");
      negativeItem.addActionListener(new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               short[] negative = new short[256 * 1];
               for (int i = 0; i < 256; i++)
                  negative[i] = (short) (255 - i);
               ShortLookupTable table = new ShortLookupTable(0, negative);
               LookupOp op = new LookupOp(table, null);
               filter(op);
            }
         });
```

UNREGISTERED VERSION OF CHM TO PDF CONVERTER PRO BY THETA-SOFTWARE

```
      editMenu.add(negativeItem);

      JMenuItem rotateItem = new JMenuItem("Rotate");
      rotateItem.addActionListener(new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               if (image == null) return;
               AffineTransform transform = AffineTransform.getRotateInstance(
                     Math.toRadians(5), image.getWidth() / 2, image.getHeight() / 2);
               AffineTransformOp op = new AffineTransformOp(transform,
                     AffineTransformOp.TYPE_BICUBIC);
               filter(op);
            }
         });
      editMenu.add(rotateItem);

      JMenuBar menuBar = new JMenuBar();
      menuBar.add(fileMenu);
      menuBar.add(editMenu);
      setJMenuBar(menuBar);
   }

   /**
    * Open a file and load the image.
    */
   public void openFile()
   {
      JFileChooser chooser = new JFileChooser();
      chooser.setCurrentDirectory(new File("."));
      String[] extensions = ImageIO.getReaderFileSuffixes();
      chooser.setFileFilter(new FileNameExtensionFilter("Image files", extensions));
      int r = chooser.showOpenDialog(this);
      if (r != JFileChooser.APPROVE_OPTION) return;

      try
      {
         Image img = ImageIO.read(chooser.getSelectedFile());
         image = new BufferedImage(img.getWidth(null), img.getHeight(null),
               BufferedImage.TYPE_INT_RGB);
         image.getGraphics().drawImage(img, 0, 0, null);
      }
      catch (IOException e)
      {
         JOptionPane.showMessageDialog(this, e);
      }
      repaint();
   }

   /**
    * Apply a filter and repaint.
    * @param op the image operation to apply
    */
   private void filter(BufferedImageOp op)
   {
      if (image == null) return;
      image = op.filter(image, null);
      repaint();
   }
```

```

   /**
    * Apply a convolution and repaint.
    * @param elements the convolution kernel (an array of 9 matrix elements)
    */
   private void convolve(float[] elements)
   {
      Kernel kernel = new Kernel(3, 3, elements);
      ConvolveOp op = new ConvolveOp(kernel);
      filter(op);
   }

   private BufferedImage image;
   private static final int DEFAULT_WIDTH = 400;
   private static final int DEFAULT_HEIGHT = 400;
}
```

API

`java.awt.image.BufferedImageOp` 1.2

- `BufferedImage filter(BufferedImage source, BufferedImage dest)`

  applies the image operation to the source image and stores the result in the destination image. If `dest` is `null`, a new destination image is created. The destination image is returned.

API

`java.awt.image.AffineTransformOp` 1.2

- `AffineTransformOp(AffineTransform t, int interpolationType)`

  constructs an affine transform operator. The interpolation type is one of `TYPE_BILINEAR`, `TYPE_BICUBIC`, or `TYPE_NEAREST_NEIGHBOR`

API

`java.awt.image.RescaleOp` 1.2

- `RescaleOp(float a, float b, RenderingHints hints)`

- `RescaleOp(float[] as, float[] bs, RenderingHints hints)`

  constructs a rescale operator that carries out the scaling operation $x_{new} = a \cdot x + b$. When using the first constructor, all color components (but not the alpha component) are scaled with the same coefficients. When using the second constructor, you supply values for each color component, in which case the alpha component is unaffected, or values for both alpha and color components.

API

`java.awt.image.LookupOp` 1.2

- `LookupOp(LookupTable table, RenderingHints hints)`

  constructs a lookup operator for the given lookup table.

API

`java.awt.image.ByteLookupTable` 1.2

- `ByteLookupTable(int offset, byte[] data)`
- `ByteLookupTable(int offset, byte[][] data)`

  constructs a lookup table for converting `byte` values. The offset is subtracted from the input before the lookup. The values in the first constructor are applied to all color components but not the alpha component. When using the second constructor, you supply values for each color component, in which case the alpha component is unaffected, or values for both alpha and color components.

java.awt.image.ShortLookupTable 1.2

- `ShortLookupTable(int offset, short[] data)`
- `ShortLookupTable(int offset, short[][] data)`

  constructs a lookup table for converting `short` values. The offset is subtracted from the input before the lookup. The values in the first constructor are applied to all color components but not the alpha component. When using the second constructor, you supply values for each color component, in which case the alpha component is unaffected, or values for both alpha and color components.

java.awt.image.ConvolveOp 1.2

- `ConvolveOp(Kernel kernel)`
- `ConvolveOp(Kernel kernel, int edgeCondition, RenderingHints hints)`

  constructs a convolution operator. The edge condition specified is one of `EDGE_NO_OP` and `EDGE_ZERO_FILL`. Edge values need to be treated specially because they don't have sufficient neighboring values to compute the convolution. The default is `EDGE_ZERO_FILL`.

java.awt.image.Kernel 1.2

- `Kernel(int width, int height, float[] matrixElements)`

  constructs a kernel for the given matrix.

Printing

The original JDK had no support for printing at all. It was not possible to print from applets, and you had to get a t you wanted to print in an application. JDK 1.1 introduced very lightweight printing support, just enough to produce as long as you were not too particular about the print quality. The 1.1 printing model was designed to allow brows the surface of an applet as it appears on a web page (which, however, the browser vendors have not embraced).

Java SE 1.2 introduced the beginnings of a robust printing model that is fully integrated with 2D graphics. Java SE important enhancements, such as discovery of printer features and streaming print jobs for server-side print mana

In this section, we show you how you can easily print a drawing on a single sheet of paper, how you can manage a printout, and how you can benefit from the elegance of the Java 2D imaging model and easily generate a print pre

Note

The Java platform also supports the printing of user interface components. We do not cover this topi because it is mostly of interest to implementors of browsers, screen grabbers, and so on. For more information on printing components, see http://java.sun.com/developer/onlineTraining/Programming/JDCBook/render.html .

Graphics Printing

In this section, we tackle what is probably the most common printing situation: printing a 2D graphic. Of course, t contain text in various fonts or even consist entirely of text.

To generate a printout, you take care of these two tasks:

- Supply an object that implements the `Printable` interface.
- Start a print job.

The `Printable` interface has a single method:

```
int print(Graphics g, PageFormat format, int page)
```

That method is called whenever the print engine needs to have a page formatted for printing. Your code draws the that are to be printed onto the graphics context. The page format tells you the paper size and the print margins. T tells you which page to render.

To start a print job, you use the `PrinterJob` class. First, you call the static `getPrinterJob` method to get a print j the `Printable` object that you want to print.

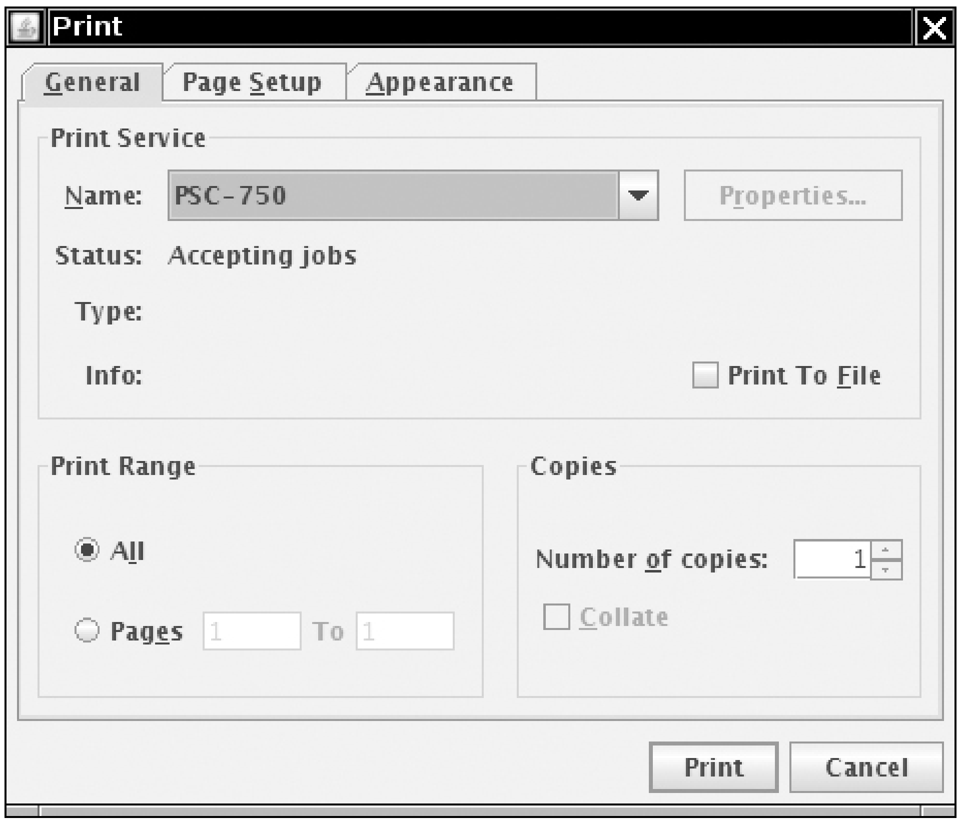
```
Printable canvas = . . .;
PrinterJob job = PrinterJob.getPrinterJob();
job.setPrintable(canvas);
```

Caution

The class PrintJob handles JDK 1.1-style printing. That class is now obsolete. Do not confuse it with the PrinterJob class.

Before starting the print job, you should call the printDialog method to display a print dialog box (see Figure 7-3 box gives the user a chance to select the printer to be used (in case multiple printers are available), the page rang printed, and various printer settings.

Figure 7-32. A cross-platform print dialog box



You collect printer settings in an object of a class that implements the PrintRequestAttributeSet interface, such HashPrintRequestAttributeSet class.

Code View:
```
HashPrintRequestAttributeSet attributes = new HashPrintRequestAttributeSet();
```

Add attribute settings and pass the `attributes` object to the `printDialog` method.

The `printDialog` method returns `true` if the user clicked OK and `false` if the user canceled the dialog box. If the the `print` method of the `PrinterJob` class to start the printing process. The `print` method might throw a `Printe` is the outline of the printing code:


```
if (job.printDialog(attributes))
{
   try
   {
      job.print(attributes);
   }
   catch (PrinterException exception)
   {
     . . .
   }
}
```


Note

Prior to JDK 1.4, the printing system used the native print and page setup dialog boxes of the host platform. To show a native print dialog box, call the `printDialog` method with no parameters. (Ther is no way to collect user settings in an attribute set.)

During printing, the `print` method of the `PrinterJob` class makes repeated calls to the `print` method of the `Prin` associated with the job.

Because the job does not know how many pages you want to print, it simply keeps calling the `print` method. As lo method returns the value `Printable.PAGE_EXISTS` , the print job keeps producing pages. When the `print` methoc `Printable.NO_SUCH_PAGE` , the print job stops.

Caution

The page numbers that the print job passes to the `print` method start with page 0.

Therefore, the print job doesn't have an accurate page count until after the printout is complete. For that reason, t can't display the correct page range and instead displays a page range of "Pages 1 to 1." You will see in the next s avoid this blemish by supplying a `Book` object to the print job.

During the printing process, the print job repeatedly calls the `print` method of the `Printable` object. The print jol make multiple calls *for the same page.* You should therefore not count pages inside the `print` method but always number parameter. There is a good reason why the print job might call the `print` method repeatedly for the same printers, in particular dot-matrix and inkjet printers, use *banding.* They print one band at a time, advance the pap the next band. The print job might use banding even for laser printers that print a full page at a time—it gives the managing the size of the spool file.

If the print job needs the `Printable` object to print a band, then it sets the clip area of the graphics context to the and calls the `print` method. Its drawing operations are clipped against the band rectangle, and only those drawing show up in the band are rendered. Your `print` method need not be aware of that process, with one caveat: It shou with the clip area.

Caution

The `Graphics` object that your `print` method gets is also clipped against the page margins. If you replace the clip area, you can draw outside the margins. Especially in a printer graphics context, the clipping area must be respected. Call `clip` , not `setClip` , to further restrict the clipping area. If you must remove a clip area, then make sure to call `getClip` at the beginning of your `print` method and restore that clip area.

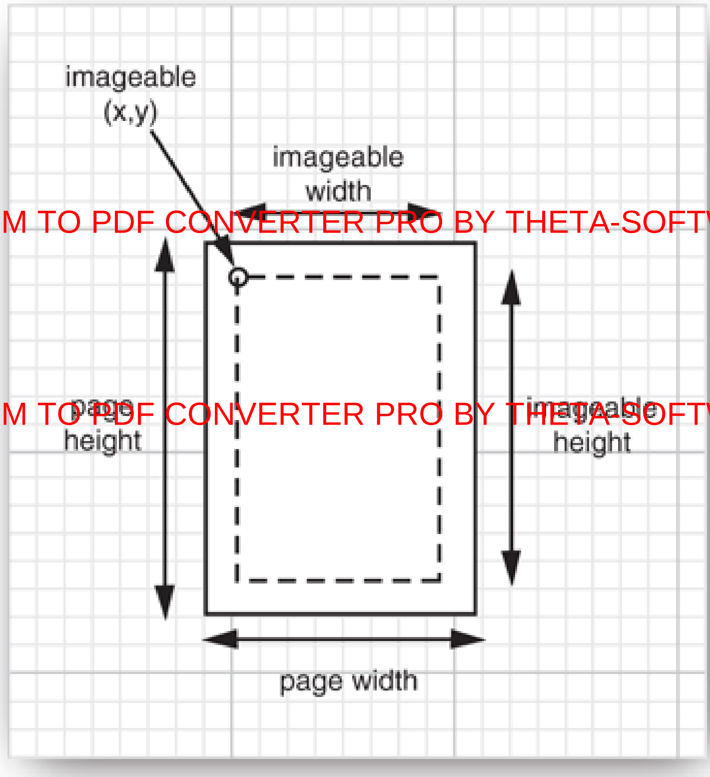
The `PageFormat` parameter of the `print` method contains information about the printed page. The methods `getWi` return the paper size, measured in *points.* One point is 1/72 of an inch. (An inch equals 25.4 millimeters.) For exa approximately 595 x 842 points, and U.S. letter-size paper is 612 x 792 points.

Points are a common measurement in the printing trade in the United States. Much to the chagrin of the rest of th printing package uses point units for two purposes. Paper sizes and paper margins are measured in points. And th print graphics contexts is one point. You can verify that in the example program at the end of this section. The pro lines of text that are 72 units apart. Run the example program and measure the distance between the baselines. T inch or 25.4 millimeters apart.

The `getWidth` and `getHeight` methods of the `PageFormat` class give you the complete paper size. Not all of the pa printable. Users typically select margins, and even if they don't, printers need to somehow grip the sheets of pape print and therefore have a small unprintable area around the edges.

The methods `getImageableWidth` and `getImageableHeight` tell you the dimensions of the area that you can actua the margins need not be symmetrical, so you must also know the top-left corner of the imageable area (see Figure obtain by the methods `getImageableX` and `getImageableY` .

Figure 7-33. Page format measurements



Tip

The graphics context that you receive in the `print` method is clipped to exclude the margins, but the origin of the coordinate system is nevertheless the top-left corner of the paper. It makes sense to translate the coordinate system to start at the top-left corner of the imageable area. Simply start yo `print` method with

```
g.translate(pageFormat.getImageableX(), pageFormat.getImageableY());
```

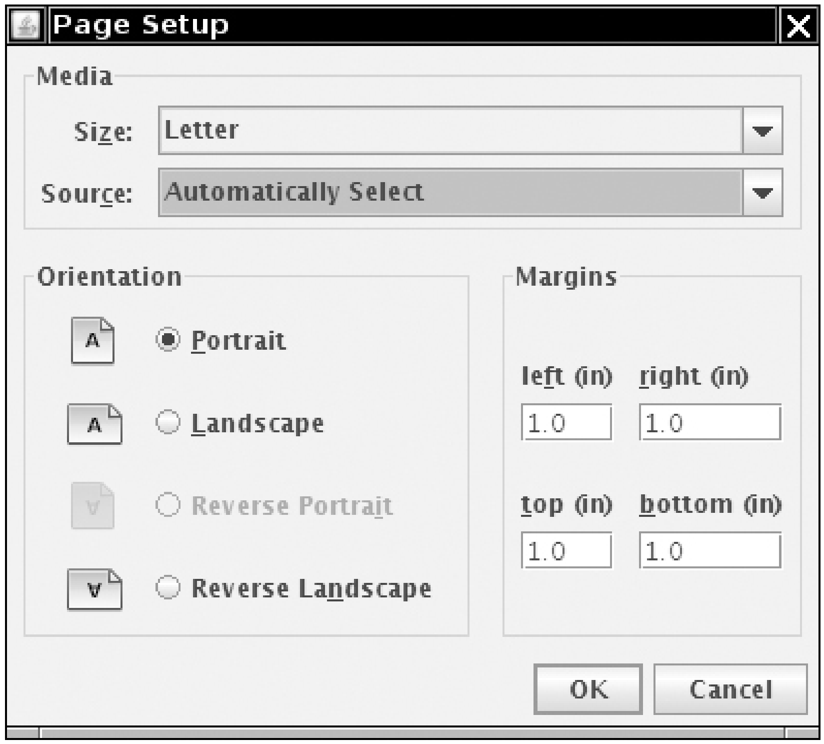
If you want your users to choose the settings for the page margins or to switch between portrait and landscape or setting other printing attributes, you can call the `pageDialog` method of the `PrinterJob` class:

```
PageFormat format = job.pageDialog(attributes);
```

Note

One of the tabs of the print dialog box contains the page setup dialog box (see Figure 7-34). You might still want to give users an option to set the page format before printing, especially if your program presents a "what you see is what you get" display of the pages to be printed. The `pageDial` method returns a `PageFormat` object with the user settings.

Figure 7-34. A cross-platform page setup dialog box



Listing 7-8 shows how to render the same set of shapes on the screen and on the printed page. A subclass of `JPan` `Printable` interface. Both the `paintComponent` and the `print` methods call the same method to carry out the actu

Code View:
```
class PrintPanel extends JPanel implements Printable
{
   public void paintComponent(Graphics g)
   {
      super.paintComponent(g);
      Graphics2D g2 = (Graphics2D) g;
      drawPage(g2);
   }

   public int print(Graphics g, PageFormat pf, int page)
      throws PrinterException
```

```
   {
      if (page >= 1) return Printable.NO_SUCH_PAGE;
      Graphics2D g2 = (Graphics2D) g;
      g2.translate(pf.getImageableX(), pf.getImageableY());
      drawPage(g2);
      return Printable.PAGE_EXISTS;
   }

   public void drawPage(Graphics2D g2)
   {
      // shared drawing code goes here
      . . .
   }
   . . .
}
```


This example displays and prints the image shown in Figure 7-20 on page 558 , namely, the outline of the messag that is used as a clipping area for a pattern of lines.

Click the Print button to start printing, or click the Page setup button to open the page setup dialog box. Listing 7-

Note

To show a native page setup dialog box, you pass a default `PageFormat` object to the `pageDialog` method. The method clones that object, modifies it according to the user selections in the dialog box and returns the cloned object.

```
PageFormat defaultFormat = printJob.defaultPage();
PageFormat selectedFormat = printJob.pageDialog(defaultFormat);
```

Listing 7-8. **PrintTest.java**

Code View:

```
import java.awt.*;
import java.awt.event.*;
import java.awt.font.*;
import java.awt.geom.*;
import java.awt.print.*;
import javax.print.attribute.*;
import javax.swing.*;

/**
 * This program demonstrates how to print 2D graphics
 * @version 1.12 2007-08-16
 * @author Cay Horstmann
 */
public class PrintTest
{
   public static void main(String[] args)
```

```
   {
      EventQueue.invokeLater(new Runnable()
         {
            public void run()
            {
               JFrame frame = new PrintTestFrame();
               frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
               frame.setVisible(true);
            }
         });
   }
}

/**
 * This frame shows a panel with 2D graphics and buttons to print the graphics and to
 * set up the page format.
 */
class PrintTestFrame extends JFrame
{
   public PrintTestFrame()
   {
      setTitle("PrintTest");
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

      canvas = new PrintComponent();
      add(canvas, BorderLayout.CENTER);

      attributes = new HashPrintRequestAttributeSet();

      JPanel buttonPanel = new JPanel();
      JButton printButton = new JButton("Print");
      buttonPanel.add(printButton);
      printButton.addActionListener(new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               try
               {
                  PrinterJob job = PrinterJob.getPrinterJob();
                  job.setPrintable(canvas);
                  if (job.printDialog(attributes)) job.print(attributes);
               }
               catch (PrinterException e)
               {
                  JOptionPane.showMessageDialog(PrintTestFrame.this, e);
               }
            }
         });

      JButton pageSetupButton = new JButton("Page setup");
      buttonPanel.add(pageSetupButton);
      pageSetupButton.addActionListener(new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               PrinterJob job = PrinterJob.getPrinterJob();
               job.pageDialog(attributes);
            }
         });
```

```

      add(buttonPanel, BorderLayout.NORTH);
   }

   private PrintComponent canvas;
   private PrintRequestAttributeSet attributes;

   private static final int DEFAULT_WIDTH = 300;
   private static final int DEFAULT_HEIGHT = 300;
```



```
}

/**
 * This component generates a 2D graphics image for screen display and printing.
 */
class PrintComponent extends JComponent implements Printable
```



```
{
   public void paintComponent(Graphics g)
   {
      Graphics2D g2 = (Graphics2D) g;
      drawPage(g2);
   }

   public int print(Graphics g, PageFormat pf, int page) throws PrinterException
   {
      if (page >= 1) return Printable.NO_SUCH_PAGE;
      Graphics2D g2 = (Graphics2D) g;
      g2.translate(pf.getImageableX(), pf.getImageableY());
      g2.draw(new Rectangle2D.Double(0, 0, pf.getImageableWidth(), pf.getImageableHeight()));

      drawPage(g2);
      return Printable.PAGE_EXISTS;
   }

   /**
    * This method draws the page both on the screen and the printer graphics context.
    * @param g2 the graphics context
    */
   public void drawPage(Graphics2D g2)
   {
      FontRenderContext context = g2.getFontRenderContext();
      Font f = new Font("Serif", Font.PLAIN, 72);
      GeneralPath clipShape = new GeneralPath();

      TextLayout layout = new TextLayout("Hello", f, context);
      AffineTransform transform = AffineTransform.getTranslateInstance(0, 72);
      Shape outline = layout.getOutline(transform);
      clipShape.append(outline, false);

      layout = new TextLayout("World", f, context);
      transform = AffineTransform.getTranslateInstance(0, 144);
      outline = layout.getOutline(transform);
      clipShape.append(outline, false);

      g2.draw(clipShape);
      g2.clip(clipShape);

      final int NLINES = 50;
      Point2D p = new Point2D.Double(0, 0);
      for (int i = 0; i < NLINES; i++)
```

```
        {
           double x = (2 * getWidth() * i) / NLINES;
           double y = (2 * getHeight() * (NLINES - 1 - i)) / NLINES;
           Point2D q = new Point2D.Double(x, y);
           g2.draw(new Line2D.Double(p, q));
        }
    }
}
```

API

`java.awt.print.Printable` 1.2

- `int print(Graphics g, PageFormat format, int pageNumber)`

  renders a page and returns `PAGE_EXISTS` , or returns `NO_SUCH_PAGE` .

Parameters:	`g`	The graphics context onto which the page is rendered
	`format`	The format of the page to draw on
	`pageNumber`	The number of the requested page

API

`java.awt.print.PrinterJob` 1.2

- `static PrinterJob getPrinterJob()`

  returns a printer job object.

- `PageFormat defaultPage()`

  returns the default page format for this printer.

- `boolean printDialog(PrintRequestAttributeSet attributes)`

- `boolean printDialog()`

  opens a print dialog box to allow a user to select the pages to be printed and to change print settings. The first method displays a cross-platform dialog box, the second a native dialog box. The first method modifies the `attributes` object to reflect the user settings. Both methods return `true` if the user accepts the dialog box.

- `PageFormat pageDialog(PrintRequestAttributeSet attributes)`
- `PageFormat pageDialog(PageFormat defaults)`

  displays a page setup dialog box. The first method displays a cross-platform dialog box, the second a native dialog box. Both methods return a `PageFormat` object with the format that the user requested in the dialog box. The first method modifies the `attributes` object to reflect the user settings. The second method does not modify the `defaults` object.

- `void setPrintable(Printable p)`

- `void setPrintable(Printable p, PageFormat format)`

  sets the `Printable` of this print job and an optional page format.

- `void print()`
- `void print(PrintRequestAttributeSet attributes)`

  prints the current `Printable` by repeatedly calling its `print` method and sending the rendered pages to the printer, until no more pages are available.

API `java.awt.print.PageFormat` 1.2

- `double getWidth()`
- `double getHeight()`

  returns the width and height of the page.

- `double getImageableWidth()`
- `double getImageableHeight()`

  returns the width and height of the imageable area of the page.

- `double getImageableX()`

- `double getImageableY()`

  returns the position of the top-left corner of the imageable area.

- `int getOrientation()`

  returns one of `PORTRAIT` , `LANDSCAPE` , or `REVERSE_LANDSCAPE` . Page orientation is transparent to programmers because the page format and graphics context settings automatically reflect the page orientation.

Multiple-Page Printing

In practice, you usually shouldn't pass a raw `Printable` object to a print job. Instead, you should obtain an object implements the `Pageable` interface. The Java platform supplies one such class, called `Book` . A book is made up of which is a `Printable` object. You make a book by adding `Printable` objects and their page counts.

```
Book book = new Book();
Printable coverPage = . . .;
Printable bodyPages = . . .;
book.append(coverPage, pageFormat); // append 1 page
book.append(bodyPages, pageFormat, pageCount);
```

Then, you use the `setPageable` method to pass the `Book` object to the print job.

```
printJob.setPageable(book);
```

Now the print job knows exactly how many pages to print. Then, the print dialog box displays an accurate page ra can select the entire range or subranges.

Caution

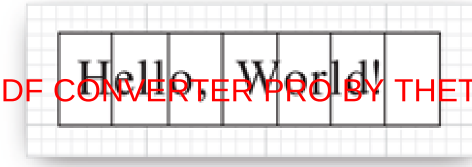
When the print job calls the `print` methods of the `Printable` sections, it passes the current page number of the *book* , and not of each *section* , as the current page number. That is a huge pain—eac section must know the page counts of the preceding sections to make sense of the page number parameter.

From your perspective as a programmer, the biggest challenge about using the `Book` class is that you must know I each section will have when you print it. Your `Printable` class needs a *layout algorithm* that computes the layout the printed pages. Before printing starts, invoke that algorithm to compute the page breaks and the page count. Y layout information so you have it handy during the printing process.

You must guard against the possibility that the user has changed the page format. If that happens, you must reco even if the information that you want to print has not changed.

Listing 7-9 shows how to produce a multipage printout. This program prints a message in very large characters on (see Figure 7-35). You can then trim the margins and tape the pages together to form a banner.

Figure 7-35. A banner



The `layoutPages` method of the `Banner` class computes the layout. We first lay out the message string in a 72-poi compute the height of the resulting string and compare it with the imageable height of the page. We derive a scale two measurements. When printing the string, we magnify it by that scale factor.

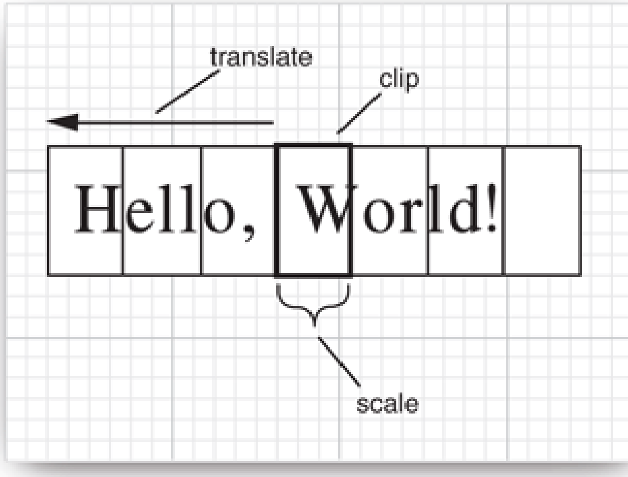
Caution

To lay out your information precisely, you usually need access to the printer graphics context. Unfortunately, there is no way to obtain that graphics context until printing actually starts. In our example program, we make do with the screen graphics context and hope that the font metrics of th screen and printer match.

The `getPageCount` method of the `Banner` class first calls the layout method. Then it scales up the width of the strin the imageable width of each page. The quotient, rounded up to the next integer, is the page count.

It sounds like it might be difficult to print the banner because characters can be broken across multiple pages. How the power of the Java 2D API, this turns out not to be a problem at all. When a particular page is requested, we si `translate` method of the `Graphics2D` class to shift the top-left corner of the string to the left. Then, we set a clip equals the current page (see Figure 7-36). Finally, we scale the graphics context with the scale factor that the lay computed.

Figure 7-36. Printing a page of a banner

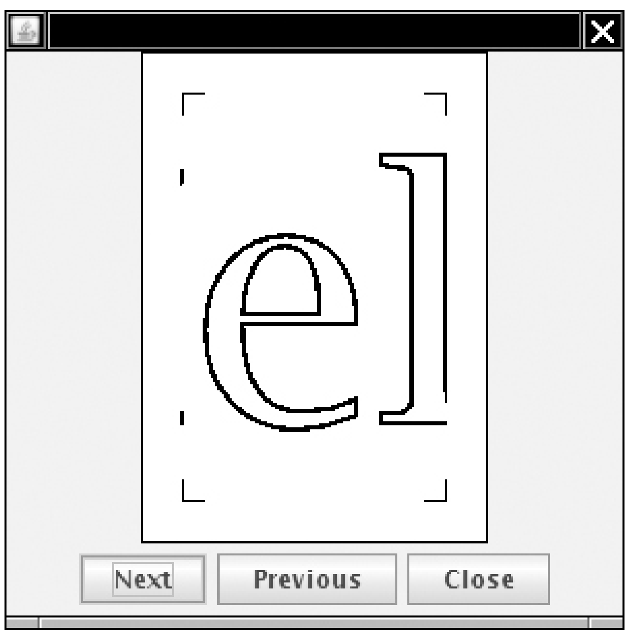


This example shows the power of transformations. The drawing code is kept simple, and the transformation does a
placing the drawing at the appropriate place. Finally, the clip cuts away the part of the image that falls outside the
section, you will see another compelling use of transformations, to display a print preview.

Print Preview

Most professional programs have a print preview mechanism that lets you look at your pages on the screen so tha
paper on a printout that you don't like. The printing classes of the Java platform do not supply a standard "print pr
but it is easy to design your own (see Figure 7-37). In this section, we show you how. The `PrintPreviewDialog` c
is completely generic—you can reuse it to preview any kind of printout.

Figure 7-37. The print preview dialog, showing a banner page



To construct a `PrintPreviewDialog` , you supply either a `Printable` or a `Book` , together with a `PageFormat` objec
the dialog box contains a `PrintPreviewCanvas` . As you use the Next and Previous buttons to flip through the pag
`paintComponent` method calls the `print` method of the `Printable` object for the requested page.

Normally, the `print` method draws the page context on a printer graphics context. However, we supply the screen
suitably scaled so that the entire printed page fits inside a small screen rectangle.


```
float xoff = . . .; // left of page
float yoff = . . .; // top of page
float scale = . . .; // to fit printed page onto screen
g2.translate(xoff, yoff);
g2.scale(scale, scale);
Printable printable = book.getPrintable(currentPage);
printable.print(g2, pageFormat, currentPage);
```


The `print` method never knows that it doesn't actually produce printed pages. It simply draws onto the graphics c
producing a microscopic print preview on the screen. This is a compelling demonstration of the power of the Java 2

Listing 7-9 contains the code for the banner printing program and the print preview dialog box. Type "Hello, World
field and look at the print preview, then print the banner.

Listing 7-9. **BookTest.java**

Code View:

```
import java.awt.*;
import java.awt.event.*;
import java.awt.font.*;
import java.awt.geom.*;
import java.awt.print.*;
import javax.print.attribute.*;
import javax.swing.*;

/**
 * This program demonstrates the printing of a multipage book. It prints a "banner", by
 * blowing up a text string to fill the entire page vertically. The program also contains a
 * generic print preview dialog.
 * @version 1.12 2007-08-16
 * @author Cay Horstmann
 */
public class BookTest
{
   public static void main(String[] args)
   {
      EventQueue.invokeLater(new Runnable()
         {
            public void run()
            {
               JFrame frame = new BookTestFrame();
               frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
               frame.setVisible(true);
            }
         });
   }
}

```

```
/**
 * This frame has a text field for the banner text and buttons for printing, page setup,
 * and print preview.
 */
class BookTestFrame extends JFrame
{
   public BookTestFrame()
   {
      setTitle("BookTest");

      text = new JTextField();
      add(text, BorderLayout.NORTH);

      attributes = new HashPrintRequestAttributeSet();

      JPanel buttonPanel = new JPanel();

      JButton printButton = new JButton("Print");
      buttonPanel.add(printButton);
      printButton.addActionListener(new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               try
               {
                  PrinterJob job = PrinterJob.getPrinterJob();
                  job.setPageable(makeBook());
                  if (job.printDialog(attributes))
                  {
                    job.print(attributes);
                  }
               }
               catch (PrinterException e)
               {
                  JOptionPane.showMessageDialog(BookTestFrame.this, e);
               }
            }
         });

      JButton pageSetupButton = new JButton("Page setup");
      buttonPanel.add(pageSetupButton);
      pageSetupButton.addActionListener(new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               PrinterJob job = PrinterJob.getPrinterJob();
               pageFormat = job.pageDialog(attributes);
            }
         });

      JButton printPreviewButton = new JButton("Print preview");
      buttonPanel.add(printPreviewButton);
      printPreviewButton.addActionListener(new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               PrintPreviewDialog dialog = new PrintPreviewDialog(makeBook());
               dialog.setVisible(true);
            }
```

```
         });

      add(buttonPanel, BorderLayout.SOUTH);
      pack();
   }

   /**
    * Makes a book that contains a cover page and the pages for the banner.
    */
```

```
   public Book makeBook()
   {
      if (pageFormat == null)
      {
         PrinterJob job = PrinterJob.getPrinterJob();
         pageFormat = job.defaultPage();
      }
```

```
      Book book = new Book();
      String message = text.getText();
      Banner banner = new Banner(message);
      int pageCount = banner.getPageCount((Graphics2D) getGraphics(), pageFormat);
      book.append(new CoverPage(message + " (" + pageCount + " pages)"), pageFormat);
      book.append(banner, pageFormat, pageCount);
      return book;
   }

   private JTextField text;
   private PageFormat pageFormat;
   private PrintRequestAttributeSet attributes;
}

/**
 * A banner that prints a text string on multiple pages.
 */
class Banner implements Printable
{
   /**
    * Constructs a banner
    * @param m the message string
    */
   public Banner(String m)
   {
      message = m;
   }

   /**
    * Gets the page count of this section.
    * @param g2 the graphics context
    * @param pf the page format
    * @return the number of pages needed
    */
   public int getPageCount(Graphics2D g2, PageFormat pf)
   {
      if (message.equals("")) return 0;
      FontRenderContext context = g2.getFontRenderContext();
      Font f = new Font("Serif", Font.PLAIN, 72);
      Rectangle2D bounds = f.getStringBounds(message, context);
      scale = pf.getImageableHeight() / bounds.getHeight();
      double width = scale * bounds.getWidth();
      int pages = (int) Math.ceil(width / pf.getImageableWidth());
```

```
      return pages;
   }

   public int print(Graphics g, PageFormat pf, int page) throws PrinterException
   {
      Graphics2D g2 = (Graphics2D) g;
      if (page > getPageCount(g2, pf)) return Printable.NO_SUCH_PAGE;
      g2.translate(pf.getImageableX(), pf.getImageableY());

      drawPage(g2, pf, page);
      return Printable.PAGE_EXISTS;
   }

   public void drawPage(Graphics2D g2, PageFormat pf, int page)
   {
      if (message.equals("")) return;
      page--; // account for cover page

      drawCropMarks(g2, pf);
      g2.clip(new Rectangle2D.Double(0, 0, pf.getImageableWidth(), pf.getImageableHeight()));
      g2.translate(-page * pf.getImageableWidth(), 0);
      g2.scale(scale, scale);
      FontRenderContext context = g2.getFontRenderContext();
      Font f = new Font("Serif", Font.PLAIN, 72);
      TextLayout layout = new TextLayout(message, f, context);
      AffineTransform transform = AffineTransform.getTranslateInstance(0, layout.getAscent());
      Shape outline = layout.getOutline(transform);
      g2.draw(outline);
   }

   /**
    * Draws 1/2" crop marks in the corners of the page.
    * @param g2 the graphics context
    * @param pf the page format
    */
   public void drawCropMarks(Graphics2D g2, PageFormat pf)
   {
      final double C = 36; // crop mark length = 1/2 inch
      double w = pf.getImageableWidth();
      double h = pf.getImageableHeight();
      g2.draw(new Line2D.Double(0, 0, 0, C));
      g2.draw(new Line2D.Double(0, 0, C, 0));
      g2.draw(new Line2D.Double(w, 0, w, C));
      g2.draw(new Line2D.Double(w, 0, w - C, 0));
      g2.draw(new Line2D.Double(0, h, 0, h - C));
      g2.draw(new Line2D.Double(0, h, C, h));
      g2.draw(new Line2D.Double(w, h, w, h - C));
      g2.draw(new Line2D.Double(w, h, w - C, h));
   }

   private String message;
   private double scale;
}

/**
 * This class prints a cover page with a title.
 */
class CoverPage implements Printable
{
```

```
   /**
    * Constructs a cover page.
    * @param t the title
    */
   public CoverPage(String t)
   {
      title = t;
   }

```

```
   public int print(Graphics g, PageFormat pf, int page) throws PrinterException
   {
      if (page >= 1) return Printable.NO_SUCH_PAGE;
      Graphics2D g2 = (Graphics2D) g;
      g2.setPaint(Color.black);
      g2.translate(pf.getImageableX(), pf.getImageableY());
```

```
      FontRenderContext context = g2.getFontRenderContext();
      Font f = g2.getFont();
      TextLayout layout = new TextLayout(title, f, context);
      float ascent = layout.getAscent();
      g2.drawString(title, 0, ascent);
      return Printable.PAGE_EXISTS;
   }

   private String title;
}

/**
 * This class implements a generic print preview dialog.
 */
class PrintPreviewDialog extends JDialog
{
   /**
    * Constructs a print preview dialog.
    * @param p a Printable
    * @param pf the page format
    * @param pages the number of pages in p
    */
   public PrintPreviewDialog(Printable p, PageFormat pf, int pages)
   {
      Book book = new Book();
      book.append(p, pf, pages);
      layoutUI(book);
   }

   /**
    * Constructs a print preview dialog.
    * @param b a Book
    */
   public PrintPreviewDialog(Book b)
   {
      layoutUI(b);
   }

   /**
    * Lays out the UI of the dialog.
    * @param book the book to be previewed
    */
   public void layoutUI(Book book)
   {
```

```java
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

      canvas = new PrintPreviewCanvas(book);
      add(canvas, BorderLayout.CENTER);

      JPanel buttonPanel = new JPanel();

      JButton nextButton = new JButton("Next");
      buttonPanel.add(nextButton);
      nextButton.addActionListener(new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               canvas.flipPage(1);
            }
         });

      JButton previousButton = new JButton("Previous");
      buttonPanel.add(previousButton);
      previousButton.addActionListener(new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               canvas.flipPage(-1);
            }
         });

      JButton closeButton = new JButton("Close");
      buttonPanel.add(closeButton);
      closeButton.addActionListener(new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               setVisible(false);
            }
         });

      add(buttonPanel, BorderLayout.SOUTH);
   }

   private PrintPreviewCanvas canvas;

   private static final int DEFAULT_WIDTH = 300;
   private static final int DEFAULT_HEIGHT = 300;
}

/**
 * The canvas for displaying the print preview.
 */
class PrintPreviewCanvas extends JComponent
{
   /**
    * Constructs a print preview canvas.
    * @param b the book to be previewed
    */
   public PrintPreviewCanvas(Book b)
   {
      book = b;
      currentPage = 0;
```

```
   }

   public void paintComponent(Graphics g)
   {
      Graphics2D g2 = (Graphics2D) g;
      PageFormat pageFormat = book.getPageFormat(currentPage);

      double xoff; // x offset of page start in window
```



```
      double yoff; // y offset of page start in window
      double scale; // scale factor to fit page in window
      double px = pageFormat.getWidth();
      double py = pageFormat.getHeight();
      double sx = getWidth() - 1;
      double sy = getHeight() - 1;
      if (px / py < sx / sy) // center horizontally
```



```
      {
         scale = sy / py;
         xoff = 0.5 * (sx - scale * px);
         yoff = 0;
      }
      else
      // center vertically
      {
         scale = sx / px;
         xoff = 0;
         yoff = 0.5 * (sy - scale * py);
      }
      g2.translate((float) xoff, (float) yoff);
      g2.scale((float) scale, (float) scale);

      // draw page outline (ignoring margins)
      Rectangle2D page = new Rectangle2D.Double(0, 0, px, py);
      g2.setPaint(Color.white);
      g2.fill(page);
      g2.setPaint(Color.black);
      g2.draw(page);

      Printable printable = book.getPrintable(currentPage);
      try
      {
         printable.print(g2, pageFormat, currentPage);
      }
      catch (PrinterException e)
      {
         g2.draw(new Line2D.Double(0, 0, px, py));
         g2.draw(new Line2D.Double(px, 0, 0, py));
      }
   }

   /**
    * Flip the book by the given number of pages.
    * @param by the number of pages to flip by. Negative values flip backwards.
    */
   public void flipPage(int by)
   {
      int newPage = currentPage + by;
      if (0 <= newPage && newPage < book.getNumberOfPages())
      {
         currentPage = newPage;
```

```
            repaint();
        }
    }

    private Book book;
    private int currentPage;
}
```

java.awt.print.PrinterJob 1.2

- `void setPageable(Pageable p)`

  sets a `Pageable` (such as a `Book`) to be printed.

java.awt.print.Book 1.2

- `void append(Printable p, PageFormat format)`
- `void append(Printable p, PageFormat format, int pageCount)`

  appends a section to this book. If the page count is not specified, the first page is added.

- `Printable getPrintable(int page)`

  gets the printable for the specified page.

Print Services

So far, you have seen how to print 2D graphics. However, the printing API introduced in Java SE 1.4 affords far gr
The API defines a number of data types and lets you find print services that are able to print them. Among the dat

- Images in GIF, JPEG, or PNG format.
- Documents in text, HTML, PostScript, or PDF format.
- Raw printer code data.

- Objects of a class that implements `Printable` , `Pageable` , or `RenderableImage` .

The data themselves can be stored in a source of bytes or characters such as an input stream, a URL, or an array. describes the combination of a data source and a data type. The `DocFlavor` class defines a number of inner classe data sources. Each of the inner classes defines constants to specify the flavors. For example, the constant


```
DocFlavor.INPUT_STREAM.GIF
```

describes a GIF image that is read from an input stream. Table 7-3 lists the combinations.

Table 7-3. Document Flavors for Print Services

Data Source	Data Type	MIME Type
`INPUT_STREAM`	`GIF`	`image/gif`
`URL`	`JPEG`	`image/jpeg`
`BYTE_ARRAY`	`PNG`	`image/png`
	`POSTSCRIPT`	`application/postscript`
	`PDF`	`application/pdf`
	`TEXT_HTML_HOST`	`text/html` (using host encoding)
	`TEXT_HTML_US_ASCII`	`text/html; charset=us-ascii`
	`TEXT_HTML_UTF_8`	`text/html; charset=utf-8`
	`TEXT_HTML_UTF_16`	`text/html; charset=utf-16`
	`TEXT_HTML_UTF_16LE`	`text/html; charset=utf-16le` (little-endian)
	`TEXT_HTML_UTF_16BE`	`text/html; charset=utf-16be` (big-endian)
	`TEXT_PLAIN_HOST`	`text/plain` (using host encoding)
	`TEXT_PLAIN_US_ASCII`	`text/plain; charset=us-ascii`
	`TEXT_PLAIN_UTF_8`	`text/plain; charset=utf-8`
	`TEXT_PLAIN_UTF_16`	`text/plain; charset=utf-16`
	`TEXT_PLAIN_UTF_16LE`	`text/plain; charset=utf-16le` (little-endian)
	`TEXT_PLAIN_UTF_16BE`	`text/plain; charset=utf-16be` (big-endian)
	`PCL`	`application/vnd.hp-PCL` (Hewlett Packard Printer Control Language)
	`AUTOSENSE`	`application/octet-stream` (raw printer data)

Data Source	Data Type	MIME Type
`READER`	`TEXT_HTML`	`text/html; charset=utf-16`
`STRING`	`TEXT_PLAIN`	`text/plain; charset=utf-16`
`CHAR_ARRAY`		
`SERVICE_FORMATTED`	`PRINTABLE`	N/A
	`PAGEABLE`	N/A
	`RENDERABLE_IMAGE`	N/A

Suppose you want to print a GIF image that is located in a file. First find out whether there is a *print service* that is handling the task. The static `lookupPrintServices` method of the `PrintServiceLookup` class returns an array of objects that can handle the given document flavor.

```
DocFlavor flavor = DocFlavor.INPUT_STREAM.GIF;
PrintService[] services
   = PrintServiceLookup.lookupPrintServices(flavor, null);
```

The second parameter of the `lookupPrintServices` method is `null` to indicate that we don't want to constrain the specifying printer attributes. We cover attributes in the next section.

Note

Java SE 6 supplies print services for basic document flavors such as images and 2D graphics, but if y try to print text or HTML documents, the lookup will return an empty array.

If the lookup yields an array with more than one element, you select from the listed print services. You can call the of the `PrintService` class to get the printer names, and then let the user choose.

Next, get a document print job from the service:

```
DocPrintJob job = services[i].createPrintJob();
```

For printing, you need an object that implements the `Doc` interface. The Java library supplies a class `SimpleDoc` fo `SimpleDoc` constructor requires the data source object, the document flavor, and an optional attribute set. For exa

```
InputStream in = new FileInputStream(fileName);
Doc doc = new SimpleDoc(in, flavor, null);
```

Finally, you are ready to print:

```
job.print(doc, null);
```

As before, the `null` parameter can be replaced by an attribute set.

Note that this printing process is quite different from that of the preceding section. There is no user interaction thr boxes. For example, you can implement a server-side printing mechanism in which users submit print jobs througl

The program in Listing 7-10 demonstrates how to use a print service to print an image file.

Listing 7-10. **PrintServiceTest.java**

Code View:

```
import java.io.*;
import javax.print.*;

/**
 * This program demonstrates the use of print services. The program lets you print a GIF
 * image to any of the print services that support the GIF document flavor.
 * @version 1.10 2007-08-16
 * @author Cay Horstmann
 */
public class PrintServiceTest
{
   public static void main(String[] args)
   {
      DocFlavor flavor = DocFlavor.URL.GIF;
      PrintService[] services = PrintServiceLookup.lookupPrintServices(flavor, null);
      if (args.length == 0)
      {
         if (services.length == 0) System.out.println("No printer for flavor " + flavor);
         else
         {
            System.out.println("Specify a file of flavor " + flavor
                  + "\nand optionally the number of the desired printer.");
            for (int i = 0; i < services.length; i++)
               System.out.println((i + 1) + ": " + services[i].getName());
         }
         System.exit(0);
      }
      String fileName = args[0];
      int p = 1;
      if (args.length > 1) p = Integer.parseInt(args[1]);
      try
      {
         if (fileName == null) return;
         FileInputStream in = new FileInputStream(fileName);
         Doc doc = new SimpleDoc(in, flavor, null);
         DocPrintJob job = services[p - 1].createPrintJob();
         job.print(doc, null);
      }
      catch (FileNotFoundException e)
      {
         e.printStackTrace();
      }
      catch (PrintException e)
      {
         e.printStackTrace();
      }
   }
}
```

javax.print.PrintServiceLookup 1.4

- `PrintService[] lookupPrintServices(DocFlavor flavor, AttributeSet attributes)`

  looks up the print services that can handle the given document flavor and attributes.

Parameters:	`flavor`	The document flavor
	`attributes`	The required printing attributes, or `null` if attributes should not be considered

javax.print.PrintService 1.4

- `DocPrintJob createPrintJob()`

  creates a print job for printing an object of a class that implements the `Doc` interface, such as a `SimpleDoc` .

javax.print.DocPrintJob 1.4

- `void print(Doc doc, PrintRequestAttributeSet attributes)`

  prints the given document with the given attributes.

Parameters:	`doc`	The `Doc` to be printed
	`attributes`	The required printing attributes, or `null` if no printing attributes are required

API javax.print.SimpleDoc 1.4

- `SimpleDoc(Object data, DocFlavor flavor, DocAttributeSet attributes)`

  constructs a `SimpleDoc` object that can be printed with a `DocPrintJob` .

Parameters:	`data`	The object with the print data, such as an input stream or a `Printable`
	`flavor`	The document flavor of the print data
	`attributes`	Document attributes, or `null` if attributes are not required

Stream Print Services

A print service sends print data to a printer. A stream print service generates the same print data but instead senc stream, perhaps for delayed printing or because the print data format can be interpreted by other programs. In pa data format is PostScript, then it is useful to save the print data to a file because many programs can process Post Java platform includes a stream print service that can produce PostScript output from images and 2D graphics. Yo service on all systems, even if there are no local printers.

Enumerating stream print services is a bit more tedious than locating regular print services. You need both the `Doc` object to be printed and the MIME type of the stream output. You then get a `StreamPrintServiceFactory` array c

Code View:
```
DocFlavor flavor = DocFlavor.SERVICE_FORMATTED.PRINTABLE;
String mimeType = "application/postscript";
StreamPrintServiceFactory[] factories
   = StreamPrintServiceFactory.lookupStreamPrintServiceFactories(flavor, mimeType);
```

The `StreamPrintServiceFactory` class has no methods that would help us distinguish any one factory from anoth `factories[0]` . We call the `getPrintService` method with an output stream parameter to get a `StreamPrintServ`

```
OutputStream out = new FileOutputStream(fileName);
StreamPrintService service = factories[0].getPrintService(out);
```

The `StreamPrintService` class is a subclass of `PrintService` . To produce a printout, simply follow the steps of tl section.

API	`javax.print.StreamPrintServiceFactory` 1.4

- `StreamPrintServiceFactory[] lookupStreamPrintServiceFactories(DocFlavor flavor, String mimeType)`

  looks up the stream print service factories that can print the given document flavor and produces an output stream of the given MIME type.

- `StreamPrintService getPrintService(OutputStream out)`

  gets a print service that sends the printing output to the given output stream.

Printing Attributes

The print service API contains a complex set of interfaces and classes to specify various kinds of attributes. There groups of attributes. The first two specify requests to the printer.

- *Print request attributes* request particular features for all `doc` objects in a print job, such as two-sided printin size.

- *Doc attributes* are request attributes that apply only to a single `doc` object.

The other two attributes contain information about the printer and job status.

- *Print service attributes* give information about the print service, such as the printer make and model or whet currently accepting jobs.

- *Print job attributes* give information about the status of a particular print job, such as whether the job is alre

To describe the various attributes there is an interface `Attribute` with subinterfaces:

```
PrintRequestAttribute
DocAttribute
PrintServiceAttribute
PrintJobAttribute
SupportedValuesAttribute
```

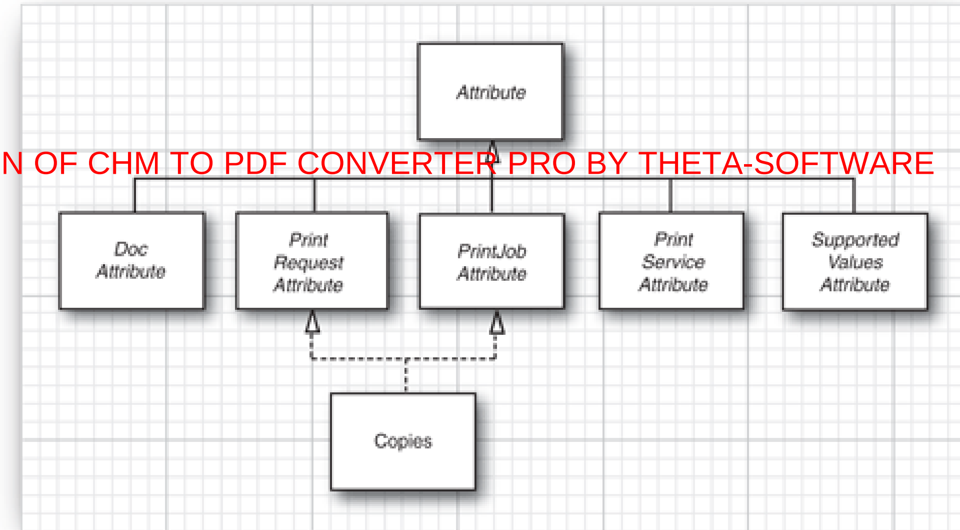
Individual attribute classes implement one or more of these interfaces. For example, objects of the `Copies` class d of copies of a printout. That class implements both the `PrintRequestAttribute` and the `PrintJobAttribute` inter print request can contain a request for multiple copies. Conversely, an attribute of the print job might be how man were actually printed. That number might be lower, perhaps because of printer limitations or because the printer r

The `SupportedValuesAttribute` interface indicates that an attribute value does not reflect actual request or statu the capability of a service. For example, the `CopiesSupported` class implements the `SupportedValuesAttribute` i of that class might describe that a printer supports 1 through 99 copies of a printout.

Figure 7-38 shows a class diagram of the attribute hierarchy.

Figure 7-38. The attribute hierarchy

[View full size image]



In addition to the interfaces and classes for individual attributes, the print service API defines interfaces and classe sets. A superinterface, `AttributeSet` , has four subinterfaces:

```
PrintRequestAttributeSet
DocAttributeSet
PrintServiceAttributeSet
PrintJobAttributeSet
```

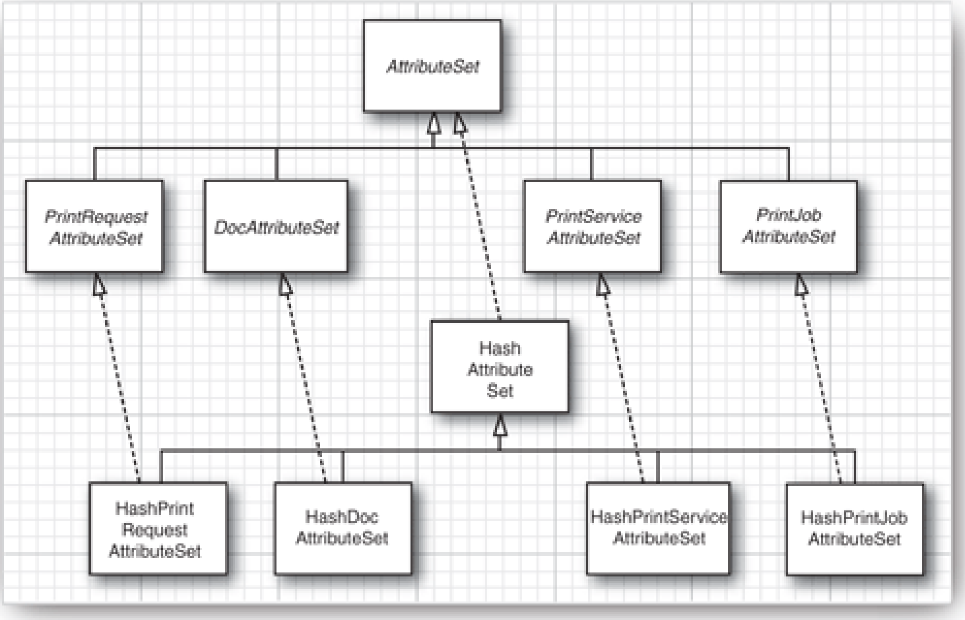
Each of these interfaces has an implementing class, yielding the five classes:

```
HashAttributeSet
HashPrintRequestAttributeSet
HashDocAttributeSet
HashPrintServiceAttributeSet
HashPrintJobAttributeSet
```

Figure 7-39 shows a class diagram of the attribute set hierarchy.

Figure 7-39. The attribute set hierarchy

[View full size image]



For example, you construct a print request attribute set like this:

```
PrintRequestAttributeSet attributes = new HashPrintRequestAttributeSet();
```

After constructing the set, you are freed from worry about the `Hash` prefix.

Why have all these interfaces? They make it possible to check for correct attribute usage. For example, a `DocAttr` only objects that implement the `DocAttribute` interface. Any attempt to add another attribute results in a runtime

An attribute set is a specialized kind of map, where the keys are of type `Class` and the values belong to a class th `Attribute` interface. For example, if you insert an object

```
new Copies(10)
```

into an attribute set, then its key is the `Class` object `Copies.class` . That key is called the *category* of the attribu interface declares a method

```
Class getCategory()
```

that returns the category of an attribute. The `Copies` class defines the method to return the object `Copies.class` requirement that the category be the same as the class of the attribute.

When an attribute is added to an attribute set, the category is extracted automatically. You just add the attribute

```
attributes.add(new Copies(10));
```

If you subsequently add another attribute with the same category, it overwrites the first one.

To retrieve an attribute, you need to use the category as the key, for example,

```
AttributeSet attributes = job.getAttributes();
Copies copies = (Copies) attribute.get(Copies.class);
```

Finally, attributes are organized by the values they can have. The `Copies` attribute can have any integer value. Th extends the `IntegerSyntax` class that takes care of all integer-valued attributes. The `getValue` method returns th the attribute, for example,


```
int n = copies.getValue();
```

The classes

```
TextSyntax
DateTimeSyntax
URISyntax
```


encapsulate a string, date and time, or URI.

Finally, many attributes can take a finite number of values. For example, the `PrintQuality` attribute has three set normal, and high. They are represented by three constants:

```
PrintQuality.DRAFT
PrintQuality.NORMAL
PrintQuality.HIGH
```

Attribute classes with a finite number of values extend the `EnumSyntax` class, which provides a number of convenie set up these enumerations in a typesafe manner. You need not worry about the mechanism when using such an a the named values to attribute sets:

```
attributes.add(PrintQuality.HIGH);
```

Here is how you check the value of an attribute:

```
if (attributes.get(PrintQuality.class) == PrintQuality.HIGH)
   . . .
```

Table 7-4 lists the printing attributes. The second column lists the superclass of the attribute class (for example, `I` the `Copies` attribute) or the set of enumeration values for the attributes with a finite set of values. The last four co whether the attribute class implements the `DocAttribute` (DA), `PrintJobAttribute` (PJA), `PrintRequestAttribu` `PrintServiceAttribute` (PSA) interfaces.

Table 7-4. Printing Attributes

Attribute	Superclass or Enumeration Constants	DA	PJA	PRA
`Chromaticity`	`MONOCHROME, COLOR`	✓	✓	✓
`ColorSupported`	`SUPPORTED, NOT_SUPPORTED`			
`Compression`	`COMPRESS, DEFLATE, GZIP, NONE`	✓		

Attribute	Superclass or Enumeration Constants	DA	PJA	PRA
Copies	IntegerSyntax		✓	✓
DateTimeAtCompleted	DateTimeSyntax		✓	
DateTimeAtCreation	DateTimeSyntax		✓	
DateTimeAtProcessing	DateTimeSyntax		✓	
Destination	URISyntax		✓	✓
DocumentName	TextSyntax	✓		
Fidelity	FIDELITY_TRUE, FIDELITY_FALSE		✓	✓
Finishings	NONE, STAPLE, EDGE_STITCH, BIND, SADDLE_STITCH, COVER, . . .	✓	✓	✓
JobHoldUntil	DateTimeSyntax		✓	✓
JobImpressions	IntegerSyntax		✓	✓
JobImpressionsCompleted	IntegerSyntax		✓	
JobKOctets	IntegerSyntax		✓	✓
JobKOctetsProcessed	IntegerSyntax		✓	
JobMediaSheets	IntegerSyntax		✓	✓
JobMediaSheetsCompleted	IntegerSyntax		✓	
JobMessageFromOperator	TextSyntax		✓	
JobName	TextSyntax		✓	✓
JobOriginatingUserName	TextSyntax		✓	
JobPriority	IntegerSyntax		✓	✓
JobSheets	STANDARD, NONE		✓	✓
JobState	ABORTED, CANCELED, COMPLETED, PENDING, PENDING_HELD, PROCESSING, PROCESSING_STOPPED		✓	
JobStateReason	ABORTED_BY_SYSTEM, DOCUMENT_FORMAT_ERROR, many others			
JobStateReasons	HashSet		✓	
MediaName	ISO_A4_WHITE, ISO_A4_TRANSPARENT, NA_LETTER_WHITE, NA_LETTER_TRANSPARENT	✓	✓	✓
MediaSize	ISO.A0 - ISO.A10, ISO.B0 - ISO.B10, ISO.C0 - ISO.C10, NA.LETTER, NA.LEGAL, various other paper and envelope sizes			

Attribute	Superclass or Enumeration Constants	DA	PJA	PRA
MediaSizeName	ISO_A0 - ISO_A10, ISO_B0 - ISO_B10, ISO_C0 - ISO_C10, NA_LETTER, NA_LEGAL, various other paper and envelope size names	✓	✓	✓
MediaTray	TOP, MIDDLE, BOTTOM, SIDE, ENVELOPE, LARGE_CAPACITY, MAIN, MANUAL	✓	✓	✓
MultipleDocumentHandling	SINGLE_DOCUMENT, SINGLE_DOCUMENT_NEW_SHEET, SEPARATE_DOCUMENTS_COLLATED_COPIES, SEPARATE_DOCUMENTS_UNCOLLATED_COPIES		✓	✓
NumberOfDocuments	IntegerSyntax		✓	
NumberOfInterveningJobs	IntegerSyntax		✓	
NumberUp	IntegerSyntax	✓	✓	✓
OrientationRequested	PORTRAIT, LANDSCAPE, REVERSE_PORTRAIT, REVERSE_LANDSCAPE	✓	✓	✓
OutputDeviceAssigned	TextSyntax		✓	
PageRanges	SetOfInteger	✓	✓	✓
PagesPerMinute	IntegerSyntax			
PagesPerMinuteColor	IntegerSyntax			
PDLOverrideSupported	ATTEMPTED, NOT_ATTEMPTED			
PresentationDirection	TORIGHT_TOBOTTOM, TORIGHT_TOTOP, TOBOTTOM_TORIGHT, TOBOTTOM_TOLEFT, TOLEFT_TOBOTTOM, TOLEFT_TOTOP, TOTOP_TORIGHT, TOTOP_TOLEFT		✓	✓
PrinterInfo	TextSyntax			
PrinterIsAcceptingJobs	ACCEPTING_JOBS, NOT_ACCEPTING_JOBS			
PrinterLocation	TextSyntax			
PrinterMakeAndModel	TextSyntax			
PrinterMessageFromOperator	TextSyntax			
PrinterMoreInfo	URISyntax			
PrinterMoreInfoManufacturer	URISyntax			
PrinterName	TextSyntax			
PrinterResolution	ResolutionSyntax	✓	✓	✓
PrinterState	PROCESSING, IDLE, STOPPED, UNKNOWN			

Attribute	Superclass or Enumeration Constants	DA	PJA	PRA
PrinterStateReason	COVER_OPEN, FUSER_OVER_TEMP, MEDIA_JAM, and many others			
PrinterStateReasons	HashMap			
PrinterURI	URISyntax			
PrintQuality	DRAFT, NORMAL, HIGH	✓	✓	✓
QueuedJobCount	IntegerSyntax			
ReferenceUriSchemesSupported	FILE, FTP, GOPHER, HTTP, HTTPS, NEWS, NNTP, WAIS			
RequestingUserName	TextSyntax			✓
Severity	ERROR, REPORT, WARNING			
SheetCollate	COLLATED, UNCOLLATED	✓	✓	✓
Sides	ONE_SIDED, DUPLEX (=TWO_SIDED_LONG_EDGE), TUMBLE (=TWO_SIDED_SHORT_EDGE)	✓	✓	✓

Note

As you can see, there are lots of attributes, many of which are quite specialized. The source for most the attributes is the Internet Printing Protocol 1.1 (RFC 2911).

Note

An earlier version of the printing API introduced the JobAttributes and PageAttributes classes, th purpose of which is similar to the printing attributes covered in this section. These classes are now obsolete.

API

`javax.print.attribute.Attribute` 1.4

- `Class getCategory()`

  gets the category of this attribute.

- `String getName()`

  gets the name of this attribute.

API

`javax.print.attribute.AttributeSet` 1.4

- `boolean add(Attribute attr)`

  adds an attribute to this set. If the set has another attribute with the same category, that attribute is replaced by the given attribute. Returns `true` if the set changed as a result of this operation.

- `Attribute get(Class category)`

  retrieves the attribute with the given category key, or `null` if no such attribute exists.

- `boolean remove(Attribute attr)`

- `boolean remove(Class category)`

  removes the given attribute, or the attribute with the given category, from the set. Returns `true` if the set changed as a result of this operation.

- `Attribute[] toArray()`

  returns an array with all attributes in this set.

API `javax.print.PrintService` 1.4

- `PrintServiceAttributeSet getAttributes()`

  gets the attributes of this print service.

API `javax.print.DocPrintJob` 1.4

- `PrintJobAttributeSet getAttributes()`

  gets the attributes of this print job.

This concludes our discussion on printing. You now know how to print 2D graphics and other document types, how printers and stream print services, and how to set and retrieve attributes. Next, we turn to two important user inte clipboard and support for the drag-and-drop mechanism.

The Clipboard

One of the most useful and convenient user interface mechanisms of GUI environments (such as Windows and the X Window System) is *cut and paste*. You select some data in one program and cut or copy them to the clipboard. Then, you select another program and paste the clipboard contents into that application. Using the clipboard, you can transfer text, images, or other data from one document to another, or, of course, from one place in a document to another place in the same document. Cut and paste is so natural that most computer users never think about it.

Even though the clipboard is conceptually simple, implementing clipboard services is actually harder than you might think. Suppose you copy text from a word processor to the clipboard. If you paste that text into another word processor, then you expect that the fonts and formatting will stay intact. That is, the text in the clipboard needs to retain the formatting information. However, if you paste the text into a plain text field, then you expect that just the characters are pasted in, without additional formatting codes. To support this flexibility, the data provider can offer the clipboard data in multiple formats, and the data consumer can pick one of them.

The system clipboard implementations of Microsoft Windows and the Macintosh are similar, but, of course, there are slight differences. However, the X Window System clipboard mechanism is much more limited—cutting and pasting of anything but plain text is only sporadically supported. You should consider these limitations when trying out the programs in this section.

Note

Check out the file *jre/*`lib/flavormap.properties` on your platform to get an idea about what kinds of objects can be transferred between Java programs and the system clipboard.

Often, programs need to support cut and paste of data types that the system clipboard cannot handle. The data transfer API supports the transfer of arbitrary local object references in the same virtual machine. Between different virtual machines, you can transfer serialized objects and references to remote objects.

Table 7-5 summarizes the data transfer capabilities of the clipboard mechanism.

Table 7-5. Capabilities of the Java Data Transfer Mechanism

Transfer	Format
Between a Java program and a native program	Text, images, file lists, . . . (depending on the host platform)
Between two cooperating Java programs	Serialized and remote objects
Within one Java program	Any object

Classes and Interfaces for Data Transfer

Data transfer in the Java technology is implemented in a package called `java.awt.datatransfer`. Here is an overview of the most important classes and interfaces of that package.

- Objects that can be transferred via a clipboard must implement the `Transferable` interface.
- The `Clipboard` class describes a clipboard. Transferable objects are the only items that can be put on or taken off a clipboard. The system clipboard is a concrete example of a `Clipboard`.
- The `DataFlavor` class describes data flavors that can be placed on the clipboard.
- The `StringSelection` class is a concrete class that implements the `Transferable` interface. It transfers text strings.
- A class must implement the `ClipboardOwner` interface if it wants to be notified when the clipboard contents have been overwritten by somooeone else. Clipboard ownership enables "delayed formatting" of complex data. If a program transfers simple data (such as a string), then it simply sets the clipboard contents and moves on to do the next thing. However, if a program will place complex data that can be formatted in multiple flavors onto the clipboard, then it might not actually want to prepare all the flavors, because there is a good chance that most of them are never needed. However, then it needs to hang on to the clipboard data so that it can create the flavors later when they are requested. The clipboard owner is notified (by a call to its `lostOwnership` method) when the contents of the clipboard change. That tells it that the information is no longer needed. In our sample programs, we don't worry about clipboard ownership.

Transferring Text

The best way to get comfortable with the data transfer classes is to start with the simplest situation: transferring text to and from the system clipboard. First, get a reference to the system clipboard.

```
Clipboard clipboard = Toolkit.getDefaultToolkit().getSystemClipboard();
```

For strings to be transferred to the clipboard, they must be wrapped into `StringSelection` objects.

```
String text = . . .
StringSelection selection = new StringSelection(text);
```

The actual transfer is done by a call to `setContents`, which takes a `StringSelection` object and a `ClipBoardOwner` as parameters. If you are not interested in designating a clipboard owner, set the second parameter to `null`.

```
clipboard.setContents(selection, null);
```

Here is the reverse operation, reading a string from the clipboard:

```
DataFlavor flavor = DataFlavor.stringFlavor;
if (clipboard.isDataFlavorAvailable(flavor)
   String text = (String) clipboard.getData(flavor);
```

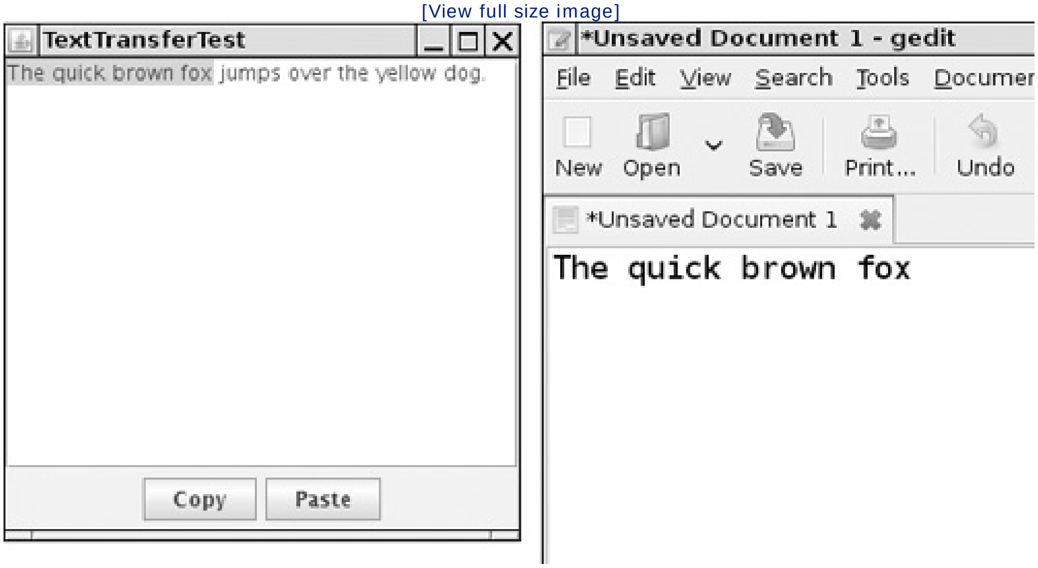
The parameter of the getContents call is an Object reference of the requesting object, but because the current implementation of the Clipboard class ignores it, we just pass null.

The return value of getContents can be null. That indicates that the clipboard is either empty or that it has no data that the Java platform knows how to retrieve as text.

Listing 7-11 is a program that demonstrates cutting and pasting between a Java application and the system clipboard. If you select an area of text in the text area and click Copy, then the selected text is copied to the system clipboard. You can then paste it into any text editor (see Figure 7-40). Conversely, when you copy text from the text editor, you can paste it into our sample program.

Figure 7-40. The TextTransferTest program

[View full size image]



Listing 7-11. **TextTransferTest.java**

Code View:
```
import java.awt.*;
import java.awt.datatransfer.*;
import java.awt.event.*;
import java.io.*;

import javax.swing.*;

/**
 * This program demonstrates the transfer of text between a Java application and the systen
 * clipboard.
 * @version 1.13 2007-08-16
 * @author Cay Horstmann
 */
public class TextTransferTest
{
   public static void main(String[] args)
   {
      EventQueue.invokeLater(new Runnable()
```

```
         {
            public void run()
            {
               JFrame frame = new TextTransferFrame();
               frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
               frame.setVisible(true);
            }
         });
   }
}

/**
 * This frame has a text area and buttons for copying and pasting text.
 */
class TextTransferFrame extends JFrame
{
   public TextTransferFrame()
   {
      setTitle("TextTransferTest");
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

      textArea = new JTextArea();
      add(new JScrollPane(textArea), BorderLayout.CENTER);
      JPanel panel = new JPanel();

      JButton copyButton = new JButton("Copy");
      panel.add(copyButton);
      copyButton.addActionListener(new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               copy();
            }
         });

      JButton pasteButton = new JButton("Paste");
      panel.add(pasteButton);
      pasteButton.addActionListener(new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               paste();
            }
         });

      add(panel, BorderLayout.SOUTH);
   }

   /**
    * Copies the selected text to the system clipboard.
    */
   private void copy()
   {
      Clipboard clipboard = Toolkit.getDefaultToolkit().getSystemClipboard();
      String text = textArea.getSelectedText();
      if (text == null) text = textArea.getText();
      StringSelection selection = new StringSelection(text);
      clipboard.setContents(selection, null);
   }
```

```
   /**
    * Pastes the text from the system clipboard into the text area.
    */
   private void paste()
   {
      Clipboard clipboard = Toolkit.getDefaultToolkit().getSystemClipboard();
      DataFlavor flavor = DataFlavor.stringFlavor;
      if (clipboard.isDataFlavorAvailable(flavor))
      {
         try
         {
            String text = (String) clipboard.getData(flavor);
            textArea.replaceSelection(text);
         }
         catch (UnsupportedFlavorException e)
         {
            JOptionPane.showMessageDialog(this, e);
         }
         catch (IOException e)
         {
            JOptionPane.showMessageDialog(this, e);
         }
      }
   }

   private JTextArea textArea;

   private static final int DEFAULT_WIDTH = 300;
   private static final int DEFAULT_HEIGHT = 300;
}
```


API

`java.awt.Toolkit` 1.0

- `Clipboard getSystemClipboard()` **1.1**

  gets the system clipboard.

`java.awt.datatransfer.Clipboard` 1.1

- `Transferable getContents(Object requester)`

  gets the clipboard contents.

Parameters:	`requester`	The object requesting the clipboard contents; this value is not actually used

- `void setContents(Transferable contents, ClipboardOwner owner)`

  puts contents on the clipboard.

Parameters:	`contents`	The `Transferable` encapsulating the contents
	`owner`	The object to be notified (via its `lostOwnership` method) when new information is placed on the clipboard, or `null` if no notification is desired

- `boolean isDataFlavorAvailable(DataFlavor flavor)` **5.0**

  returns `true` if the clipboard has data in the given flavor.

- `Object getData(DataFlavor flavor)` **5.0**

  gets the data in the given flavor, or throws an `UnsupportedFlavorException` if no data are available in the given flavor.

`java.awt.datatransfer.ClipboardOwner` 1.1

- `void lostOwnership(Clipboard clipboard, Transferable contents)`

  notifies this object that it is no longer the owner of the contents of the clipboard.

Parameters:	`clipboard`	The clipboard onto which the contents were placed
	`contents`	The item that this owner had placed onto the clipboard

java.awt.datatransfer.Transferable 1.1

- boolean isDataFlavorSupported(DataFlavor flavor)

  returns true if the specified flavor is one of the supported data flavors; false otherwise.

- Object getTransferData(DataFlavor flavor)

  returns the data, formatted in the requested flavor. Throws an UnsupportedFlavorException if the flavor requested is not supported.

The Transferable Interface and Data Flavors

A DataFlavor is defined by two characteristics:

- A MIME type name (such as "image/gif").
- A representation class for accessing the data (such as java.awt.Image).

In addition, every data flavor has a human-readable name (such as "GIF Image").

The representation class can be specified with a class parameter in the MIME type, for example,

```
image/gif;class=java.awt.Image
```

Note

This is just an example to show the syntax. There is no standard data flavor for transferring GIF image data.

If no class parameter is given, then the representation class is InputStream.

For transferring local, serialized, and remote Java objects, Sun Microsystems defines three MIME types:

```
application/x-java-jvm-local-objectref
application/x-java-serialized-object
application/x-java-remote-object
```

Note

The `x-` prefix indicates that this is an experimental name, not one that is sanctioned by IANA, the organization that assigns standard MIME type names.

For example, the standard `stringFlavor` data flavor is described by the MIME type

```
application/x-java-serialized-object;class=java.lang.String
```

You can ask the clipboard to list all available flavors:

```
DataFlavor[] flavors = clipboard.getAvailableDataFlavors()
```

You can also install a `FlavorListener` onto the clipboard. The listener is notified when the collection of data flavors on the clipboard changes. See the API notes for details.

API `java.awt.datatransfer.DataFlavor` 1.1

- `DataFlavor(String mimeType, String humanPresentableName)`

  creates a data flavor that describes stream data in a format described by a MIME type.

Parameters:	`mimeType`	A MIME type string
	`humanPresentableName`	A more readable version of the name

- `DataFlavor(Class class, String humanPresentableName)`

  creates a data flavor that describes a Java platform class. Its MIME type is `application/x-java-serialized-object;class=className`.

Parameters:	`class`	The class that is retrieved from the `Transferable`
	`humanPresentableName`	A readable version of the name

- `String getMimeType()`

  returns the MIME type string for this data flavor.

- `boolean isMimeTypeEqual(String mimeType)` tests whether this data flavor has the given MIME type.

- String getHumanPresentableName()

  returns the human-presentable name for the data format of this data flavor.

- Class getRepresentationClass()

  returns a Class object that represents the class of the object that a Transferable object will return when called with this data flavor. This is either the class parameter of the MIME type or InputStream.

API java.awt.datatransfer.Clipboard 1.1

- DataFlavor[] getAvailableDataFlavors() **5.0**

  returns an array of the available flavors.

- void addFlavorListener(FlavorListener listener) **5.0**

  adds a listener that is notified when the set of available flavors changes.

API java.awt.datatransfer.Transferable 1.1

- DataFlavor[] getTransferDataFlavors()

  returns an array of the supported flavors.

API java.awt.datatransfer.FlavorListener 5.0

- void flavorsChanged(FlavorEvent event)

  is called when a clipboard's set of available flavors changes.

Building an Image Transferable

Objects that you want to transfer via the clipboard must implement the Transferable interface. The StringSelection class is currently the only public class in the Java standard library that implements the Transferable interface. In this section, you will see how to transfer images into the clipboard. Because Java does not supply a class for image transfer, you must implement it yourself.

The class is completely trivial. It simply reports that the only available data format is DataFlavor.imageFlavor, and it holds an image object.

Code View:
```
class ImageTransferable implements Transferable
{
   public ImageTransferable(Image image)
   {
      theImage = image;
   }

   public DataFlavor[] getTransferDataFlavors()
   {
      return new DataFlavor[] { DataFlavor.imageFlavor };
   }

   public boolean isDataFlavorSupported(DataFlavor flavor)
   {
      return flavor.equals(DataFlavor.imageFlavor);
   }

   public Object getTransferData(DataFlavor flavor)
      throws UnsupportedFlavorException
   {
      if(flavor.equals(DataFlavor.imageFlavor))
      {
         return theImage;
      }
      else
      {
         throw new UnsupportedFlavorException(flavor);
      }
   }

   private Image theImage;
}
```

Note

Java SE supplies the DataFlavor.imageFlavor constant and does all the heavy lifting to convert between Java images and native clipboard images. But, curiously, it does not supply the wrapper class that is necessary to place images onto the clipboard.

The program of Listing 7-12 demonstrates the transfer of images between a Java application and the system clipboard. When the program starts, it generates an image containing a red circle. Click the Copy button to copy the image to the clipboard and then paste it into another application (see Figure 7-41). From another application, copy an image into the system clipboard. Then click the Paste button and see the image being pasted into the example program (see Figure 7-42).

Figure 7-41. Copying from a Java program to a native program

[View full size image]

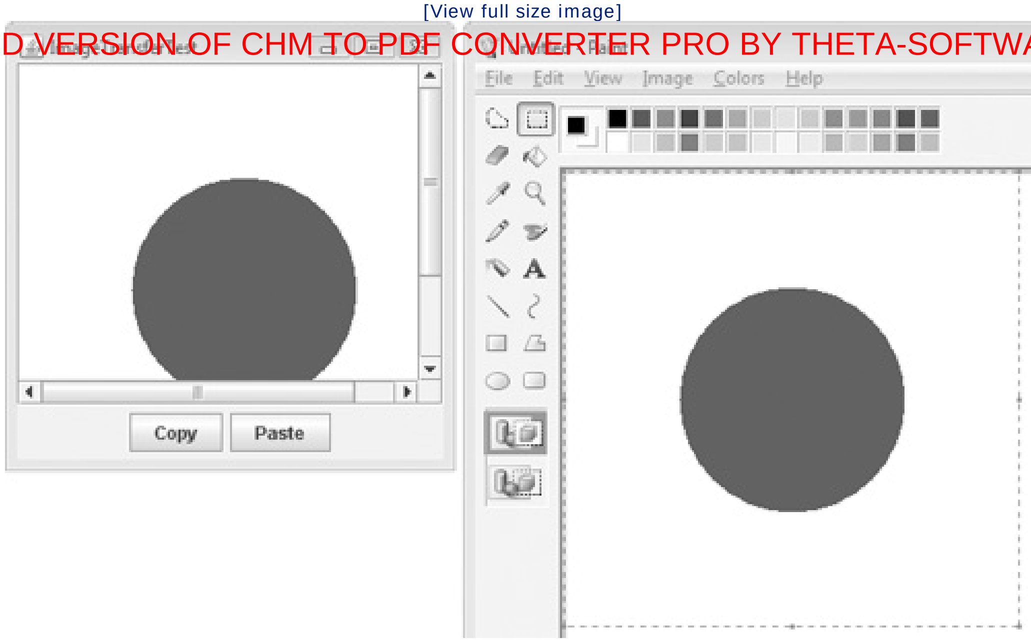


Figure 7-42. Copying from a native program to a Java program

[View full size image]

The program is a straightforward modification of the text transfer program. The data flavor is now `DataFlavor.imageFlavor`, and we use the `ImageTransferable` class to transfer an image to the system clipboard.

Listing 7-12. **`ImageTransferTest.java`**

Code View:

```
import java.io.*;
import java.awt.*;
import java.awt.datatransfer.*;
import java.awt.event.*;
import java.awt.image.*;
import javax.swing.*;

/**
 * This program demonstrates the transfer of images between a Java application and the system
 * clipboard.
 * @version 1.22 2007-08-16
 * @author Cay Horstmann
 */
public class ImageTransferTest
{
   public static void main(String[] args)
   {
      EventQueue.invokeLater(new Runnable()
         {
            public void run()
            {
               JFrame frame = new ImageTransferFrame();
               frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
               frame.setVisible(true);
            }
         });
   }
```

```
}

/**
 * This frame has an image label and buttons for copying and pasting an image.
 */
class ImageTransferFrame extends JFrame
{
   public ImageTransferFrame()
   {
```

```
      setTitle("ImageTransferTest");
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

      label = new JLabel();
      image = new BufferedImage(DEFAULT_WIDTH, DEFAULT_HEIGHT, BufferedImage.TYPE_INT_ARGB);
      Graphics g = image.getGraphics();
      g.setColor(Color.WHITE);
```

```
      g.fillRect(0, 0, DEFAULT_WIDTH, DEFAULT_HEIGHT);
      g.setColor(Color.RED);
      g.fillOval(DEFAULT_WIDTH / 4, DEFAULT_WIDTH / 4, DEFAULT_WIDTH / 2, DEFAULT_HEIGHT / 2);

      label.setIcon(new ImageIcon(image));
      add(new JScrollPane(label), BorderLayout.CENTER);
      JPanel panel = new JPanel();

      JButton copyButton = new JButton("Copy");
      panel.add(copyButton);
      copyButton.addActionListener(new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               copy();
            }
         });

      JButton pasteButton = new JButton("Paste");
      panel.add(pasteButton);
      pasteButton.addActionListener(new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               paste();
            }
         });

      add(panel, BorderLayout.SOUTH);
   }

   /**
    * Copies the current image to the system clipboard.
    */
   private void copy()
   {
      Clipboard clipboard = Toolkit.getDefaultToolkit().getSystemClipboard();
      ImageTransferable selection = new ImageTransferable(image);
      clipboard.setContents(selection, null);
   }

   /**
    * Pastes the image from the system clipboard into the image label.
```

```
    */
   private void paste()
   {
      Clipboard clipboard = Toolkit.getDefaultToolkit().getSystemClipboard();
      DataFlavor flavor = DataFlavor.imageFlavor;
      if (clipboard.isDataFlavorAvailable(flavor))
      {
         try
         {
            image = (Image) clipboard.getData(flavor);
            label.setIcon(new ImageIcon(image));
         }
         catch (UnsupportedFlavorException exception)
         {
            JOptionPane.showMessageDialog(this, exception);
         }
         catch (IOException exception)
         {
            JOptionPane.showMessageDialog(this, exception);
         }
      }
   }

   private JLabel label;
   private Image image;

   private static final int DEFAULT_WIDTH = 300;
   private static final int DEFAULT_HEIGHT = 300;
}

/**
 * This class is a wrapper for the data transfer of image objects.
 */
class ImageTransferable implements Transferable
{
   /**
    * Constructs the selection.
    * @param image an image
    */
   public ImageTransferable(Image image)
   {
      theImage = image;
   }

   public DataFlavor[] getTransferDataFlavors()
   {
      return new DataFlavor[] { DataFlavor.imageFlavor };
   }

   public boolean isDataFlavorSupported(DataFlavor flavor)
   {
      return flavor.equals(DataFlavor.imageFlavor);
   }

   public Object getTransferData(DataFlavor flavor) throws UnsupportedFlavorException
   {
      if (flavor.equals(DataFlavor.imageFlavor))
      {
         return theImage;
```

```
        }
        else
        {
            throw new UnsupportedFlavorException(flavor);
        }
    }

    private Image theImage;
}
```


Transferring Java Objects via the System Clipboard

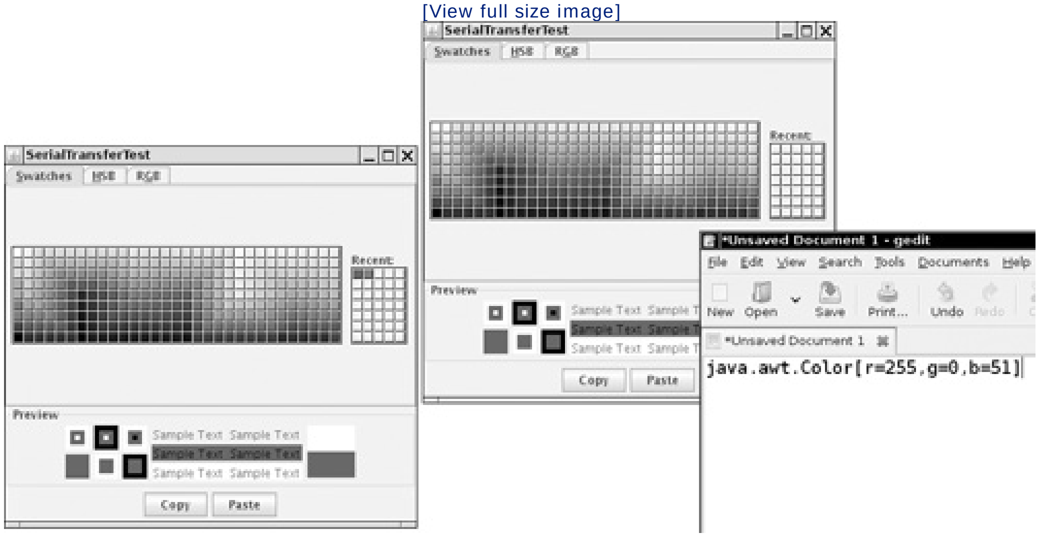
Suppose you want to copy and paste objects from one Java application to another. In that case, you cannot use local clipboards. Fortunately, you can place serialized Java objects onto the system clipboard.

The program in Listing 7-13 demonstrates this capability. The program shows a color chooser. The Copy button copies the current color to the system clipboard as a serialized `Color` object. The Paste button checks whether the system clipboard contains a serialized `Color` object. If so, it fetches the color and sets it as the current choice of the color chooser.

You can transfer the serialized object between two Java applications (see Figure 7-43). Run two copies of the `SerialTransferTest` program. Click Copy in the first program, then click Paste in the second program. The `Color` object is transferred from one virtual machine to the other.

Figure 7-43. Data are copied between two instances of a Java application

[View full size image]



To enable the data transfer, the Java platform places binary data on the system clipboard that contains the serialized object. Another Java program—not necessarily of the same type as the one that generated the clipboard data—can retrieve the clipboard data and deserialize the object.

Of course, a non-Java application will not know what to do with the clipboard data. For that reason, the example program offers the clipboard data in a second flavor, as text. The text is simply the result of the `toString` method, applied to the transferred object. To see the second flavor, run the program, click on a color, and then select the Paste command in your text editor. A string such as

```
java.awt.Color[r=255,g=0,b=51]
```

will be inserted into your document.

Essentially no additional programming is required to transfer a serializable object. You use the MIME type

`application/x-java-serialized-object;class=`*className*

As before, you have to build your own transfer wrapper—see the example code for details.

Listing 7-13. **SerialTransferTest.java**

Code View:

```
import java.io.*;
import java.awt.*;
import java.awt.datatransfer.*;
import java.awt.event.*;
import javax.swing.*;

/**
 * This program demonstrates the transfer of serialized objects between virtual machines.
 * @version 1.02 2007-08-16
 * @author Cay Horstmann
 */
public class SerialTransferTest
{
   public static void main(String[] args)
   {
      EventQueue.invokeLater(new Runnable()
         {
            public void run()
            {
               JFrame frame = new SerialTransferFrame();
               frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
               frame.setVisible(true);
            }
         });
   }
}

/**
 * This frame contains a color chooser, and copy and paste buttons.
 */
class SerialTransferFrame extends JFrame
{
   public SerialTransferFrame()
   {
      setTitle("SerialTransferTest");

      chooser = new JColorChooser();
```

```
      add(chooser, BorderLayout.CENTER);
      JPanel panel = new JPanel();

      JButton copyButton = new JButton("Copy");
      panel.add(copyButton);
      copyButton.addActionListener(new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
```

```
               copy();
            }
         });

      JButton pasteButton = new JButton("Paste");
      panel.add(pasteButton);
      pasteButton.addActionListener(new ActionListener()
```

```
         {
             public void actionPerformed(ActionEvent event)
             {
                paste();
             }
          });

       add(panel, BorderLayout.SOUTH);
       pack();
   }

   /**
    * Copies the chooser's color into the system clipboard.
    */
   private void copy()
   {
      Clipboard clipboard = Toolkit.getDefaultToolkit().getSystemClipboard();
      Color color = chooser.getColor();
      SerialTransferable selection = new SerialTransferable(color);
      clipboard.setContents(selection, null);
   }

   /**
    * Pastes the color from the system clipboard into the chooser.
    */
   private void paste()
   {
      Clipboard clipboard = Toolkit.getDefaultToolkit().getSystemClipboard();
      try
      {
         DataFlavor flavor = new DataFlavor(
               "application/x-java-serialized-object;class=java.awt.Color");
         if (clipboard.isDataFlavorAvailable(flavor))
         {
            Color color = (Color) clipboard.getData(flavor);
            chooser.setColor(color);
         }
      }
      catch (ClassNotFoundException e)
      {
         JOptionPane.showMessageDialog(this, e);
      }
      catch (UnsupportedFlavorException e)
```

```
      {
         JOptionPane.showMessageDialog(this, e);
      }
      catch (IOException e)
      {
         JOptionPane.showMessageDialog(this, e);
      }
   }

   private JColorChooser chooser;
}

/**
 * This class is a wrapper for the data transfer of serialized objects.
 */
class SerialTransferable implements Transferable
{
   /**
    * Constructs the selection.
    * @param o any serializable object
    */
   SerialTransferable(Serializable o)
   {
      obj = o;
   }

   public DataFlavor[] getTransferDataFlavors()
   {
      DataFlavor[] flavors = new DataFlavor[2];
      Class<?> type = obj.getClass();
      String mimeType = "application/x-java-serialized-object;class=" + type.getName();
      try
      {
         flavors[0] = new DataFlavor(mimeType);
         flavors[1] = DataFlavor.stringFlavor;
         return flavors;
      }
      catch (ClassNotFoundException e)
      {
         return new DataFlavor[0];
      }
   }

   public boolean isDataFlavorSupported(DataFlavor flavor)
   {
      return DataFlavor.stringFlavor.equals(flavor)
            || "application".equals(flavor.getPrimaryType())
            && "x-java-serialized-object".equals(flavor.getSubType())
            && flavor.getRepresentationClass().isAssignableFrom(obj.getClass());
   }

   public Object getTransferData(DataFlavor flavor) throws UnsupportedFlavorException
   {
      if (!isDataFlavorSupported(flavor)) throw new UnsupportedFlavorException(flavor);

      if (DataFlavor.stringFlavor.equals(flavor)) return obj.toString();

      return obj;
   }
```

```
    private Serializable obj;
}
```


Using a Local Clipboard to Transfer Object References

Occasionally, you might need to copy and paste a data type that isn't one of the data types supported by the system clipboard, and that isn't serializable. To transfer an arbitrary Java object reference within the same JVM, you use the MIME type

`application/x-java-jvm-local-objectref;class=`*className*

You need to define a `Transferable` wrapper for this type. The process is entirely analogous to the `SerialTransferable` wrapper of the preceding example.

An object reference is only meaningful within a single virtual machine. For that reason, you cannot copy the shape object to the system clipboard. Instead, use a local clipboard:

```
Clipboard clipboard = new Clipboard("local");
```

The construction parameter is the clipboard name.

However, using a local clipboard has one major disadvantage. You need to synchronize the local and the system clipboard, so that users don't confuse the two. Currently, the Java platform doesn't do that synchronization for you.

API `java.awt.datatransfer.Clipboard` 1.1

- `Clipboard(String name)`

  constructs a local clipboard with the given name.

Drag and Drop

When you use cut and paste to transmit information between two programs, the clipboard acts as an intermediary The *drag and drop* metaphor cuts out the middleman and lets two programs communicate directly. The Java platform offers basic support for drag and drop. You can carry out drag and drop operations between Java applications and native applications. This section shows you how to write a Java application that is a drop target, and an application that is a drag source.

Before going deeper into the Java platform support for drag and drop, let us quickly look at the drag-and-drop use interface. We use the Windows Explorer and WordPad programs as examples—on another platform, you can experiment with locally available programs with drag-and-drop capabilities.

You initiate a *drag operation* with a *gesture* inside a *drag source* —by first selecting one or more elements and the dragging the selection away from its initial location. When you release the mouse button over a drop target that accepts the drop operation, the drop target queries the drag source for information about the dropped elements ai carries out an appropriate operation. For example, if you drop a file icon from a file manager on top of a directory icon, then the file is moved into that directory. However, if you drag it to a text editor, then the text editor opens the file. (This requires, of course, that you use a file manager and text editor that are enabled for drag and drop, such as Explorer/WordPad in Windows or Nautilus/gedit in Gnome).

If you hold down the CTRL key while dragging, then the type of the drop action changes from a *move action* to a *copy action,* and a copy of the file is placed into the directory. If you hold down *both* SHIFT and CTRL keys, then a *lin* to the file is placed into the directory. (Other platforms might use other keyboard combinations for these operations.)

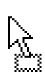
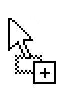
Thus, there are three types of drop actions with different gestures:

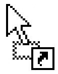
- Move
- Copy
- Link

The intention of the link action is to establish a reference to the dropped element. Such links typically require support from the host operating system (such as symbolic links for files, or object linking for document components) and don't usually make a lot of sense in cross-platform programs. In this section, we focus on using drag and drop for copying and moving.

There is usually some visual feedback for the drag operation. Minimally, the cursor shape changes. As the cursor moves over possible *drop targets,* the cursor shape indicates whether the drop is possible or not. If a drop is possible, the cursor shape also indicates the type of the drop action. Table 7-6 shows several drop cursor shapes.

Table 7-6. Drop Cursor Shapes

Action	Windows Icon	Gnome Icon
Move		
Copy		

Action	Windows Icon	Gnome Icon
Link		
Drop not allowed		

You can also drag other elements besides file icons. For example, you can select text in WordPad or gedit and drag it. Try dropping text fragments into willing drop targets and see how they react.

Note

This experiment shows a disadvantage of drag and drop as a user interface mechanism. It can be difficult for users to anticipate what they can drag, where they can drop it, and what happens when they do. Because the default "move" action can remove the original, many users are understandably cautious about experimenting with drag and drop.

Data Transfer Support in Swing

Starting with Java SE 1.4, several Swing components have built-in support for drag and drop (see Table 7-7). You can drag selected text from a number of components, and you can drop text into text components. For backward compatibility, you must call the `setDragEnabled` method to activate dragging. Dropping is always enabled.

Table 7-7. Data Transfer Support in Swing Components

Component	Drag Source	Drop Target
`JFileChooser`	Exports file list	N/A
`JColorChooser`	Exports color object	Accepts color objects
`JTextField` `JFormattedTextField`	Exports selected text	Accepts text
`JPasswordField`	N/A (for security)	Accepts text

`JTextArea`
`JTextPane`
`JEditorPane`
	Exports selected text	Accepts text and file lists

`JList`
`JTable`
`JTree`
	Exports text description of selection (copy only)	N/A

Note

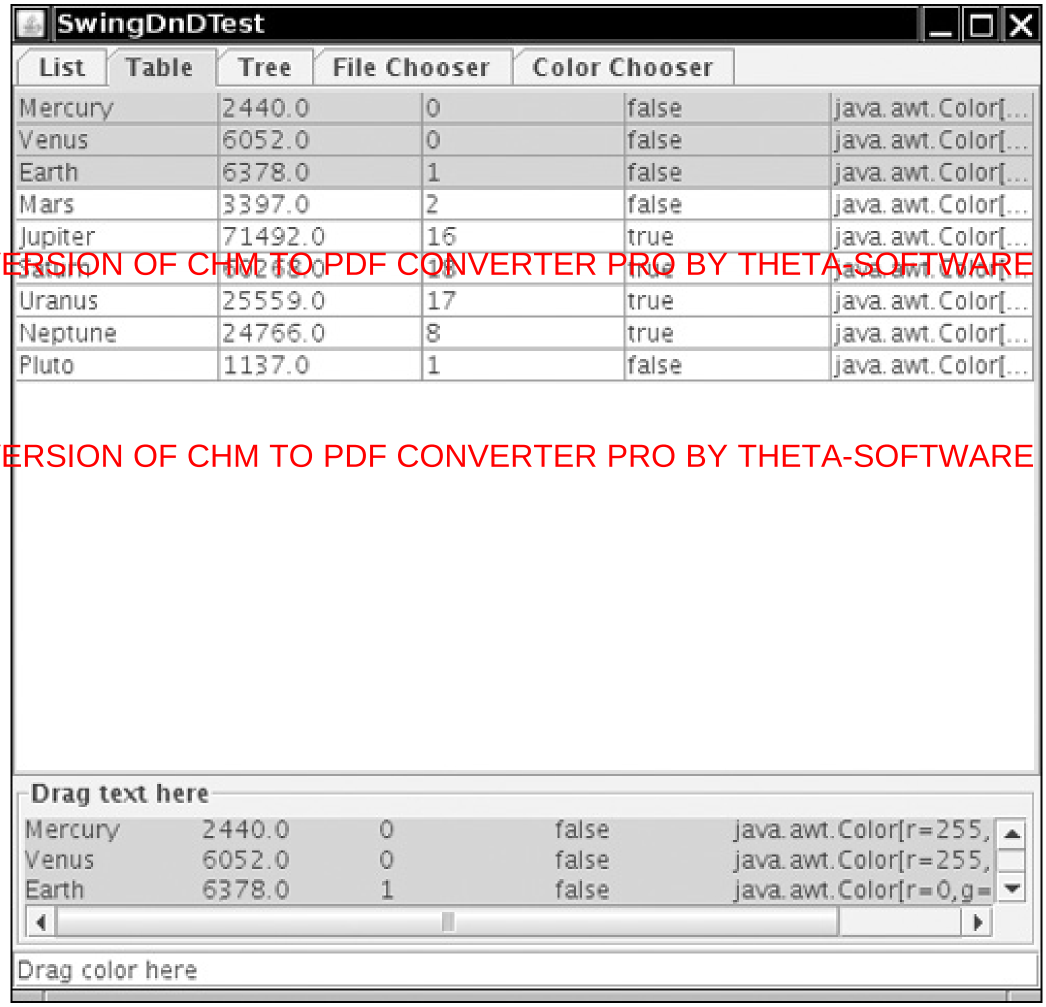
The `java.awt.dnd` package provides a lower-level drag-and-drop API that forms the basis for the Swing drag and drop. We do not discuss that API in this book.

The program in Listing 7-14 demonstrates the behavior. As you run the program, note these points:

- You can select multiple items in the list, table, or tree and drag them.
- Dragging items from the table is a bit awkward. You first select with the mouse, then you let go of the mous button, then click it again, and then you drag.
- When you drop the items in the text area, you can see how the dragged information is formatted. Table cells are separated by tabs, and each selected row is on a separate line (see Figure 7-44).

Figure 7-44. The Swing drag-and-drop test program

[View full size image]



- You can only copy, not move, items, from the list, table, tree, file chooser, or color chooser. Removing items from a list, table, or tree is not possible with all data models. You will see in the next section how to implement this capability when the data model is editable.
- You cannot drag into the list, table, tree, or file chooser.
- If you run two copies of the program, you can drag a color from one color chooser to the other.
- You cannot drag text out of the text area because we didn't call `setDragEnabled` on it.

The Swing package provides a potentially useful mechanism to quickly turn a component into a drag source and drop target. You can install a *transfer handler* for a given property. For example, in our sample program, we call

```
textField.setTransferHandler(new TransferHandler("background"));
```

You can now drag a color into the text field, and its background color changes.

When a drop occurs, then the transfer handler checks whether one of the data flavors has representation class

`Color` . If so, it invokes the `setBackground` method.

By installing this transfer handler into the text field, you disable the standard transfer handler. You can no longer cut, copy, paste, drag, or drop text in the text field. However, you can now drag color out of this text field. You sti need to select some text to initiate the drag gesture. When you drag the text, you will find that you can drop it int the color chooser and change its color value to the text field's background color. However, you cannot drop the te› into the text area.

Listing 7-14. **SwingDnDTest.java**

Code View:

```
import java.awt.*;

import javax.swing.*;
import javax.swing.border.*;
import javax.swing.event.*;

/**
 * This program demonstrates the basic Swing support for drag and drop.
 * @version 1.10 2007-09-20
 * @author Cay Horstmann
 */
public class SwingDnDTest
{
   public static void main(String[] args)
   {
      EventQueue.invokeLater(new Runnable()
         {
            public void run()
            {
               JFrame frame = new SwingDnDFrame();
               frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
               frame.setVisible(true);
            }
         });
   }
}

class SwingDnDFrame extends JFrame
{
   public SwingDnDFrame()
   {
      setTitle("SwingDnDTest");
      JTabbedPane tabbedPane = new JTabbedPane();

      JList list = SampleComponents.list();
      tabbedPane.addTab("List", list);
      JTable table = SampleComponents.table();
      tabbedPane.addTab("Table", table);
      JTree tree = SampleComponents.tree();
      tabbedPane.addTab("Tree", tree);
      JFileChooser fileChooser = new JFileChooser();
      tabbedPane.addTab("File Chooser", fileChooser);
      JColorChooser colorChooser = new JColorChooser();
      tabbedPane.addTab("Color Chooser", colorChooser);

      final JTextArea textArea = new JTextArea(4, 40);
```

```
      JScrollPane scrollPane = new JScrollPane(textArea);
      scrollPane.setBorder(new TitledBorder(new EtchedBorder(), "Drag text here"));

      JTextField textField = new JTextField("Drag color here");
      textField.setTransferHandler(new TransferHandler("background"));

      tabbedPane.addChangeListener(new ChangeListener()
         {
            public void stateChanged(ChangeEvent e)
            {
               textArea.setText("");
            }
         });

      tree.setDragEnabled(true);
      table.setDragEnabled(true);
      list.setDragEnabled(true);
      fileChooser.setDragEnabled(true);
      colorChooser.setDragEnabled(true);
      textField.setDragEnabled(true);

      add(tabbedPane, BorderLayout.NORTH);
      add(scrollPane, BorderLayout.CENTER);
      add(textField, BorderLayout.SOUTH);
      pack();
   }
}
```


UNREGISTERED VERSION OF CHM TO PDF CONVERTER PRO BY THETA-SOFTWARE

API `javax.swing.JComponent` 1.2

- `void setTransferHandler(TransferHandler handler)` **1.4**

  sets a transfer handler to handle data transfer operations (cut, copy, paste, drag, drop).

API `javax.swing.TransferHandler` 1.4

- `TransferHandler(String propertyName)`

  constructs a transfer handler that reads or writes the JavaBeans component property with the given name when a data transfer operation is executed.

API

javx.swing.JFileChooser 1.2
javax.swing.JColorChooser 1.2
javax.swing.JTextComponent 1.2
javax.swing.JList 1.2
javax.swing.JTable 1.2
javax.swing.JTree 1.2

- `void setDragEnabled(boolean b)` **1.4**

  enables or disables dragging of data out of this component.

Drag Sources

In the previous section, you saw how to take advantage of the basic drag-and-drop support in Swing. In this section, we show you how to configure any component as a drag source. In the next section, we discuss drop targets and present a sample component that is both a source and a target for images.

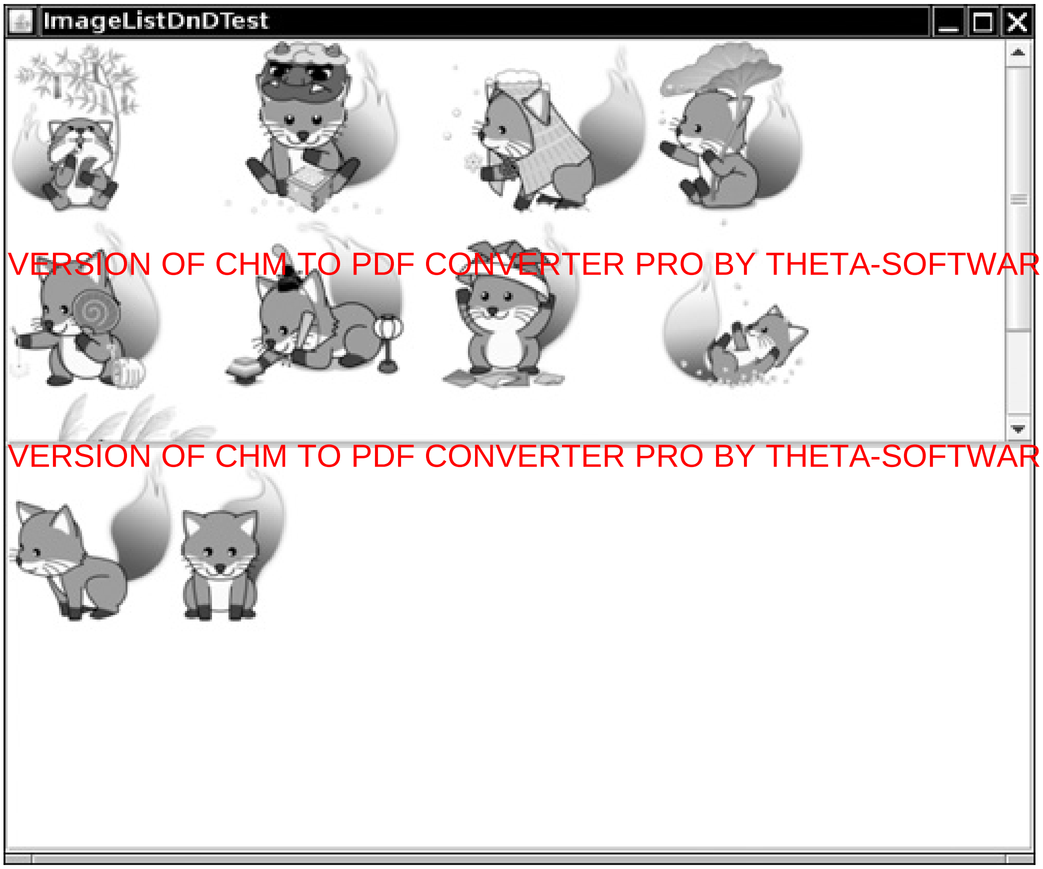
To customize the drag-and-drop behavior of a Swing component, you subclass the `TransferHandler` class. First, override the `getSourceActions` method to indicate which actions (copy, move, link) your component supports. Next, override the `getTransferable` method that produces a `Transferable` object, following the same process tha you use for copying to the clipboard.

In our sample program, we drag images out of a `JList` that is filled with image icons (see Figure 7-45). Here is th implementation of the `createTransferable` method. The selected image is simply placed into an `ImageTransferable` wrapper.

```
protected Transferable createTransferable(JComponent source)
{
   JList list = (JList) source;
   int index = list.getSelectedIndex();
   if (index < 0) return null;
   ImageIcon icon = (ImageIcon) list.getModel().getElementAt(index);
   return new ImageTransferable(icon.getImage());
}
```

Figure 7-45. The ImageList drag-and-drop application
"Foxkeh" 2006 Mozilla Japan.

[View full size image]



UNREGISTERED VERSION OF CHM TO PDF CONVERTER PRO BY THETA-SOFTWARE

In our example, we are fortunate that a `JList` is already wired for initiating a drag gesture. You simply activate th mechanism by calling the `setDragEnabled` method. If you add drag support to a component that does not recognize a drag gesture, you need to initiate the transfer yourself. For example, here is how you can initiate dragging on a `JLabel` :

Code View:
```
label.addMouseListener(new MouseAdapter()
{
   public void mousePressed(MouseEvent evt)
   {
      int mode;
      if ((evt.getModifiers() & (InputEvent.CTRL_MASK | InputEvent.SHIFT_MASK)) != 0)
               mode = TransferHandler.COPY;
      else mode = TransferHandler.MOVE;
      JComponent comp = (JComponent) evt.getSource();
      TransferHandler th = comp.getTransferHandler();
      th.exportAsDrag(comp, evt, mode);
   }
});
```

Here, we simply start the transfer when the user clicks on the label. A more sophisticated implementation would watch for a mouse motion that drags the mouse by a small amount.

When the user completes the drop action, the exportDone method of the source transfer handler is invoked. In tha method, you need to remove the transferred object if the user carried out a move action. Here is the implementation for the image list:

```
protected void exportDone(JComponent source, Transferable data, int action)
{
   if (action == MOVE)
   {
      JList list = (JList) source;
      int index = list.getSelectedIndex();
      if (index < 0) return;
      DefaultListModel model = (DefaultListModel) list.getModel();
      model.remove(index);
   }
}
```

To summarize, to turn a component into a drag source, you add a transfer handler that specifies the following:

- Which actions are supported.
- Which data is transferred.
- And how the original data is removed after a move action.

In addition, if your drag source is a component other than those listed in Table 7-7 on page 654 , you need to wat for a mouse gesture and initiate the transfer.

API javax.swing.TransferHandler 1.4

- int getSourceActions(JComponent c)

  override to return the allowable source actions (bitwise or combination of COPY , MOVE , and LINK) when dragging from the given component.

- protected Transferable createTransferable(JComponent source)

  override to create the Transferable for the data that is to be dragged.

- void exportAsDrag(JComponent comp, InputEvent e, int action)

  starts a drag gesture from the given component. The action is COPY , MOVE , or LINK .

- protected void exportDone(JComponent source, Transferable data, int action) override to adjust the drag source after a successful transfer.

Drop Targets

In this section, we show you how to implement a drop target. Our example is again a JList with image icons. We add drop support so that users can drop images into the list.

To make a component into a drop target, you set a TransferHandler and implement the canImport and importData methods.

Note

As of Java SE 6, you can add a transfer handler to a JFrame . This is most commonly used for dropping files into an application. Valid drop locations include the frame decorations and the menu bar, but not components contained in the frame (which have their own transfer handlers).

The canImport method is called continuously as the user moves the mouse over the drop target component. Retu true if a drop is allowed. This information affects the cursor icon that gives visual feedback whether the drop is allowed.

As of Java SE 6, the canImport method has a parameter of type TransferHandler.TransferSupport . Through th parameter, you can obtain the drop action chosen by the user, the drop location, and the data to be transferred. (Before Java SE 6, a different canImport method was called that only supplies a list of data flavors.)

In the canImport method, you can also override the user drop action. For example, if a user chose the move actio but it would be inappropriate to remove the original, you can force the transfer handler to use a copy action instead.

Here is a typical example. The image list component is willing to accept drops of file lists and images. However, if file list is dragged into the component, then a user-selected MOVE action is changed into a COPY action, so that the image files do not get deleted.

```
public boolean canImport(TransferSupport support)
{
   if (support.isDataFlavorSupported(DataFlavor.javaFileListFlavor))
   {
      if (support.getUserDropAction() == MOVE) support.setDropAction(COPY);
      return true;
   }
   else return support.isDataFlavorSupported(DataFlavor.imageFlavor);
}
```

A more sophisticated implementation could check that the files actually contain images.

The Swing components JList , JTable , JTree , and JTextComponent give visual feedback about insertion positio as the mouse is moved over the drop target. By default, the selection (for JList , JTable , and JTree) or the car (for JTextComponent) is used to indicate the drop location. That approach is neither user-friendly nor flexible, and it is the default solely for backward compatibility. You should call the setDropMode method to choose a more appropriate visual feedback.

You can control whether the dropped data should overwrite existing items or be inserted between them. For example, in our sample program, we call

```
setDropMode(DropMode.ON_OR_INSERT);
```

to allow the user to drop onto an item (thereby replacing it), or to insert between two items (see Figure 7-46). Table 7-8 shows the drop modes supported by the Swing components.

Figure 7-46. Visual indicators for dropping onto an item and between two items
"Foxkeh" 2006 Mozilla Japan.

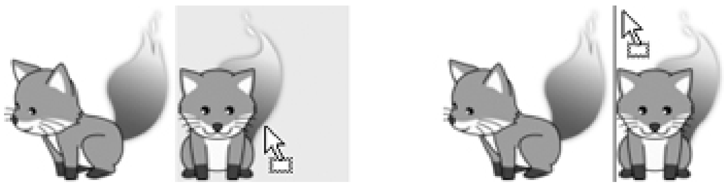


Table 7-8. Drop Modes

Component	Supported Drop Modes
JList , JTree	ON , INSERT , ON_OR_INSERT , USE_SELECTION
JTable	ON , INSERT , ON_OR_INSERT , INSERT_ROWS , INSERT_COLS , ON_OR_INSERT_ROWS , ON_OR_INSERT_COLS , USE_SELECTION
JTextComponent	INSERT , USE_SELECTION (actually moves the caret, not the selection)

Once the user completes the drop gesture, the importData method is invoked. You need to obtain the data from the drag source. Invoke the getTransferable method on the TransferSupport parameter to obtain a reference to a Transferable object. This is the same interface that is used for copy and paste.

One data type that is commonly used for drag and drop is the DataFlavor.javaFileListFlavor . A file list describes a set of files that is dropped onto the target. The transfer data is an object of type List<File> . Here is the code for retrieving the files:

Code View:
```
DataFlavor[] flavors = transferable.getTransferDataFlavors();
if (Arrays.asList(flavors).contains(DataFlavor.javaFileListFlavor))
{
   List<File> fileList = (List<File>) transferable.getTransferData(DataFlavor.javaFileListFlavor
   for (File f : fileList)
   {
      do something with f;
   }
}
```

When dropping into one of the components listed in Table 7-8 , you need to know precisely where to drop the data
Invoke the getDropLocation method on the TransferSupport parameter to find where the drop occurred. This method returns an object of a subclass of TransferHandler.DropLocation . The JList , JTable , JTree , and JTextComponent classes define subclasses that specify location in the particular data model. For example, a locatio in a list is simply an integer index, but a location in a tree is a tree path. Here is how we obtain the drop location i our image list:

Code View:
```
int index;
if (support.isDrop())
{
   JList.DropLocation location = (JList.DropLocation) support.getDropLocation();
   index = location.getIndex();
}
else index = model.size();
```


The JList.DropLocation subclass has a method getIndex that returns the index of the drop. (The JTree.DropLocation subclass has a method getPath instead.)

The importData method is also called when data is pasted into the component with the CTRL+V keystroke. In that case, the getDropLocation method would throw an IllegalStateException . Therefore, if the isDrop method returns false , we simply append the pasted data to the end of the list.

When inserting into a list, table, or tree, you also need to check whether the data is supposed to be inserted between items or whether it should replace the item at the drop location. For a list, invoke the isInsert method o the JList.DropLocation . For the other components, see the API notes for their drop location classes at the end o this section.

To summarize, to turn a component into a drop target, you add a transfer handler that specifies the following:

- When a dragged item can be accepted.
- How the dropped data is imported.

In addition, if you add drop support to a JList , JTable , JTree , or JTextComponent , you should set the drop mode.

Listing 7-15 shows the complete program. Note that the ImageList class is both a drag source and a drop target. Try dragging images between the two lists. You can also drag image files from a file chooser of another program into the lists.

Listing 7-15. **ImageListDragDrop.java**

Code View:
```
import java.awt.*;
import java.awt.datatransfer.*;
import java.io.*;
```

```
import java.util.*;
import javax.imageio.*;
import javax.swing.*;
import java.util.List;

/**
 * This program demonstrates drag and drop in an image list.
 * @version 1.00 2007-09-20
 * @author Cay Horstmann
 */
public class ImageListDnDTest
{
   public static void main(String[] args)
   {
      EventQueue.invokeLater(new Runnable()
         {
            public void run()
            {
               JFrame frame = new ImageListDnDFrame();
               frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
               frame.setVisible(true);
            }
         });
   }
}

class ImageListDnDFrame extends JFrame
{
   public ImageListDnDFrame()
   {
      setTitle("ImageListDnDTest");
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

      list1 = new ImageList(new File("images1").listFiles());
      list2 = new ImageList(new File("images2").listFiles());
      setLayout(new GridLayout(2, 1));
      add(new JScrollPane(list1));
      add(new JScrollPane(list2));
   }

   private ImageList list1;
   private ImageList list2;
   private static final int DEFAULT_WIDTH = 600;
   private static final int DEFAULT_HEIGHT = 500;
}

class ImageList extends JList
{
   public ImageList(File[] imageFiles)
   {
      DefaultListModel model = new DefaultListModel();
      for (File f : imageFiles)
         model.addElement(new ImageIcon(f.getPath()));

      setModel(model);
      setVisibleRowCount(0);
      setLayoutOrientation(JList.HORIZONTAL_WRAP);
      setDragEnabled(true);
      setDropMode(DropMode.ON_OR_INSERT);
```

```java
      setTransferHandler(new ImageListTransferHandler());
   }
}

class ImageListTransferHandler extends TransferHandler
{
   // Support for drag

   public int getSourceActions(JComponent source)
   {
      return COPY_OR_MOVE;
   }

   protected Transferable createTransferable(JComponent source)
   {
      JList list = (JList) source;
      int index = list.getSelectedIndex();
      if (index < 0) return null;
      ImageIcon icon = (ImageIcon) list.getModel().getElementAt(index);
      return new ImageTransferable(icon.getImage());
   }

   protected void exportDone(JComponent source, Transferable data, int action)
   {
      if (action == MOVE)
      {
         JList list = (JList) source;
         int index = list.getSelectedIndex();
         if (index < 0) return;
         DefaultListModel model = (DefaultListModel) list.getModel();
         model.remove(index);
      }
   }

   // Support for drop

   public boolean canImport(TransferSupport support)
   {
      if (support.isDataFlavorSupported(DataFlavor.javaFileListFlavor))
      {
         if (support.getUserDropAction() == MOVE) support.setDropAction(COPY);
         return true;
      }
      else return support.isDataFlavorSupported(DataFlavor.imageFlavor);
   }

   public boolean importData(TransferSupport support)
   {
      JList list = (JList) support.getComponent();
      DefaultListModel model = (DefaultListModel) list.getModel();

      Transferable transferable = support.getTransferable();
      List<DataFlavor> flavors = Arrays.asList(transferable.getTransferDataFlavors());

      List<Image> images = new ArrayList<Image>();

      try
      {
         if (flavors.contains(DataFlavor.javaFileListFlavor))
```

```
         {
            List<File> fileList = (List<File>) transferable
                  .getTransferData(DataFlavor.javaFileListFlavor);
            for (File f : fileList)
            {
               try
               {
                  images.add(ImageIO.read(f));
               }
               catch (IOException ex)
               {
                  // couldn't read image--skip
               }
            }
         }
         else if (flavors.contains(DataFlavor.imageFlavor))
         {
            images.add((Image) transferable.getTransferData(DataFlavor.imageFlavor));
         }

         int index;
         if (support.isDrop())
         {
            JList.DropLocation location = (JList.DropLocation) support.getDropLocation();
            index = location.getIndex();
            if (!location.isInsert()) model.remove(index); // replace location
         }
         else index = model.size();
         for (Image image : images)
         {
            model.add(index, new ImageIcon(image));
            index++;
         }
         return true;
      }
      catch (IOException ex)
      {
         return false;
      }
      catch (UnsupportedFlavorException ex)
      {
         return false;
      }
   }
}
```

javax.swing.TransferHandler 1.4

- `boolean canImport(TransferSupport support)` **6**

  override to indicate whether the target component can accept the drag described by the `TransferSupport` parameter

- `boolean importData(TransferSupport support)` **6**

  override to carry out the drop or paste gesture described by the `TransferSupport` parameter, and return `true` if the import was successful

javax.swing.JFrame 1.2

- `void setTransferHandler(TransferHandler handler)` **6**

  sets a transfer handler to handle drop and paste operations only

javax.swing.JList 1.2
javax.swing.JTable 1.2
javax.swing.JTree 1.2
javax.swing.JTextComponent 1.2

- `void setDropMode(DropMode mode)` **6**

  set the drop mode of this component to one of the values specified in Table 7-8 .

API

`javax.swing.TransferHandler.TransferSupport` 6

- `Component getComponent()`

  gets the target component of this transfer.

- `DataFlavor[] getDataFlavors()`

  gets the data flavors of the data to be transferred.

- `boolean isDrop()`

  `true` if this transfer is a drop, `false` if it is a paste.

- `int getUserDropAction()`

  gets the drop action chosen by the user (`MOVE` , `COPY` , or `LINK`).

- `getSourceDropActions()`

  gets the drop actions that are allowed by the drag source.

- `getDropAction()`

- `setDropAction()`

  gets or sets the drop action of this transfer. Initially, this is the user drop action, but it can be overridden by the transfer handler.

- `DropLocation getDropLocation()`

  gets the location of the drop, or throws an `IllegalStateException` if this transfer is not a drop.

API

`javax.swing.TransferHandler.DropLocation` 6

- `Point getDropPoint()`

  gets the mouse location of the drop in the target component.

API **javax.swing.JList.DropLocation 6**

- `boolean isInsert()`

  returns `true` if the data are to be inserted before a given location, `false` if they are to replace existing data.

- `int getIndex()`

  gets the model index for the insertion or replacement.

API **javax.swing.JTable.DropLocation 6**

- `boolean isInsertRow()`
- `boolean isInsertColumn()`

  returns `true` if data are to be inserted before a row or column.

- `int getRow()`
- `int getColumn()`

  gets the model row or column index for the insertion or replacement, or -1 if the drop occurred in an empty area.

API `javax.swing.JTree.DropLocation` 6

- `TreePath getPath()`
- `int getChildIndex()`

  returns the tree path and child that, together with the drop mode of the target component, define the drop location, as described in Table 7-9 .

  Table 7-9. Drop Location Handling in **JTree**

Drop Mode	Tree Edit Action
`INSERT`	Insert as child of the path, before the child index.
`ON` or `USE_SELECTION`	Replace the data of the path (child index not used).
`INSERT_OR_ON`	If the child index is -1, do as in `ON` , otherwise as in `INSERT` .

API `javax.swing.JTextComponent.DropLocation` 6

- `int getIndex()`

  the index at which to insert the data.

Platform Integration

We finish this chapter with several features that were added to Java SE 6 to make Java applications feel more like native applications. The splash screen feature allows your application to display a splash screen as the virtual machine starts up. The `java.awt.Desktop` class lets you launch native applications such as the default browser and e-mail program. Finally, you now have access to the system tray and can clutter it up with icons, just like so many native applications do.

Splash Screens

A common complaint about Java applications is their long startup time. The Java virtual machine takes some time to load all required classes, particularly for a Swing application that needs to pull in large amounts of Swing and AWT library code. Users dislike applications that take a long time to bring up an initial screen, and they might even try launching the application multiple times if they don't know whether the first launch was successful. The remedy is a *splash screen*, a small window that appears quickly, telling the user that the application has been launched successfully.

UNREGISTERED VERSION OF CHM TO PDF CONVERTER PRO BY THETA-SOFTWARE

Traditionally, this has been difficult for Java applications. Of course, you can put up a window as soon as your `main` method starts. However, the `main` method is only launched after the class loader has loaded all dependent classes, which might take a while.

Java SE 6 solves this problem by enabling the virtual machine to show an image immediately on launch. There are two mechanisms for specifying that image. You can use the `-splash` command-line option:

```
java -splash:myimage.png MyApp
```

Alternatively, you can specify it in the manifest of a JAR file:

```
Main-Class: MyApp
SplashScreen-Image: myimage.gif
```

The image is displayed immediately and automatically disappears when the first AWT window is made visible. You can supply any GIF, JPEG, or PNG image. Animation (in GIF) and transparency (GIF and PNG) are supported.

If your application is ready to go as soon as it reaches `main`, you can skip the remainder of this section. However, many applications use a plug-in architecture in which a small core loads a set of plugins at startup. Eclipse and NetBeans are typical examples. In that case, you can indicate the loading progress on the splash screen.

There are two approaches. You can draw directly on the splash screen, or you can replace it with a borderless frame with identical contents, and then draw inside the frame. Our sample program shows both techniques.

To draw directly on the splash screen, get a reference to the splash screen and get its graphics context and dimensions:

```
SplashScreen splash = SplashScreen.getSplashScreen();
Graphics2D g2 = splash.createGraphics();
Rectangle bounds = splash.getBounds();
```

You can now draw in the usual way. When you are done, call `update` to ensure that the drawing is refreshed.

Our sample program draws a simple progress bar, as seen in the left image in Figure 7-47.

```
g.fillRect(x, y, width * percent / 100, height);
splash.update();
```

Figure 7-47. The initial splash screen and a borderless follow-up window

Note

The splash screen is a singleton object. You cannot construct your own. If no splash screen was set on the command line or in the manifest, the `getSplashScreen` method returns null.

Drawing directly on the splash screen has a drawback. It is tedious to compute all pixel positions, and your progress indicator won't match the native progress bar. To avoid these problems, you can replace the initial splash screen with a follow-up window of the same size and content as soon as the `main` method starts. That window can contain arbitrary Swing components.

Our sample program in Listing 7-16 demonstrates this technique. The right image in Figure 7-47 shows a borderless frame with a panel that paints the splash screen and contains a `JProgressBar`. Now we have full access to the Swing API and can easily add message strings without having to fuss with pixel positions.

Note that we do not need to remove the initial splash screen. It is automatically removed as soon as the follow-up window is made visible.

Caution

Unfortunately, there is a noticeable flash when the splash screen is replaced by the follow-up window.

Listing 7-16. **`SplashScreenTest.java`**

Code View:

```
import java.awt.*;
import java.util.List;
import javax.swing.*;

/**
 * This program demonstrates the splash screen API.
 * @version 1.00 2007-09-21
 * @author Cay Horstmann
 */
public class SplashScreenTest
{
   private static void drawOnSplash(int percent)
   {
      Rectangle bounds = splash.getBounds();
      Graphics2D g = splash.createGraphics();
      int height = 20;
      int x = 2;
      int y = bounds.height - height - 2;
      int width = bounds.width - 4;
      Color brightPurple = new Color(76, 36, 121);
      g.setColor(brightPurple);
      g.fillRect(x, y, width * percent / 100, height);
      splash.update();
```

```
    }

    /**
     * This method draws on the splash screen.
     */
    private static void init1()
    {
       splash = SplashScreen.getSplashScreen();
       if (splash == null)
       {
          System.err.println("Did you specify a splash image with -splash or in the manifest?");
          System.exit(1);
       }

       try
       {
          for (int i = 0; i <= 100; i++)
          {
             drawOnSplash(i);
             Thread.sleep(100); // simulate startup work
          }
       }
       catch (InterruptedException e)
       {
       }
    }

   /**
    * This method displays a frame with the same image as the splash screen.
    */
   private static void init2()
   {
      final Image img = Toolkit.getDefaultToolkit().getImage(splash.getImageURL());

      final JFrame splashFrame = new JFrame();
      splashFrame.setUndecorated(true);

      final JPanel splashPanel = new JPanel()
         {
            public void paintComponent(Graphics g)
            {
               g.drawImage(img, 0, 0, null);
            }
         };

      final JProgressBar progressBar = new JProgressBar();
      progressBar.setStringPainted(true);
      splashPanel.setLayout(new BorderLayout());
      splashPanel.add(progressBar, BorderLayout.SOUTH);

      splashFrame.add(splashPanel);
      splashFrame.setBounds(splash.getBounds());
      splashFrame.setVisible(true);

      new SwingWorker<Void, Integer>()
      {
         protected Void doInBackground() throws Exception
         {
            try
```

```java
            {
               for (int i = 0; i <= 100; i++)
               {
                  publish(i);
                  Thread.sleep(100);
               }
            }
            catch (InterruptedException e)
            {
```

```java
            }
            return null;
         }

         protected void process(List<Integer> chunks)
         {
            for (Integer chunk : chunks)
```

```java
            {
               progressBar.setString("Loading module " + chunk);
               progressBar.setValue(chunk);
               splashPanel.repaint(); // because img is loaded asynchronously
            }
         }

         protected void done()
         {
            splashFrame.setVisible(false);

            JFrame frame = new JFrame();
            frame.setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
            frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            frame.setTitle("SplashScreenTest");
            frame.setVisible(true);
         }
      }.execute();
   }

   public static void main(String args[])
   {
      init1();

      EventQueue.invokeLater(new Runnable()
         {
            public void run()
            {
               init2();
            }
         });
   }

   private static SplashScreen splash;
   private static final int DEFAULT_WIDTH = 300;
   private static final int DEFAULT_HEIGHT = 300;
}
```

API

java.awt.SplashScreen 6

- `static SplashScreen getSplashScreen()`

  gets a reference to the splash screen, or `null` if no splash screen is present.

- `URL getImageURL()`

- `void setImageURL(URL imageURL)`

  gets or sets the URL of the splash screen image. Setting the image updates the splash screen.

- `Rectangle getBounds()`

  gets the bounds of the splash screen.

- `Graphics2D createGraphics()`

  gets a graphics context for drawing on the splash screen.

- `void update()`

  updates the display of the splash screen.

- `void close()`

  closes the splash screen. The splash screen is automatically closed when the first AWT window is made visible.

Launching Desktop Applications

The `java.awt.Desktop` class lets you launch the default browser and e-mail program. You can also open, edit, and print files, using the applications that are registered for the file type.

The API is very straightforward. First, call the static `isDesktopSupported` method. If it returns `true`, the current platform supports the launching of desktop applications. Then call the static `getDesktop` method to obtain a `Desktop` instance.

Not all desktop environments support all API operations. For example, in the Gnome desktop on Linux, it is possible to open files, but you cannot print them. (There is no support for "verbs" in file associations.) To find out what is supported on your platform, call the `isSupported` method, passing a value in the `Desktop.Action` enumeration. Our sample program contains tests such as the following:

Code View:
```
if (desktop.isSupported(Desktop.Action.PRINT)) printButton.setEnabled(true);
```

To open, edit, or print a file, first check that the action is supported, and then call the `open`, `edit`, or `print` method. To launch the browser, pass a `URI`. (See Chapter 3 for more information on URIs.) You can simply call the `URI` constructor with a string containing an `http` or `https` URL.

To launch the default e-mail program, you need to construct a `URI` of a particular format, namely

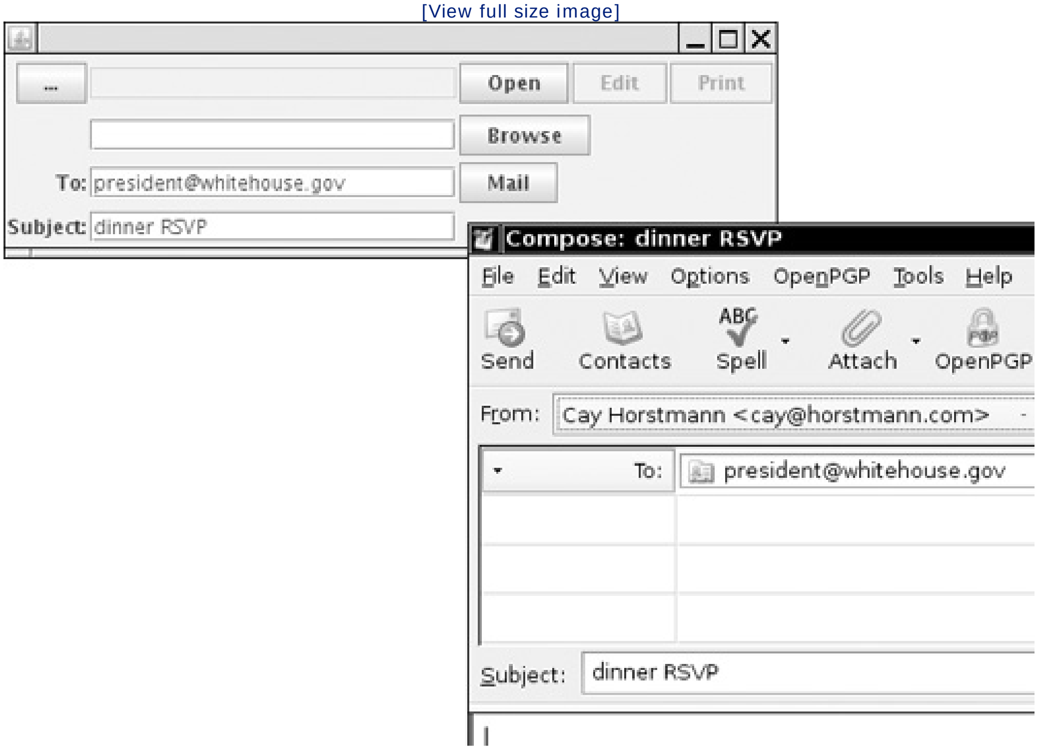
`mailto:`*recipient*`?`*query*

Here *recipient* is the e-mail address of the recipient, such as president@whitehouse.gov, and *query* contains `&`-separated *name=value* pairs, with percent-encoded values. (Percent encoding is essentially the same as the URL encoding algorithm described in Chapter 3, but a space is encoded as `%20`, not `+`). An example is `subject=dinner%20RSVP&bcc=putin%40kremvax.ru`. The format is documented in RFC 2368 (http://www.ietf.org/rfc/rfc2368.txt). Unfortunately, the `URI` class does not know anything about `mailto` URIs, so you have to assemble and encode your own. To make matters worse, at the time of this writing, there is no standard for dealing with non-ASCII characters. A common approach (which we take as well) is to convert each character to UTF-8 and percent-encode the resulting bytes.

Our sample program in Listing 7-17 lets you open, edit, or print a file of your choice, browse a URL, or launch your e-mail program (see Figure 7-48).

Figure 7-48. Launching a desktop application

[View full size image]



Listing 7-17. **DesktopAppTest.java**

Code View:

```
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.net.*;
import javax.swing.*;

/**
 * This program demonstrates the desktop app API.
 * @version 1.00 2007-09-22
 * @author Cay Horstmann
 */
public class DesktopAppTest
{
   public static void main(String[] args)
   {
      EventQueue.invokeLater(new Runnable()
         {
            public void run()
            {
               JFrame frame = new DesktopAppFrame();
               frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
               frame.setVisible(true);
            }
         });
   }
}

class DesktopAppFrame extends JFrame
{
   public DesktopAppFrame()
   {
      setLayout(new GridBagLayout());
      final JFileChooser chooser = new JFileChooser();
      JButton fileChooserButton = new JButton("...");
      final JTextField fileField = new JTextField(20);
      fileField.setEditable(false);
      JButton openButton = new JButton("Open");
      JButton editButton = new JButton("Edit");
      JButton printButton = new JButton("Print");
      final JTextField browseField = new JTextField();
      JButton browseButton = new JButton("Browse");
      final JTextField toField = new JTextField();
      final JTextField subjectField = new JTextField();
      JButton mailButton = new JButton("Mail");

      openButton.setEnabled(false);
      editButton.setEnabled(false);
      printButton.setEnabled(false);
      browseButton.setEnabled(false);
      mailButton.setEnabled(false);

      if (Desktop.isDesktopSupported())
      {
         Desktop desktop = Desktop.getDesktop();
         if (desktop.isSupported(Desktop.Action.OPEN)) openButton.setEnabled(true);
```

```
   if (desktop.isSupported(Desktop.Action.EDIT)) editButton.setEnabled(true);
   if (desktop.isSupported(Desktop.Action.PRINT)) printButton.setEnabled(true);
   if (desktop.isSupported(Desktop.Action.BROWSE)) browseButton.setEnabled(true);
   if (desktop.isSupported(Desktop.Action.MAIL)) mailButton.setEnabled(true);
}

fileChooserButton.addActionListener(new ActionListener()
   {
      public void actionPerformed(ActionEvent e)
```

```
      {
         if (chooser.showOpenDialog(DesktopAppFrame.this) ==
               JFileChooser.APPROVE_OPTION)
            fileField.setText(chooser.getSelectedFile().getAbsolutePath());
      }
   });

```

```
openButton.addActionListener(new ActionListener()
   {
      public void actionPerformed(ActionEvent e)
      {
         try
         {
            Desktop.getDesktop().open(chooser.getSelectedFile());
         }
         catch (IOException ex)
         {
            ex.printStackTrace();
         }
      }
   });

editButton.addActionListener(new ActionListener()
   {
      public void actionPerformed(ActionEvent e)
      {
         try
         {
            Desktop.getDesktop().edit(chooser.getSelectedFile());
         }
         catch (IOException ex)
         {
            ex.printStackTrace();
         }
      }
   });

printButton.addActionListener(new ActionListener()
   {
      public void actionPerformed(ActionEvent e)
      {
         try
         {
            Desktop.getDesktop().print(chooser.getSelectedFile());
         }
         catch (IOException ex)
         {
            ex.printStackTrace();
         }
      }
```

```
         });

      browseButton.addActionListener(new ActionListener()
         {
            public void actionPerformed(ActionEvent e)
            {
               try
               {
                  Desktop.getDesktop().browse(new URI(browseField.getText()));
               }
               catch (URISyntaxException ex)
               {
                  ex.printStackTrace();
               }
               catch (IOException ex)
               {
                  ex.printStackTrace();
               }
            }
         });

      mailButton.addActionListener(new ActionListener()
         {
            public void actionPerformed(ActionEvent e)
            {
               try
               {
                  String subject = percentEncode(subjectField.getText());
                  URI uri = new URI("mailto:" + toField.getText() + "?subject=" + subject);

                  System.out.println(uri);
                  Desktop.getDesktop().mail(uri);
               }
               catch (URISyntaxException ex)
               {
                  ex.printStackTrace();
               }
               catch (IOException ex)
               {
                  ex.printStackTrace();
               }
            }
         });

      JPanel buttonPanel = new JPanel();
      ((FlowLayout) buttonPanel.getLayout()).setHgap(2);
      buttonPanel.add(openButton);
      buttonPanel.add(editButton);
      buttonPanel.add(printButton);

      add(fileChooserButton, new GBC(0, 0).setAnchor(GBC.EAST).setInsets(2));
      add(fileField, new GBC(1, 0).setFill(GBC.HORIZONTAL));
      add(buttonPanel, new GBC(2, 0).setAnchor(GBC.WEST).setInsets(0));
      add(browseField, new GBC(1, 1).setFill(GBC.HORIZONTAL));
      add(browseButton, new GBC(2, 1).setAnchor(GBC.WEST).setInsets(2));
      add(new JLabel("To:"), new GBC(0, 2).setAnchor(GBC.EAST).setInsets(5, 2, 5, 2));
      add(toField, new GBC(1, 2).setFill(GBC.HORIZONTAL));
      add(mailButton, new GBC(2, 2).setAnchor(GBC.WEST).setInsets(2));
      add(new JLabel("Subject:"), new GBC(0, 3).setAnchor(GBC.EAST).setInsets(5, 2, 5, 2));
```

```
      add(subjectField, new GBC(1, 3).setFill(GBC.HORIZONTAL));

      pack();
   }

   private static String percentEncode(String s)
   {
      try
      {
         return URLEncoder.encode(s, "UTF-8").replaceAll("[+]", "%20");
      }
      catch (UnsupportedEncodingException ex)
      {
         return null; // UTF-8 is always supported
      }
   }
}
```


java.awt.Desktop 6

- `static boolean isDesktopSupported()`

  returns `true` if launching of desktop applications is supported on this platform.

- `static Desktop getDesktop()`

  returns the `Desktop` object for launching desktop operations. Throws an `UnsupportedOperationException` if this platform does not support launching of desktop operations.

- `boolean isSupported(Desktop.Action action)`

  returns `true` if the given action is supported. `action` is one of `OPEN`, `EDIT`, `PRINT`, `BROWSE`, or `MAIL`.

- `void open(File file)`

  launches the application that is registered for viewing the given file.

- `void edit(File file)`

  launches the application that is registered for editing the given file.

- `void print(File file)`

  prints the given file.

- `void browse(URI uri)`

  launches the default browser on the given URI.

- `void mail()`

- `void mail(URI uri)`

  launches the default mailer. The second version can can be used to fill in parts of the e-mail message.

The System Tray

Many desktop environments have an area for icons of programs that run in the background and occasionally notify users of events. In Windows, this area is called the *system tray*, and the icons are called *tray icons*. The Java API adopts the same terminology. A typical example of such a program is a monitor that checks for software updates. If new software updates are available, the monitor program can change the appearance of the icon or display a message near the icon.

Frankly, the system tray is somewhat overused, and computer users are not usually filled with joy when they discover yet another tray icon. Our sample system tray application—a program that dispenses virtual fortune cookies—is no exception to that rule.

The `java.awt.SystemTray` class is the cross-platform conduit to the system tray. Similar to the `Desktop` class discussed in the preceding section, you first call the static `isSupported` method to check that the local Java platform supports the system tray. If so, you get a `SystemTray` singleton by calling the static `getSystemTray` method.

The most important method of the `SystemTray` class is the `add` method that lets you add a `TrayIcon` instance. A tray icon has three key properties:

- The icon image.

- The tooltip that is visible when the mouse hovers over the icon.

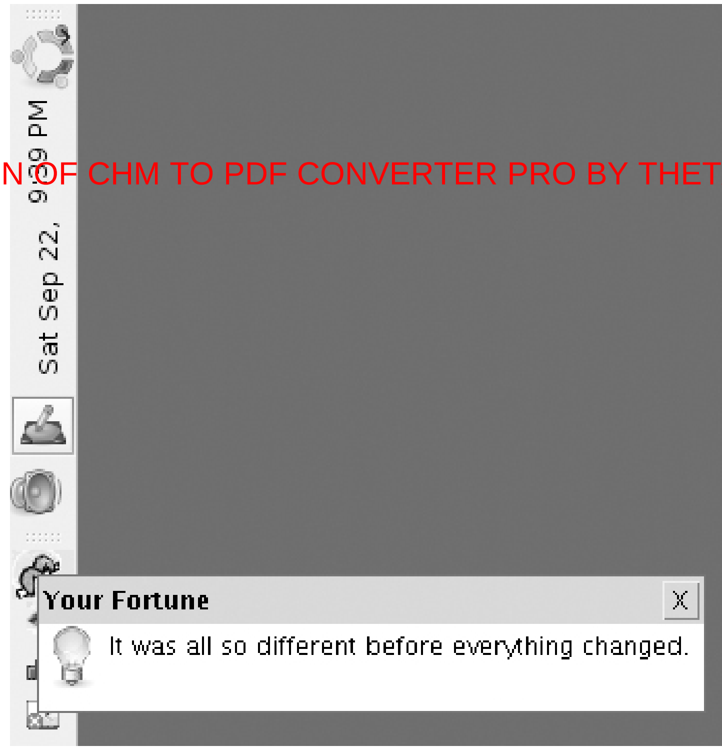
- The pop-up menu that is displayed when the user clicks on the icon with the right mouse button.

The pop-up menu is an instance of the `PopupMenu` class of the AWT library, representing a native pop-up menu, not a Swing menu. You add AWT `MenuItem` instances, each of which has an action listener just like the Swing counterpart.

Finally, a tray icon can display notifications to the user (see Figure 7-49). Call the `displayMessage` method of the `TrayIcon` class and specify the caption, message, and message type.

Code View:
```
trayIcon.displayMessage("Your Fortune", fortunes.get(index), TrayIcon.MessageType.INFO);
```

Figure 7-49. A notification from a tray icon



Listing 7-18 shows the application that places a fortune cookie icon into the system tray. The program reads a fortune cookie file (from the venerable UNIX `fortune` program) in which each fortune is terminated by a line containing a `%` character. It displays a message every ten seconds. Mercifully, there is a pop-up menu with an item to exit the application. If only all tray icons were so considerate!

Listing 7-18. **SystemTrayTest.java**

Code View:
```
import java.awt.*;
import java.util.*;
import java.util.List;
import java.awt.event.*;
import java.io.*;
import javax.swing.Timer;

/**
 * This program demonstrates the system tray API.
 * @version 1.00 2007-09-22
 * @author Cay Horstmann
```

```
 */
public class SystemTrayTest
{
   public static void main(String[] args)
   {
      final TrayIcon trayIcon;

      if (!SystemTray.isSupported())
      {
         System.err.println("System tray is not supported.");
         return;
      }

      SystemTray tray = SystemTray.getSystemTray();
      Image image = Toolkit.getDefaultToolkit().getImage("cookie.png");

      PopupMenu popup = new PopupMenu();
      MenuItem exitItem = new MenuItem("Exit");
      exitItem.addActionListener(new ActionListener()
         {
            public void actionPerformed(ActionEvent e)
            {
               System.exit(0);
            }
         });
      popup.add(exitItem);

      trayIcon = new TrayIcon(image, "Your Fortune", popup);

      trayIcon.setImageAutoSize(true);
      trayIcon.addActionListener(new ActionListener()
         {
            public void actionPerformed(ActionEvent e)
            {
               trayIcon.displayMessage("How do I turn this off?",
                     "Right-click on the fortune cookie and select Exit.",
                     TrayIcon.MessageType.INFO);
            }
         });

      try
      {
         tray.add(trayIcon);
      }
      catch (AWTException e)
      {
         System.err.println("TrayIcon could not be added.");
         return;
      }

      final List<String> fortunes = readFortunes();
      Timer timer = new Timer(10000, new ActionListener()
         {
            public void actionPerformed(ActionEvent e)
            {
               int index = (int) (fortunes.size() * Math.random());
               trayIcon.displayMessage("Your Fortune", fortunes.get(index),
                     TrayIcon.MessageType.INFO);
            }
```

```
         });
      timer.start();
   }

   private static List<String> readFortunes()
   {
      List<String> fortunes = new ArrayList<String>();
      try
```



```
      {
         Scanner in = new Scanner(new File("fortunes"));
         StringBuilder fortune = new StringBuilder();
         while (in.hasNextLine())
         {
            String line = in.nextLine();
            if (line.equals("%"))
```



```
            {
               fortunes.add(fortune.toString());
               fortune = new StringBuilder();
            }
            else
            {
               fortune.append(line);
               fortune.append(' ');
            }
         }
      }
      catch (IOException ex)
      {
        ex.printStackTrace();
      }
      return fortunes;
   }

}
```

java.awt.SystemTray 6

- `static boolean isSupported()`

  returns `true` if system tray access is supported on this platform.

- `static SystemTray getSystemTray()`

  returns the `SystemTray` object for accessing the system tray. Throws an `UnsupportedOperationException` if this platform does not support system tray access.

- `Dimension getTrayIconSize()`

  gets the dimensions for an icon in the system tray.

- `void add(TrayIcon trayIcon)`

- `void remove(TrayIcon trayIcon)`

  adds or removes a system tray icon.

java.awt.TrayIcon 6

- `TrayIcon(Image image)`

- `TrayIcon(Image image, String tooltip)`

- `TrayIcon(Image image, String tooltip, PopupMenu popupMenu)`

  constructs a tray icon with the given image, tooltip, and pop-up menu.

- `Image getImage()`

- `void setImage(Image image)`

- `String getTooltip()`

- `void setTooltip(String tooltip)`
- `PopupMenu getPopupMenu()`
- `void setPopupMenu(PopupMenu popupMenu)`

  gets or sets the image, tooltip, or pop-up menu of this tooltip.

- `boolean isImageAutoSize()`

- `void setImageAutoSize(boolean autosize)`

  gets or sets the `imageAutoSize` property. If set, the image is scaled to fit the tooltip icon area. If not (the default), it is cropped (if too large) or centered (if too small).

- `void displayMessage(String caption, String text, TrayIcon.MessageType messageType)`

  displays a message near the tray icon. The message type is one of `INFO`, `WARNING`, `ERROR`, or `NONE`.

- `public void addActionListener(ActionListener listener)`
- `public void removeActionListener(ActionListener listener)`

  adds or removes an action listener when the listenercalled is platform-dependent. Typical cases are clicking on a notification or double-clicking on the tray icon.

You have now reached the end of this long chapter covering advanced AWT features. In the next chapter, we discuss the JavaBeans specification and its use for GUI builders.

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

The official definition of a bean, as given in the JavaBeans specification, is: "A bean is a reusable software component based on Sun's JavaBeans specification that can be manipulated visually in a builder tool."

Once you implement a bean, others can use it in a builder environment (such as NetBeans). Instead of having to write tedious code, they can simply drop your bean into a GUI form and customize it with dialog boxes.

This chapter explains how you can implement beans so that other developers can use them easily.

Note

We'd like to address a common confusion before going any further: The JavaBeans that we discuss in this chapter have little in common with Enterprise JavaBeans (EJB). Enterprise JavaBeans are server-side components with support for transactions, persistence, replication, and security. At a very basic level, they too are components that can be manipulated in builder tools. However, the Enterprise JavaBeans technology is quite a bit more complex than the "Standard Edition" JavaBeans technology.

That does not mean that standard JavaBeans components are limited to client-side programming. Web technologies such as JavaServer Faces (JSF) and JavaServer Pages (JSP) rely heavily on the JavaBeans component model.

Why Beans?

Programmers with experience in Visual Basic will immediately know why beans are so important. Programmers coming from an environment in which the tradition is to "roll your own" for everything often find it hard to believe that Visual Basic is one of the most successful examples of reusable object technology. For those who have never worked with Visual Basic, here, in a nutshell, is how you build a Visual Basic application:

1. You build the interface by dropping components (called *controls* in Visual Basic) onto a form window.

2. Through *property inspectors,* you set properties of the components such as height, color, or other behavior.

3. The property inspectors also list the events to which components can react. Some events can be hooked up through dialog boxes. For other events, you write short snippets of event handling code.

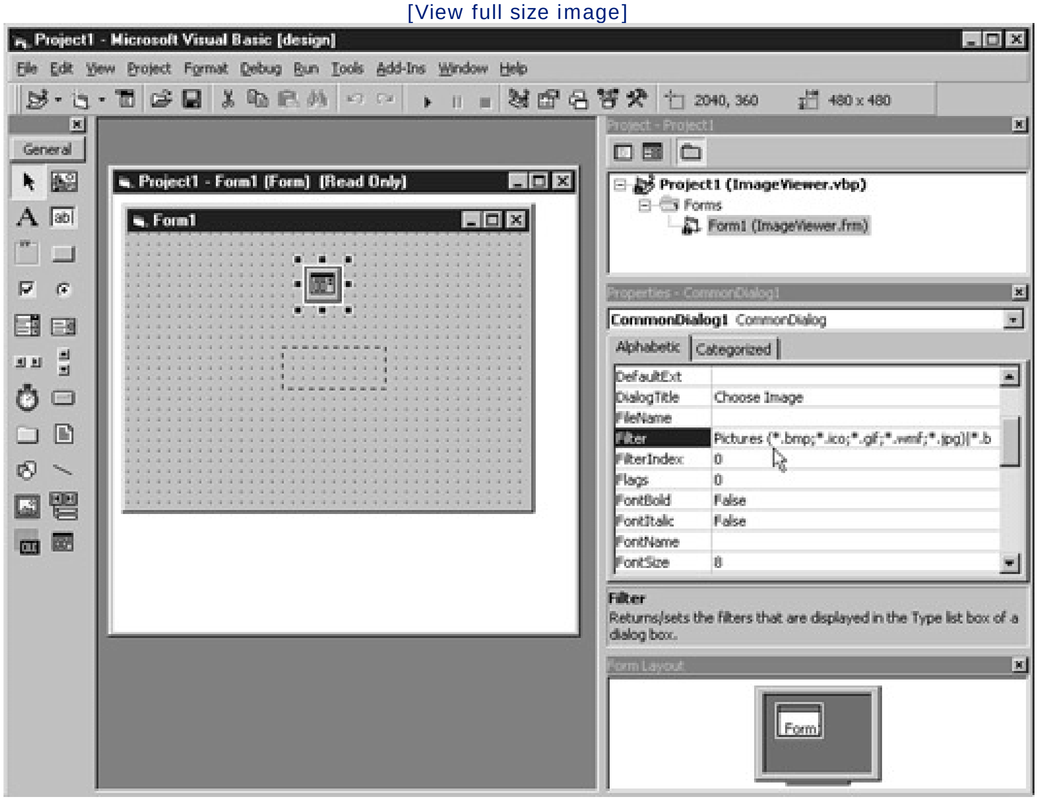
For example, in Volume I, Chapter 2, we wrote a program that displays an image in a frame. It took over a page of code. Here's what you would do in Visual Basic to create a program with pretty much the same functionality:

1. Add two controls to a window: an *Image* control for displaying graphics and a *Common Dialog* control for selecting a file.

2. Set the *Filter* properties of the CommonDialog control so that only files that the Image control can handle will show up, as shown in Figure 8-1.

Figure 8-1. The Properties window in Visual Basic for an image application

[View full size image]



3. Write four lines of Visual Basic code that will be activated when the project first starts running. All the code you need for this sequence looks like this:

```
Private Sub Form_Load()
  CommonDialog1.ShowOpen
  Image1.Picture = LoadPicture(CommonDialog1.FileName)
End Sub
```

The code pops up the file dialog box—but only files with the right extension are shown because of how we set the filter property. After the user selects an image file, the code then tells the Image control to display it.

That's it. The layout activity, combined with these statements, gives essentially the same functionality as a page of Java code. Clearly, it is a lot easier to learn how to drop down components and set properties than it is to write a page of code.

We do not want to imply that Visual Basic is a good solution for every problem. It is clearly optimized for a particular kind of problem—UI-intensive Windows programs. The JavaBeans technology was invented to make Java technology competitive in this arena. It enables vendors to create Visual Basic-style development environments. These environments make it possible to build user interfaces with a minimum of programming.

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

The official definition of a bean, as given in the JavaBeans specification, is: "A bean is a reusable software component based on Sun's JavaBeans specification that can be manipulated visually in a builder tool."

Once you implement a bean, others can use it in a builder environment (such as NetBeans). Instead of having to write tedious code, they can simply drop your bean into a GUI form and customize it with dialog boxes.

This chapter explains how you can implement beans so that other developers can use them easily.

Note

We'd like to address a common confusion before going any further: The JavaBeans that we discuss in this chapter have little in common with Enterprise JavaBeans (EJB). Enterprise JavaBeans are server-side components with support for transactions, persistence, replication, and security. At a very basic level, they too are components that can be manipulated in builder tools. However, the Enterprise JavaBeans technology is quite a bit more complex than the "Standard Edition" JavaBeans technology.

That does not mean that standard JavaBeans components are limited to client-side programming. Web technologies such as JavaServer Faces (JSF) and JavaServer Pages (JSP) rely heavily on the JavaBeans component model.

Why Beans?

Programmers with experience in Visual Basic will immediately know why beans are so important. Programmers coming from an environment in which the tradition is to "roll your own" for everything often find it hard to believe that Visual Basic is one of the most successful examples of reusable object technology. For those who have never worked with Visual Basic, here, in a nutshell, is how you build a Visual Basic application:

1. You build the interface by dropping components (called *controls* in Visual Basic) onto a form window.

2. Through *property inspectors,* you set properties of the components such as height, color, or other behavior.

3. The property inspectors also list the events to which components can react. Some events can be hooked up through dialog boxes. For other events, you write short snippets of event handling code.

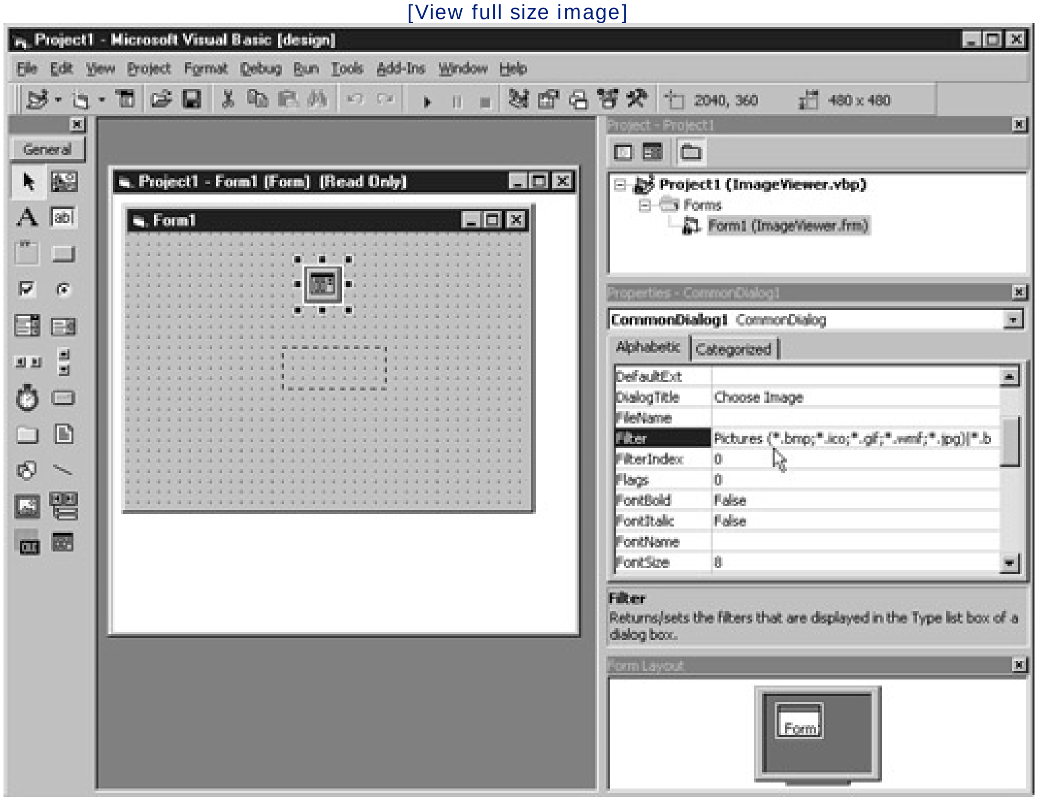
For example, in Volume I, Chapter 2, we wrote a program that displays an image in a frame. It took over a page of code. Here's what you would do in Visual Basic to create a program with pretty much the same functionality:

1. Add two controls to a window: an *Image* control for displaying graphics and a *Common Dialog* control for selecting a file.

2. Set the *Filter* properties of the CommonDialog control so that only files that the Image control can handle will show up, as shown in Figure 8-1.

Figure 8-1. The Properties window in Visual Basic for an image application

[View full size image]



3. Write four lines of Visual Basic code that will be activated when the project first starts running. All the code you need for this sequence looks like this:

```
Private Sub Form_Load()
  CommonDialog1.ShowOpen
  Image1.Picture = LoadPicture(CommonDialog1.FileName)
End Sub
```

   The code pops up the file dialog box—but only files with the right extension are shown because of how we set the filter property. After the user selects an image file, the code then tells the Image control to display it.

That's it. The layout activity, combined with these statements, gives essentially the same functionality as a page of Java code. Clearly, it is a lot easier to learn how to drop down components and set properties than it is to write a page of code.

We do not want to imply that Visual Basic is a good solution for every problem. It is clearly optimized for a particular kind of problem—UI-intensive Windows programs. The JavaBeans technology was invented to make Java technology competitive in this arena. It enables vendors to create Visual Basic-style development environments. These environments make it possible to build user interfaces with a minimum of programming.

The Bean-Writing Process

Writing a bean is not technically difficult—there are only a few new classes and interfaces for you to master. In particular, the simplest kind of bean is nothing more than a Java class that follows some fairly strict naming conventions for its methods.

Note

Some authors claim that a bean must have a default constructor. The JavaBeans specification is actually silent on this issue. However, most builder tools require a default constructor for each bean, so that they can instantiate beans without construction parameters.

Listing 8-1 at the end of this section shows the code for an ImageViewer bean that could give a Java builder environment the same functionality as the Visual Basic image control we mentioned in the previous section. When you look at this code, notice that the `ImageViewerBean` class really doesn't look any different from any other class. For example, all accessor methods begin with `get`, and all mutator methods begin with `set`. As you will soon see, builder tools use this standard naming convention to discover *properties.* For example, `fileName` is a property of this bean because it has `get` and `set` methods.

Note that a property is not the same as an instance field. In this particular example, the `fileName` property is computed from the `file` instance field. Properties are conceptually at a higher level than instance fields—they are features of the interface, whereas instance fields belong to the implementation of the class.

One point that you need to keep in mind when you read through the examples in this chapter is that real-world beans are much more elaborate and tedious to code than our brief examples, for two reasons.

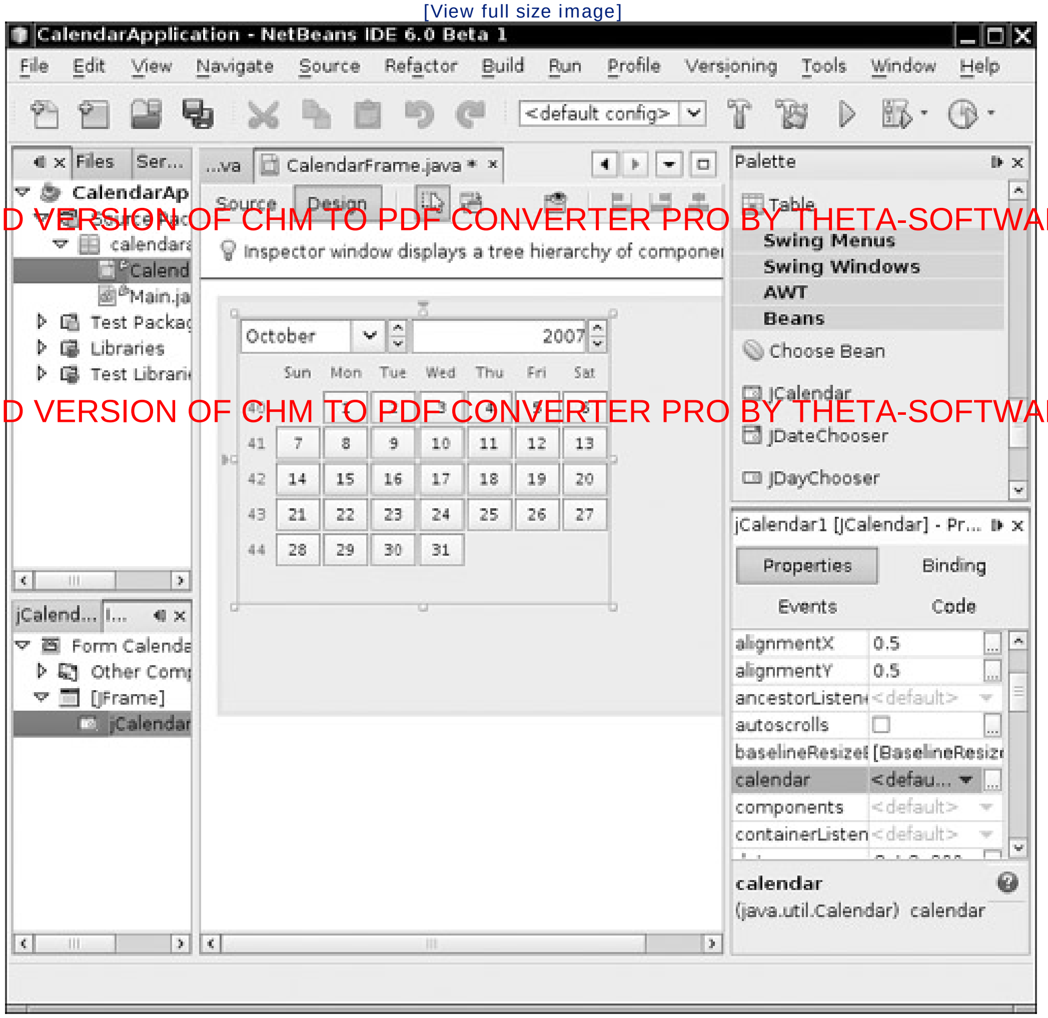
1. Beans must be usable by less-than-expert programmers. You need to expose *lots of properties* so that your users can access most of the functionality of your bean with a visual design tool and without programming.

2. The same bean must be usable in a wide *variety of contexts.* Both the behavior and the appearance of your bean must be customizable. Again, this means exposing lots of properties.

A good example of a bean with rich behavior is `CalendarBean` by Kai Tödter (see Figure 8-2). The bean and its source code are freely available from http://www.toedter.com/en/jcalendar. This bean gives users a convenient way of entering dates, by locating them in a calendar display. This is obviously pretty complex and not something one would want to program from scratch. By using a bean such as this one, you can take advantage of the work of others, simply by dropping the bean into a builder tool.

Figure 8-2. A calendar bean

[View full size image]



UNREGISTERED VERSION OF CHM TO PDF CONVERTER PRO BY THETA-SOFTWARE

Fortunately, you need to master only a small number of concepts to write beans with a rich set of behaviors. The example beans in this chapter, although not trivial, are kept simple enough to illustrate the necessary concepts.

Listing 8-1. **ImageViewerBean.java**

Code View:

```
package com.horstmann.corejava;

import java.awt.*;
import java.io.*;
import javax.imageio.*;
import javax.swing.*;

/**
 * A bean for viewing an image.
 * @version 1.21 2001-08-15
 * @author Cay Horstmann
 */
```

```
public class ImageViewerBean extends JLabel
{

   public ImageViewerBean()
   {
      setBorder(BorderFactory.createEtchedBorder());
   }

   /**
    * Sets the fileName property.
    * @param fileName the image file name
    */
   public void setFileName(String fileName)
   {
      try
      {
         file = new File(fileName);
         setIcon(new ImageIcon(ImageIO.read(file)));
      }
      catch (IOException e)
      {
         file = null;
         setIcon(null);
      }
   }

   /**
    * Gets the fileName property.
    * @return the image file name
    */
   public String getFileName()
   {
      if (file == null) return "";
      else return file.getPath();
   }

   public Dimension getPreferredSize()
   {
      return new Dimension(XPREFSIZE, YPREFSIZE);
   }

   private File file = null;
   private static final int XPREFSIZE = 200;
   private static final int YPREFSIZE = 200;
}
```

Using Beans to Build an Application

Before we get into the mechanics of writing beans, we want you to see how you might use or test them. ImageViewerBean is a perfectly usable bean, but outside a builder environment it can't show off its special features.

Each builder environment uses its own set of strategies to ease the programmer's life. We cover one environment, the NetBeans integrated development environment, available from http://netbeans.org.

In this example, we use two beans, ImageViewerBean and FileNameBean. You have already seen the code for ImageViewerBean. We will analyze the code for FileNameBean later in this chapter. For now, all you have to know is that clicking the button with the ". . ." label opens a file chooser.

Packaging Beans in JAR Files

To make any bean usable in a builder tool, package into a JAR file all class files that are used by the bean code. Unlike the JAR files for an applet, a JAR file for a bean needs a manifest file that specifies which class files in the archive are beans and should be included in the builder's toolbox. For example, here is the manifest file ImageViewerBean.mf for ImageViewerBean.

```
Manifest-Version: 1.0

Name: com/horstmann/corejava/ImageViewerBean.class
Java-Bean: True
```

Note the blank line between the manifest version and bean name.

Note

We place our example beans into the package com.horstmann.corejava because some builder environments have problems loading beans from the default package.

If your bean contains multiple class files, you just mention in the manifest those class files that are beans and that you want to have displayed in the toolbox. For example, you could place ImageViewerBean and FileNameBean into the same JAR file and use the manifest

```
Manifest-Version: 1.0

Name: com/horstmann/corejava/ImageViewerBean.class
Java-Bean: True

Name: com/horstmann/corejava/FileNameBean.class
Java-Bean: True
```

Caution

Some builder tools are extremely fussy about manifests. Make sure that there are no spaces after the ends of each line, that there are blank lines after the version and between bean entries, and that the last line ends in a newline.

To make the JAR file, follow these steps:

1. Edit the manifest file.

2. Gather all needed class files in a directory.

3. Run the `jar` tool as follows:

   `jar cvfm` *JarFile ManifestFile ClassFiles*

   For example,

   Code View:
   ```
   jar cvfm ImageViewerBean.jar ImageViewerBean.mf com/horstmann/corejava/*.class
   ```

You can also add other items, such as icon images, to the JAR file. We discuss bean icons later in this chapter.

Caution

Make sure to include all files that your bean needs in the JAR file. In particular, pay attention to inner class files such as `FileNameBean$1.class`.

Builder environments have a mechanism for adding new beans, typically by loading JAR files. Here is what you do to import beans into NetBeans version 6.

Compile the `ImageViewerBean` and `FileNameBean` classes and package them into JAR files. Then start NetBeans and follow these steps.

1. Select Tools -> Palette -> Swing/AWT Components from the menu.

2. Click the Add from JAR button.

3. In the file dialog box, move to the `ImageViewerBean` directory and select `ImageViewerBean.jar`.

4. Now a dialog box pops up that lists all the beans that were found in the JAR file. Select `ImageViewerBean`.

5. Finally, you are asked into which palette you want to place the beans. Select Beans. (There are other palettes for Swing components, AWT components, and so on.)

6. Have a look at the Beans palette. It now contains an icon representing the new bean However, the icon is just a default icon—you will see later how to add icons to a bean.

Repeat these steps with `FileNameBean`. Now you are ready to compose these beans into an application.

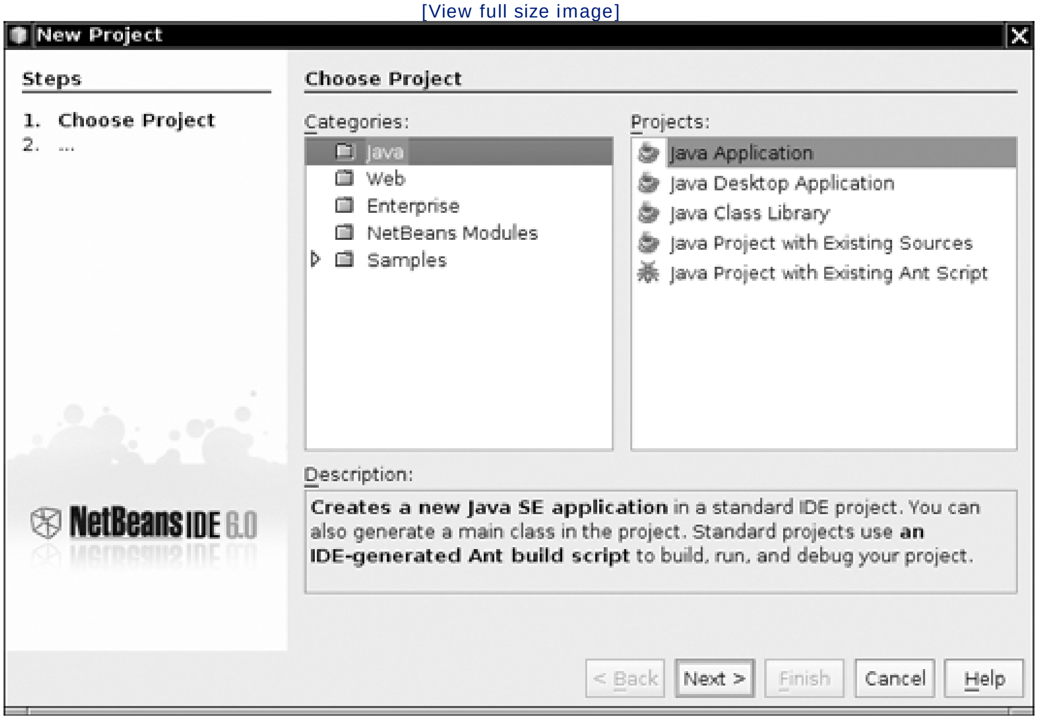
Composing Beans in a Builder Environment

The promise of component-based development is to compose your application from prefabricated components, with a minimum of programming. In this section, you will see how to compose an application from the `ImageViewerBean` and `FileNameBean` components.

In NetBeans 6, select File -> New Project from the menu. A dialog box pops up. Select Java, then Java Application (see Figure 8-3).

Figure 8-3. Creating a new project

[View full size image]

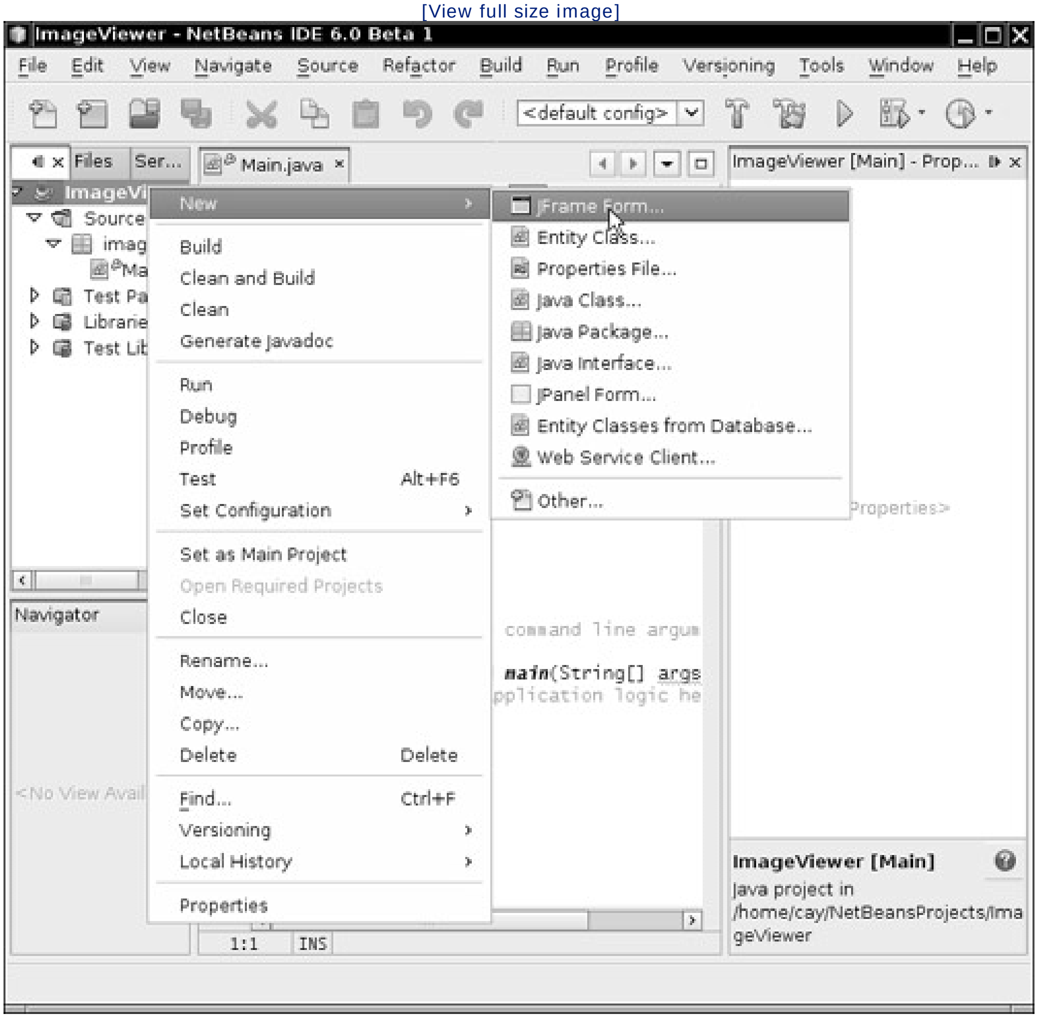


Click the Next button. On the following screen, set a name for your application (such as ImageViewer), and click the Finish button. Now you see a project viewer on the left and the source code editor in the middle.

Right-click the project name in the project viewer and select New -> JFrame Form from the menu (see Figure 8-4).

Figure 8-4. Creating a form view

[View full size image]

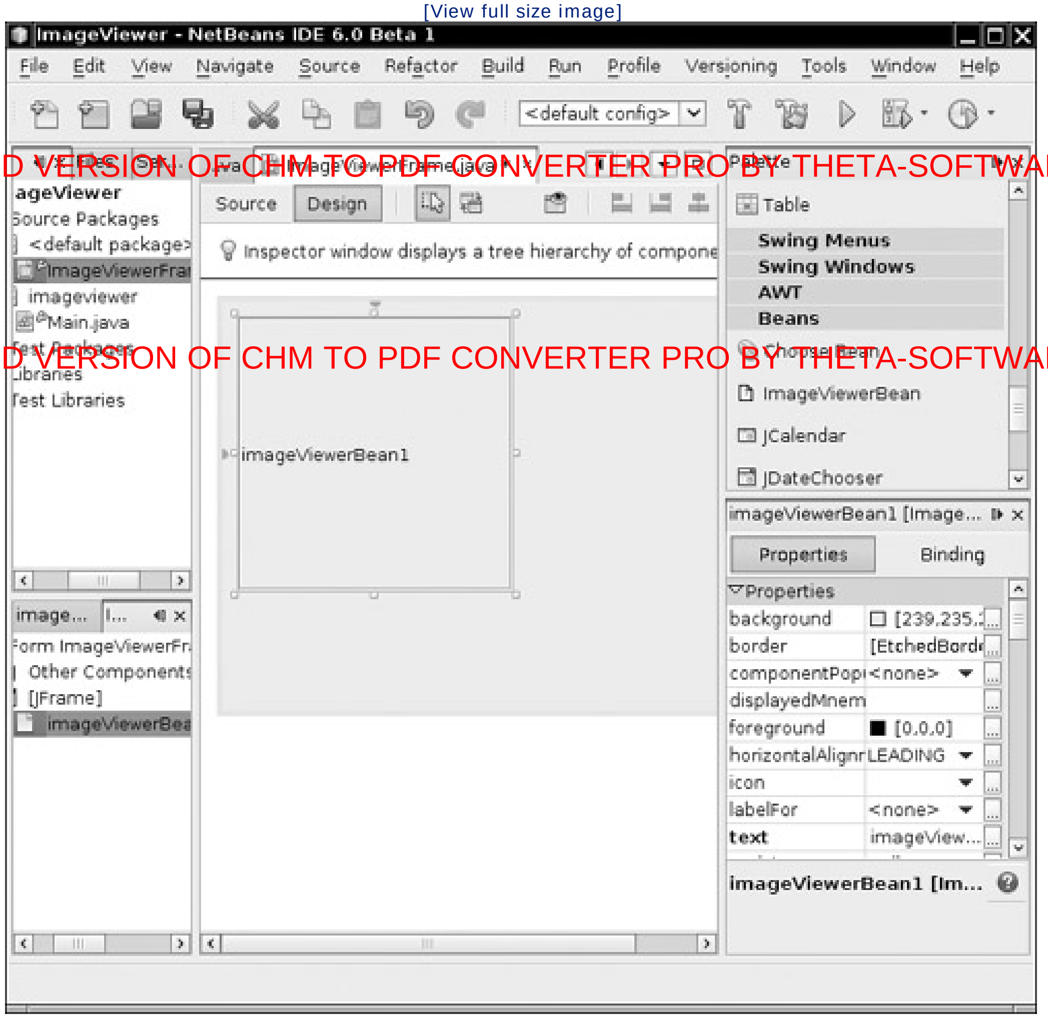


A dialog box pops up. Enter a name for the frame class (such as ImageViewerFrame), and click the Finish button. You now get a form editor with a blank frame. To add a bean to the form, select the bean in the palette that is located to the right of the form editor. Then click the frame.

Figure 8-5 shows the result of adding an ImageViewerBean onto the frame.

Figure 8-5. Adding a bean

[View full size image]

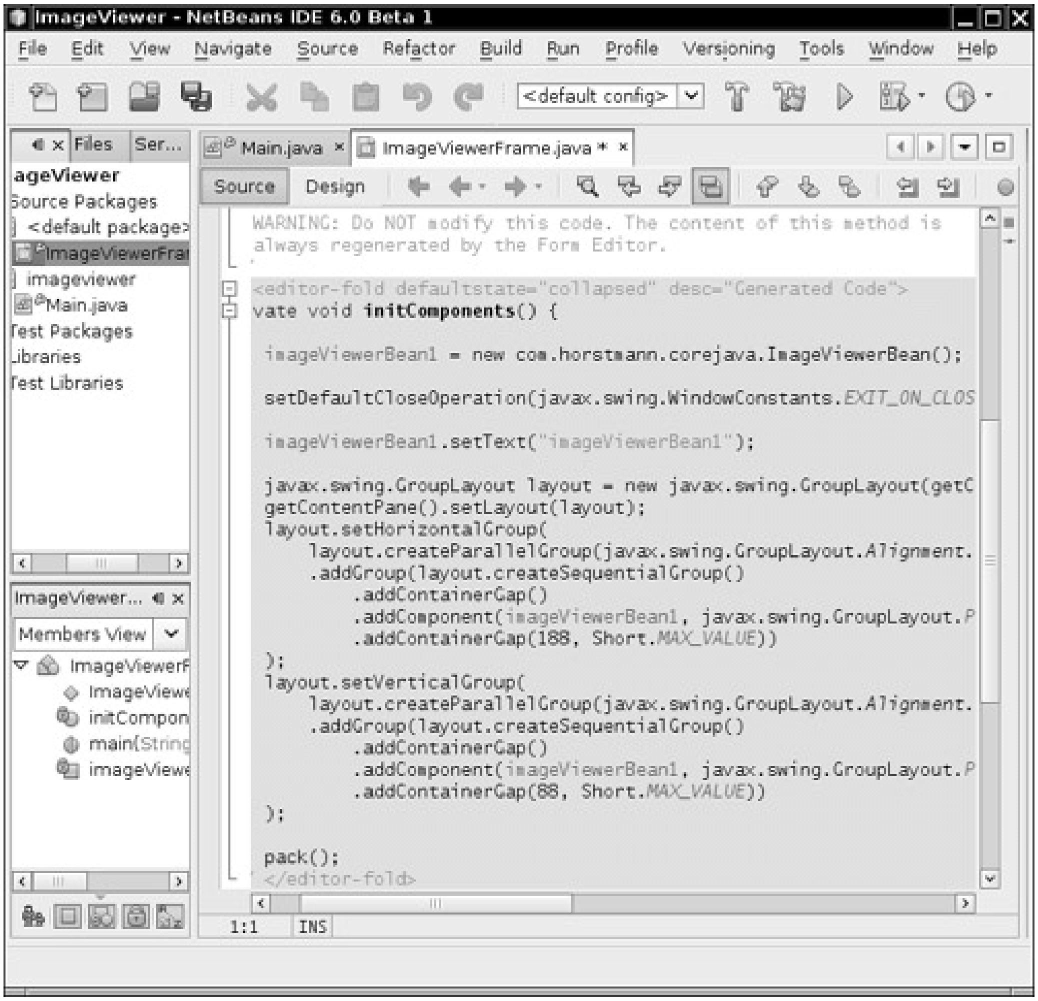


UNREGISTERED VERSION OF CHM TO PDF CONVERTER PRO BY THETA-SOFTWARE

If you look into the source window, you will find that the source code now contains the Java instructions to add the bean objects to the frame (see Figure 8-6). The source code is bracketed by dire warnings that you should not edit it. Any edits would be lost when the builder environment updates the code as you modify the form.

Figure 8-6. The source code for adding the bean

[View full size image]



Note

A builder environment is not required to update source code as you build an application. A builder environment can generate source code when you are done editing, serialize the beans you customized, or perhaps produce an entirely different description of your building activity.

For example, the experimental Bean Builder at http://bean-builder.dev.java.net lets you design GUI applications without writing any source code at all.

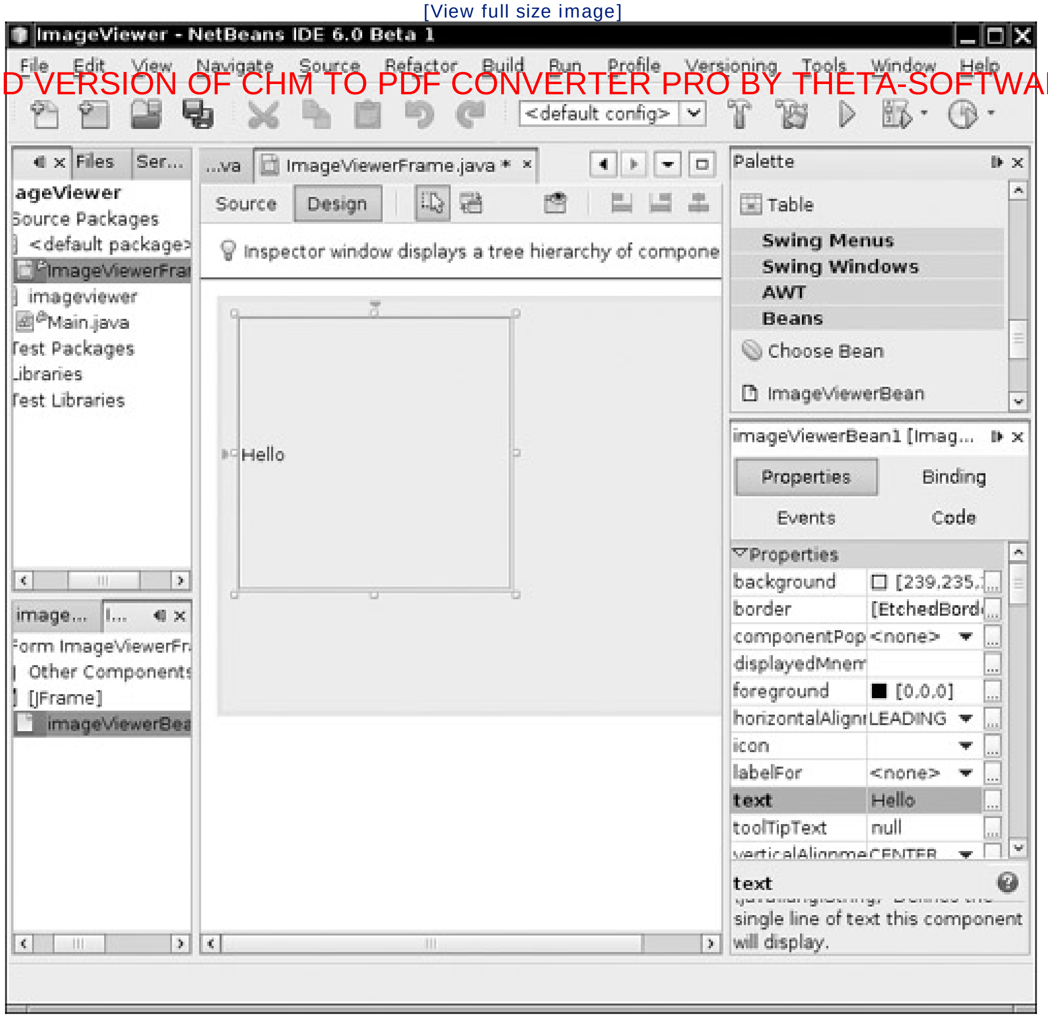
The JavaBeans mechanism doesn't attempt to force an implementation strategy on a builder tool. Instead, it aims to supply information about beans to builder tools that can choose to take advantage of the information in one way or another.

Now go back to the design view and click `ImageViewerBean` in the form. On the right-hand side is a property inspector that lists the bean property names and their current values. This is a vital part of component-based development tools because setting properties at design time is how you set the initial state of a component.

For example, you can modify the `text` property of the label used for the image bean by simply typing a new name into the property inspector. Changing the `text` property is simple—you just edit a string in a text field. Try it out—set the label text to `"Hello"`. The form is immediately updated to reflect your change (see Figure 8-7).

Figure 8-7. Changing a property in the property inspector

[View full size image]



Note

When you change the setting of a property, the NetBeans environment updates the source code to reflect your action. For example, if you set the `text` field to `Hello`, the instruction

```
imageViewerBean.setText("Hello");
```

is added to the `initComponents` method. As already mentioned, other builder tools might have different strategies for recording property settings.

Properties don't have to be strings; they can be values of any Java type. To make it possible for users to set values for properties of any type, builder tools use specialized *property editors.* (Property editors either come with the builder or are supplied by the bean developer. You see how to write your own property editors later in this chapter.)

To see a simple property editor at work, look at the `foreground` property. The property type is `Color`. You can see the color editor, with a text field containing a string `[0,0,0]` and a button labeled ". . ." that brings up a color chooser. Go ahead and change the foreground color. Notice that you'll immediately see the change to the property value—the label text changes color.

More interestingly, choose a file name for an image file in the property inspector. Once you do so, `ImageViewerBean` automatically displays the image.

Note

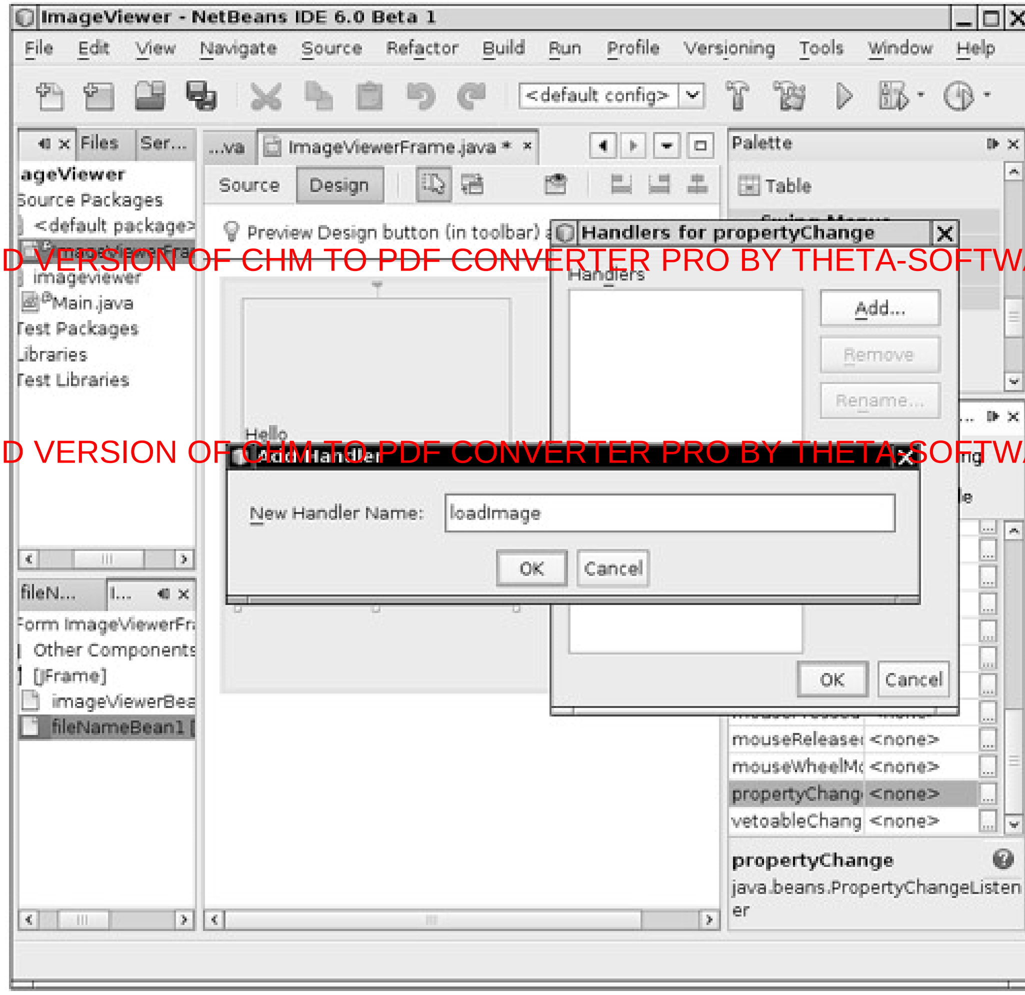
If you look closely at the property inspector in NetBeans, you will find a large number of mysterious properties such as `focusCycleRoot` and `paintingForPrint`. These are inherited from the `JLabel` superclass. You will see later in this chapter how you can suppress them from the property inspector.

To complete our application, place a `FileNameBean` object into the frame. Now we want the image to be loaded when the `fileName` property of `FileNameBean` is changed. This happens through a `PropertyChange` event; we discuss these kinds of events later in this chapter.

To react to the event, select `FileNameBean` and select the Events tab from its property inspector. Then click the "..." button next to the `propertyChange` entry. A dialog box appears that shows that no handlers are currently associated with this event. Click the Add button in the dialog box. You are prompted for a method name (see Figure 8-8). Type `loadImage`.

Figure 8-8. Adding an event to a bean

[View full size image]



Now look at the code editor. Event handling code has been added, and there is a new method:

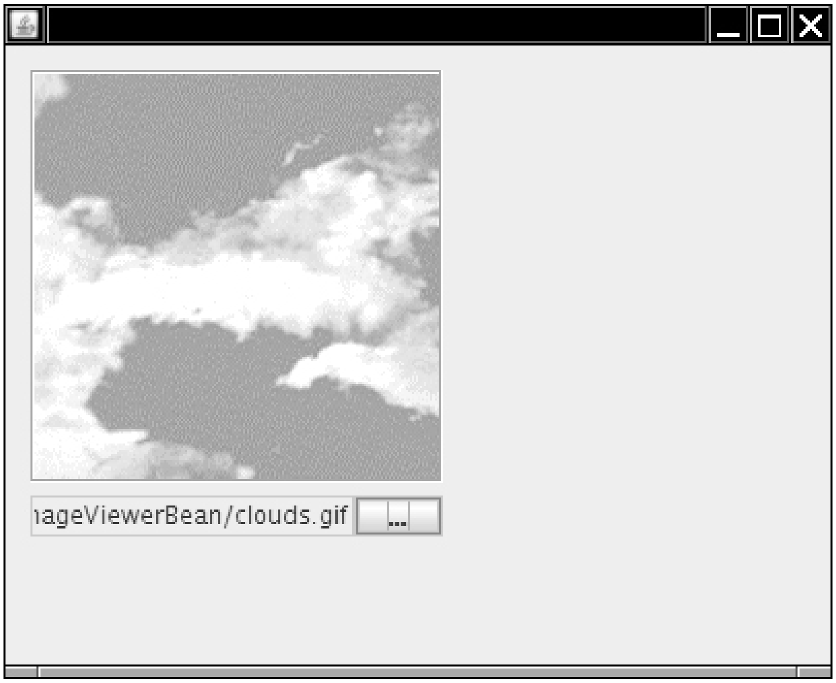
```
private void loadImage(java.beans.PropertyChange evt)
{
   // TODO add your handling code here
}
```

Add the following line of code to that method:

```
imageViewerBean1.setFileName(fileNameBean1.getFileName());
```

Then compile and execute the frame class. You now have a complete image viewer application. Click the button with the ". . ." label and select an image file. The image is displayed in the image viewer (see Figure 8-9).

Figure 8-9. The image viewer application



This process demonstrates that you can create a Java application from beans by setting properties and providing a small amount of code for event handlers.

Naming Patterns for Bean Properties and Events

In this section, we cover the basic rules for designing your own beans. First, we want to stress there is *no* cosmic beans class that you extend to build your beans. Visual beans directly or indirectly extend the `Component` class, but nonvisual beans don't have to extend any particular superclass. Remember, a bean is simply *any* class that can be manipulated in a builder tool. The builder tool does not look at the superclass to determine the bean nature of a class, but it analyzes the names of its methods. To enable this analysis, the method names for beans must follow certain patterns.

Note

There is a `java.beans.Beans` class, but all methods in it are static. Extending it would, therefore, be rather pointless, even though you will see it done occasionally, supposedly for greater "clarity." Clearly, because a bean can't extend both `Beans` and `Component`, this approach can't work for visual beans. In fact, the `Beans` class contains methods that are designed to be called by builder tools, for example, to check whether the tool is operating at design time or run time.

Other languages for visual design environments, such as Visual Basic and C#, have special keywords such as "Property" and "Event" to express these concepts directly. The designers of the Java specification decided not to add keywords to the language to support visual programming. Therefore, they needed an alternative so that a builder tool could analyze a bean to learn its properties or events. Actually, there are two alternative mechanisms. If the bean writer uses standard naming patterns for properties and events, then the builder tool can use the reflection mechanism to understand what properties and events the bean is supposed to expose. Alternatively, the bean writer can supply a *bean information* class that tells the builder tool about the properties and events of the bean. We start out using the naming patterns because they are easy to use. You'll see later in this chapter how to supply a bean information class.

Note

Although the documentation calls these standard naming patterns "design patterns," these are really only naming conventions and have nothing to do with the design patterns that are used in object-oriented programming.

The naming pattern for properties is simple: Any pair of methods

```
public Type getPropertyName()
public void setPropertyName(Type newValue)
```

corresponds to a read/write property.

For example, in our `ImageViewerBean`, there is only one read/write property (for the file name to be viewed), with the following methods:

```
public String getFileName()
public void setFileName(String newValue)
```

If you have a `get` method but not an associated `set` method, you define a read-only property. Conversely, a `set` method without an associated `get` method defines a write-only property.

Note

The `get` and `set` methods you create can do more than simply get and set a private data field. Like any Java method, they can carry out arbitrary actions. For example, the `setFileName` method of the `ImageViewerBean` class not only sets the value of the `fileName` data field, but also opens the file and loads the image.

Note

In Visual Basic and C#, properties also come from `get` and `set` methods. However, in both these languages, you explicitly define properties rather than having builder tools second-guess the programmer's intentions by analyzing method names. In those languages, properties have another advantage: Using a property name on the left side of an assignment automatically calls the `set` method. Using a property name in an expression automatically calls the `get` method. For example, in Visual Basic you can write

```
imageBean.fileName = "corejava.gif"
```

instead of

```
imageBean.setFileName("corejava.gif");
```

This syntax was considered for Java, but the language designers felt that it was a poor idea to hide a method call behind syntax that looks like field access.

There is one exception to the `get/set` naming pattern. Properties that have `boolean` values should use an `is/set` naming pattern, as in the following examples:

```
public boolean isPropertyName()
```

```
public void setPropertyName(boolean b)
```

For example, an animation might have a property `running`, with two methods

```
public boolean isRunning()
public void setRunning(boolean b)
```


The `setRunning` method would start and stop the animation. The `isRunning` method would report its current status.

Note

✔ It is legal to use a `get` prefix for a `boolean` property accessor (such as `getRunning`), but the `is` prefix is preferred.

Be careful with the capitalization pattern you use for your method names. The designers of the JavaBeans specification decided that the name of the property in our example would be `fileName`, with a lowercase `f`, even though the `get` and `set` methods contain an uppercase `F` (`getFileName`, `setFileName`). The bean analyzer performs a process called *decapitalization* to derive the property name. (That is, the first character after `get` or `set` is converted to lower case.) The rationale is that this process results in method and property names that are more natural to programmers.

However, if the first *two* letters are upper case (such as in `getURL`), then the first letter of the property is not changed to lower case. After all, a property name of `uRL` would look ridiculous.

Note

✔ What do you do if your class has a pair of `get` and `set` methods that doesn't correspond to a property that you want users to manipulate in a property inspector? In your own classes, you can of course avoid that situation by renaming your methods. However, if you extend another class, then you inherit the method names from the superclass. This happens, for example, when your bean extends `JPanel` or `JLabel`—a large number of uninteresting properties show up in the property inspector. You will see later in this chapter how you can override the automatic property discovery process by supplying *bean information*. In the bean information, you can specify exactly which properties your bean should expose.

For events, the naming patterns are equally simple. A bean builder environment will infer that your bean generates events when you supply methods to add and remove event listeners. All event class names must end in `Event`, and the classes must extend the `EventObject` class.

Suppose your bean generates events of type *EventName*`Event`. The listener interface must be called *EventName*`Listener`, and the methods to manage the listeners must be called

```
public void addEventNameListener(EventNameListener e)
public void removeEventNameListener(EventNameListener e)
public EventNameListener getEventNameListeners()
```

If you look at the code for `ImageViewerBean`, you'll see that it has no events to expose. However, many Swing components generate events, and they follow this pattern. For example, the `AbstractButton` class generates `ActionEvent` objects, and it has the following methods to manage `ActionListener` objects:

```
public void addActionListener(ActionListener e)
public void removeActionListener(ActionListener e)
ActionListener[] getActionListeners()
```

Caution

If your event class doesn't extend `EventObject`, chances are that your code will compile just fine because none of the methods of the `EventObject` class are actually needed. However, your bean will mysteriously fail—the introspection mechanism will not recognize the events.

Bean Property Types

A sophisticated bean will expose lots of different properties and events. Properties can be as simple as the `fileName` property that you saw in `ImageViewerBean` and `FileNameBean` or as sophisticated as a color value or even an array of data points—we encounter both of these cases later in this chapter. The JavaBeans specification allows four types of properties, which we illustrate by various examples.

Simple Properties

A simple property is one that takes a single value such as a string or a number. The `fileName` property of the ImageViewer is an example of a simple property. Simple properties are easy to program: Just use the `set/get` naming convention we indicated earlier. For example, if you look at the code in Listing 8-1, you can see that all it took to implement a simple string property is the following:


```
public void setFileName(String f)
{
   fileName = f;
   image = . . .
   repaint();
}

public String getFileName()
{
   if (file == null) return "";
   else return file.getPath();
}
```

Indexed Properties

An indexed property specifies an array. With an indexed property, you supply two pairs of `get` and `set` methods: one for the array and one for individual entries. They must follow this pattern:

```
Type[] getPropertyName()
void setPropertyName(Type[] newValue)
Type getPropertyName(int i)
void setPropertyName(int i, Type newValue)
```

For example, the `FileNameBean` uses an indexed property for the file extensions. It provides these four methods:

```
public String[] getExtensions() { return extensions; }
public void setExtensions(String[] newValue) { extensions = newValue; }
public String getExtensions(int i)
{
   if (0 <= i && i < extensions.length) return extensions[i];
   else return "";
}
public void setExtensions(int i, String newValue)
{
   if (0 <= i && i < extensions.length) extensions[i] = value;
}
```

```
. . .
private String[] extensions;
```

The `set`*PropertyName*`(int,` *Type*`)` method cannot be used to *grow* the array. To grow the array, you must manually build a new array and then pass it to the `set`*PropertyName*`(`*Type*`[])` method.

Bound Properties

Bound properties tell interested listeners that their value has changed. For example, the `fileName` property in `FileNameBean` is a bound property. When the file name changes, then `ImageViewerBean` is automatically notified and it loads the new file.

To implement a bound property, you must implement two mechanisms:

1. Whenever the value of the property changes, the bean must send a `PropertyChange` event to all registered listeners. This change can occur when the `set` method is called or when some other method (such as the action listener of the "..." button) changes the value.

2. To enable interested listeners to register themselves, the bean has to implement the following two methods:

   ```
   void addPropertyChangeListener(PropertyChangeListener listener)
   void removePropertyChangeListener(PropertyChangeListener listener)
   ```

   It is also recommended (but not required) to provide the method

   ```
   PropertyChangeListener[] getPropertyChangeListeners()
   ```

The `java.beans` package has a convenience class, called `PropertyChangeSupport`, that manages the listeners for you. To use this convenience class, add an instance field of this class:

Code View:
```
private PropertyChangeSupport changeSupport = new PropertyChangeSupport(this);
```

Delegate the task of adding and removing property change listeners to that object.

```
public void addPropertyChangeListener(PropertyChangeListener listener)
{
   changeSupport.addPropertyChangeListener(listener);
}

public void removePropertyChangeListener(PropertyChangeListener listener)
{
   changeSupport.removePropertyChangeListener(listener);
}

public PropertyChangeListener[] getPropertyChangeListeners()
{
   return changeSupport.getPropertyChangeListeners();
```

```
}
```

Whenever the value of the property changes, use the `firePropertyChange` method of the `PropertyChangeSupport` object to deliver an event to all the registered listeners. That method has three parameters: the name of the property, the old value, and the new value. Here is the boilerplate code for a typical setter of a bound property:


```
public void setValue(Type newValue)
{
   Type oldValue = getValue();
   value = newValue;
   changeSupport.firePropertyChange("propertyName", oldValue, newValue);
}
```


To fire a change of an indexed property, you call

Code View:

```
changeSupport.fireIndexedPropertyChange("propertyName", index, oldValue, newValue);
```

Tip

If your bean extends any class that ultimately extends the `Component` class, then you do *not* need to implement the `addPropertyChangeListener`, `removePropertyChangeListener`, and `getPropertyChangeListeners` methods. These methods are already implemented in the `Component` superclass. To notify the listeners of a property change, simply call the `firePropertyChange` method of the `JComponent` superclass. Unfortunately, firing of indexed property changes is not supported.

Other beans that want to be notified when the property value changes must add a `PropertyChangeListener`. That interface contains only one method:

```
void propertyChange(PropertyChangeEvent event)
```

The `PropertyChangeEvent` object holds the name of the property and the old and new values, obtainable with the `getPropertyName`, `getOldValue`, and `getNewValue` methods.

If the property type is not a class type, then the property value objects are instances of the usual wrapper classes.

Constrained Properties

A *constrained property* is constrained by the fact that *any* listener can "veto" proposed changes, forcing it to revert to the old setting. The Java library contains only a few examples of constrained properties. One of them is the `closed` property of the `JInternalFrame` class. If someone tries to call `setClosed(true)` on an internal frame, then all of its `VetoableChangeListeners` are notified. If any of them throws a `PropertyVetoException`, then the `closed` property is *not* changed, and the `setClosed` method throws the same exception. In particular, a `VetoableChangeListener` may veto closing the frame if its contents have not been saved.

To build a constrained property, your bean must have the following two methods to manage `VetoableChangeListener` objects:

```
public void addVetoableChangeListener(VetoableChangeListener listener);
public void removeVetoableChangeListener(VetoableChangeListener listener);
```

It also should have a method for getting all listeners:

```
VetoableChangeListener[] getVetoableChangeListeners()
```

Just as there is a convenience class to manage property change listeners, there is a convenience class, called `VetoableChangeSupport`, that manages vetoable change listeners. Your bean should contain an object of this class.

Code View:
```
private VetoableChangeSupport vetoSupport = new VetoableChangeSupport(this);
```

Adding and removing listeners should be delegated to this object. For example:

```
public void addVetoableChangeListener(VetoableChangeListener listener)
{
   vetoSupport.addVetoableChangeListener(listener);
}
public void removeVetoableChangeListener(VetoableChangeListener listener)
{
   vetoSupport.removeVetoableChangeListener(listener);
}
```

To update a constrained property value, a bean uses the following three-phase approach:

1. Notify all vetoable change listeners of the *intent* to change the property value. (Use the `fireVetoableChange` method of the `VetoableChangeSupport` class.)

2. If none of the vetoable change listeners has thrown a `PropertyVetoException`, then update the value of the property.

3. Notify all property change listeners to *confirm* that a change has occurred.

For example,

```
public void setValue(Type newValue) throws PropertyVetoException
```

```
{
   Type oldValue = getValue();
   vetoSupport.fireVetoableChange("value", oldValue, newValue);
   // survived, therefore no veto
   value = newValue;
   changeSupport.firePropertyChange("value", oldValue, newValue);
}
```


It is important that you don't change the property value until all the registered vetoable change listeners have agreed to the proposed change. Conversely, a vetoable change listener should never assume that a change that it agrees to is actually happening. The only reliable way to get notified when a change is actually happening is through a property change listener.

Note

If your bean extends the `JComponent` class, you do not need a separate `VetoableChangeSupport` object. Simply call the `fireVetoableChange` method of the `JComponent` superclass. Note that you cannot install a vetoable change listener for a specific property into a `JComponent`. You need to listen to all vetoable changes.

We end our discussion of JavaBeans properties by showing the full code for `FileNameBean` (see Listing 8-2). The `FileNameBean` has an indexed `extensions` property and a constrained `filename` property. Because `FileNameBean` extends the `JPanel` class, we did not have to explicitly use a `PropertyChangeSupport` object. Instead, we rely on the ability of the `JPanel` class to manage property change listeners.

Listing 8-2. **FileNameBean.java**

Code View:
```
package com.horstmann.corejava;

import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.util.*;
import javax.swing.*;
import javax.swing.filechooser.*;

/**
 * A bean for specifying file names.
 * @version 1.30 2007-10-03
 * @author Cay Horstmann
 */
public class FileNameBean extends JPanel
{
   public FileNameBean()
   {
      dialogButton = new JButton("...");
      nameField = new JTextField(30);
```

```java

      chooser = new JFileChooser();
      setPreferredSize(new Dimension(XPREFSIZE, YPREFSIZE));

      setLayout(new GridBagLayout());
      GridBagConstraints gbc = new GridBagConstraints();
      gbc.weightx = 100;
      gbc.weighty = 100;
      gbc.anchor = GridBagConstraints.WEST;
      gbc.fill = GridBagConstraints.BOTH;
      gbc.gridwidth = 1;
      gbc.gridheight = 1;
      add(nameField, gbc);

      dialogButton.addActionListener(new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               chooser.setFileFilter(new FileNameExtensionFilter(Arrays.toString(extensions),
                     extensions));
               int r = chooser.showOpenDialog(null);
               if (r == JFileChooser.APPROVE_OPTION)
               {
                  File f = chooser.getSelectedFile();
                  String name = f.getAbsolutePath();
                  setFileName(name);
              }
            }
         });
      nameField.setEditable(false);

      gbc.weightx = 0;
      gbc.anchor = GridBagConstraints.EAST;
      gbc.fill = GridBagConstraints.NONE;
      gbc.gridx = 1;
      add(dialogButton, gbc);
   }

   /**
    * Sets the fileName property.
    * @param newValue the new file name
    */
   public void setFileName(String newValue)
   {
      String oldValue = nameField.getText();
      nameField.setText(newValue);
      firePropertyChange("fileName", oldValue, newValue);
   }

   /**
    * Gets the fileName property.
    * @return the name of the selected file
    */
   public String getFileName()
   {
      return nameField.getText();
   }

   /**
```

```
     * Gets the extensions property.
     * @return the default extensions in the file chooser
     */
    public String[] getExtensions()
    {
       return extensions;
    }

    /**
```

```
     * Sets the extensions property.
     * @param newValue the new default extensions
     */
    public void setExtensions(String[] newValue)
    {
       extensions = newValue;
    }
```

```

    /**
     * Gets one of the extensions property values.
     * @param i the index of the property value
     * @return the value at the given index
     */
    public String getExtensions(int i)
    {
       if (0 <= i && i < extensions.length) return extensions[i];
       else return "";
    }

    /**
     * Sets one of the extensions property values.
     * @param i the index of the property value
     * @param newValue the new value at the given index
     */
    public void setExtensions(int i, String newValue)
    {
       if (0 <= i && i < extensions.length) extensions[i] = newValue;
    }

    private static final int XPREFSIZE = 200;
    private static final int YPREFSIZE = 20;
    private JButton dialogButton;
    private JTextField nameField;
    private JFileChooser chooser;
    private String[] extensions = { "gif", "png" };
 }
```

API

`java.beans.PropertyChangeListener` 1.1

- `void propertyChange(PropertyChangeEvent event)`

  is called when a property change event is fired.

API

`java.beans.PropertyChangeSupport` 1.1

- `PropertyChangeSupport(Object sourceBean)`

  constructs a `PropertyChangeSupport` object that manages listeners for bound property changes of the given bean.

- `void addPropertyChangeListener(PropertyChangeListener listener)`

- `void addPropertyChangeListener(String propertyName, PropertyChangeListener listener)` **1.2**

  registers an interested listener for changes in all bound properties, or only the named bound property.

- `void removePropertyChangeListener(PropertyChangeListener listener)`

- `void removePropertyChangeListener(String propertyName, PropertyChangeListener listener)` **1.2**

  removes a previously registered property change listener.

- `void firePropertyChange(String propertyName, Object oldValue, Object newValue)`

- `void firePropertyChange(String propertyName, int oldValue, int newValue)` **1.2**

- `void firePropertyChange(String propertyName, boolean oldValue, boolean newValue)` **1.2**

  sends a `PropertyChangeEvent` to registered listeners.

- `void fireIndexedPropertyChange(String propertyName, int index, Object oldValue, Object newValue)` **5.0**

- `void fireIndexedPropertyChange(String propertyName, int index, int oldValue, int newValue)` **5.0**

- `void fireIndexedPropertyChange(String propertyName, int index, boolean oldValue, boolean newValue)` **5.0**

  sends an `IndexedPropertyChangeEvent` to registered listeners.

- `PropertyChangeListener[] getPropertyChangeListeners()` **1.4**

- `PropertyChangeListener[] getPropertyChangeListeners(String propertyName)` **1.4**

  gets the listeners for changes in all bound properties, or only the named bound property.

API

`java.beans.PropertyChangeEvent` **1.1**

- `PropertyChangeEvent(Object sourceBean, String propertyName, Object oldValue, Object newValue)`

  constructs a new `PropertyChangeEvent` object, describing that the given property has changed from `oldValue` to `newValue`.

- `String getPropertyName()`

  returns the name of the property.

- `Object getOldValue();`

- `Object getNewValue()`

  returns the old and new value of the property.

API

`java.beans.IndexedPropertyChangeEvent` 5.0

- `IndexedPropertyChangeEvent(Object sourceBean, String propertyName, int index, Object oldValue, Object newValue)`

  constructs a new `IndexedPropertyChangeEvent` object, describing that the given property has changed from `oldValue` to `newValue` at the given index.

- `int getIndex()`

  returns the index at which the change occurred.

API

`java.beans.VetoableChangeListener` 1.1

- `void vetoableChange(PropertyChangeEvent event)`

  is called when a property is about to be changed. It should throw a `PropertyVetoException` if the change is not acceptable.

API

`java.beans.VetoableChangeSupport` 1.1

- `VetoableChangeSupport(Object sourceBean)`

  constructs a `PropertyChangeSupport` object that manages listeners for constrained property changes of the given bean.

- `void addVetoableChangeListener(VetoableChangeListener listener)`

- `void addVetoableChangeListener(String propertyName, VetoableChangeListener listener)` **1.2**

  registers an interested listener for changes in all constrained properties, or only the named constrained property.

- `void removeVetoableChangeListener(VetoableChangeListener listener)`

- `void removeVetoableChangeListener(String propertyName, VetoableChangeListener listener)` **1.2**

  removes a previously registered vetoable change listener.

- `void fireVetoableChange(String propertyName, Object oldValue, Object newValue)`

- `void fireVetoableChange(String propertyName, int oldValue, int newValue)` **1.2**

- `void fireVetoableChange(String propertyName, boolean oldValue, boolean newValue)` **1.2**

  sends a `VetoableChangeEvent` to registered listeners.

- `VetoableChangeListener[] getVetoableChangeListeners()` **1.4**

- `VetoableChangeListener[] getVetoableChangeListeners(String propertyName)` **1.4**

  gets the listeners for changes in all constrained properties, or only the named bound property.

API

`java.awt.Component` 1.0

- `void addPropertyChangeListener(PropertyChangeListener listener)` **1.2**

- `void addPropertyChangeListener(String propertyName, PropertyChangeListener listener)` **1.2**

  registers an interested listener for changes in all bound properties, or only the named bound property.

- `void removePropertyChangeListener(PropertyChangeListener listener)` **1.2**

- `void removePropertyChangeListener(String propertyName, PropertyChangeListener listener)` **1.2** removes a previously registered property change listener.

- `void firePropertyChange(String propertyName, Object oldValue, Object newValue)` **1.2**

  sends a `PropertyChangeEvent` to registered listeners.

API

`javax.swing.JComponent` 1.2

- `void addVetoableChangeListener(VetoableChangeListener listener)`

  registers an interested listener for changes in all constrained properties, or only the named constrained property.

- `void removeVetoableChangeListener(VetoableChangeListener listener)`

  removes a previously registered vetoable change listener.

- `void fireVetoableChange(String propertyName, Object oldValue, Object newValue)`

  sends a `VetoableChangeEvent` to registered listeners.

API

`java.beans.PropertyVetoException` 1.1

- `PropertyVetoException(String message, PropertyChangeEvent event)`

  creates a new `PropertyVetoException`.

- `PropertyChangeEvent getPropertyChangeEvent()`

  returns the `PropertyChangeEvent` that was vetoed.

`BeanInfo` Classes

If you use the standard naming patterns for the methods of your bean class, then a builder tool can use reflection to determine features such as properties and events. This process makes it simple to get started with bean programming, but naming patterns are rather limiting. As your beans become complex, there might be features of your bean that naming patterns will not reveal. Moreover, as we already mentioned, many beans have `get/set` method pairs that should *not* correspond to bean properties.

If you need a more flexible mechanism for describing information about your bean, define an object that implements the `BeanInfo` interface. When you provide such an object, a builder tool will consult it about the features that your bean supports.

The name of the bean info class must be formed by adding `BeanInfo` to the name of the bean. For example, the bean info class associated to the class `ImageViewerBean` *must* be named `ImageViewerBeanBeanInfo`. The bean info class must be part of the same package as the bean itself.

You won't normally write a class that implements all methods of the `BeanInfo` interface. Instead, you should extend the `SimpleBeanInfo` convenience class that has default implementations for all the methods in the `BeanInfo` interface.

The most common reason for supplying a `BeanInfo` class is to gain control of the bean properties. You construct a `PropertyDescriptor` for each property by supplying the name of the property and the class of the bean that contains it.

Code View:
```
PropertyDescriptor descriptor = new PropertyDescriptor("fileName", ImageViewerBean.class);
```

Then implement the `getPropertyDescriptors` method of your `BeanInfo` class to return an array of all property descriptors.

For example, suppose `ImageViewerBean` wants to hide all properties that it inherits from the `JLabel` superclass and expose only the `fileName` property. The following `BeanInfo` class does just that:

Code View:
```
// bean info class for ImageViewerBean
class ImageViewerBeanBeanInfo extends SimpleBeanInfo
{
   public PropertyDescriptor[] getPropertyDescriptors()
   {
      return propertyDescriptors;
   }
   private PropertyDescriptor[] propertyDescriptors =  new PropertyDescriptor[]
      {
         new PropertyDescriptor("fileName", ImageViewerBean.class);
      };
}
```

Other methods also return `EventSetDescriptor` and `MethodDescriptor` arrays, but they are less commonly used. If one of these methods returns `null` (as is the case for the `SimpleBeanInfo` methods), then the standard naming patterns apply. However, if you override a method to return a non-null array, then you must include *all* properties, events, or methods in your array.

Note

Sometimes, you might want to write generic code that discovers properties or events of an arbitrary bean. Call the static `getBeanInfo` method of the `Introspector` class. The `Introspector` constructs a `BeanInfo` class that completely describes the bean, taking into account the information in `BeanInfo` companion classes.

Another useful method in the `BeanInfo` interface is the `getIcon` method that lets you give your bean a custom icon. Builder tools will display the icon in a palette. Actually, you can specify four separate icon bitmaps. The `BeanInfo` interface has four constants that cover the standard sizes:

```
ICON_COLOR_16x16
ICON_COLOR_32x32
ICON_MONO_16x16
ICON_MONO_32x32
```

In the following class, we use the `loadImage` convenience method in the `SimpleBeanInfo` class to load the icon images:

Code View:

```
public class ImageViewerBeanBeanInfo extends SimpleBeanInfo
{
   public ImageViewerBeanBeanInfo()
   {
      iconColor16 = loadImage("ImageViewerBean_COLOR_16x16.gif");
      iconColor32 = loadImage("ImageViewerBean_COLOR_32x32.gif");
      iconMono16 = loadImage("ImageViewerBean_MONO_16x16.gif");
      iconMono32 = loadImage("ImageViewerBean_MONO_32x32.gif");
   }

   public Image getIcon(int iconType)
   {
      if (iconType == BeanInfo.ICON_COLOR_16x16) return iconColor16;
      else if (iconType == BeanInfo.ICON_COLOR_32x32) return iconColor32;
      else if (iconType == BeanInfo.ICON_MONO_16x16) return iconMono16;
      else if (iconType == BeanInfo.ICON_MONO_32x32) return iconMono32;
      else return null;
   }

   private Image iconColor16;
   private Image iconColor32;
   private Image iconMono16;
   private Image iconMono32;
```

```
}
```

API `java.beans.Introspector` 1.1

- `static BeanInfo getBeanInfo(Class<?> beanClass)`

  gets the bean information of the given class.

API `java.beans.BeanInfo` 1.1

- `PropertyDescriptor[] getPropertyDescriptors()`

  returns the descriptors for the bean properties. A return of `null` indicates that the naming conventions should be used to find the properties.

- `Image getIcon(int iconType)`

  returns an image object that can represent the bean in toolboxes, tool bars, and the like. There are four constants, as described earlier, for the standard types of icons.

API `java.beans.SimpleBeanInfo` 1.1

- `Image loadImage(String resourceName)`

  returns an image object file associated to the resource. The resource name is a path name, taken relative to the directory containing the bean info class.

API

`java.beans.FeatureDescriptor` 1.1

- `String getName()`
- `void setName(String name)`

  gets or sets the programmatic name for the feature.

- `String getDisplayName()`
- `void setDisplayName(String displayName)`

  gets or sets a display name for the feature. The default value is the value returned by `getName`. However, currently there is no explicit support for supplying feature names in multiple locales.

- `String getShortDescription()`
- `void setShortDescription(String text)`

  gets or sets a string that a builder tool can use to provide a short description for this feature. The default value is the return value of `getDisplayName`.

- `boolean isExpert()`
- `void setExpert(boolean b)`

  gets or sets an expert flag that a builder tool can use to determine whether to hide the feature from a naive user.

- `boolean isHidden()`
- `void setHidden(boolean b)`

  gets or sets a flag that a builder tool should hide this feature.

`java.beans.PropertyDescriptor` 1.1

- `PropertyDescriptor(String propertyName, Class<?> beanClass)`

- `PropertyDescriptor(String propertyName, Class<?> beanClass, String getMethod, String setMethod)`

  constructs a `PropertyDescriptor` object. The methods throw an `IntrospectionException` if an error occurred during introspection. The first constructor assumes that you follow the standard convention for the names of the `get` and `set` methods.

- `Class<?> getPropertyType()`

  returns a `Class` object for the property type.

- `Method getReadMethod()`

- `Method getWriteMethod()`

  returns the method to get or set the property.

`java.beans.IndexedPropertyDescriptor` 1.1

- `IndexedPropertyDescriptor(String propertyName, Class<?> beanClass)`

- `IndexedPropertyDescriptor(String propertyName, Class<?> beanClass, String getMethod, String setMethod, String indexedGetMethod, String indexedSetMethod)`

  constructs an `IndexedPropertyDescriptor` for the index property. The first constructor assumes that you follow the standard convention for the names of the `get` and `set` methods.

- `Method getIndexedReadMethod()`

- `Method getIndexedWriteMethod()`

  returns the method to get or set an indexed value in the property.

Property Editors

If you add an integer or string property to a bean, then that property is automatically displayed in the bean's property inspector. But what happens if you add a property whose values cannot easily be edited in a text field, for example, a `Date` or a `Color`? Then, you need to provide a separate component by which the user can specify the property value. Such components are called *property editors*. For example, a property editor for a date object might be a calendar that lets the user scroll through the months and pick a date. A property editor for a `Color` object would let the user select the red, green, and blue components of the color.

Actually, NetBeans already has a property editor for colors. Also, of course, there are property editors for basic types such as `String` (a text field) and `boolean` (a checkbox).

The process for supplying a new property editor is slightly involved. First, you create a bean info class to accompany your bean. Override the `getPropertyDescriptors` method. That method returns an array of `PropertyDescriptor` objects. You create one object for each property that should be displayed on a property editor, *even those for which you just want the default editor.*

You construct a `PropertyDescriptor` by supplying the name of the property and the class of the bean that contains it.

Code View:
```
PropertyDescriptor descriptor = new PropertyDescriptor("titlePosition", ChartBean.class);
```

Then you call the `setPropertyEditorClass` method of the `PropertyDescriptor` class.

```
descriptor.setPropertyEditorClass(TitlePositionEditor.class);
```

Next, you build an array of descriptors for properties of your bean. For example, the chart bean that we discuss in this section has five properties:

- A `Color` property, `graphColor`
- A `String` property, `title`
- An `int` property, `titlePosition`
- A `double[]` property, `values`
- A `boolean` property, `inverse`

The code in Listing 8-3 shows the `ChartBeanBeanInfo` class that specifies the property editors for these properties. It achieves the following:

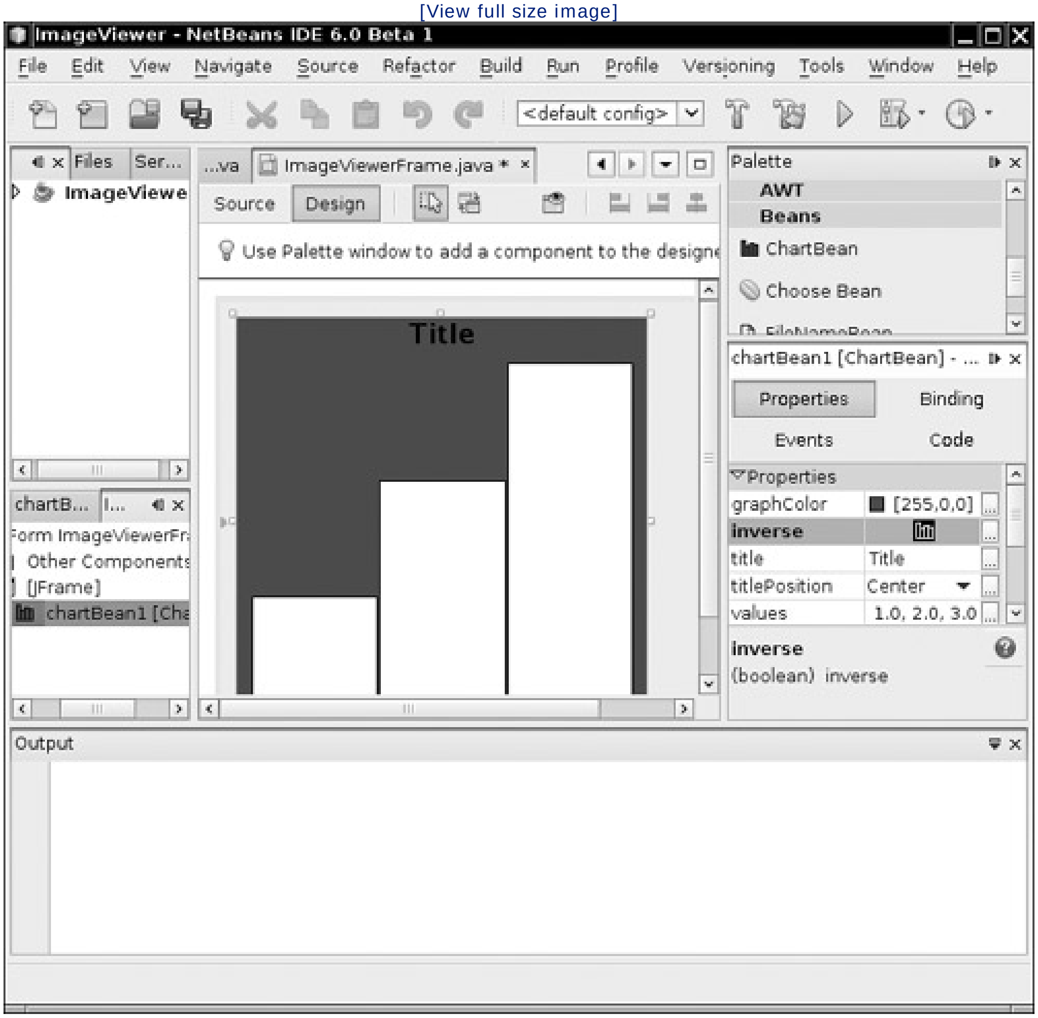
1. The getPropertyDescriptors method returns a descriptor for each property. The title and graphColor properties are used with the default editors; that is, the string and color editors that come with the builder tool.

2. The titlePosition, values, and inverse properties use special editors of type TitlePositionEditor, DoubleArrayEditor, and InverseEditor, respectively.

Figure 8-10 shows the chart bean. You can see the title on the top. Its position can be set to left, center, or right. The values property specifies the graph values. If the inverse property is true, then the background is colored and the bars of the chart are white. You can find the code for the chart bean with the book's companion code; the bean is simply a modification of the chart applet in Volume I, Chapter 10.

Figure 8-10. The chart bean

[View full size image]



Listing 8-3. **ChartBeanBeanInfo.java**

Code View:

```
package com.horstmann.corejava;

import java.awt.*;
import java.beans.*;

/**
 * The bean info for the chart bean, specifying the property editors.
 * @version 1.20 2007-10-05
 * @author Cay Horstmann
 */
public class ChartBeanBeanInfo extends SimpleBeanInfo
{
   public ChartBeanBeanInfo()
   {
      iconColor16 = loadImage("ChartBean_COLOR_16x16.gif");
      iconColor32 = loadImage("ChartBean_COLOR_32x32.gif");
      iconMono16 = loadImage("ChartBean_MONO_16x16.gif");
      iconMono32 = loadImage("ChartBean_MONO_32x32.gif");

      try
      {
         PropertyDescriptor titlePositionDescriptor = new PropertyDescriptor("titlePosition",
               ChartBean.class);
         titlePositionDescriptor.setPropertyEditorClass(TitlePositionEditor.class);
         PropertyDescriptor inverseDescriptor = new PropertyDescriptor("inverse", ChartBean.class);
         inverseDescriptor.setPropertyEditorClass(InverseEditor.class);
         PropertyDescriptor valuesDescriptor = new PropertyDescriptor("values", ChartBean.class);
         valuesDescriptor.setPropertyEditorClass(DoubleArrayEditor.class);
         propertyDescriptors = new PropertyDescriptor[] {
               new PropertyDescriptor("title", ChartBean.class), titlePositionDescriptor,
               valuesDescriptor, new PropertyDescriptor("graphColor", ChartBean.class),
               inverseDescriptor };
       }
       catch (IntrospectionException e)
       {
          e.printStackTrace();
       }
   }

   public PropertyDescriptor[] getPropertyDescriptors()
   {
      return propertyDescriptors;
   }

   public Image getIcon(int iconType)
   {
      if (iconType == BeanInfo.ICON_COLOR_16x16) return iconColor16;
      else if (iconType == BeanInfo.ICON_COLOR_32x32) return iconColor32;
      else if (iconType == BeanInfo.ICON_MONO_16x16) return iconMono16;
      else if (iconType == BeanInfo.ICON_MONO_32x32) return iconMono32;
      else return null;
   }

   private PropertyDescriptor[] propertyDescriptors;
   private Image iconColor16;
   private Image iconColor32;
   private Image iconMono16;
   private Image iconMono32;
```

```
}
```

java.beans.PropertyDescriptor 1.1

- `PropertyDescriptor(String name, Class<?> beanClass)`

  constructs a `PropertyDescriptor` object.

Parameters:		
	`name`	The name of the property
	`beanClass`	The class of the bean to which the property belongs

- `void setPropertyEditorClass(Class<?> editorClass)`

  sets the class of the property editor to be used with this property.

java.beans.BeanInfo 1.1

- `PropertyDescriptor[] getPropertyDescriptors()`

  returns a descriptor for each property that should be displayed in the property inspector for the bean.

Writing Property Editors

Before we get into the mechanics of writing property editors, we should point out that a editor is under the control of the builder, not the bean. When the builder displays the property inspector, it carries out the following steps for each bean property.

1. It instantiates a property editor.
2. It asks the bean to tell it the current value of the property.
3. It then asks the property editor to display the value.

A property editor must supply a default constructor, and it must implement the `PropertyEditor` interface. You will usually want to extend the convenience `PropertyEditorSupport` class that provides default versions of these methods.

For every property editor you write, you choose one of three ways to display and edit the property value:

- As a text string (define `getAsText` and `setAsText`)

- As a choice field (define `getAsText`, `setAsText`, and `getTags`)

- Graphically, by painting it (define `isPaintable`, `paintValue`, `supportsCustomEditor`, and `getCustomEditor`)

We have a closer look at these choices in the following sections.

String-Based Property Editors

Simple property editors work with text strings. You override the `setAsText` and `getAsText` methods. For example, our chart bean has a property that lets you choose where the title should be displayed: Left, Center, or Right. These choices are implemented as an enumeration

```
public enum Position { LEFT, CENTER, RIGHT };
```

But of course, we don't want them to appear as uppercase strings `LEFT`, `CENTER`, `RIGHT`—unless we are trying to enter the User Interface Hall of Horrors. Instead, we define a property editor whose `getAsText` method picks a string that looks pleasing to the developer:

```
class TitlePositionEditor extends PropertyEditorSupport
{
   public String getAsText()
   {
      int index = ((ChartBean.Position) getValue()).ordinal();
      return tags[index];
   }
   . . .
   private String[] tags = { "Left", "Center", "Right" };
}
```

Ideally, these strings should appear in the current locale, not necessarily in English, but we leave that as an exercise to the reader.

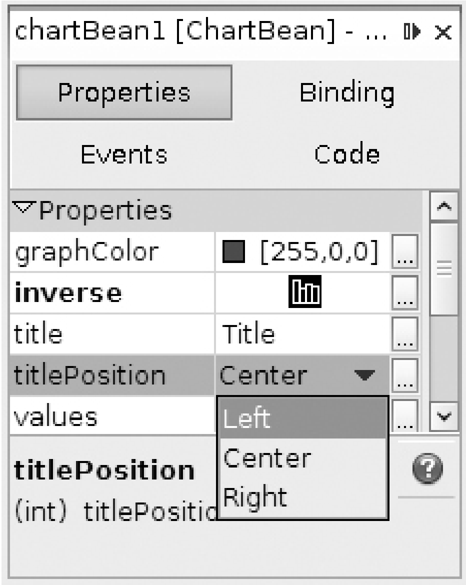
Conversely, we need to supply a method that converts a text string back to the property value:

```
public void setAsText(String s)
{
   int index = Arrays.asList(tags).indexOf(s);
   if (index >= 0) setValue(ChartBean.Position.values()[index]);
}
```

If we simply supply these two methods, the property inspector will provide a text field. It is initialized by a call to `getAsText`, and the `setAsText` method is called when we are done editing. Of course, in our situation, this is not a good choice for the titlePosition property, unless, of course, we are also competing for entry into the User Interface Hall of Shame. It is better to display all valid settings in a combo box (see Figure 8-11). The PropertyEditorSupport class gives a simple mechanism for indicating that a combo box is appropriate. Simply write a getTags method that returns an array of strings.

```
public String[] getTags() { return tags; }
```

Figure 8-11. Custom property editors at work



The default getTags method returns null, indicating that a text field is appropriate for editing the property value.

When supplying the getTags method, you still need to supply the getAsText and setAsText methods. The getTags method simply specifies the strings that should be offered to the user. The getAsText/setAsText methods translate between the strings and the data type of the property (which can be a string, an integer, an enumeration, or a completely different type).

Finally, property editors should implement the getJavaInitializationString method.With this method, you can give the builder tool the Java code that sets a property to its current value. The builder tool uses this string for automatic code generation. Here is the method for the TitlePositionEditor:

Code View:
```
public String getJavaInitializationString()
{
   return ChartBean.Position.class.getName().replace('$', '.') + "." + getValue();
}
```

This method returns a string such as "com.horstmann.corejava.ChartBean.Position.LEFT". Try it out in NetBeans: If you edit the titlePosition property, NetBeans inserts code such as

Code View:
```
chartBean1.setTitlePosition(com.horstmann.corejava.ChartBean.Position.LEFT);
```

In our situation, the code is a bit cumbersome because `ChartBean.Position.class.getName()` is the string `"com.horstmann.corejava.ChartBean$Position"`. We replace the `$` with a period, and add the result of invoking `toString` on the enumeration value.

Note

If a property has a custom editor that does not implement the `getJavaInitializationString` method, NetBeans does not know how to generate code and produces a setter with parameter `???`.

Listing 8-4 shows the code for this property editor.

Listing 8-4. **`TitlePositionEditor.java`**

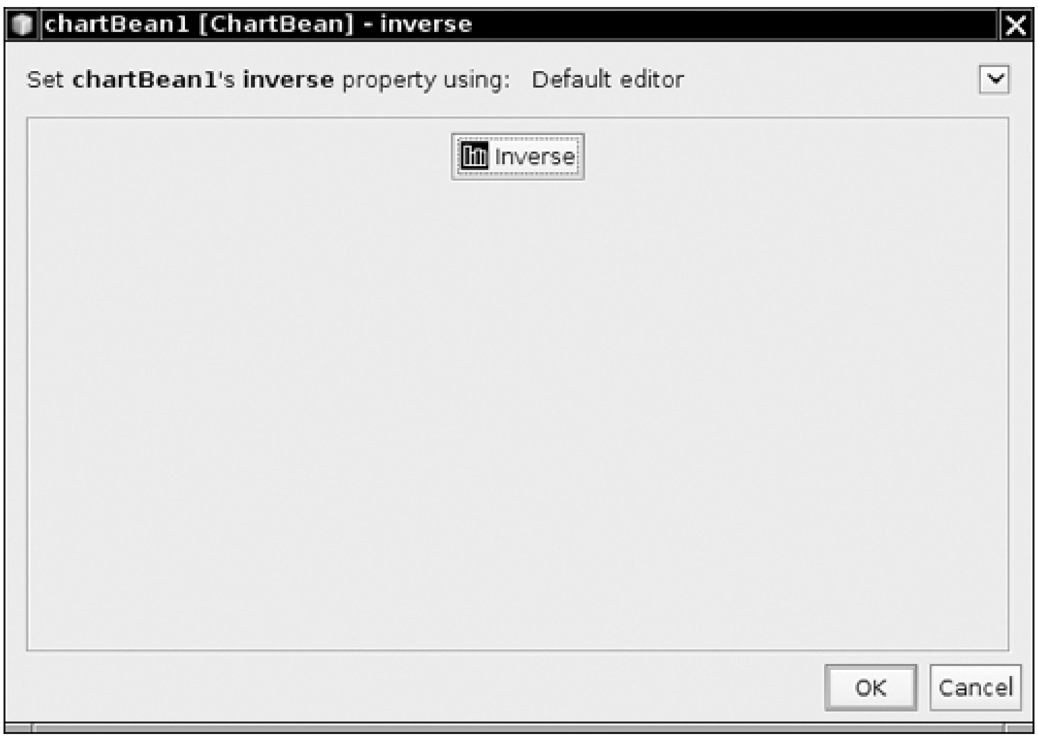
Code View:
```
package com.horstmann.corejava;

import java.beans.*;
import java.util.*;

/**
 * A custom editor for the titlePosition property of the ChartBean. The editor lets the user
 * choose between Left, Center, and Right
 * @version 1.20 2007-12-14
 * @author Cay Horstmann
 */
public class TitlePositionEditor extends PropertyEditorSupport
{
   public String[] getTags()
   {
      return tags;
   }

   public String getJavaInitializationString()
   {
      return ChartBean.Position.class.getName().replace('$', '.') + "." + getValue();
   }

   public String getAsText()
   {
      int index = ((ChartBean.Position) getValue()).ordinal();
      return tags[index];
   }

```

```
   public void setAsText(String s)
   {
      int index = Arrays.asList(tags).indexOf(s);
      if (index >= 0) setValue(ChartBean.Position.values()[index]);
   }

   private String[] tags = { "Left", "Center", "Right" };
}
```

GUI-Based Property Editors

A sophisticated property should not be edited as text. Instead, a graphical representation is displayed in the property inspector, in the small area that would otherwise hold a text field or combo box. When the user clicks on that area, a custom editor dialog box pops up (see Figure 8-12). The dialog box contains a component to edit the property values, supplied by the property editor, and various buttons, supplied by the builder environment. In our example, the customizer is rather spare, containing a single button. The book's companion code contains a more elaborate editor for editing the chart values.

Figure 8-12. A custom editor dialog box



To build a GUI-based property editor, you first tell the property inspector that you will paint the value and not use a string.

Override the `getAsText` method in the `PropertyEditor` interface to return `null` and the `isPaintable` method to return `true`.

Then, you implement the `paintValue` method. It receives a `Graphics` context and the coordinates of the rectangle inside which you can paint. Note that this rectangle is typically small, so you can't have a very elaborate representation. We simply draw one of two icons (which you can see in Figure 8-11 on page 717).

```
public void paintValue(Graphics g, Rectangle box)
{
   ImageIcon icon = (Boolean) getValue() ? inverseIcon : normalIcon;
   int x = bounds.x + (bounds.width - icon.getIconWidth()) / 2;
   int y = bounds.y + (bounds.height - icon.getIconHeight()) / 2;
   g.drawImage(icon.getImage(), x, y, null);
}
```


This graphical representation is not editable. The user must click on it to pop up a custom editor.

You indicate that you will have a custom editor by overriding the `supportsCustomEditor` in the `PropertyEditor` interface to return `true`.

Next, the `getCustomEditor` method of the `PropertyEditor` interface constructs and returns an object of the custom editor class.

Listing 8-5 shows the code for the `InverseEditor` that displays the current property value in the property inspector. Listing 8-6 shows the code for the custom editor panel for changing the value.

Listing 8-5. **`InverseEditor.java`**

Code View:

```
package com.horstmann.corejava;

import java.awt.*;
import java.beans.*;
import javax.swing.*;

/**
 * The property editor for the inverse property of the ChartBean. The inverse property toggles
 * between colored graph bars and colored background.
 * @version 1.30 2007-10-03
 * @author Cay Horstmann
 */
public class InverseEditor extends PropertyEditorSupport
{
   public Component getCustomEditor()
   {
      return new InverseEditorPanel(this);
   }

   public boolean supportsCustomEditor()
   {
      return true;
   }

   public boolean isPaintable()
   {
      return true;
   }

   public String getAsText()
```

```
   {
      return null;
   }

   public String getJavaInitializationString()
   {
      return "" + getValue();
   }

   public void paintValue(Graphics g, Rectangle bounds)
   {
      ImageIcon icon = (Boolean) getValue() ? inverseIcon : normalIcon;
      int x = bounds.x + (bounds.width - icon.getIconWidth()) / 2;
      int y = bounds.y + (bounds.height - icon.getIconHeight()) / 2;
      g.drawImage(icon.getImage(), x, y, null);
   }

   private ImageIcon inverseIcon = new ImageIcon(getClass().getResource(
         "ChartBean_INVERSE_16x16.gif"));
   private ImageIcon normalIcon =
         new ImageIcon(getClass().getResource("ChartBean_MONO_16x16.gif"));
}
```

Listing 8-6. **InverseEditorPanel.java**

Code View:

```
package com.horstmann.corejava;

import java.awt.event.*;
import java.beans.*;
import javax.swing.*;

/**
 * The panel for setting the inverse property. It contains a button to toggle between normal
 * and inverse coloring.
 * @version 1.30 2007-10-03
 * @author Cay Horstmann
 */
public class InverseEditorPanel extends JPanel
{
   public InverseEditorPanel(PropertyEditorSupport ed)
   {
      editor = ed;
      button = new JButton();
      updateButton();
      button.addActionListener(new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               editor.setValue(!(Boolean) editor.getValue());
               updateButton();
            }
         });
      add(button);
```

```
   }

   private void updateButton()
   {
      if ((Boolean) editor.getValue())
      {
         button.setIcon(inverseIcon);
         button.setText("Inverse");
      }
      else
      {
         button.setIcon(normalIcon);
         button.setText("Normal");
      }
   }

   private JButton button;
   private PropertyEditorSupport editor;
   private ImageIcon inverseIcon = new ImageIcon(getClass().getResource(
         "ChartBean_INVERSE_16x16.gif"));
   private ImageIcon normalIcon =
         new ImageIcon(getClass().getResource("ChartBean_MONO_16x16.gif"));
}
```


API

`java.beans.PropertyEditor` 1.1

- `Object getValue()`

  returns the current value of the property. Basic types are wrapped into object wrappers.

- `void setValue(Object newValue)`

  sets the property to a new value. Basic types must be wrapped into object wrappers.

Parameters:	`newValue`	The new value of the object; should be a newly created object that the property can own

- `String getAsText()`

  override this method to return a string representation of the current value of the property. The default returns `null` to indicate that the property cannot be represented as a string.

- `void setAsText(String text)`

  override this method to set the property to a new value that is obtained by parsing the text. May throw an `IllegalArgumentException` if the text does not represent a legal value or if this property cannot be represented as a string.

- `String[] getTags()`

  override this method to return an array of all possible string representations of the property values so they can be displayed in a Choice box. The default returns `null` to indicate that there is not a finite set of string values.

- `boolean isPaintable()`

  override this method to return `true` if the class uses the `paintValue` method to display the property.

- `void paintValue(Graphics g, Rectangle bounds)`

  override this method to represent the value by drawing into a graphics context in the specified place on the component used for the property inspector.

- `boolean supportsCustomEditor()`

  override this method to return `true` if the property editor has a custom editor.

- `Component getCustomEditor()`

  override this method to return the component that contains a customized GUI for editing the property value.

- `String getJavaInitializationString()`

  override this method to return a Java code string that can be used to generate code that initializes the property value. Examples are `"0"`, `"new Color(64, 64, 64)"`.

Customizers

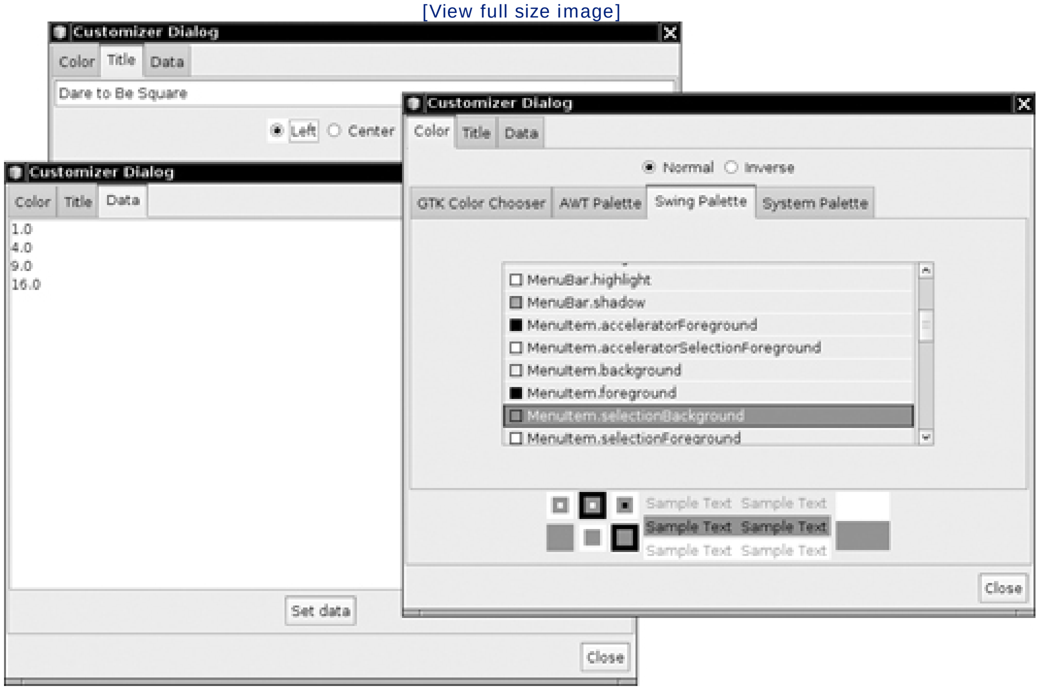
A property editor is responsible for allowing the user to set one property at a time. Especially if certain properties of a bean relate to each other, it might be more user friendly to give users a way to edit multiple properties at the same time. To enable this feature, you supply a *customizer* instead of (or in addition to) multiple property editors.

Moreover, some beans might have features that are not exposed as properties and that therefore cannot be edited through the property inspector. For those beans, a customizer is essential.

In the example program for this section, we develop a customizer for the chart bean. The customizer lets you set several properties of the chart bean in one dialog box, as shown in Figure 8-13.

Figure 8-13. The customizer for the ChartBean

[View full size image]



To add a customizer to your bean, you must supply a `BeanInfo` class and override the `getBeanDescriptor` method, as shown in the following example.

```
public ChartBean2BeanInfo extends SimpleBeanInfo
{
   public BeanDescriptor getBeanDescriptor()
   {
      return beanDescriptor;
   }
   . . .
   private BeanDescriptor beanDescriptor
```

```
      = new BeanDescriptor(ChartBean2.class, ChartBean2Customizer.class);
}
```

Note that you need not follow any naming pattern for the customizer class. (Nevertheless, it is customary to name the customizer as *BeanName*`Customizer`.)

You will see in the next section how to implement a customizer.

API **`java.beans.BeanInfo` 1.1**

- `BeanDescriptor getBeanDescriptor()`

  returns a `BeanDescriptor` object that describes features of the bean.

API **`java.beans.BeanDescriptor` 1.1**

- `BeanDescriptor(Class<?> beanClass, Class<?> customizerClass)`

  constructs a `BeanDescriptor` object for a bean that has a customizer.

Parameters:	`beanClass`	The `Class` object for the bean
	`customizerClass`	The `Class` object for the bean's customizer

Writing a Customizer Class

Any customizer class you write must have a default constructor, extend the `Component` class, and implement the `Customizer` interface. That interface has only three methods:

- The `setObject` method, which takes a parameter that specifies the bean being customized
- The `addPropertyChangeListener` and `removePropertyChangeListener` methods, which manage the collection of listeners that are notified when a property is changed in the customizer

It is a good idea to update the visual appearance of the target bean by broadcasting a `PropertyChangeEvent` whenever the user changes any of the property values, not just when the user is at the end of the customization process.

Unlike property editors, customizers are not automatically displayed. In NetBeans, you must right-click on the bean and select the Customize menu option to pop up the customizer. At that point, the builder calls the `setObject` method of the customizer. Notice that your customizer is created before it is actually linked to an instance of your bean. Therefore, you cannot assume any information about the state of a bean in the constructor.

Because customizers typically present the user with many options, it is often handy to use the tabbed pane user interface. We use this approach and have the customizer extend the `JTabbedPane` class.

The customizer gathers the following information in three panes:

- Graph color and inverse mode
- Title and title position

- Data points

Of course, developing this kind of user interface can be tedious to code—our example devotes over 100 lines just to set it up in the constructor. However, this task requires only the usual Swing programming skills, and we don't dwell on the details here.

One trick is worth keeping in mind. You often need to edit property values in a customizer. Rather than implementing a new interface for setting the property value of a particular class, you can simply locate an existing property editor and add it to your user interface! For example, in our `ChartBean2` customizer, we need to set the graph color. Because we know that NetBeans has a perfectly good property editor for colors, we locate it as follows:

```
PropertyEditor colorEditor = PropertyEditorManager.findEditor(Color.Class);
Component colorEditorComponent = colorEditor.getCustomEditor();
```

Once we have all components laid out, we initialize their values in the `setObject` method. The `setObject` method is called when the customizer is displayed. Its parameter is the bean that is being customized. To proceed, we store that bean reference—we'll need it later to notify the bean of property changes. Then, we initialize each user interface component. Here is a part of the `setObject` method of the chart bean customizer that does this initialization:

```
public void setObject(Object obj)
{
   bean = (ChartBean2) obj;
   titleField.setText(bean.getTitle());
   colorEditor.setValue(bean.getGraphColor());
   . . .
}
```

Finally, we hook up event handlers to track the user's activities. Whenever the user changes the value of a component, the component fires an event that our customizer must handle. The event handler must update the value of the property in the bean and must also fire a `PropertyChangeEvent` so that other listeners (such as the property inspector) can be updated. Let us follow that process with a couple of user interface elements in the chart bean customizer.

When the user types a new title, we want to update the title property. We attach a `DocumentListener` to the text field into which the user types the title.

```
titleField.getDocument().addDocumentListener(new
```

```
   DocumentListener()
   {
      public void changedUpdate(DocumentEvent event)
      {
         setTitle(titleField.getText());
      }
      public void insertUpdate(DocumentEvent event)
      {
         setTitle(titleField.getText());
      }
      public void removeUpdate(DocumentEvent event)
      {
         setTitle(titleField.getText());
      }
   });
```

The three listener methods call the `setTitle` method of the customizer. That method calls the bean to update the property value and then fires a property change event. (This update is necessary only for properties that are not bound.) Here is the code for the `setTitle` method.

```
public void setTitle(String newValue)
{
   if (bean == null) return;
   String oldValue = bean.getTitle();
   bean.setTitle(newValue);
   firePropertyChange("title", oldValue, newValue);
}
```

When the color value changes in the color property editor, we want to update the graph color of the bean. We track the color changes by attaching a listener to the property editor. Perhaps confusingly, that editor also sends out property change events.

```
colorEditor.addPropertyChangeListener(new
   PropertyChangeListener()
   {
      public void propertyChange(PropertyChangeEvent event)
      {
         setGraphColor((Color) colorEditor.getValue());
      }
   });
```

Listing 8-7 provides the full code of the chart bean customizer.

Listing 8-7. **ChartBean2Customizer.java**

Code View:

```
package com.horstmann.corejava;

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

```java
import javax.swing.event.*;

/**
 * A customizer for the chart bean that allows the user to edit all chart properties in a
 * single tabbed dialog.
 * @version 1.12 2007-10-03
 * @author Cay Horstmann
 */
public class ChartBean2Customizer extends JTabbedPane implements Customizer
```

```java
{
   public ChartBean2Customizer()
   {
      data = new JTextArea();
      JPanel dataPane = new JPanel();
      dataPane.setLayout(new BorderLayout());
      dataPane.add(new JScrollPane(data), BorderLayout.CENTER);
```

```java
      JButton dataButton = new JButton("Set data");
      dataButton.addActionListener(new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               setData(data.getText());
            }
         });
      JPanel panel = new JPanel();
      panel.add(dataButton);
      dataPane.add(panel, BorderLayout.SOUTH);

      JPanel colorPane = new JPanel();
      colorPane.setLayout(new BorderLayout());

      normal = new JRadioButton("Normal", true);
      inverse = new JRadioButton("Inverse", false);
      panel = new JPanel();
      panel.add(normal);
      panel.add(inverse);
      ButtonGroup group = new ButtonGroup();
      group.add(normal);
      group.add(inverse);
      normal.addActionListener(new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               setInverse(false);
            }
         });

      inverse.addActionListener(new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               setInverse(true);
            }
         });

      colorEditor = PropertyEditorManager.findEditor(Color.class);
      colorEditor.addPropertyChangeListener(new PropertyChangeListener()
         {
            public void propertyChange(PropertyChangeEvent event)
```

```
            {
               setGraphColor((Color) colorEditor.getValue());
            }
         });

      colorPane.add(panel, BorderLayout.NORTH);
      colorPane.add(colorEditor.getCustomEditor(), BorderLayout.CENTER);

      JPanel titlePane = new JPanel();
      titlePane.setLayout(new BorderLayout());

      group = new ButtonGroup();
      position = new JRadioButton[3];
      position[0] = new JRadioButton("Left");
      position[1] = new JRadioButton("Center");
      position[2] = new JRadioButton("Right");

      panel = new JPanel();
      for (int i = 0; i < position.length; i++)
      {
         final ChartBean2.Position pos = ChartBean2.Position.values()[i];
         panel.add(position[i]);
         group.add(position[i]);
         position[i].addActionListener(new ActionListener()
            {
               public void actionPerformed(ActionEvent event)
               {
                  setTitlePosition(pos);
               }
            });
      }

      titleField = new JTextField();
      titleField.getDocument().addDocumentListener(new DocumentListener()
         {
            public void changedUpdate(DocumentEvent evt)
            {
               setTitle(titleField.getText());
            }

            public void insertUpdate(DocumentEvent evt)
            {
               setTitle(titleField.getText());
            }

            public void removeUpdate(DocumentEvent evt)
            {
               setTitle(titleField.getText());
            }
         });

      titlePane.add(titleField, BorderLayout.NORTH);
      JPanel panel2 = new JPanel();
      panel2.add(panel);
      titlePane.add(panel2, BorderLayout.CENTER);
      addTab("Color", colorPane);
      addTab("Title", titlePane);
      addTab("Data", dataPane);

```

```
    }

    /**
     * Sets the data to be shown in the chart.
     * @param s a string containing the numbers to be displayed, separated by white space
     */
    public void setData(String s)
    {
       StringTokenizer tokenizer = new StringTokenizer(s);
```



```

       int i = 0;
       double[] values = new double[tokenizer.countTokens()];
       while (tokenizer.hasMoreTokens())
       {
          String token = tokenizer.nextToken();
          try
```



```
          {
             values[i] = Double.parseDouble(token);
             i++;
          }
          catch (NumberFormatException e)
          {
          }
       }
       setValues(values);
    }

    /**
     * Sets the title of the chart.
     * @param newValue the new title
     */
    public void setTitle(String newValue)
    {
       if (bean == null) return;
       String oldValue = bean.getTitle();
       bean.setTitle(newValue);
       firePropertyChange("title", oldValue, newValue);
    }

    /**
     * Sets the title position of the chart.
     * @param i the new title position (ChartBean2.LEFT, ChartBean2.CENTER, or ChartBean2.RIGHT)
     */
    public void setTitlePosition(ChartBean2.Position pos)
    {
       if (bean == null) return;
       ChartBean2.Position oldValue = bean.getTitlePosition();
       bean.setTitlePosition(pos);
       firePropertyChange("titlePosition", oldValue, pos);
    }

    /**
     * Sets the inverse setting of the chart.
     * @param b true if graph and background color are inverted
     */
    public void setInverse(boolean b)
    {
       if (bean == null) return;
       boolean oldValue = bean.isInverse();
```

```
      bean.setInverse(b);
      firePropertyChange("inverse", oldValue, b);
   }

   /**
    * Sets the values to be shown in the chart.
    * @param newValue the new value array
    */
   public void setValues(double[] newValue)
   {
      if (bean == null) return;
      double[] oldValue = bean.getValues();
      bean.setValues(newValue);
      firePropertyChange("values", oldValue, newValue);
   }

   /**
    * Sets the color of the chart
    * @param newValue the new color
    */
   public void setGraphColor(Color newValue)
   {
      if (bean == null) return;
      Color oldValue = bean.getGraphColor();
      bean.setGraphColor(newValue);
      firePropertyChange("graphColor", oldValue, newValue);
   }

   public void setObject(Object obj)
   {
      bean = (ChartBean2) obj;

      data.setText("");
      for (double value : bean.getValues())
         data.append(value + "\n");

      normal.setSelected(!bean.isInverse());
      inverse.setSelected(bean.isInverse());

      titleField.setText(bean.getTitle());

      for (int i = 0; i < position.length; i++)
         position[i].setSelected(i == bean.getTitlePosition().ordinal());

      colorEditor.setValue(bean.getGraphColor());
   }

   private ChartBean2 bean;
   private PropertyEditor colorEditor;

   private JTextArea data;
   private JRadioButton normal;
   private JRadioButton inverse;
   private JRadioButton[] position;
   private JTextField titleField;
}
```

API `java.beans.Customizer` 1.1

- `void setObject(Object bean)`

  specifies the bean to customize.

JavaBeans Persistence

JavaBeans persistence uses JavaBeans properties to save beans to a stream and to read them back at a later time or in a different virtual machine. In this regard, JavaBeans persistence is similar to object serialization. (See Chapter 1 for more information on serialization.) However, there is an important difference: JavaBeans persistence is *suitable for long-term storage*.

When an object is serialized, its instance fields are written to a stream. If the implementation of a class changes, then its instance fields can change. You cannot simply read files that contain serialized objects of older versions. It is possible to detect version differences and translate between old and new data representations. However, the process is extremely tedious and should only be applied in desperate situations. Plainly, serialization is unsuitable for long-term storage. For that reason, all Swing components have the following message in their documentation: "Warning: Serialized objects of this class will not be compatible with future Swing releases. The current serialization support is appropriate for short term storage or RMI between applications."

The long-term persistence mechanism was invented as a solution for this problem. It was originally intended for drag-and-drop GUI design tools. The design tool saves the result of mouse clicks—a collection of frames, panels, buttons, and other Swing components—in a file, using the long-term persistence format. The running program simply opens that file. This approach cuts out the tedious source code for laying out and wiring up Swing components. Sadly, it has not been widely implemented.

Note

The Bean Builder at http://bean-builder.dev.java.net is an experimental GUI builder with support for long-term persistence.

The basic idea behind JavaBeans persistence is simple. Suppose you want to save a `JFrame` object to a file so that you can retrieve it later. If you look into the source code of the `JFrame` class and its superclasses, then you see dozens of instance fields. If the frame were to be serialized, all of the field values would need to be written. But think about how a frame is constructed:

```
JFrame frame = new JFrame();
frame.setTitle("My Application");
frame.setVisible(true);
```

The default constructor initializes all instance fields, and a couple of properties are set. If you archive the `frame` object, the JavaBeans persistence mechanism saves exactly these statements in XML format:

```
<object class="javax.swing.JFrame">
   <void property="title">
     <string>My Application</string>
   </void>
   <void property="visible">
     <boolean>true</boolean>
   </void>
```

```
</object>
```

When the object is read back, the statements are *executed*: A JFrame object is constructed, and its title and visible properties are set to the given values. It does not matter if the internal representation of the JFrame has changed in the meantime. All that matters is that you can restore the object by setting properties.

Note that only those properties that are different from the default are archived. The XMLEncoder makes a default JFrame and compares its property with the frame that is being archived. Property setter statements are generated only for properties that are different from the default. This process is called *redundancy elimination*. As a result, the archives are generally smaller than the result of serialization. (When serializing Swing components, the difference is particularly dramatic because Swing objects have a lot of state, most of which is never changed from the default.)

Of course, there are minor technical hurdles with this approach. For example, the call


```
frame.setSize(600, 400);
```

is not a property setter. However, the XMLEncoder can cope with this: It writes the statement

```
<void property="bounds">
   <object class="java.awt.Rectangle">
       <int>0</int>
       <int>0</int>
       <int>600</int>
       <int>400</int>
   </object>
</void>
```

To save an object to a stream, use an XMLEncoder:

```
XMLEncoder out = new XMLEncoder(new FileOutputStream(. . .));
out.writeObject(frame);
out.close();
```

To read it back, use an XMLDecoder:

```
XMLDecoder in = new XMLDecoder(new FileInputStream(. . .));
JFrame newFrame = (JFrame) in.readObject();
in.close();
```

The program in Listing 8-8 shows how a frame can load and save *itself* (see Figure 8-14). When you run the program, first click the Save button and save the frame to a file. Then move the original frame to a different position and click Load to see another frame pop up at the original location. Have a look inside the XML file that the program produces.

Figure 8-14. The PersistentFrameTest program

[View full size image]

If you look closely at the XML output, you will find that the XMLEncoder carries out an amazing amount of work when it saves the frame. The XMLEncoder produces statements that carry out the following actions:

- Set various frame properties: size, layout, defaultCloseOperation, title, and so on.
- Add buttons to the frame.
- Add action listeners to the buttons.

Here, we had to construct the action listers with the EventHandler class. The XMLEncoder cannot archive arbitrary inner classes, but it knows how to handle EventHandler objects.

Listing 8-8. **PersistentFrameTest.java**

Code View:
```
import java.awt.*;
import java.awt.event.*;
import java.beans.*;
import java.io.*;
import javax.swing.*;

/**
 * This program demonstrates the use of an XML encoder and decoder to save and restore a frame.
 * @version 1.01 2007-10-03
 * @author Cay Horstmann
 */
public class PersistentFrameTest
{
   public static void main(String[] args)
   {
      chooser = new JFileChooser();
      chooser.setCurrentDirectory(new File("."));
      PersistentFrameTest test = new PersistentFrameTest();
      test.init();
   }

   public void init()
```

```
   {
      frame = new JFrame();
      frame.setLayout(new FlowLayout());
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.setTitle("PersistentFrameTest");
      frame.setSize(400, 200);

      JButton loadButton = new JButton("Load");
      frame.add(loadButton);
      loadButton.addActionListener(EventHandler.create(ActionListener.class, this, "load"));

      JButton saveButton = new JButton("Save");
      frame.add(saveButton);
      saveButton.addActionListener(EventHandler.create(ActionListener.class, this, "save"));

      frame.setVisible(true);
   }

   public void load()
   {
      // show file chooser dialog
      int r = chooser.showOpenDialog(null);

      // if file selected, open
      if(r == JFileChooser.APPROVE_OPTION)
      {
         try
         {
            File file = chooser.getSelectedFile();
            XMLDecoder decoder = new XMLDecoder(new FileInputStream(file));
            decoder.readObject();
            decoder.close();
         }
         catch (IOException e)
         {
            JOptionPane.showMessageDialog(null, e);
         }
      }
   }

   public void save()
   {
      if (chooser.showSaveDialog(null) == JFileChooser.APPROVE_OPTION)
      {
         try
         {
            File file = chooser.getSelectedFile();
            XMLEncoder encoder = new XMLEncoder(new FileOutputStream(file));
            encoder.writeObject(frame);
            encoder.close();
         }
         catch (IOException e)
         {
            JOptionPane.showMessageDialog(null, e);
         }
      }
   }

   private static JFileChooser chooser;
```

```
   private JFrame frame;
}
```

Using JavaBeans Persistence for Arbitrary Data

JavaBeans persistence is not limited to the storage of Swing components. You can use the mechanism to store *any* collection of objects, provided you follow a few simple rules. In the following sections, you learn how you can use JavaBeans persistence as a long-term storage format for your own data.

The `XMLEncoder` has built-in support for the following types:

- `null`
- All primitive types and their wrappers
- Enumerations (since Java SE 6)
- `String`
- Arrays
- Collections and maps
- The reflection types `Class`, `Field`, `Method`, and `Proxy`
- The AWT types `Color`, `Cursor`, `Dimension`, `Font`, `Insets`, `Point`, `Rectangle`, and `ImageIcon`
- AWT and Swing components, borders, layout managers, and models
- Event handlers

Writing a Persistence Delegate to Construct an Object

Using JavaBeans persistence is trivial if one can obtain the state of every object by setting properties. But in real programs, there are always a few classes that don't work that way. Consider, for example, the `Employee` class of Volume I, Chapter 4. `Employee` isn't a well-behaved bean. It doesn't have a default constructor, and it doesn't have methods `setName`, `setSalary`, `setHireDay`. To overcome this problem, you define a *persistence delegate*. Such a delegate is responsible for generating an XML encoding of an object.

The persistence delegate for the `Employee` class overrides the `instantiate` method to produce an *expression* that constructs an object.

```
PersistenceDelegate delegate = new
    DefaultPersistenceDelegate()
    {
       protected Expression instantiate(Object oldInstance, Encoder out)
       {
          Employee e = (Employee) oldInstance;
          GregorianCalendar c = new GregorianCalendar();
          c.setTime(e.getHireDay());
          return new Expression(oldInstance, Employee.class, "new",
             new Object[]
             {
                e.getName(),
                e.getSalary(),
                c.get(Calendar.YEAR),
                c.get(Calendar.MONTH),
                c.get(Calendar.DATE)
             });
       }
    };
```


This means: "To re-create `oldInstance`, call the `new` method (i.e., the constructor) on the `Employee.class` object, and supply the given parameters." The parameter name `oldInstance` is a bit misleading—this is simply the instance that is being saved.

To install the persistence delegate, you have two choices. You can associate it with a specific `XMLWriter`:

```
out.setPersistenceDelegate(Employee.class, delegate);
```

Alternatively, you can set the `persistenceDelegate` attribute of the *bean descriptor* of the `BeanInfo`:

```
BeanInfo info = Introspector.getBeanInfo(GregorianCalendar.class);
info.getBeanDescriptor().setValue("persistenceDelegate", delegate);
```

Once the delegate is installed, you can save `Employee` objects. For example, the statements

```
Object myData = new Employee("Harry Hacker", 50000, 1989, 10, 1);
out.writeObject(myData);
```

generate the following output:

```
<object class="Employee">
   <string>Harry Hacker</string>
   <double>50000.0</double>
   <int>1989</int>
   <int>10</int>
   <int>1</int>
</object>
```

Note

You only need to tweak the *encoding* process. There are no special decoding methods. The decoder simply executes the statements and expressions that it finds in its XML input.

Constructing an Object from Properties

If all constructor parameters can be obtained by accessing properties of `oldInstance`, then you need not write the `instantiate` method yourself. Instead, simply construct a `DefaultPersistenceDelegate` and supply the property names.

For example, the following statement sets the persistence delegate for the `Rectangle2D.Double` class:

Code View:
```
out.setPersistenceDelegate(Rectangle2D.Double.class,
   new DefaultPersistenceDelegate(new String[] { "x", "y", "width", "height" }));
```

This tells the encoder: "To encode a `Rectangle2D.Double` object, get its `x`, `y`, `width`, and `height` properties and call the constructor with those four values." As a result, the output contains an element such as the following:

```
<object class="java.awt.geom.Rectangle2D$Double">
   <double>5.0</double>
   <double>10.0</double>
   <double>20.0</double>
   <double>30.0</double>
</object>
```

If you are the author of the class, you can do even better. Annotate the constructor with the `@ConstructorProperties` annotation. Suppose, for example, the `Employee` class had a constructor with three parameters (name, salary, and hire day). Then we could have annotated the constructor as follows:

```
@ConstructorProperties({"name", "salary", "hireDay"})
public Employee(String n, double s, Date d)
```

This tells the encoder to call the `getName`, `getSalary`, and `getHireDay` property getters and write the resulting values into the `object` expression.

The `@ConstructorProperties` annotation was introduced in Java SE 6, and has so far only been used for classes in the Java Management Extensions (JMX) API.

Constructing an Object with a Factory Method

Sometimes, you need to save objects that are obtained from factory methods, not constructors. Consider, for example, how you get an `InetAddress` object:

```
byte[] bytes = new byte[] { 127, 0, 0, 1};
InetAddress address = InetAddress.getByAddress(bytes);
```

The `instantiate` method of the `PersistenceDelegate` produces a call to the factory method.

```
protected Expression instantiate(Object oldInstance, Encoder out)
{
   return new Expression(oldInstance, InetAddress.class, "getByAddress",
      new Object[] { ((InetAddress) oldInstance).getAddress() });
}
```


A sample output is


```
<object class="java.net.Inet4Address" method="getByAddress">
   <array class="byte" length="4">
      <void index="0">
         <byte>127</byte>
      </void>
      <void index="3">
         <byte>1</byte>
      </void>
   </array>
</object>
```

Caution

You must install this delegate with the concrete subclass, such as `Inet4Address`, not with the abstract `InetAddress` class!

Postconstruction Work

The state of some classes is built up by calls to methods that are not property setters. You can cope with that situation by overriding the `initialize` method of the `DefaultPersistenceDelegate`. The `initialize` method is called after the `instantiate` method. You can generate a sequence of *statements* that are recorded in the archive.

For example, consider the `BitSet` class. To re-create a `BitSet` object, you set all the bits that were present in the original. The following `initialize` method generates the necessary statements:

Code View:

```
protected void initialize(Class<?> type, Object oldInstance, Object newInstance, Encoder out)
{
   super.initialize(type, oldInstance, newInstance, out);
   BitSet bs = (BitSet) oldInstance;
   for (int i = bs.nextSetBit(0); i >= 0; i = bs.nextSetBit(i + 1))
   out.writeStatement(new Statement(bs, "set", new Object[] { i, i + 1, true } ));
}
```

A sample output is

```
<object class="java.util.BitSet">
   <void method="set">
      <int>1</int>
      <int>2</int>
      <boolean>true</boolean>
   </void>
   <void method="set">
      <int>4</int>
      <int>5</int>
      <boolean>true</boolean>
   </void>
</object>
```

Note

It would make more sense to write `new Statement(bs, "set", new Object[] { i } )`, but then the `XMLWriter` produces an unsightly statement that sets a property with an empty name.

Transient Properties

Occasionally, a class has a property with a getter and setter that the `XMLDecoder` discovers, but you don't want to include the property value in the archive. To suppress archiving of a property, mark it as *transient* in the property descriptor. For example, the following statement marks the `removeMode` property of the `DamageReporter` class (which you will see in detail in the next section) as transient.

```
BeanInfo info = Introspector.getBeanInfo(DamageReport.class);
for (PropertyDescriptor desc : info.getPropertyDescriptors())
   if (desc.getName().equals("removeMode"))
      desc.setValue("transient", Boolean.TRUE);
```

The program in Listing 8-9 shows the various persistence delegates at work. Keep in mind that this program shows a worst-case scenario—in actual applications, many classes can be archived without the use of delegates.

Listing 8-9. **PersistenceDelegateTest.java**

Code View:

```
import java.awt.geom.*;
import java.beans.*;
import java.net.*;
import java.util.*;

/**
 * This program demonstrates various persistence delegates.
```

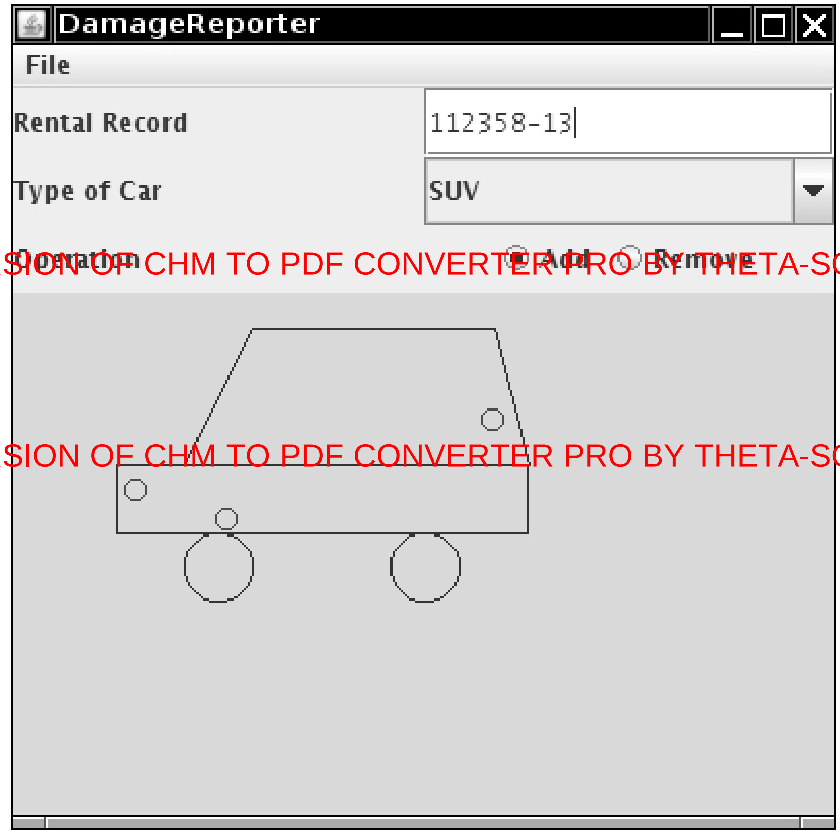
```
 * @version 1.01 2007-10-03
 * @author Cay Horstmann
 */
public class PersistenceDelegateTest
{
   public static class Point
   {
      @ConstructorProperties( { "x", "y" })
      public Point(int x, int y)
      {
         this.x = x;
         this.y = y;
      }

      public int getX()
      {
         return x;
      }

      public int getY()
      {
         return y;
      }

      private final int x, y;
   }

   public static void main(String[] args) throws Exception
   {
      PersistenceDelegate delegate = new PersistenceDelegate()
         {
            protected Expression instantiate(Object oldInstance, Encoder out)
            {
               Employee e = (Employee) oldInstance;
               GregorianCalendar c = new GregorianCalendar();
               c.setTime(e.getHireDay());
               return new Expression(oldInstance, Employee.class, "new", new Object[] {
                     e.getName(), e.getSalary(), c.get(Calendar.YEAR), c.get(Calendar.MONTH),
                     c.get(Calendar.DATE) });
            }
         };
      BeanInfo info = Introspector.getBeanInfo(Employee.class);
      info.getBeanDescriptor().setValue("persistenceDelegate", delegate);

      XMLEncoder out = new XMLEncoder(System.out);
      out.setExceptionListener(new ExceptionListener()
         {
            public void exceptionThrown(Exception e)
            {
               e.printStackTrace();
            }
         });

      out.setPersistenceDelegate(Rectangle2D.Double.class, new DefaultPersistenceDelegate(
            new String[] { "x", "y", "width", "height" }));

      out.setPersistenceDelegate(Inet4Address.class, new DefaultPersistenceDelegate()
         {
            protected Expression instantiate(Object oldInstance, Encoder out)
```

```
            {
               return new Expression(oldInstance, InetAddress.class, "getByAddress",
                     new Object[] { ((InetAddress) oldInstance).getAddress() });
            }
         });

      out.setPersistenceDelegate(BitSet.class, new DefaultPersistenceDelegate()
         {
            protected void initialize(Class<?> type, Object oldInstance, Object newInstance,
                  Encoder out)
            {
               super.initialize(type, oldInstance, newInstance, out);
               BitSet bs = (BitSet) oldInstance;
               for (int i = bs.nextSetBit(0); i >= 0; i = bs.nextSetBit(i + 1))
                  out.writeStatement(new Statement(bs, "set",
                                                   new Object[] { i, i + 1, true }));
            }
         });

      out.writeObject(new Employee("Harry Hacker", 50000, 1989, 10, 1));
      out.writeObject(new Point(17, 29));
      out.writeObject(new java.awt.geom.Rectangle2D.Double(5, 10, 20, 30));
      out.writeObject(InetAddress.getLocalHost());
      BitSet bs = new BitSet();
      bs.set(1, 4);
      bs.clear(2, 3);
      out.writeObject(bs);
      out.close();
   }
}
```

A Complete Example for JavaBeans Persistence

We end the description of JavaBeans persistence with a complete example (see Figure 8-15). This application writes a damage report for a rental car. The rental car agent enters the rental record, selects the car type, uses the mouse to click on damaged areas on the car, and saves the report. The application can also load existing damage reports. Listing 8-10 contains the code for the program.

Figure 8-15. The DamageReporter application



The application uses JavaBeans persistence to save and load `DamageReport` objects (see Listing 8-11). It illustrates the following aspects of the persistence technology:

- Properties are automatically saved and restored. Nothing needs to be done for the `rentalRecord` and `carType` properties.
- Postconstruction work is required to restore the damage locations. The persistence delegate generates statements that call the `click` method.
- The `Point2D.Double` class needs a `DefaultPersistenceDelegate` that constructs a point from its `x` and `y` properties.
- The `removeMode` property (which specifies whether mouse clicks add or remove damage marks) is transient because it should not be saved in damage reports.

Here is a sample damage report:

Code View:
```
<?xml version="1.0" encoding="UTF-8"?>
<java version="1.5.0" class="java.beans.XMLDecoder">
   <object class="DamageReport">
```

```
      <object class="java.lang.Enum" method="valueOf">
         <class>DamageReport$CarType</class>
         <string>SEDAN</string>
      </object>
      <void property="rentalRecord">
         <string>12443-19</string>
      </void>
      <void method="click">
         <object class="java.awt.geom.Point2D$Double">
            <double>181.0</double>
            <double>84.0</double>
         </object>
      </void>
      <void method="click">
         <object class="java.awt.geom.Point2D$Double">
            <double>162.0</double>
            <double>66.0</double>
         </object>
      </void>
   </object>
</java>
```

Note

The sample application does *not* use JavaBeans persistence to save the GUI of the application. That might be of interest to creators of development tools, but here we are focusing on how to use the persistence mechanism to store *application data*.

This example ends our discussion of JavaBeans persistence. In summary, JavaBeans persistence archives are

- Suitable for long-term storage.
- Small and fast.
- Easy to create.
- Human editable.
- A part of standard Java.

Listing 8-10. **DamageReporter.java**

Code View:

```
import java.awt.*;
import java.awt.event.*;
import java.awt.geom.*;
import java.beans.*;
import java.io.*;
import java.util.*;
import javax.swing.*;
```



```

/**
 * This program demonstrates the use of an XML encoder and decoder. All GUI and drawing
 * code is collected in this class. The only interesting pieces are the action listeners for
 * openItem and saveItem. Look inside the DamageReport class for encoder customizations.
 * @version 1.01 2004-10-03
 * @author Cay Horstmann
```



```
 */
public class DamageReporter extends JFrame
{
   public static void main(String[] args)
   {
      JFrame frame = new DamageReporter();
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.setVisible(true);
   }

   public DamageReporter()
   {
      setTitle("DamageReporter");
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

      chooser = new JFileChooser();
      chooser.setCurrentDirectory(new File("."));

      report = new DamageReport();
      report.setCarType(DamageReport.CarType.SEDAN);

      // set up the menu bar
      JMenuBar menuBar = new JMenuBar();
      setJMenuBar(menuBar);

      JMenu menu = new JMenu("File");
      menuBar.add(menu);

      JMenuItem openItem = new JMenuItem("Open");
      menu.add(openItem);
      openItem.addActionListener(new ActionListener()
         {
            public void actionPerformed(ActionEvent evt)
            {
               // show file chooser dialog
               int r = chooser.showOpenDialog(null);

               // if file selected, open
               if (r == JFileChooser.APPROVE_OPTION)
               {
                  try
                  {
                     File file = chooser.getSelectedFile();
                     XMLDecoder decoder = new XMLDecoder(new FileInputStream(file));
```

```
                     report = (DamageReport) decoder.readObject();
                     decoder.close();
                     rentalRecord.setText(report.getRentalRecord());
                     carType.setSelectedItem(report.getCarType());
                     repaint();
                  }
                  catch (IOException e)
                  {
                     JOptionPane.showMessageDialog(null, e);
                  }
               }
            }
         });

      JMenuItem saveItem = new JMenuItem("Save");
      menu.add(saveItem);
      saveItem.addActionListener(new ActionListener()
         {
            public void actionPerformed(ActionEvent evt)
            {
               report.setRentalRecord(rentalRecord.getText());
               chooser.setSelectedFile(new File(rentalRecord.getText() + ".xml"));

               // show file chooser dialog
               int r = chooser.showSaveDialog(null);

               // if file selected, save
               if (r == JFileChooser.APPROVE_OPTION)
               {
                  try
                  {
                     File file = chooser.getSelectedFile();
                     XMLEncoder encoder = new XMLEncoder(new FileOutputStream(file));
                     report.configureEncoder(encoder);
                     encoder.writeObject(report);
                     encoder.close();
                  }
                  catch (IOException e)
                  {
                     JOptionPane.showMessageDialog(null, e);
                  }
               }
            }
         });

      JMenuItem exitItem = new JMenuItem("Exit");
      menu.add(exitItem);
      exitItem.addActionListener(new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               System.exit(0);
            }
         });

      // combo box for car type
      rentalRecord = new JTextField();
      carType = new JComboBox();
      carType.addItem(DamageReport.CarType.SEDAN);
```

```
      carType.addItem(DamageReport.CarType.WAGON);
      carType.addItem(DamageReport.CarType.SUV);

      carType.addActionListener(new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               DamageReport.CarType item = (DamageReport.CarType) carType.getSelectedItem();
               report.setCarType(item);
               repaint();
            }
         });

      // component for showing car shape and damage locations
      carComponent = new JComponent()
         {
            public void paintComponent(Graphics g)
            {
               Graphics2D g2 = (Graphics2D) g;
               g2.setColor(new Color(0.9f, 0.9f, 0.45f));
               g2.fillRect(0, 0, getWidth(), getHeight());
               g2.setColor(Color.BLACK);
               g2.draw(shapes.get(report.getCarType()));
               report.drawDamage(g2);
            }
         };
      carComponent.addMouseListener(new MouseAdapter()
         {
            public void mousePressed(MouseEvent event)
            {
               report.click(new Point2D.Double(event.getX(), event.getY()));
               repaint();
            }
         });

      // radio buttons for click action
      addButton = new JRadioButton("Add");
      removeButton = new JRadioButton("Remove");
      ButtonGroup group = new ButtonGroup();
      JPanel buttonPanel = new JPanel();
      group.add(addButton);
      buttonPanel.add(addButton);
      group.add(removeButton);
      buttonPanel.add(removeButton);
      addButton.setSelected(!report.getRemoveMode());
      removeButton.setSelected(report.getRemoveMode());
      addButton.addActionListener(new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               report.setRemoveMode(false);
            }
         });
      removeButton.addActionListener(new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               report.setRemoveMode(true);
            }
```

```
         });

      // layout components
      JPanel gridPanel = new JPanel();
      gridPanel.setLayout(new GridLayout(0, 2));
      gridPanel.add(new JLabel("Rental Record"));
      gridPanel.add(rentalRecord);
      gridPanel.add(new JLabel("Type of Car"));
      gridPanel.add(carType);
      gridPanel.add(new JLabel("Operation"));
      gridPanel.add(buttonPanel);

      add(gridPanel, BorderLayout.NORTH);
      add(carComponent, BorderLayout.CENTER);
   }

   private JTextField rentalRecord;
   private JComboBox carType;
   private JComponent carComponent;
   private JRadioButton addButton;
   private JRadioButton removeButton;
   private DamageReport report;
   private JFileChooser chooser;

   private static final int DEFAULT_WIDTH = 400;
   private static final int DEFAULT_HEIGHT = 400;

   private static Map<DamageReport.CarType, Shape> shapes =
        new EnumMap<DamageReport.CarType, Shape>(DamageReport.CarType.class);

   static
   {
      int width = 200;
      int x = 50;
      int y = 50;
      Rectangle2D.Double body = new Rectangle2D.Double(x, y + width / 6, width - 1, width / 6);
      Ellipse2D.Double frontTire = new Ellipse2D.Double(x + width / 6, y + width / 3,
            width / 6, width / 6);
      Ellipse2D.Double rearTire = new Ellipse2D.Double(x + width * 2 / 3, y + width / 3,
            width / 6, width / 6);

      Point2D.Double p1 = new Point2D.Double(x + width / 6, y + width / 6);
      Point2D.Double p2 = new Point2D.Double(x + width / 3, y);
      Point2D.Double p3 = new Point2D.Double(x + width * 2 / 3, y);
      Point2D.Double p4 = new Point2D.Double(x + width * 5 / 6, y + width / 6);

      Line2D.Double frontWindshield = new Line2D.Double(p1, p2);
      Line2D.Double roofTop = new Line2D.Double(p2, p3);
      Line2D.Double rearWindshield = new Line2D.Double(p3, p4);

      GeneralPath sedanPath = new GeneralPath();
      sedanPath.append(frontTire, false);
      sedanPath.append(rearTire, false);
      sedanPath.append(body, false);
      sedanPath.append(frontWindshield, false);
      sedanPath.append(roofTop, false);
      sedanPath.append(rearWindshield, false);
      shapes.put(DamageReport.CarType.SEDAN, sedanPath);

```

```
      Point2D.Double p5 = new Point2D.Double(x + width * 11 / 12, y);
      Point2D.Double p6 = new Point2D.Double(x + width, y + width / 6);
      roofTop = new Line2D.Double(p2, p5);
      rearWindshield = new Line2D.Double(p5, p6);

      GeneralPath wagonPath = new GeneralPath();
      wagonPath.append(frontTire, false);
      wagonPath.append(rearTire, false);
      wagonPath.append(body, false);
      wagonPath.append(frontWindshield, false);
      wagonPath.append(roofTop, false);
      wagonPath.append(rearWindshield, false);
      shapes.put(DamageReport.CarType.WAGON, wagonPath);

      Point2D.Double p7 = new Point2D.Double(x + width / 3, y - width / 6);
      Point2D.Double p8 = new Point2D.Double(x + width * 11 / 12, y - width / 6);
      frontWindshield = new Line2D.Double(p1, p7);
      roofTop = new Line2D.Double(p7, p8);
      rearWindshield = new Line2D.Double(p8, p6);

      GeneralPath suvPath = new GeneralPath();
      suvPath.append(frontTire, false);
      suvPath.append(rearTire, false);
      suvPath.append(body, false);
      suvPath.append(frontWindshield, false);
      suvPath.append(roofTop, false);
      suvPath.append(rearWindshield, false);
      shapes.put(DamageReport.CarType.SUV, suvPath);
   }
}
```


Listing 8-11. **DamageReport.java**

Code View:

```
import java.awt.*;
import java.awt.geom.*;
import java.beans.*;
import java.util.*;

/**
 * This class describes a vehicle damage report that will be saved and loaded with the
 * long-term persistence mechanism.
 * @version 1.21 2004-08-30
 * @author Cay Horstmann
 */
public class DamageReport
{
   public enum CarType
   {
      SEDAN, WAGON, SUV
   }

```

```
   // this property is saved automatically
   public void setRentalRecord(String newValue)
   {
      rentalRecord = newValue;
   }

   public String getRentalRecord()
   {
      return rentalRecord;
   }

   // this property is saved automatically
   public void setCarType(CarType newValue)
   {
      carType = newValue;
   }

   public CarType getCarType()
   {
      return carType;
   }

   // this property is set to be transient
   public void setRemoveMode(boolean newValue)
   {
      removeMode = newValue;
   }

   public boolean getRemoveMode()
   {
      return removeMode;
   }

   public void click(Point2D p)
   {
      if (removeMode)
      {
         for (Point2D center : points)
         {
            Ellipse2D circle = new Ellipse2D.Double(center.getX() - MARK_SIZE, center.getY()
                  - MARK_SIZE, 2 * MARK_SIZE, 2 * MARK_SIZE);
            if (circle.contains(p))
            {
               points.remove(center);
               return;
            }
         }
      }
      else points.add(p);
   }

   public void drawDamage(Graphics2D g2)
   {
      g2.setPaint(Color.RED);
      for (Point2D center : points)
      {
         Ellipse2D circle = new Ellipse2D.Double(center.getX() - MARK_SIZE, center.getY()
               - MARK_SIZE, 2 * MARK_SIZE, 2 * MARK_SIZE);
         g2.draw(circle);
```

```
      }
   }

   public void configureEncoder(XMLEncoder encoder)
   {
      // this step is necessary to save Point2D.Double objects
      encoder.setPersistenceDelegate(Point2D.Double.class, new DefaultPersistenceDelegate(
            new String[] { "x", "y" }));

      // this step is necessary because the array list of points is not
      // (and should not be) exposed as a property
      encoder.setPersistenceDelegate(DamageReport.class, new DefaultPersistenceDelegate()
         {
            protected void initialize(Class<?> type, Object oldInstance, Object newInstance,
                  Encoder out)
            {
               super.initialize(type, oldInstance, newInstance, out);
               DamageReport r = (DamageReport) oldInstance;

               for (Point2D p : r.points)
                 out.writeStatement(new Statement(oldInstance, "click", new Object[] { p }));
            }
         });

   }

   // this step is necessary to make the removeMode property transient
   static
   {
      try
      {
         BeanInfo info = Introspector.getBeanInfo(DamageReport.class);
         for (PropertyDescriptor desc : info.getPropertyDescriptors())
           if (desc.getName().equals("removeMode")) desc.setValue("transient", Boolean.TRUE);
      }
      catch (IntrospectionException e)
      {
         e.printStackTrace();
      }
   }

   private String rentalRecord;
   private CarType carType;
   private boolean removeMode;
   private ArrayList<Point2D> points = new ArrayList<Point2D>();

   private static final int MARK_SIZE = 5;
}
```

`java.beans.XMLEncoder` 1.4

- `XMLEncoder(OutputStream out)`

  constructs an `XMLEncoder` that sends its output to the given stream.

- `void writeObject(Object obj)`

  archives the given object.

- `void writeStatement(Statement stat)`

  writes the given statement to the archive. This method should only be called from a persistence delegate.

`java.beans.Encoder` 1.4

- `void setPersistenceDelegate(Class<?> type, PersistenceDelegate delegate)`

- `PersistenceDelegate getPersistenceDelegate(Class<?> type)`

  sets or gets the delegate for archiving objects of the given type.

- `void setExceptionListener(ExceptionListener listener)`

- `ExceptionListener getExceptionListener()`

  sets or gets the exception listener that is notified if an exception occurs during the encoding process.

`java.beans.ExceptionListener` 1.4

- `void exceptionThrown(Exception e)`

  is called when an exception was thrown during the encoding or decoding process.

API

`java.beans.XMLDecoder` 1.4

- `XMLDecoder(InputStream in)`

  constructs an `XMLDecoder` that reads an archive from the given input stream.

- `Object readObject()`

  reads the next object from the archive.

- `void setExceptionListener(ExceptionListener listener)`

- `ExceptionListener getExceptionListener()`

  sets or gets the exception listener that is notified if an exception occurs during the encoding process.

API

`java.beans.PersistenceDelegate` 1.4

- `protected abstract Expression instantiate(Object oldInstance, Encoder out)`

  returns an expression for instantiating an object that is equivalent to `oldInstance`.

- `protected void initialize(Class<?> type, Object oldInstance, Object newInstance, Encoder out)`

  writes statements to `out` that turn `newInstance` into an object that is equivalent to `oldInstance`.

`java.beans.DefaultPersistenceDelegate` 1.4

- `DefaultPersistenceDelegate()`

  constructs a persistence delegate for a class with a zero-parameter constructor.

- `DefaultPersistenceDelegate(String[] propertyNames)`

  constructs a persistence delegate for a class whose construction parameters are the values of the given properties.

- `protected Expression instantiate(Object oldInstance, Encoder out)`

  returns an expression for invoking the constructor with either no parameters or the values of the properties specified in the constructor.

- `protected void initialize(Class<?> type, Object oldInstance, Object newInstance, Encoder out)`

  writes statements to `out` that apply property setters to `newInstance`, attempting to turn it into an object that is equivalent to `oldInstance`.

`java.beans.Expression` 1.4

- `Expression(Object value, Object target, String methodName, Object[] parameters)`

  constructs an expression that calls the given method on `target`, with the given parameters. The result of the expression is assumed to be `value`. To call a constructor, `target` should be a `Class` object and `methodName` should be `"new"`.

java.beans.Statement 1.4

- `Statement(Object target, String methodName, Object[] parameters)`

  constructs a statement that calls the given method on `target`, with the given parameters.

You have now worked your way through three long chapters on GUI programming with Swing, AWT, and JavaBeans. In the next chapter, we move on to an entirely different topic: security. Security has always been a core feature of the Java platform. As the world in which we live and compute gets more dangerous, a thorough understanding of Java security will be of increasing importance for many developers.

Chapter 9. Security

- Class Loaders
- Bytecode Verification
- Security Managers and Permissions
- User Authentication
- Digital Signatures
- Code Signing
- Encryption

When Java technology first appeared on the scene, the excitement was not about a well-crafted programming language but about the possibility of safely executing applets that are delivered over the Internet (see Volume I, Chapter 10 for more information about applets). Obviously, delivering executable applets is practical only when the recipients are sure that the code can't wreak havoc on their machines. For this reason, security was and is a major concern of both the designers and the users of Java technology. This means that unlike other languages and systems, where security was implemented as an afterthought or a reaction to break-ins, security mechanisms are an integral part of Java technology.

Three mechanisms help ensure safety:

- Language design features (bounds checking on arrays, no unchecked type conversions, no pointer arithmetic, and so on).
- An access control mechanism that controls what the code can do (such as file access, network access, and so on).
- Code signing, whereby code authors can use standard cryptographic algorithms to authenticate Java code. Then, the users of the code can determine exactly who created the code and whether the code has been altered after it was signed.

We will first discuss *class loaders* that check class files for integrity when they are loaded into the virtual machine. We will demonstrate how that mechanism can detect tampering with class files.

For maximum security, both the default mechanism for loading a class and a custom class loader need to work with a *security manager* class that controls what actions code can perform. You'll see in detail how to configure Java platform security.

Finally, you'll see the cryptographic algorithms supplied in the `java.security` package, which allow for code signing and user authentication.

As always, we focus on those topics that are of greatest interest to application programmers. For an in-depth view, we recommend the book *Inside Java 2 Platform Security: Architecture, API Design, and Implementation*, 2nd ed., by Li Gong, Gary Ellison, and Mary Dageforde (Prentice Hall PTR 2003).

Class Loaders

A Java compiler converts source instructions for the Java virtual machine. The virtual machine code is stored in a class file with a `.class` extension. Each class file contains the definition and implementation code for one class or interface. These class files must be interpreted by a program that can translate the instruction set of the virtual machine into the machine language of the target machine.

Note that the virtual machine loads only those class files that are needed for the execution of a program. For example, suppose program execution starts with `MyProgram.class`. Here are the steps that the virtual machine carries out.

1. The virtual machine has a mechanism for loading class files, for example, by reading the files from disk or by requesting them from the Web; it uses this mechanism to load the contents of the `MyProgram` class file.

2. If the `MyProgram` class has fields or superclasses of another class type, their class files are loaded as well. (The process of loading all the classes that a given class depends on is called *resolving* the class.)

3. The virtual machine then executes the `main` method in `MyProgram` (which is static, so no instance of a class needs to be created).

4. If the `main` method or a method that `main` calls requires additional classes, these are loaded next.

The class loading mechanism doesn't just use a single class loader, however. Every Java program has at least three class loaders:

- The bootstrap class loader

- The extension class loader

- The system class loader (also sometimes called the application class loader)

The bootstrap class loader loads the system classes (typically, from the JAR file `rt.jar`). It is an integral part of the virtual machine and is usually implemented in C. There is no `ClassLoader` object corresponding to the bootstrap class loader. For example,

```
String.class.getClassLoader()
```

returns `null`.

The extension class loader loads "standard extensions" from the *jre/lib/ext* directory. You can drop JAR files into that directory, and the extension class loader will find the classes in them, even without any class path. (Some people recommend this mechanism to avoid the "class path from hell," but see the next cautionary note.)

The system class loader loads the application classes. It locates classes in the directories and JAR/ZIP files on the class path, as set by the `CLASSPATH` environment variable or the `-classpath` command-line option.

In Sun's Java implementation, the extension and system class loaders are implemented in Java. Both are instances of the `URLClassLoader` class.

Caution

You can run into grief if you drop a JAR file into the *jre/lib/ext* directory and one of its classes needs to load a class that is not a system or extension class. The extension class loader *does not use the class path*. Keep that in mind before you use the extension directory as a way to manage your class file hassles.

Note

In addition to all the places already mentioned, classes can be loaded from the *jre/lib/endorsed* directory. This mechanism can only be used to replace certain standard Java libraries (such as those for XML and CORBA support) with newer versions. See http://java.sun.com/javase/6/docs/technotes/guides/standards/index.html for details.

The Class Loader Hierarchy

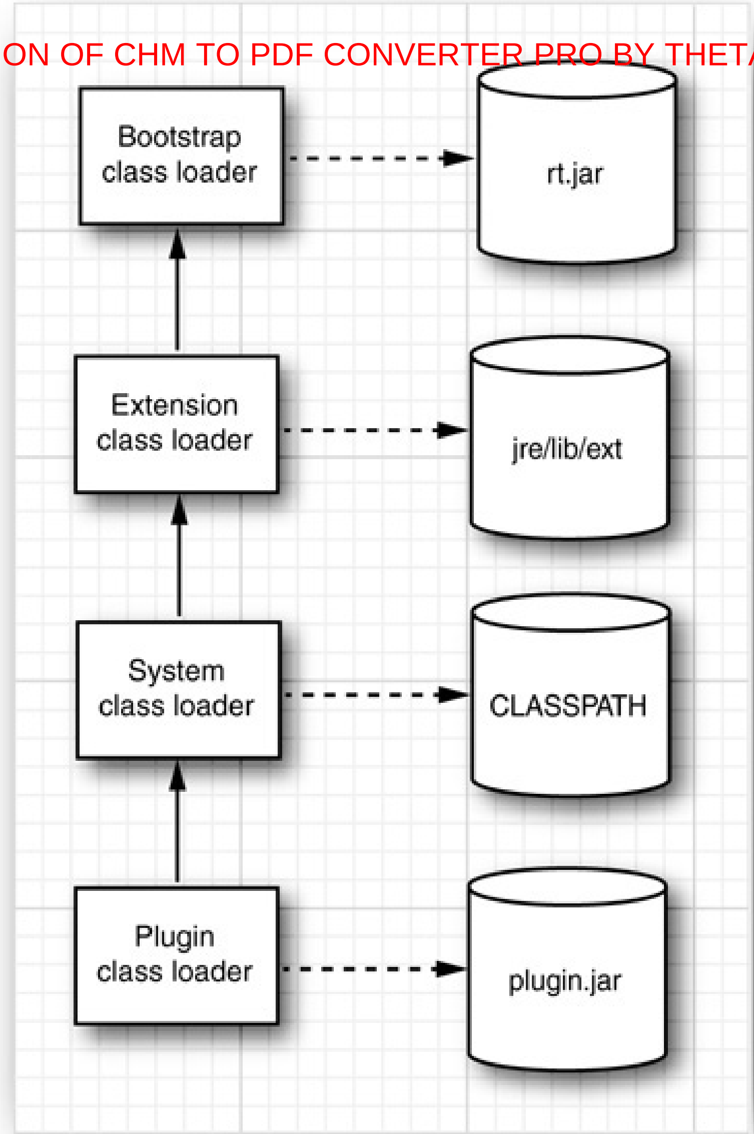
Class loaders have a *parent/child* relationship. Every class loader except for the bootstrap class loader has a parent class loader. A class loader is supposed to give its parent a chance to load any given class and only load it if the parent has failed. For example, when the system class loader is asked to load a system class (say, `java.util.ArrayList`), then it first asks the extension class loader. That class loader first asks the bootstrap class loader. The bootstrap class loader finds and loads the class in `rt.jar`, and neither of the other class loaders searches any further.

Some programs have a plugin architecture in which certain parts of the code are packaged as optional plugins. If the plugins are packaged as JAR files, you can simply load the plugin classes with an instance of `URLClassLoader`.

```
URL url = new URL("file:///path/to/plugin.jar");
URLClassLoader pluginLoader = new URLClassLoader(new URL[] { url });
Class<?> cl = pluginLoader.loadClass("mypackage.MyClass");
```

Because no parent was specified in the `URLClassLoader` constructor, the parent of the `pluginLoader` is the system class loader. Figure 9-1 shows the hierarchy.

Figure 9-1. The class loader hierarchy



Most of the time, you don't have to worry about the class loader hierarchy. Generally, classes are loaded because they are required by other classes, and that process is transparent to you.

Occasionally, you need to intervene and specify a class loader. Consider this example.

- Your application code contains a helper method that calls `Class.forName(classNameString)`.
- That method is called from a plugin class.
- The `classNameString` specifies a class that is contained in the plugin JAR.

The author of the plugin has the reasonable expectation that the class should be loaded. However, the helper method's class was loaded by the system class loader, and that is the class loader used by `Class.forName`. The classes in the plugin JAR are not visible. This phenomenon is called *classloader inversion*.

To overcome this problem, the helper method needs to use the correct class loader. It can require the class loader as a parameter. Alternatively, it can require that the correct class loader is set as the *context class loader* of the current thread. This strategy is used by many frameworks (such as the JAXP and JNDI frameworks that we discussed in Chapters 2 and 4).

Each thread has a reference to a class loader, called the context class loader. The main thread's context class loader is the system class loader. When a new thread is created, its context class loader is set to the creating thread's context class loader. Thus, if you don't do anything, then all threads have their context class loader set to the system class loader.

However, you can set any class loader by calling

```
Thread t = Thread.currentThread();
t.setContextClassLoader(loader);
```

The helper method can then retrieve the context class loader:

```
Thread t = Thread.currentThread();
ClassLoader loader = t.getContextClassLoader();
Class cl = loader.loadClass(className);
```

The question remains when the context class loader is set to the plugin class loader. The application designer must make this decision. Generally, it is a good idea to set the context class loader when invoking a method of a plugin class that was loaded with a different class loader. Alternatively, the caller of the helper method can set the context class loader.

Tip

If you write a method that loads a class by name, it is a good idea to offer the caller the choice between passing an explicit class loader and using the context class loader. Don't simply use the class loader of the method's class.

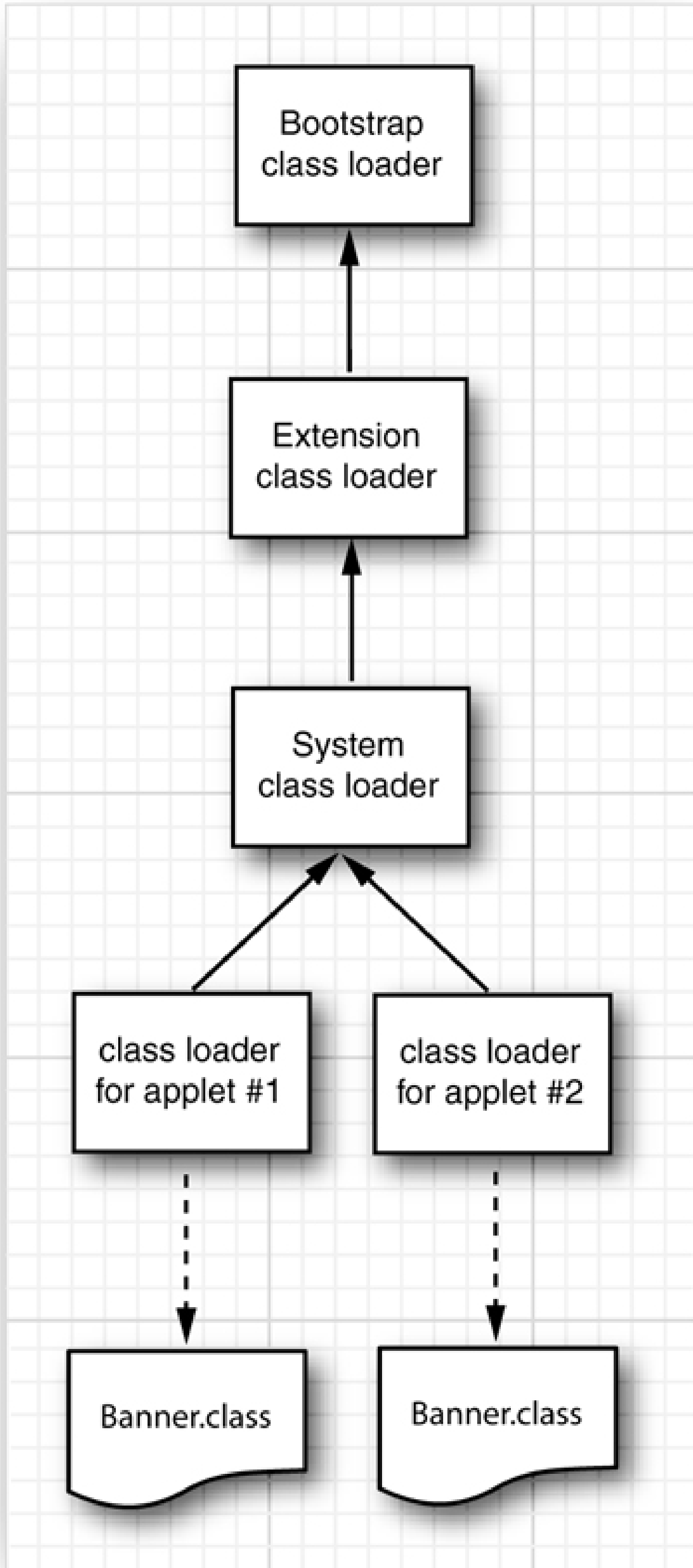
Using Class Loaders as Namespaces

Every Java programmer knows that package names are used to eliminate name conflicts. There are two classes called `Date` in the standard library, but of course their real names are `java.util.Date` and `java.sql.Date`. The simple name is only a programmer convenience and requires the inclusion of appropriate `import` statements. In a running program, all class names contain their package name.

It might surprise you, however, that you can have two classes in the same virtual machine that have the same class *and package* name. A class is determined by its full name *and* the class loader. This technique is useful for loading code from multiple sources. For example, a browser uses separate instances of the applet class loader class for each web page. This allows the virtual machine to separate classes from different web pages, no matter what they are named. Figure 9-2 shows an example. Suppose a web page contains two applets, provided by different advertisers, and each applet has a class called `Banner`. Because each applet is loaded by a separate class loader, these classes are entirely distinct and do not conflict with each other.

UNREGISTERED VERSION OF CHM TO PDF CONVERTER PRO BY THETA-SOFTWARE

Figure 9-2. Two class loaders load different classes with the same name


Bootstrap class loader
Extension class loader
System class loader
class loader for applet #1
class loader for applet #2
Banner.class
Banner.class

Note

This technique has other uses as well, such as "hot deployment" of servlets and Enterprise JavaBeans. See http://java.sun.com/developer/TechTips/2000/tt1027.html for more information.

Writing Your Own Class Loader

You can write your own class loader for specialized purposes. That lets you carry out custom checks before you pass the bytecodes to the virtual machine. For example, you can write a class loader that can refuse to load a class that has not been marked as "paid for."

To write your own class loader, you simply extend the `ClassLoader` class and override the method.

```
findClass(String className)
```

The `loadClass` method of the `ClassLoader` superclass takes care of the delegation to the parent and calls `findClass` only if the class hasn't already been loaded and if the parent class loader was unable to load the class.

Your implementation of this method must do the following:

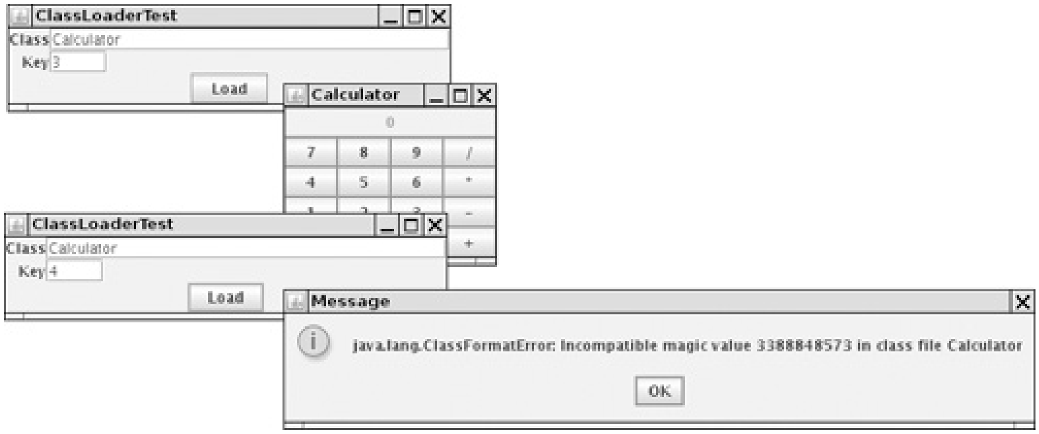
1. Load the bytecodes for the class from the local file system or from some other source.

2. Call the `defineClass` method of the `ClassLoader` superclass to present the bytecodes to the virtual machine.

In the program of Listing 9-1, we implement a class loader that loads encrypted class files. The program asks the user for the name of the first class to load (that is, the class containing `main`) and the decryption key. It then uses a special class loader to load the specified class and calls the `main` method. The class loader decrypts the specified class and all nonsystem classes that are referenced by it. Finally, the program calls the `main` method of the loaded class (see Figure 9-3).

Figure 9-3. The ClassLoaderTest program

[View full size image]



For simplicity, we ignore 2,000 years of progress in the field of cryptography and use the venerable Caesar cipher for encrypting the class files.

Note

David Kahn's wonderful book *The Codebreakers* (Macmillan, 1967, p. 84) refers to Suetonius as a historical source for the Caesar cipher. Caesar shifted the 24 letters of the Roman alphabet by 3 letters, which at the time baffled his adversaries.

When this chapter was first written, the U.S. government restricted the export of strong encryption methods. Therefore, we used Caesar's method for our example because it was clearly legal for export.

Our version of the Caesar cipher has as a key a number between 1 and 255. To decrypt, simply add that key to every byte and reduce modulo 256. The `Caesar.java` program of Listing 9-2 carries out the encryption.

So that we do not confuse the regular class loader, we use a different extension, `.caesar`, for the encrypted class files.

To decrypt, the class loader simply subtracts the key from every byte. In the companion code for this book, you will find four class files, encrypted with a key value of 3—the traditional choice. To run the encrypted program, you need the custom class loader defined in our `ClassLoaderTest` program.

Encrypting class files has a number of practical uses (provided, of course, that you use a cipher stronger than the Caesar cipher). Without the decryption key, the class files are useless. They can neither be executed by a standard virtual machine nor readily disassembled.

This means that you can use a custom class loader to authenticate the user of the class or to ensure that a program has been paid for before it will be allowed to run. Of course, encryption is only one application of a custom class loader. You can use other types of class loaders to solve other problems, for example, storing class files in a database.

Listing 9-1. **`ClassLoaderTest.java`**

Code View:

```
import java.io.*;
import java.lang.reflect.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

/**
 * This program demonstrates a custom class loader that decrypts class files.
```



```
 * @version 1.22 2007-10-05
 * @author Cay Horstmann
 */
public class ClassLoaderTest
{
   public static void main(String[] args)
```



```
   {
      EventQueue.invokeLater(new Runnable()
         {
            public void run()
            {

               JFrame frame = new ClassLoaderFrame();
               frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
               frame.setVisible(true);
            }
         });
   }
}

/**
 * This frame contains two text fields for the name of the class to load and the decryption key
 */
class ClassLoaderFrame extends JFrame
{
   public ClassLoaderFrame()
   {
      setTitle("ClassLoaderTest");
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
      setLayout(new GridBagLayout());
      add(new JLabel("Class"), new GBC(0, 0).setAnchor(GBC.EAST));
      add(nameField, new GBC(1, 0).setWeight(100, 0).setAnchor(GBC.WEST));
      add(new JLabel("Key"), new GBC(0, 1).setAnchor(GBC.EAST));
      add(keyField, new GBC(1, 1).setWeight(100, 0).setAnchor(GBC.WEST));
      JButton loadButton = new JButton("Load");
      add(loadButton, new GBC(0, 2, 2, 1));
      loadButton.addActionListener(new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               runClass(nameField.getText(), keyField.getText());
            }
         });
      pack();
   }

   /**
    * Runs the main method of a given class.
    * @param name the class name
    * @param key the decryption key for the class files
```

```
    */
   public void runClass(String name, String key)
   {
      try
      {
         ClassLoader loader = new CryptoClassLoader(Integer.parseInt(key));
         Class<?> c = loader.loadClass(name);
         Method m = c.getMethod("main", String[].class);
         m.invoke(null, (Object) new String[] {});
      }
      catch (Throwable e)
      {
         JOptionPane.showMessageDialog(this, e);
      }
   }

   private JTextField keyField = new JTextField("3", 4);
   private JTextField nameField = new JTextField("Calculator", 30);
   private static final int DEFAULT_WIDTH = 300;
   private static final int DEFAULT_HEIGHT = 200;
}

/**
 * This class loader loads encrypted class files.
 */
class CryptoClassLoader extends ClassLoader
{
   /**
    * Constructs a crypto class loader.
    * @param k the decryption key
    */
   public CryptoClassLoader(int k)
   {
      key = k;
   }

   protected Class<?> findClass(String name) throws ClassNotFoundException
   {
      byte[] classBytes = null;
      try
      {
         classBytes = loadClassBytes(name);
      }
      catch (IOException e)
      {
         throw new ClassNotFoundException(name);
      }

      Class<?> cl = defineClass(name, classBytes, 0, classBytes.length);
      if (cl == null) throw new ClassNotFoundException(name);
      return cl;
   }

   /**
    * Loads and decrypt the class file bytes.
    * @param name the class name
    * @return an array with the class file bytes
    */
   private byte[] loadClassBytes(String name) throws IOException
```

```
   {
      String cname = name.replace('.', '/') + ".caesar";
      FileInputStream in = null;
      in = new FileInputStream(cname);
      try
      {
         ByteArrayOutputStream buffer = new ByteArrayOutputStream();
         int ch;
         while ((ch = in.read()) != -1)
```

```
         {
            byte b = (byte) (ch - key);
            buffer.write(b);
         }
         in.close();
         return buffer.toByteArray();
      }
```

```
      finally
      {
         in.close();
      }
   }

   private int key;
}
```

Listing 9-2. **Caesar.java**

Code View:
```
import java.io.*;

/**
 * Encrypts a file using the Caesar cipher.
 * @version 1.00 1997-09-10
 * @author Cay Horstmann
 */
public class Caesar
{
   public static void main(String[] args)
   {
      if (args.length != 3)
      {
         System.out.println("USAGE: java Caesar in out key");
         return;
      }

      try
      {
         FileInputStream in = new FileInputStream(args[0]);
         FileOutputStream out = new FileOutputStream(args[1]);
         int key = Integer.parseInt(args[2]);
         int ch;
         while ((ch = in.read()) != -1)
```

```
         {
            byte c = (byte) (ch + key);
            out.write(c);
         }
         in.close();
         out.close();
      }
      catch (IOException exception)
      {
         exception.printStackTrace();
      }
   }
}
```

API

java.lang.Class 1.0

- `ClassLoader getClassLoader()`

  gets the class loader that loaded this class.

API

java.lang.ClassLoader 1.0

- `ClassLoader getParent()` **1.2**

  returns the parent class loader, or `null` if the parent class loader is the bootstrap class loader.

- `static ClassLoader getSystemClassLoader()` **1.2**

  gets the system class loader; that is, the class loader that was used to load the first application class.

- `protected Class findClass(String name)` **1.2**

  should be overridden by a class loader to find the bytecodes for a class and present them to the virtual machine by calling the `defineClass` method. In the name of the class, use `.` as package name separator, and don't use a `.class` suffix.

- `Class defineClass(String name, byte[] byteCodeData, int` `offset, int length)`

adds a new class to the virtual machine whose bytecodes are provided in the given data range.

API `java.net.URLClassLoader` 1.2

- `URLClassLoader(URL[] urls)`

- `URLClassLoader(URL[] urls, ClassLoader parent)`

  constructs a class loader that loads classes from the given URLs. If a URL ends in a `/`, it is assumed to be a directory, otherwise it is assumed to be a JAR file.

API `java.lang.Thread` 1.0

- `ClassLoader getContextClassLoader()` **1.2**

  gets the class loader that the creator of this thread has designated as the most reasonable class loader to use when executing this thread.

- `void setContextClassLoader(ClassLoader loader)` **1.2**

  sets a class loader for code in this thread to retrieve for loading classes. If no context class loader is set explicitly when a thread is started, the parent's context class loader is used.

Chapter 9. Security

- Class Loaders
- Bytecode Verification
- Security Managers and Permissions
- User Authentication
- Digital Signatures
- Code Signing
- Encryption

When Java technology first appeared on the scene, the excitement was not about a well-crafted programming language but about the possibility of safely executing applets that are delivered over the Internet (see Volume I, Chapter 10 for more information about applets). Obviously, delivering executable applets is practical only when the recipients are sure that the code can't wreak havoc on their machines. For this reason, security was and is a major concern of both the designers and the users of Java technology. This means that unlike other languages and systems, where security was implemented as an afterthought or a reaction to break-ins, security mechanisms are an integral part of Java technology.

Three mechanisms help ensure safety:

- Language design features (bounds checking on arrays, no unchecked type conversions, no pointer arithmetic, and so on).
- An access control mechanism that controls what the code can do (such as file access, network access, and so on).
- Code signing, whereby code authors can use standard cryptographic algorithms to authenticate Java code. Then, the users of the code can determine exactly who created the code and whether the code has been altered after it was signed.

We will first discuss *class loaders* that check class files for integrity when they are loaded into the virtual machine. We will demonstrate how that mechanism can detect tampering with class files.

For maximum security, both the default mechanism for loading a class and a custom class loader need to work with a *security manager* class that controls what actions code can perform. You'll see in detail how to configure Java platform security.

Finally, you'll see the cryptographic algorithms supplied in the `java.security` package, which allow for code signing and user authentication.

As always, we focus on those topics that are of greatest interest to application programmers. For an in-depth view, we recommend the book *Inside Java 2 Platform Security: Architecture, API Design, and Implementation*, 2nd ed., by Li Gong, Gary Ellison, and Mary Dageforde (Prentice Hall PTR 2003).

Class Loaders

A Java compiler converts source instructions for the Java virtual machine. The virtual machine code is stored in a class file with a `.class` extension. Each class file contains the definition and implementation code for one class or interface. These class files must be interpreted by a program that can translate the instruction set of the virtual machine into the machine language of the target machine.

Note that the virtual machine loads only those class files that are needed for the execution of a program. For example, suppose program execution starts with `MyProgram.class`. Here are the steps that the virtual machine carries out.

1. The virtual machine has a mechanism for loading class files, for example, by reading the files from disk or by requesting them from the Web; it uses this mechanism to load the contents of the `MyProgram` class file.

2. If the `MyProgram` class has fields or superclasses of another class type, their class files are loaded as well. (The process of loading all the classes that a given class depends on is called *resolving* the class.)

3. The virtual machine then executes the `main` method in `MyProgram` (which is static, so no instance of a class needs to be created).

4. If the `main` method or a method that `main` calls requires additional classes, these are loaded next.

The class loading mechanism doesn't just use a single class loader, however. Every Java program has at least three class loaders:

- The bootstrap class loader

- The extension class loader

- The system class loader (also sometimes called the application class loader)

The bootstrap class loader loads the system classes (typically, from the JAR file `rt.jar`). It is an integral part of the virtual machine and is usually implemented in C. There is no `ClassLoader` object corresponding to the bootstrap class loader. For example,

```
String.class.getClassLoader()
```

returns `null`.

The extension class loader loads "standard extensions" from the *jre/*`lib/ext` directory. You can drop JAR files into that directory, and the extension class loader will find the classes in them, even without any class path. (Some people recommend this mechanism to avoid the "class path from hell," but see the next cautionary note.)

The system class loader loads the application classes. It locates classes in the directories and JAR/ZIP files on the class path, as set by the `CLASSPATH` environment variable or the `-classpath` command-line option.

In Sun's Java implementation, the extension and system class loaders are implemented in Java. Both are instances of the `URLClassLoader` class.

Caution

You can run into grief if you drop a JAR file into the *jre/*`lib/ext` directory and one of its classes needs to load a class that is not a system or extension class. The extension class loader *does not use the class path*. Keep that in mind before you use the extension directory as a way to manage your class file hassles.

Note

In addition to all the places already mentioned, classes can be loaded from the *jre/*`lib/endorsed` directory. This mechanism can only be used to replace certain standard Java libraries (such as those for XML and CORBA support) with newer versions. See http://java.sun.com/javase/6/docs/technotes/guides/standards/index.html for details.

The Class Loader Hierarchy

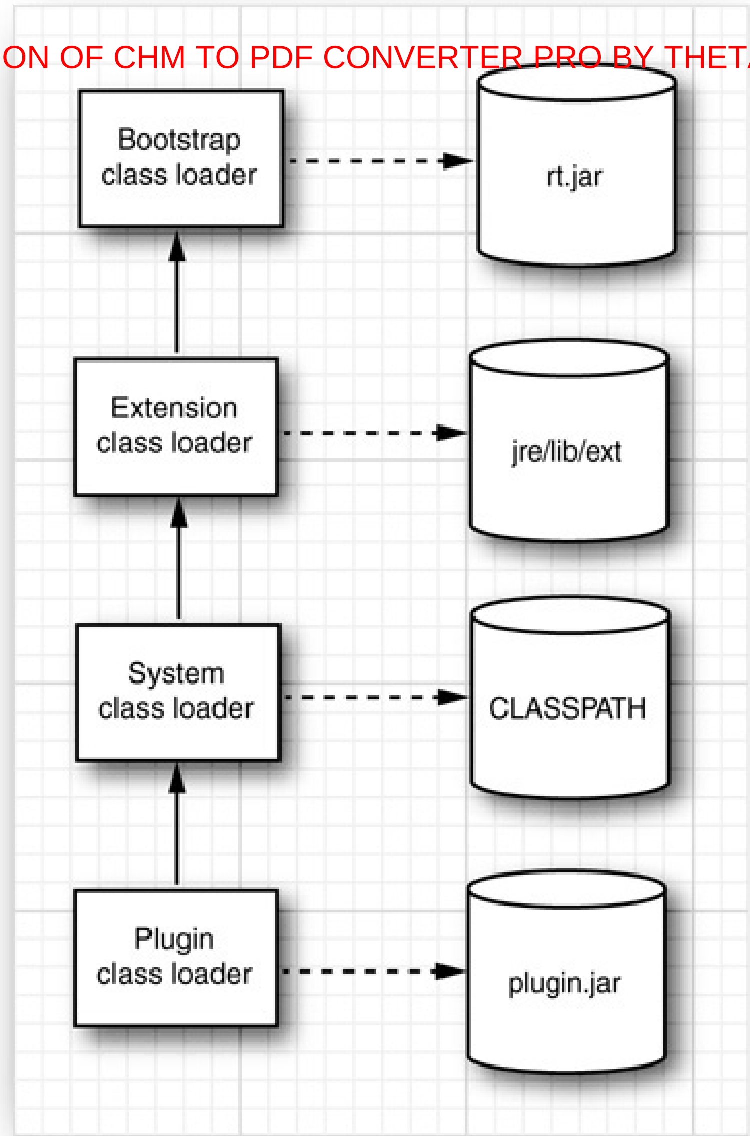
Class loaders have a *parent/child* relationship. Every class loader except for the bootstrap class loader has a parent class loader. A class loader is supposed to give its parent a chance to load any given class and only load it if the parent has failed. For example, when the system class loader is asked to load a system class (say, `java.util.ArrayList`), then it first asks the extension class loader. That class loader first asks the bootstrap class loader. The bootstrap class loader finds and loads the class in `rt.jar`, and neither of the other class loaders searches any further.

Some programs have a plugin architecture in which certain parts of the code are packaged as optional plugins. If the plugins are packaged as JAR files, you can simply load the plugin classes with an instance of `URLClassLoader`.

```
URL url = new URL("file:///path/to/plugin.jar");
URLClassLoader pluginLoader = new URLClassLoader(new URL[] { url });
Class<?> cl = pluginLoader.loadClass("mypackage.MyClass");
```

Because no parent was specified in the `URLClassLoader` constructor, the parent of the `pluginLoader` is the system class loader. Figure 9-1 shows the hierarchy.

Figure 9-1. The class loader hierarchy



Most of the time, you don't have to worry about the class loader hierarchy. Generally, classes are loaded because they are required by other classes, and that process is transparent to you.

Occasionally, you need to intervene and specify a class loader. Consider this example.

- Your application code contains a helper method that calls `Class.forName(classNameString)`.
- That method is called from a plugin class.
- The `classNameString` specifies a class that is contained in the plugin JAR.

The author of the plugin has the reasonable expectation that the class should be loaded. However, the helper method's class was loaded by the system class loader, and that is the class loader used by `Class.forName`. The classes in the plugin JAR are not visible. This phenomenon is called *classloader inversion*.

To overcome this problem, the helper method needs to use the correct class loader. It can require the class loader as a parameter. Alternatively, it can require that the correct class loader is set as the *context class loader* of the current thread. This strategy is used by many frameworks (such as the JAXP and JNDI frameworks that we discussed in Chapters 2 and 4).

Each thread has a reference to a class loader, called the context class loader. The main thread's context class loader is the system class loader. When a new thread is created, its context class loader is set to the creating thread's context class loader. Thus, if you don't do anything, then all threads have their context class loader set to the system class loader.

However, you can set any class loader by calling

```
Thread t = Thread.currentThread();
t.setContextClassLoader(loader);
```

The helper method can then retrieve the context class loader:

```
Thread t = Thread.currentThread();
ClassLoader loader = t.getContextClassLoader();
Class cl = loader.loadClass(className);
```

The question remains when the context class loader is set to the plugin class loader. The application designer must make this decision. Generally, it is a good idea to set the context class loader when invoking a method of a plugin class that was loaded with a different class loader. Alternatively, the caller of the helper method can set the context class loader.

Tip

If you write a method that loads a class by name, it is a good idea to offer the caller the choice between passing an explicit class loader and using the context class loader. Don't simply use the class loader of the method's class.

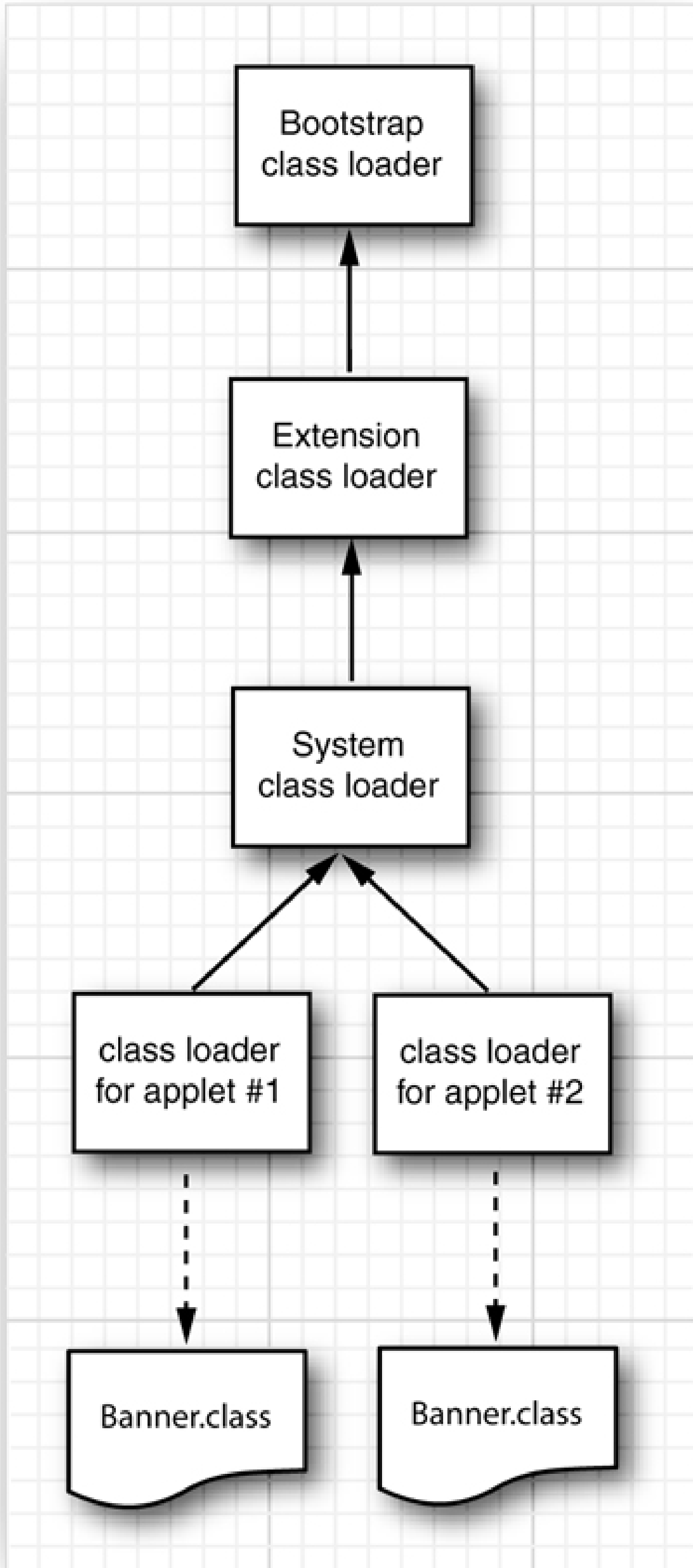
Using Class Loaders as Namespaces

Every Java programmer knows that package names are used to eliminate name conflicts. There are two classes called `Date` in the standard library, but of course their real names are `java.util.Date` and `java.sql.Date`. The simple name is only a programmer convenience and requires the inclusion of appropriate `import` statements. In a running program, all class names contain their package name.

It might surprise you, however, that you can have two classes in the same virtual machine that have the same class *and package* name. A class is determined by its full name *and* the class loader. This technique is useful for loading code from multiple sources. For example, a browser uses separate instances of the applet class loader class for each web page. This allows the virtual machine to separate classes from different web pages, no matter what they are named. Figure 9-2 shows an example. Suppose a web page contains two applets, provided by different advertisers, and each applet has a class called `Banner`. Because each applet is loaded by a separate class loader, these classes are entirely distinct and do not conflict with each other.

UNREGISTERED VERSION OF CHM TO PDF CONVERTER PRO BY THETA-SOFTWARE

Figure 9-2. Two class loaders load different classes with the same name



Bootstrap class loader
Extension class loader
System class loader
class loader for applet #1
class loader for applet #2
Banner.class
Banner.class

Note

This technique has other uses as well, such as "hot deployment" of servlets and Enterprise JavaBeans. See http://java.sun.com/developer/TechTips/2000/tt1027.html for more information.

Writing Your Own Class Loader

You can write your own class loader for specialized purposes. That lets you carry out custom checks before you pass the bytecodes to the virtual machine. For example, you can write a class loader that can refuse to load a class that has not been marked as "paid for."

To write your own class loader, you simply extend the `ClassLoader` class and override the method.

```
findClass(String className)
```

The `loadClass` method of the `ClassLoader` superclass takes care of the delegation to the parent and calls `findClass` only if the class hasn't already been loaded and if the parent class loader was unable to load the class.

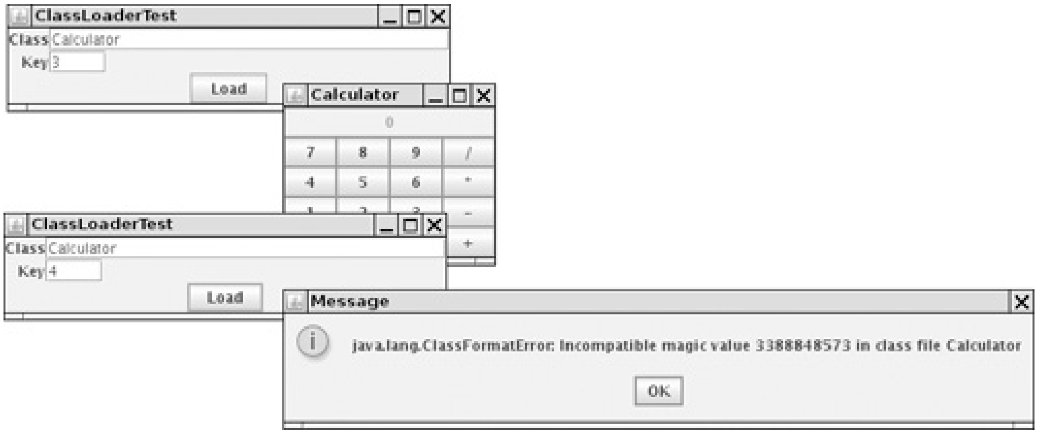
Your implementation of this method must do the following:

1. Load the bytecodes for the class from the local file system or from some other source.
2. Call the `defineClass` method of the `ClassLoader` superclass to present the bytecodes to the virtual machine.

In the program of Listing 9-1, we implement a class loader that loads encrypted class files. The program asks the user for the name of the first class to load (that is, the class containing `main`) and the decryption key. It then uses a special class loader to load the specified class and calls the `main` method. The class loader decrypts the specified class and all nonsystem classes that are referenced by it. Finally, the program calls the `main` method of the loaded class (see Figure 9-3).

Figure 9-3. The ClassLoaderTest program

[View full size image]



For simplicity, we ignore 2,000 years of progress in the field of cryptography and use the venerable Caesar cipher for encrypting the class files.

Note

David Kahn's wonderful book *The Codebreakers* (Macmillan, 1967, p. 84) refers to Suetonius as a historical source for the Caesar cipher. Caesar shifted the 24 letters of the Roman alphabet by 3 letters, which at the time baffled his adversaries.

When this chapter was first written, the U.S. government restricted the export of strong encryption methods. Therefore, we used Caesar's method for our example because it was clearly legal for export.

Our version of the Caesar cipher has as a key a number between 1 and 255. To decrypt, simply add that key to every byte and reduce modulo 256. The `Caesar.java` program of Listing 9-2 carries out the encryption.

So that we do not confuse the regular class loader, we use a different extension, `.caesar`, for the encrypted class files.

To decrypt, the class loader simply subtracts the key from every byte. In the companion code for this book, you will find four class files, encrypted with a key value of 3—the traditional choice. To run the encrypted program, you need the custom class loader defined in our `ClassLoaderTest` program.

Encrypting class files has a number of practical uses (provided, of course, that you use a cipher stronger than the Caesar cipher). Without the decryption key, the class files are useless. They can neither be executed by a standard virtual machine nor readily disassembled.

This means that you can use a custom class loader to authenticate the user of the class or to ensure that a program has been paid for before it will be allowed to run. Of course, encryption is only one application of a custom class loader. You can use other types of class loaders to solve other problems, for example, storing class files in a database.

Listing 9-1. **`ClassLoaderTest.java`**

Code View:

UNREGISTERED VERSION OF CHM TO PDF CONVERTER PRO BY THETA-SOFTWARE

```
import java.io.*;
import java.lang.reflect.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

/**
 * This program demonstrates a custom class loader that decrypts class files.
 * @version 1.22 2007-10-05
 * @author Cay Horstmann
 */
public class ClassLoaderTest
{
   public static void main(String[] args)
   {
      EventQueue.invokeLater(new Runnable()
         {
            public void run()
            {

               JFrame frame = new ClassLoaderFrame();
               frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
               frame.setVisible(true);
            }
         });
   }
}

/**
 * This frame contains two text fields for the name of the class to load and the decryption key
 */
class ClassLoaderFrame extends JFrame
{
   public ClassLoaderFrame()
   {
      setTitle("ClassLoaderTest");
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
      setLayout(new GridBagLayout());
      add(new JLabel("Class"), new GBC(0, 0).setAnchor(GBC.EAST));
      add(nameField, new GBC(1, 0).setWeight(100, 0).setAnchor(GBC.WEST));
      add(new JLabel("Key"), new GBC(0, 1).setAnchor(GBC.EAST));
      add(keyField, new GBC(1, 1).setWeight(100, 0).setAnchor(GBC.WEST));
      JButton loadButton = new JButton("Load");
      add(loadButton, new GBC(0, 2, 2, 1));
      loadButton.addActionListener(new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               runClass(nameField.getText(), keyField.getText());
            }
         });
      pack();
   }

   /**
    * Runs the main method of a given class.
    * @param name the class name
    * @param key the decryption key for the class files
```

```
    */
   public void runClass(String name, String key)
   {
      try
      {
         ClassLoader loader = new CryptoClassLoader(Integer.parseInt(key));
         Class<?> c = loader.loadClass(name);
         Method m = c.getMethod("main", String[].class);
         m.invoke(null, (Object) new String[] {});
      }
      catch (Throwable e)
      {
         JOptionPane.showMessageDialog(this, e);
      }
   }

   private JTextField keyField = new JTextField("3", 4);
   private JTextField nameField = new JTextField("Calculator", 30);
   private static final int DEFAULT_WIDTH = 300;
   private static final int DEFAULT_HEIGHT = 200;
}

/**
 * This class loader loads encrypted class files.
 */
class CryptoClassLoader extends ClassLoader
{
   /**
    * Constructs a crypto class loader.
    * @param k the decryption key
    */
   public CryptoClassLoader(int k)
   {
      key = k;
   }

   protected Class<?> findClass(String name) throws ClassNotFoundException
   {
      byte[] classBytes = null;
      try
      {
         classBytes = loadClassBytes(name);
      }
      catch (IOException e)
      {
         throw new ClassNotFoundException(name);
      }

      Class<?> cl = defineClass(name, classBytes, 0, classBytes.length);
      if (cl == null) throw new ClassNotFoundException(name);
      return cl;
   }

   /**
    * Loads and decrypt the class file bytes.
    * @param name the class name
    * @return an array with the class file bytes
    */
   private byte[] loadClassBytes(String name) throws IOException
```

```
   {
      String cname = name.replace('.', '/') + ".caesar";
      FileInputStream in = null;
      in = new FileInputStream(cname);
      try
      {
         ByteArrayOutputStream buffer = new ByteArrayOutputStream();
         int ch;
         while ((ch = in.read()) != -1)
         {
            byte b = (byte) (ch - key);
            buffer.write(b);
         }
         in.close();
         return buffer.toByteArray();
      }
      finally
      {
         in.close();
      }
   }

   private int key;
}
```


Listing 9-2. **Caesar.java**

Code View:

```
import java.io.*;

/**
 * Encrypts a file using the Caesar cipher.
 * @version 1.00 1997-09-10
 * @author Cay Horstmann
 */
public class Caesar
{
   public static void main(String[] args)
   {
      if (args.length != 3)
      {
         System.out.println("USAGE: java Caesar in out key");
         return;
      }

      try
      {
         FileInputStream in = new FileInputStream(args[0]);
         FileOutputStream out = new FileOutputStream(args[1]);
         int key = Integer.parseInt(args[2]);
         int ch;
         while ((ch = in.read()) != -1)
```

```
         {
            byte c = (byte) (ch + key);
            out.write(c);
         }
         in.close();
         out.close();
      }
      catch (IOException exception)
      {
         exception.printStackTrace();
      }
   }
}
```

API `java.lang.Class` 1.0

- `ClassLoader getClassLoader()`

  gets the class loader that loaded this class.

API `java.lang.ClassLoader` 1.0

- `ClassLoader getParent()` **1.2**

  returns the parent class loader, or `null` if the parent class loader is the bootstrap class loader.

- `static ClassLoader getSystemClassLoader()` **1.2**

  gets the system class loader; that is, the class loader that was used to load the first application class.

- `protected Class findClass(String name)` **1.2**

  should be overridden by a class loader to find the bytecodes for a class and present them to the virtual machine by calling the `defineClass` method. In the name of the class, use `.` as package name separator, and don't use a `.class` suffix.

- `Class defineClass(String name, byte[] byteCodeData, int` offset, int length)

adds a new class to the virtual machine whose bytecodes are provided in the given data range.

API java.net.URLClassLoader 1.2

- `URLClassLoader(URL[] urls)`

- `URLClassLoader(URL[] urls, ClassLoader parent)`

  constructs a class loader that loads classes from the given URLs. If a URL ends in a `/`, it is assumed to be a directory, otherwise it is assumed to be a JAR file.

API java.lang.Thread 1.0

- `ClassLoader getContextClassLoader()` 1.2

  gets the class loader that the creator of this thread has designated as the most reasonable class loader to use when executing this thread.

- `void setContextClassLoader(ClassLoader loader)` 1.2

  sets a class loader for code in this thread to retrieve for loading classes. If no context class loader is set explicitly when a thread is started, the parent's context class loader is used.

Bytecode Verification

When a class loader presents the bytecodes of a newly loaded Java platform class to the virtual machine, these bytecodes are first inspected by a *verifier.* The verifier checks that the instructions cannot perform actions that are obviously damaging. All classes except for system classes are verified. You can, however, deactivate verification with the undocumented `-noverify` option.

For example,

```
java -noverify Hello
```

Here are some of the checks that the verifier carries out:

- Variables are initialized before they are used.
- Method calls match the types of object references.
- Rules for accessing private data and methods are not violated.
- Local variable accesses fall within the runtime stack.
- The runtime stack does not overflow.

If any of these checks fails, then the class is considered corrupted and will not be loaded.

Note

If you are familiar with Gödel's theorem, you might wonder how the verifier can prove that a class file is free from type mismatches, uninitialized variables, and stack overflows. Gödel's theorem states that it is impossible to design algorithms that process program files and decide whether the input programs have a particular property (such as being free from stack overflows). Is this a conflict between the public relations department at Sun Microsystems and the laws of logic? No—in fact, the verifier is *not* a decision algorithm in the sense of Gödel. If the verifier accepts a program, it is indeed safe. However, the verifier might reject virtual machine instructions even though they would actually be safe. (You might have run into this issue when you were forced to initialize a variable with a dummy value because the compiler couldn't tell that it was going to be properly initialized.)

This strict verification is an important security consideration. Accidental errors, such as uninitialized variables, can easily wreak havoc if they are not caught. More important, in the wide open world of the Internet, you must be protected against malicious programmers who create evil effects on purpose. For example, by modifying values on the runtime stack or by writing to the private data fields of system objects, a program can break through the security system of a browser.

You might wonder, however, why a special verifier checks all these features. After all, the compiler would never allow you to generate a class file in which an uninitialized variable is used or in which a private data field is accessed from another class. Indeed, a class file generated by a compiler for the Java programming language always passes verification. However, the bytecode format used in the class files is well documented, and it is an easy matter for someone with some experience in assembly programming and a hex editor to manually produce a class file that contains valid but unsafe instructions for the Java virtual machine. Once again, keep in mind that the verifier is always guarding against maliciously altered class files, not just checking the class files produced by a compiler.

Here's an example of how to construct such an altered class file. We start with the program VerifierTest.java of Listing 9-3. This is a simple program that calls a method and displays the method result. The program can be run both as a console program and as an applet. The fun method itself just computes 1 + 2.


```
static int fun()
{
   int m;
   int n;
   m = 1;
   n = 2;
   int r = m + n;
   return r;
}
```

As an experiment, try to compile the following modification of this program:

```
static int fun()
{
   int m = 1;
   int n;
   m = 1;
   m = 2;
   int r = m + n;
   return r;
}
```

In this case, n is not initialized, and it could have any random value. Of course, the compiler detects that problem and refuses to compile the program. To create a bad class file, we have to work a little harder. First, run the javap program to find out how the compiler translates the fun method. The command

```
javap -c VerifierTest
```

shows the bytecodes in the class file in mnemonic form.

```
Method int fun()
   0 iconst_1
   1 istore_0
   2 iconst_2
   3 istore_1
   4 iload_0
   5 iload_1
```

```
6 iadd
7 istore_2
8 iload_2
9 ireturn
```

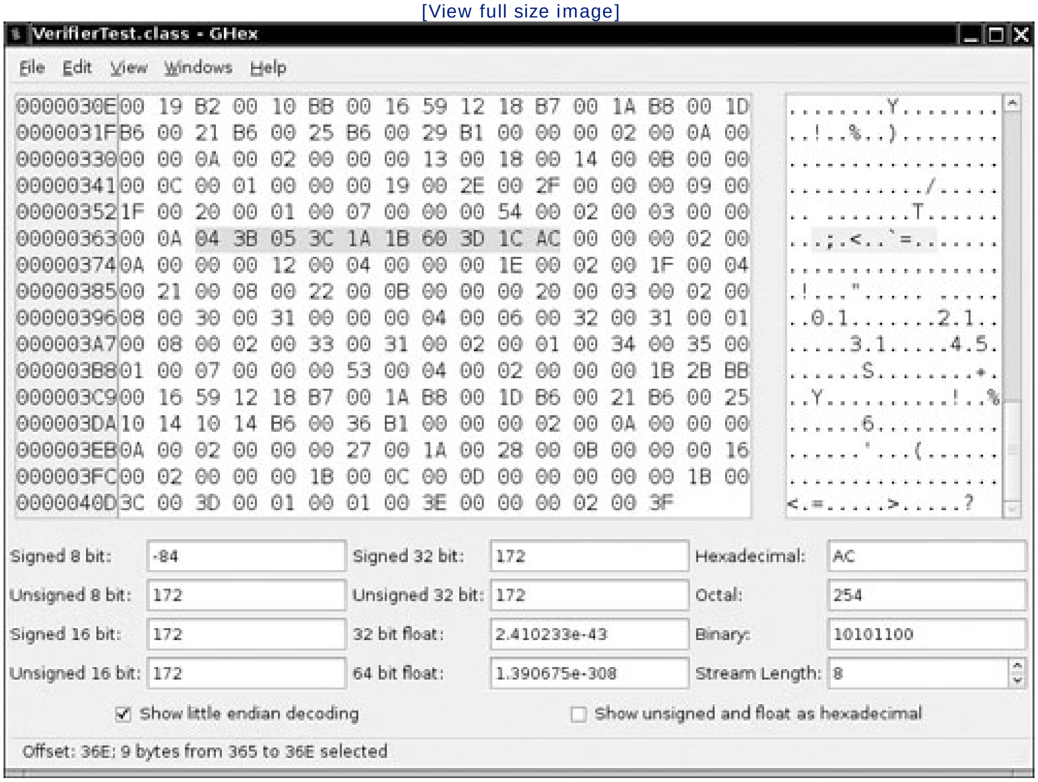
We use a hex editor to change instruction 3 from `istore_1` to `istore_0`. That is, local variable 0 (which is `m`) is initialized twice, and local variable 1 (which is `n`) is not initialized at all. We need to know the hexadecimal values for these instructions. These values are readily available from *The Java Virtual Machine Specification*, 2nd ed., by Tim Lindholm and Frank Yellin (Prentice Hall PTR 1999).

```
0 iconst_1 04
1 istore_0 3B
2 iconst_2 05
3 istore_1 3C
4 iload_0  1A
5 iload_1  1B
6 iadd     60
7 istore_2 3D
8 iload_2  1C
9 ireturn  AC
```

You can use any hex editor to carry out the modification. In Figure 9-4, you see the class file `VerifierTest.class` loaded into the Gnome hex editor, with the bytecodes of the `fun` method highlighted.

Figure 9-4. Modifying bytecodes with a hex editor

[View full size image]



Change 3C to 3B and save the class file. Then try running the VerifierTest program. You get an error message:

Code View:
```
Exception in thread "main" java.lang.VerifyError: (class: VerifierTest, method:fun signature:
()I) Accessing value from uninitialized register 1
```


That is good—the virtual machine detected our modification.

Now run the program with the -noverify (or -Xverify:none) option.


```
java -noverify VerifierTest
```

The fun method returns a seemingly random value. This is actually 2 plus the value that happened to be stored in the variable n, which never was initialized. Here is a typical printout:

```
1 + 2 == 15102330
```

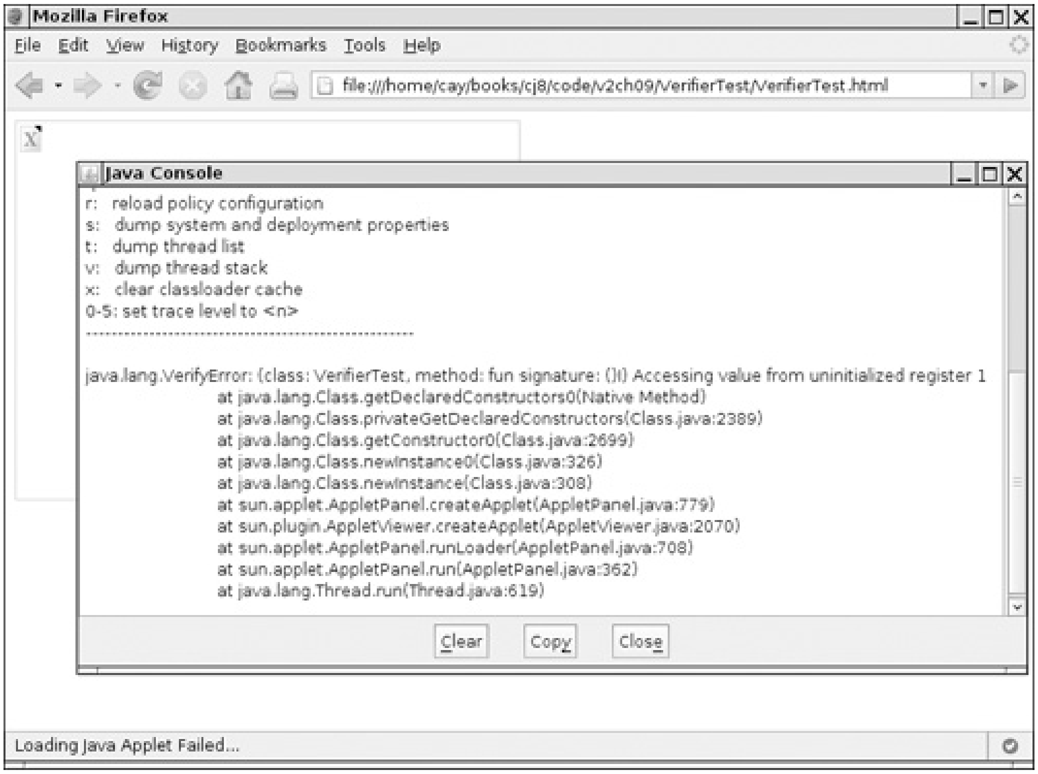
To see how browsers handle verification, we wrote this program to run either as an application or an applet. Load the applet into a browser, using a file URL such as

```
file:///C:/CoreJavaBook/v2ch9/VerifierTest/VerifierTest.html
```

You then see an error message displayed indicating that verification has failed (see Figure 9-5).

Figure 9-5. Loading a corrupted class file raises a method verification error

[View full size image]



Listing 9-3. **VerifierTest.java**

Code View:
```
import java.applet.*;
import java.awt.*;

/**
 * This application demonstrates the bytecode verifier of the virtual machine. If you use a
 * hex editor to modify the class file, then the virtual machine should detect the tampering.
 * @version 1.00 1997-09-10
 * @author Cay Horstmann
 */
public class VerifierTest extends Applet
{
   public static void main(String[] args)
   {
      System.out.println("1 + 2 == " + fun());
   }

   /**
    * A function that computes 1 + 2
    * @return 3, if the code has not been corrupted
    */
   public static int fun()
   {
      int m;
      int n;
      m = 1;
      n = 2;
      // use hex editor to change to "m = 2" in class file
      int r = m + n;
```

```
        return r;
    }

    public void paint(Graphics g)
    {
        g.drawString("1 + 2 == " + fun(), 20, 20);
    }
}
```


UNREGISTERED VERSION OF CHM TO PDF CONVERTER PRO BY THETA-SOFTWARE

Security Managers and Permissions

Once a class has been loaded into the virtual machine and checked by the verifier, the second security mechanism Java platform springs into action: the *security manager.* The security manager is a class that controls whether a sp operation is permitted. Operations checked by the security manager include the following:

- Creating a new class loader
- Exiting the virtual machine
- Accessing a field of another class by using reflection
- Accessing a file
- Opening a socket connection
- Starting a print job
- Accessing the system clipboard
- Accessing the AWT event queue
- Bringing up a top-level window

There are many other checks such as these throughout the Java library.

The default behavior when running Java applications is that *no* security manager is installed, so all these operation permitted. The applet viewer, on the other hand, enforces a security policy that is quite restrictive.

For example, applets are not allowed to exit the virtual machine. If they try calling the `exit` method, then a securi exception is thrown. Here is what happens in detail. The `exit` method of the `Runtime` class calls the `checkExit` me the security manager. Here is the entire code of the `exit` method:

```
public void exit(int status)
{
   SecurityManager security = System.getSecurityManager();
   if (security != null)
       security.checkExit(status);
   exitInternal(status);
}
```

The security manager now checks if the exit request came from the browser or an individual applet. If the security manager agrees with the exit request, then the `checkExit` method simply returns and normal processing continue

However, if the security manager doesn't want to grant the request, the `checkExit` method throws a `SecurityExc`
.

The `exit` method continues only if no exception occurred. It then calls the *private native* `exitInternal` method th
actually terminates the virtual machine. There is no other way of terminating the virtual machine, and because the
`exitInternal` method is private, it cannot be called from any other class. Thus, any code that attempts to exit the
machine must go through the `exit` method and thus through the `checkExit` security check without triggering a se
exception.

Clearly, the integrity of the security policy depends on careful coding. The providers of system services in the stan
library must always consult the security manager before attempting any sensitive operation.

The security manager of the Java platform allows both programmers and system administrators fine-grained contr
individual security permissions. We describe these features in the following section. First, we summarize the Java 2
platform security model. We then show how you can control permissions with *policy files*. Finally, we explain how y
define your own permission types.

Note

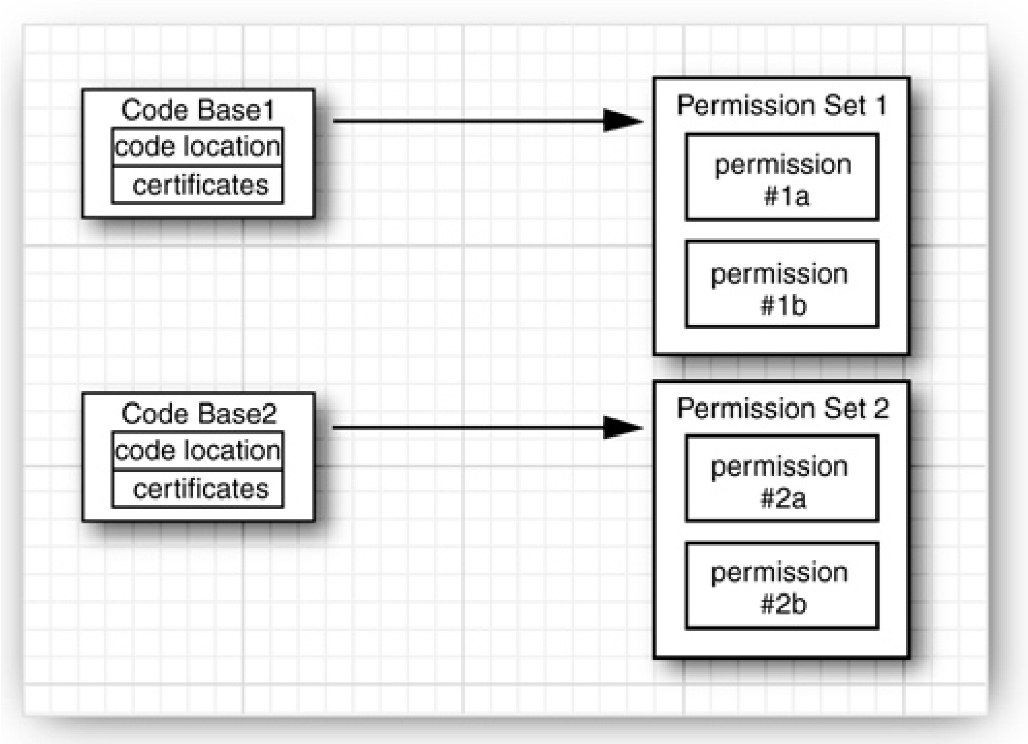
It is possible to implement and install your own security manager, but you should not attempt this unless you are an expert in computer security. It is much safer to configure the standard security manager.

Java Platform Security

JDK 1.0 had a very simple security model: Local classes had full permissions, and remote classes were confined to
sandbox. Just like a child that can only play in a sandbox, remote code was only allowed to paint on the screen an
interact with the user. The applet security manager denied all access to local resources. JDK 1.1 implemented a sli
modification: Remote code that was signed by a trusted entity was granted the same permissions as local classes.
However, both versions of the JDK provided an all-or-nothing approach. Programs either had full access or they ha
play in the sandbox.

Starting with Java SE 1.2, the Java platform has a much more flexible mechanism. A *security policy* maps *code so*
permission sets (see Figure 9-6).

Figure 9-6. A security policy



A *code source* is specified by a *code base* and a set of *certificates* . The code base specifies the origin of the code. example, the code base of remote applet code is the HTTP URL from which the applet is loaded. The code base of a JAR file is a file URL. A certificate, if present, is an assurance by some party that the code has not been tampere We cover certificates later in this chapter.

A *permission* is any property that is checked by a security manager. The Java platform supports a number of perm classes, each of which encapsulates the details of a particular permission. For example, the following instance of tl `FilePermission` class states that it is okay to read and write any file in the `/tmp` directory.

```
FilePermission p = new FilePermission("/tmp/*", "read,write");
```

More important, the default implementation of the `Policy` class reads permissions from a *permission file.* Inside a permission file, the same read permission is expressed as

```
permission java.io.FilePermission "/tmp/*", "read,write";
```

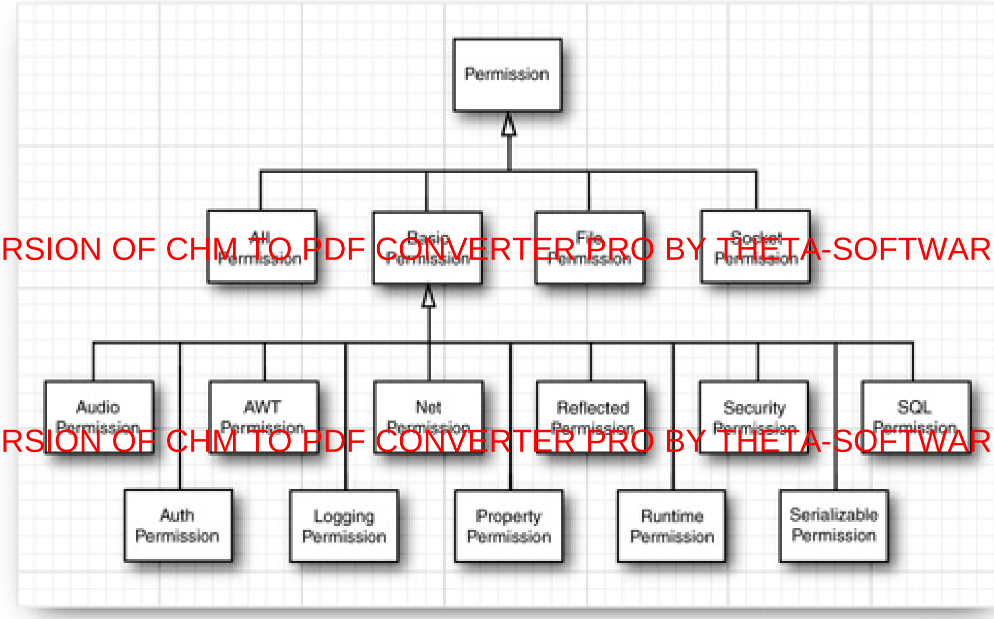
We discuss permission files in the next section.

Figure 9-7 shows the hierarchy of the permission classes that were supplied with Java SE 1.2. Many more permiss classes have been added in subsequent Java releases.

Figure 9-7. A part of the hierarchy of permission classes

[View full size image]

UNREGISTERED VERSION OF CHM TO PDF CONVERTER PRO BY THETA-SOFTWARE



In the preceding section, you saw that the `SecurityManager` class has security check methods such as `checkExit` methods exist only for the convenience of the programmer and for backward compatibility. They all map into stan permission checks. For example, here is the source code for the `checkExit` method:

```
public void checkExit()
{
   checkPermission(new RuntimePermission("exitVM"));
}
```

Each class has a *protection domain,* an object that encapsulates both the code source and the collection of permis the class. When the `SecurityManager` needs to check a permission, it looks at the classes of all methods currently call stack. It then gets the protection domains of all classes and asks each protection domain if its permission colle allows the operation that is currently being checked. If all domains agree, then the check passes. Otherwise, a `SecurityException` is thrown.

Why do all methods on the call stack need to allow a particular operation? Let us work through an example. Suppo `init` method of an applet wants to open a file. It might call

```
Reader in = new FileReader(name);
```

The `FileReader` constructor calls the `FileInputStream` constructor, which calls the `checkRead` method of the secu manager, which finally calls `checkPermission` with a `FilePermission(name, "read")` object. Table 9-1 shows the stack.

Table 9-1. Call Stack During Permission Checking

Class	Method	Code Source	Permissions
`SecurityManager`	`checkPermission`	`null`	`AllPermission`
`SecurityManager`	`checkRead`	`null`	`AllPermission`

Class	Method	Code Source	Permissions
`FileInputStream`	constructor	`null`	`AllPermission`
`FileReader`	constructor	`null`	`AllPermission`
applet	`init`	applet code source	applet permissions
. . .			

The `FileInputStream` and `SecurityManager` classes are *system classes* for which `CodeSource` is `null` and permis consist of an instance of the `AllPermission` class, which allows all operations. Clearly, their permissions alone car determine the outcome of the check. As you can see, the `checkPermission` method must take into account the re permissions of the applet class. By checking the entire call stack, the security mechanism ensures that one class c ask another class to carry out a sensitive operation on its behalf.

Note

This brief discussion of permission checking explains the basic concepts. However, we omit a number of technical details here. With security, the devil lies in the details, and we encourage you to read the book by Li Gong for more information. For a more critical view of the Java platform security model, see the book *Securing Java: Getting Down to Business with Mobile Code* , 2nd ed., by Gary McGraw and Ed W. Felten (Wiley 1999). You can find an online version of that book at http://www.securingjava.com .

API

`java.lang.SecurityManager` 1.0

- `void checkPermission(Permission p)` **1.2**

  checks whether this security manager grants the given permission. The method throws a `SecurityException` if the permission is not granted.

API

`java.lang.Class` 1.0

- `ProtectionDomain getProtectionDomain()` **1.2**

  gets the protection domain for this class, or `null` if this class was loaded without a protection domain.

java.security.ProtectionDomain 1.2

- ProtectionDomain(CodeSource source, PermissionCollection permissions)

  constructs a protection domain with the given code source and permissions.

- CodeSource getCodeSource()

  gets the code source of this protection domain.

- boolean implies(Permission p)

  returns true if the given permission is allowed by this protection domain.

java.security.CodeSource 1.2

- Certificate[] getCertificates()

  gets the certificate chain for class file signatures associated with this code source.

- URL getLocation()

  gets the code base of class files associated with this code source.

Security Policy Files

The *policy manager* reads *policy files* that contain instructions for mapping code sources to permissions. Here is a policy file:

```
grant codeBase "http://www.horstmann.com/classes"
{
   permission java.io.FilePermission "/tmp/*", "read,write";
};
```

This file grants permission to read and write files in the /tmp directory to all code that was downloaded from http://www.horstmann.com/classes .

You can install policy files in standard locations. By default, there are two locations:

- The file java.policy in the Java platform home directory

- The file `.java.policy` (notice the period at the beginning of the file name) in the user home directory

Note

You can change the locations of these files in the `java.security` configuration file in the *jre* `/lib/security` . The defaults are specified as

```
policy.url.1=file:${java.home}/lib/security/java.policy
policy.url.2=file:${user.home}/.java.policy
```

A system administrator can modify the `java.security` file and specify policy URLs that reside on another server and that cannot be edited by users. There can be any number of policy URLs (with consecutive numbers) in the policy file. The permissions of all files are combined.

If you want to store policies outside the file system, you can implement a subclass of the `Policy` class that gathers the permissions. Then change the line

```
policy.provider=sun.security.provider.PolicyFile
```

in the `java.security` configuration file.

During testing, we don't like to constantly modify the standard policy files. Therefore, we prefer to explicitly name policy file that is required for each application. Place the permissions into a separate file, say, `MyApp.policy` . To a policy, you have two choices. You can set a system property inside your applications' main method:

```
System.setProperty("java.security.policy", "MyApp.policy");
```

Alternatively, you can start the virtual machine as

```
java -Djava.security.policy=MyApp.policy MyApp
```

For applets, you instead use

```
appletviewer -J-Djava.security.policy=MyApplet.policy MyApplet.html
```

(You can use the `-J` option of the `appletviewer` to pass any command-line argument to the virtual machine.)

In these examples, the `MyApp.policy` file is added to the other policies in effect. If you add a second equal sign, s

```
java -Djava.security.policy==MyApp.policy MyApp
```

then your application uses *only* the specified policy file, and the standard policy files are ignored.

Caution

An easy mistake during testing is to accidentally leave a `.java.policy` file that grants a lot of permissions, perhaps even `AllPermission`, in the current directory. If you find that your application doesn't seem to pay attention to the restrictions in your policy file, check for a left-behind `.java.policy` file in your current directory. If you use a UNIX system, this is a particularly easy mistake to make because files with names that start with a period are not displayed by default.

As you saw previously, Java applications by default do not install a security manager. Therefore, you won't see the of policy files until you install one. You can, of course, add a line

```
System.setSecurityManager(new SecurityManager());
```

into your `main` method. Or you can add the command-line option `-Djava.security.manager` when starting the vir machine.

```
java -Djava.security.manager -Djava.security.policy=MyApp.policy MyApp
```

In the remainder of this section, we show you in detail how to describe permissions in the policy file. We describe entire policy file format, except for code certificates, which we cover later in this chapter.

A policy file contains a sequence of `grant` entries. Each entry has the following form:

```
grant codesource
{
   permission1;
   permission2;
   . . .
};
```

The code source contains a code base (which can be omitted if the entry applies to code from all sources) and the of trusted principals and certificate signers (which can be omitted if signatures are not required for this entry).

The code base is specified as

```
codeBase "url"
```

If the URL ends in a `/`, then it refers to a directory. Otherwise, it is taken to be the name of a JAR file. For exampl

```
grant codeBase "www.horstmann.com/classes/" { . . . };
grant codeBase "www.horstmann.com/classes/MyApp.jar" { . . . };
```

The code base is a URL and should always contain forward slashes as file separators, even for file URLs in Windows example,

```
grant codeBase "file:C:/myapps/classes/" { . . . };
```

Note

Everyone knows that `http` URLs start with two slashes (`http://`). But there seems sufficient confusion about `file` URLs that the policy file reader accepts two forms of file URLs, namely, `file://` *localFile* and `file:` *localFile* . Furthermore, a slash before a Windows drive letter is optional. That is, all of the following are acceptable:

```
file:C:/dir/filename.ext
file:/C:/dir/filename.ext
file://C:/dir/filename.ext
file:///C:/dir/filename.ext
```

Actually, in our tests, the `file:////C:/dir/filename.ext` is acceptable as well, and we have no explanation for that.

The permissions have the following structure:

```
permission className targetName, actionList;
```

The class name is the fully qualified class name of the permission class (such as `java.io.FilePermission`). The *t* *name* is a permission-specific value, for example, a file or directory name for the file permission, or a host and por socket permission. The *actionList* is also permission specific. It is a list of actions, such as `read` or `connect` , separ commas. Some permission classes don't need target names and action lists. Table 9-2 lists the commonly used pe classes and their actions.

Table 9-2. Permissions and Their Associated Targets and Actions

Permission	Target	Action
`java.io.FilePermission`	file target (see text)	`read, write, execute, del`
`java.net.SocketPermission`	socket target (see text)	`accept, connect, listen,`
`java.util.PropertyPermission`	property target (see text)	`read` , `write`

`java.lang.RuntimePermission`	Code View:
	`createClassLoader`
	`getClassLoader`
	`setContextClassLoader`
	`enableContextClassLoaderOverride`
	`createSecurityManager`
	`setSecurityManager`
		(none)

Permission	Target	Action
	exitVM	
	getenv.*variableName*	
	shutdownHooks	
	setFactory	
	setIO	
	modifyThread	
	stopThread	
	modifyThreadGroup	
	getProtectionDomain	
	readFileDescriptor	
	writeFileDescriptor	
	loadLibrary.*libraryName*	
	accessClassInPackage.*packageName*	
	defineClassInPackage.*packageName*	
	accessDeclaredMembers.*className*	
	queuePrintJob	
	getStackTrace	
	setDefaultUncaughtExceptionHandler	
	preferences	
	usePolicy	
java.awt.AWTPermission	showWindowWithoutWarningBanner	(none)
	accessClipboard	
	accessEventQueue	
	createRobot	
	fullScreenExclusive	
	listenToAllAWTEvents	
	readDisplayPixels	
	replaceKeyboardFocusManager	
	watchMousePointer	
	setWindowAlwaysOnTop	
	setAppletStub	
java.net.NetPermission	setDefaultAuthenticator	(none)
	specifyStreamHandler	
	requestPasswordAuthentication	
	setProxySelector	
	getProxySelector	
	setCookieHandler	
	getCookieHandler	
	setResponseCache	
	getResponseCache	
java.lang.reflect.ReflectPermission	suppressAccessChecks	(none)
java.io.SerializablePermission	enableSubclassImplementation	(none)
	enableSubstitution	

Permission	Target	Action
java.security.SecurityPermission	createAccessControlContext getDomainCombiner getPolicy setPolicy getProperty.*keyName* setProperty.*keyName* insertProvider.*providerName* removeProvider.*providerName* setSystemScope setIdentityPublicKey setIdentityInfo addIdentityCertificate removeIdentityCertificate printIdentity clearProviderProperties.*providerName* putProviderProperty.*providerName* removeProviderProperty.*providerName* getSignerPrivateKey setSignerKeyPair	(none)
java.security.AllPermission	(none)	(none)
javax.audio.AudioPermission	play record	(none)
javax.security.auth.AuthPermission	doAs doAsPrivileged getSubject getSubjectFromDomainCombiner setReadOnly modifyPrincipals modifyPublicCredentials modifyPrivateCredentials refreshCredential destroyCredential createLoginContext.*contextName* getLoginConfiguration setLoginConfiguration refreshLoginConfiguration	(none)
java.util.logging.LoggingPermission	control	(none)
java.sql.SQLPermission	setLog	(none)

As you can see from Table 9-2 , most permissions simply permit a particular operation. You can think of the opera
the target with an implied action `"permit"` . These permission classes all extend the `BasicPermission` class (see

9-7 on page 774). However, the targets for the file, socket, and property permissions are more complex, and we investigate them in detail.

File permission targets can have the following form:

file	a file
directory /	a directory
directory /*	all files in the directory
*	all files in the current directory
directory /-	all files in the directory or one of its subdirectories
-	all files in the current directory or one of its subdirectories
<<ALL FILES>>	all files in the file system

For example, the following permission entry gives access to all files in the directory /myapp and any of its subdirec

```
permission java.io.FilePermission "/myapp/-", "read,write,delete";
```

You must use the \\ escape sequence to denote a backslash in a Windows file name.

```
permission java.io.FilePermission "c:\\myapp\\-", "read,write,delete";
```

Socket permission targets consist of a host and a port range. Host specifications have the following form:

hostname or *IPaddress*	a single host
localhost or the empty string	the local host
*. *domainSuffix*	any host whose domain ends with the given suffix
*	all hosts

Port ranges are optional and have the form:

: *n*	a single port
: *n* -	all ports numbered *n* and above
:- *n*	all ports numbered *n* and below
: *n* 1- *n* 2	all ports in the given range

Here is an example:

Code View:
```
permission java.net.SocketPermission "*.horstmann.com:8000-8999", "connect";
```

Finally, property permission targets can have one of two forms:

property	a specific property
propertyPrefix `.*`	all properties with the given prefix

Examples are `"java.home"` and `"java.vm.*"` .

For example, the following permission entry allows a program to read all properties that start with `java.vm` .

```
permission java.util.PropertyPermission "java.vm.*", "read";
```

You can use system properties in policy files. The token `${` *property* `}` is replaced by the property value. For examp `${user.home}` is replaced by the home directory of the user. Here is a typical use of this system property in a perr entry.

```
permission java.io.FilePermission "${user.home}", "read,write";
```

To create platform-independent policy files, it is a good idea to use the `file.separator` property instead of explic `\\` separators. To make this simpler, the special notation `${/}` is a shortcut for `${file.separator}` . For example

```
permission java.io.FilePermission "${user.home}${/}-", "read,write";
```

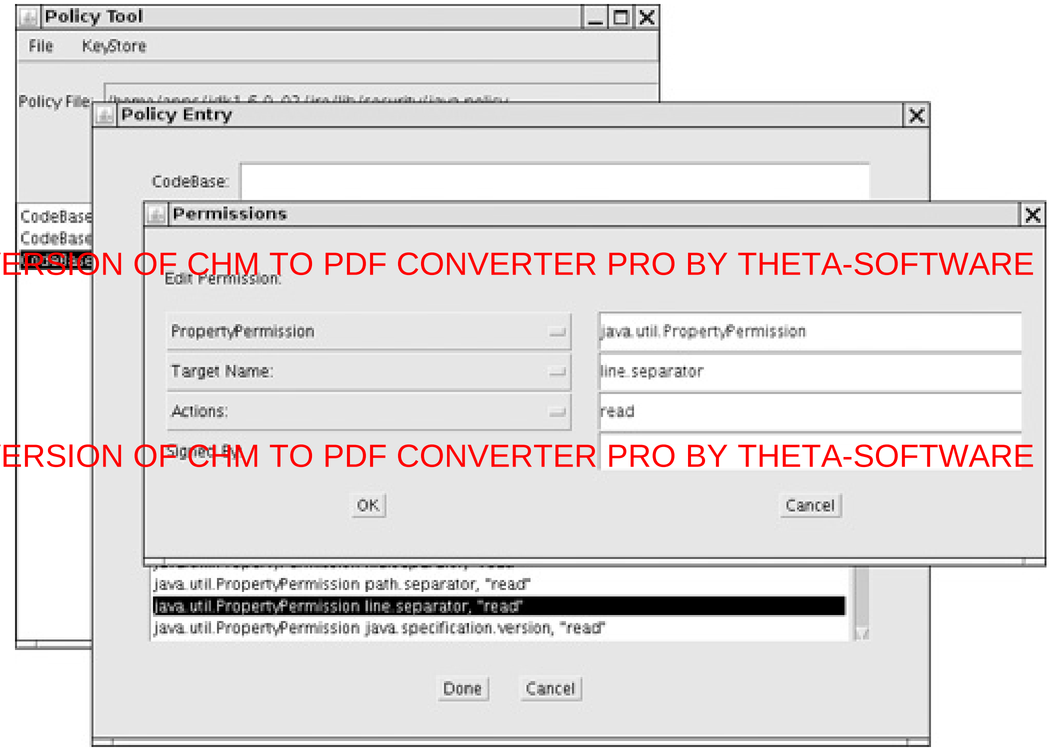
is a portable entry for granting permission to read and write in the user's home directory and any of its subdirecto

Note

The JDK comes with a rudimentary tool, called `policytool` , that you can use to edit policy files (see Figure 9-8). Of course, this tool is not suitable for end users who would be completely mystified by most of the settings. We view it as a proof of concept for an administration tool that might be used by system administrators who prefer point-and-click over syntax. Still, what's missing is a sensible set of categories (such as low, medium, or high security) that is meaningful to nonexperts. As a general observation, we believe that the Java platform certainly contains all the pieces for a fine-grained security model but that it could benefit from some polish in delivering these pieces to end users and system administrators.

Figure 9-8. The policy tool

[View full size image]



Custom Permissions

In this section, you see how you can supply your own permission class that users can refer to in their policy files.

To implement your permission class, you extend the `Permission` class and supply the following methods:

- A constructor with two `String` parameters, for the target and the action list
- `String getActions()`
- `boolean equals()`
- `int hashCode()`
- `boolean implies(Permission other)`

The last method is the most important. Permissions have an *ordering*, in which more general permissions *imply* m specific ones. Consider the file permission

```
p1 = new FilePermission("/tmp/-", "read, write");
```

This permission allows reading and writing of any file in the `/tmp` directory and any of its subdirectories.

This permission implies other, more specific permissions:

```
p2 = new FilePermission("/tmp/-", "read");
p3 = new FilePermission("/tmp/aFile", "read, write");
p4 = new FilePermission("/tmp/aDirectory/-", "write");
```

In other words, a file permission `p1` implies another file permission `p2` if

1. The target file set of `p1` contains the target file set of `p2` .
2. The action set of `p1` contains the action set of `p2` .

Consider the following example of the use of the `implies` method. When the `FileInputStream` constructor wants a file for reading, it checks whether it has permission to do so. For that check, a *specific* file permission object is pa the `checkPermission` method:

```
checkPermission(new FilePermission(fileName, "read"));
```

The security manager now asks all applicable permissions whether they imply this permission. If any one of them it, then the check passes.

In particular, the `AllPermission` implies all other permissions.

If you define your own permission classes, then you need to define a suitable notion of implication for your permis objects. Suppose, for example, that you define a `TVPermission` for a set-top box powered by Java technology. A permission

```
new TVPermission("Tommy:2-12:1900-2200", "watch,record")
```

might allow Tommy to watch and record television channels 2-12 between 19:00 and 22:00. You need to impleme `implies` method so that this permission implies a more specific one, such as

```
new TVPermission("Tommy:4:2000-2100", "watch")
```

Implementation of a Permission Class

In the next sample program, we implement a new permission for monitoring the insertion of text into a text area. program ensures that you cannot add "bad words" such as *sex, drugs,* and *C++* into a text area. We use a custom permission class so that the list of bad words can be supplied in a policy file.

The following subclass of `JTextArea` asks the security manager whether it is okay to add new text:

```
class WordCheckTextArea extends JTextArea
{
   public void append(String text)
   {
      WordCheckPermission p = new WordCheckPermission(text, "insert");
      SecurityManager manager = System.getSecurityManager();
      if (manager != null) manager.checkPermission(p);
```

```
        super.append(text);
    }
}
```

If the security manager grants the WordCheckPermission , then the text is appended. Otherwise, the checkPermi
method throws an exception.

Word check permissions have two possible actions: insert (the permission to insert a specific text) and avoid (th
permission to add any text that avoids certain bad words). You should run this program with the following policy fi

```
grant
{
   permission WordCheckPermission "sex,drugs,C++", "avoid";
};
```


This policy file grants the permission to insert any text that avoids the bad words *sex, drugs,* and *C++.*

When designing the WordCheckPermission class, we must pay particular attention to the implies method. Here a
rules that control whether permission p1 implies permission p2 .

- If p1 has action avoid and p2 has action insert , then the target of p2 must avoid all words in p1 . For exam
  permission

  ```
  WordCheckPermission "sex,drugs,C++", "avoid"
  ```

  implies the permission

  ```
  WordCheckPermission "Mary had a little lamb", "insert"
  ```

- If p1 and p2 both have action avoid , then the word set of p2 must contain all words in the word set of p1 . F
  example, the permission

  ```
  WordCheckPermission "sex,drugs", "avoid"
  ```

  implies the permission

  ```
  WordCheckPermission "sex,drugs,C++", "avoid"
  ```

- If p1 and p2 both have action insert , then the text of p1 must contain the text of p2 . For example, the per

  ```
  WordCheckPermission "Mary had a little lamb", "insert"
  ```

  implies the permission

  ```
  WordCheckPermission "a little lamb", "insert"
  ```

You can find the implementation of this class in Listing 9-4 .

Note that you retrieve the permission target with the confusingly named getName method of the Permission class

Because permissions are described by a pair of strings in policy files, permission classes need to be prepared to pa
these strings. In particular, we use the following method to transform the comma-separated list of bad words of a
permission into a genuine Set .

```
public Set<String> badWordSet()
{
   Set<String> set = new HashSet<String>();
   set.addAll(Arrays.asList(getName().split(",")));
   return set;
}
```

This code allows us to use the `equals` and `containsAll` methods to compare sets. As you saw in Chapter 2 , the e
method of a set class finds two sets to be equal if they contain the same elements in any order. For example, the s
resulting from `"sex,drugs,C++"` and `"C++,drugs,sex"` are equal.

Caution

Make sure that your permission class is a public class. The policy file loader cannot load classes with package visibility outside the boot class path, and it silently ignores any classes that it cannot find.

The program in Listing 9-5 shows how the `WordCheckPermission` class works. Type any text into the text field anc
the Insert button. If the security check passes, the text is appended to the text area. If not, an error message is d
(see Figure 9-9).

Figure 9-9. The **PermissionTest** program

Caution

If you carefully look at Figure 9-9 , you will see that the frame window has a warning border with the misleading caption `"Java Applet Window."` The window caption is determined by the `showWindowWithoutWarningBanner` target of the `java.awt.AWTPermission` . If you like, you can edit the policy file to grant that permission.

You have now seen how to configure Java platform security. Most commonly, you will simply tweak the standard permissions. For additional control, you can define custom permissions that can be configured in the same way as standard permissions.

Listing 9-4. **`WordCheckPermission.java`**

Code View:
```
import java.security.*;
import java.util.*;

/**
 * A permission that checks for bad words.
 * @version 1.00 1999-10-23
 * @author Cay Horstmann
 */
public class WordCheckPermission extends Permission
{
   /**
    * Constructs a word check permission
    * @param target a comma separated word list
    * @param anAction "insert" or "avoid"
    */
   public WordCheckPermission(String target, String anAction)
   {
      super(target);
      action = anAction;
   }

   public String getActions()
   {
      return action;
   }

   public boolean equals(Object other)
   {
      if (other == null) return false;
      if (!getClass().equals(other.getClass())) return false;
      WordCheckPermission b = (WordCheckPermission) other;
      if (!action.equals(b.action)) return false;
      if (action.equals("insert")) return getName().equals(b.getName());
      else if (action.equals("avoid")) return badWordSet().equals(b.badWordSet());
      else return false;
   }

   public int hashCode()
   {
      return getName().hashCode() + action.hashCode();
```

```
    }

    public boolean implies(Permission other)
    {
       if (!(other instanceof WordCheckPermission)) return false;
       WordCheckPermission b = (WordCheckPermission) other;
       if (action.equals("insert"))
       {
          return b.action.equals("insert") && getName().indexOf(b.getName()) >= 0;
       }
       else if (action.equals("avoid"))
       {
          if (b.action.equals("avoid")) return b.badWordSet().containsAll(badWordSet());
          else if (b.action.equals("insert"))
          {
             for (String badWord : badWordSet())
                if (b.getName().indexOf(badWord) >= 0) return false;
             return true;
          }
          else return false;
       }
       else return false;
    }

    /**
     * Gets the bad words that this permission rule describes.
     * @return a set of the bad words
     */
    public Set<String> badWordSet()
    {
       Set<String> set = new HashSet<String>();
       set.addAll(Arrays.asList(getName().split(",")));
       return set;
    }

    private String action;
 }
```

Listing 9-5. **PermissionTest.java**

Code View:

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

/**
 * This class demonstrates the custom WordCheckPermission.
 * @version 1.03 2007-10-06
 * @author Cay Horstmann
 */
public class PermissionTest
{
   public static void main(String[] args)
   {
```

```
      System.setProperty("java.security.policy", "PermissionTest.policy");
      System.setSecurityManager(new SecurityManager());
      EventQueue.invokeLater(new Runnable()
         {
            public void run()
            {
               JFrame frame = new PermissionTestFrame();
               frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
               frame.setVisible(true);
            }
         });
   }
}

/**
 * This frame contains a text field for inserting words into a text area that is protected
 * from "bad words".
 */
class PermissionTestFrame extends JFrame
{
   public PermissionTestFrame()
   {
      setTitle("PermissionTest");
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

      textField = new JTextField(20);
      JPanel panel = new JPanel();
      panel.add(textField);
      JButton openButton = new JButton("Insert");
      panel.add(openButton);
      openButton.addActionListener(new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               insertWords(textField.getText());
            }
         });

      add(panel, BorderLayout.NORTH);

      textArea = new WordCheckTextArea();
      add(new JScrollPane(textArea), BorderLayout.CENTER);
   }

   /**
    * Tries to insert words into the text area. Displays a dialog if the attempt fails.
    * @param words the words to insert
    */
   public void insertWords(String words)
   {
      try
      {
         textArea.append(words + "\n");
      }
      catch (SecurityException e)
      {
         JOptionPane.showMessageDialog(this, "I am sorry, but I cannot do that.");
      }
   }
```

```

   private JTextField textField;
   private WordCheckTextArea textArea;
   private static final int DEFAULT_WIDTH = 400;
   private static final int DEFAULT_HEIGHT = 300;
}

/**
 * A text area whose append method makes a security check to see that no bad words are added.
 */
class WordCheckTextArea extends JTextArea
{
   public void append(String text)
   {
      WordCheckPermission p = new WordCheckPermission(text, "insert");
      SecurityManager manager = System.getSecurityManager();
      if (manager != null) manager.checkPermission(p);
      super.append(text);
   }
}
```

API `java.security.Permission` 1.2

- `Permission(String name)`

  constructs a permission with the given target name.

- `String getName()`

  returns the target name of this permission.

- `boolean implies(Permission other)`

  checks whether this permission implies the other permission. That is the case if the other permission describes a more specific condition that is a consequence of the condition described by this permission.

User Authentication

The Java Authentication and Authorization Service (JAAS) is a part of Java SE 1.4 and beyond. The "authentication" part is concerned with ascertaining the identity of a program user. The "authorization" part maps users to permissions.

JAAS is a "pluggable" API that isolates Java applications from the particular technology used to implement authentication. It supports, among others, UNIX logins, NT logins, Kerberos authentication, and certificate-based authentication.

Once a user has been authenticated, you can attach a set of permissions. For example, here we grant Harry a particular set of permissions that other users do not have:


```
grant principal com.sun.security.auth.UnixPrincipal "harry"
{
   permission java.util.PropertyPermission "user.*", "read";
   . . .
};
```

The `com.sun.security.auth.UnixPrincipal` class checks the name of the UNIX user who is running this program. Its `getName` method returns the UNIX login name, and we check whether that name equals `"harry"`.

You use a `LoginContext` to allow the security manager to check such a grant statement. Here is the basic outline of the login code:

Code View:

```
try
{
   System.setSecurityManager(new SecurityManager());
   LoginContext context = new LoginContext("Login1"); // defined in JAAS configuration file
   context.login();
   // get the authenticated Subject
   Subject subject = context.getSubject();
   . . .
   context.logout();
}
catch (LoginException exception) // thrown if login was not successful
{
   exception.printStackTrace();
}
```

Now the `subject` denotes the individual who has been authenticated.

The string parameter `"Login1"` in the `LoginContext` constructor refers to an entry with the same name in the JAAS configuration file. Here is a sample configuration file:

```
Login1
{
   com.sun.security.auth.module.UnixLoginModule required;
   com.whizzbang.auth.module.RetinaScanModule sufficient;
```

```
};

Login2
{
   . . .
};
```

Of course, the JDK contains no biometric login modules. The following modules are supplied in the `com.sun.security.auth.module` package:

```
UnixLoginModule
NTLoginModule
Krb5LoginModule
JndiLoginModule
KeyStoreLoginModule
```

A login policy consists of a sequence of login modules, each of which is labeled `required`, `sufficient`, `requisite`, or `optional`. The meaning of these keywords is given by the following algorithm:

1. The modules are executed in turn, until a `sufficient` module succeeds, a `requisite` module fails, or the end of the module list is reached.

2. Authentication is successful if all `required` and `requisite` modules succeed, or if none of them were executed, if at least one `sufficient` or `optional` module succeeds.

A login authenticates a *subject*, which can have multiple *principals*. A principal describes some property of the subject, such as the user name, group ID, or role. As you saw in the `grant` statement, principals govern permissions. The `com.sun.security.auth.UnixPrincipal` describes the UNIX login name, and the `UnixNumericGroupPrincipal` can test for membership in a UNIX group.

A `grant` clause can test for a principal, with the syntax

`grant` *principalClass* "*principalName*"

For example:

```
grant com.sun.security.auth.UnixPrincipal "harry"
```

When a user has logged in, you then run, in a separate access control context, the code that requires checking of principals. Use the static `doAs` or `doAsPrivileged` method to start a new `PrivilegedAction` whose `run` method executes the code.

Both of those methods execute an action by calling the `run` method of an object that implements the `PrivilegedAction` interface, using the permissions of the subject's principals:

Code View:

```
PrivilegedAction<T> action = new
   PrivilegedAction()
```

```
    {
       public T run()
       {
         // run with permissions of subject principals
          . . .
       }
    };
 T result = Subject.doAs(subject, action); // or Subject.doAsPrivileged(subject, action, null)
```


If the actions can throw checked exceptions, then you implement the PrivilegedExceptionAction interface instead.

The difference between the doAs and doAsPrivileged methods is subtle. The doAs method starts out with the current access control context, whereas the doAsPrivileged method starts out with a new context. The latter method allows you to separate the permissions for the login code and the "business logic." In our example application, the login code has permissions

```
permission javax.security.auth.AuthPermission "createLoginContext.Login1";
permission javax.security.auth.AuthPermission "doAsPrivileged";
```

The authenticated user has a permission

```
permission java.util.PropertyPermission "user.*", "read";
```

If we had used doAs instead of doAsPrivileged, then the login code would have also needed that permission!

The program in Listing 9-6 and Listing 9-7 demonstrates how to restrict permissions to certain users. The AuthTest program authenticates a user and then runs a simple action that retrieves a system property.

To make this example work, package the code for the login and the action into two separate JAR files:

```
javac *.java
jar cvf login.jar AuthTest.class
jar cvf action.jar SysPropAction.class
```

If you look at the policy file in Listing 9-8, you will see that the UNIX user with the name harry has the permission to read all files. Change harry to your login name. Then run the command

```
java -classpath login.jar:action.jar
   -Djava.security.policy=AuthTest.policy
   -Djava.security.auth.login.config=jaas.config
   AuthTest
```

Listing 9-12 shows the login configuration.

On Windows, change Unix to NT in both AuthTest.policy and jaas.config, and use a semicolon to separate the JAR files:

```
java -classpath login.jar;action.jar . . .
```

The AuthTest program should now display the value of the user.home property. However, if you change the login name in the AuthTest.policy file, then a security exception should be thrown because you no longer have the required permission.

Caution

Be careful to follow these instructions *exactly*. It is very easy to get the setup wrong by making seemingly innocuous changes.

Listing 9-6. **AuthTest.java**

Code View:
```
import java.security.*;
import javax.security.auth.*;
import javax.security.auth.login.*;

/**
 * This program authenticates a user via a custom login and then executes the SysPropAction
 * with the user's privileges.
 * @version 1.01 2007-10-06
 * @author Cay Horstmann
 */
public class AuthTest
{
   public static void main(final String[] args)
   {
      System.setSecurityManager(new SecurityManager());
      try
      {
         LoginContext context = new LoginContext("Login1");
         context.login();
         System.out.println("Authentication successful.");
         Subject subject = context.getSubject();
         System.out.println("subject=" + subject);
         PrivilegedAction<String> action = new SysPropAction("user.home");
         String result = Subject.doAsPrivileged(subject, action, null);
         System.out.println(result);
         context.logout();
      }
      catch (LoginException e)
      {
         e.printStackTrace();
      }
   }
}
```

Listing 9-7. **SysPropAction.java**

Code View:
```
import java.security.*;

/**
   This action looks up a system property.
 * @version 1.01 2007-10-06
 * @author Cay Horstmann
*/
public class SysPropAction implements PrivilegedAction<String>
{
   /**
      Constructs an action for looking up a given property.
      @param propertyName the property name (such as "user.home")
   */
   public SysPropAction(String propertyName) { this.propertyName = propertyName; }

   public String run()
   {
      return System.getProperty(propertyName);
   }

   private String propertyName;
}
```


Listing 9-8. **AuthTest.policy**

Code View:
```
grant codebase "file:login.jar"
{
   permission javax.security.auth.AuthPermission "createLoginContext.Login1";
   permission javax.security.auth.AuthPermission "doAsPrivileged";
};

grant principal com.sun.security.auth.UnixPrincipal "harry"
{
   permission java.util.PropertyPermission "user.*", "read";
};
```

API

`javax.security.auth.login.LoginContext` 1.4

- `LoginContext(String name)`

  constructs a login context. The `name` corresponds to the login descriptor in the JAAS configuration file.

- `void login()`

  establishes a login or throws `LoginException` if the login failed. Invokes the `login` method on the managers in the JAAS configuration file.

- `void logout()`

  logs out the subject. Invokes the `logout` method on the managers in the JAAS configuration file.

- `Subject getSubject()`

  returns the authenticated subject.

API

`javax.security.auth.Subject` 1.4

- `Set<Principal> getPrincipals()`

  gets the principals of this subject.

- `static Object doAs(Subject subject, PrivilegedAction action)`

- `static Object doAs(Subject subject, PrivilegedExceptionAction action)`

- `static Object doAsPrivileged(Subject subject, PrivilegedAction action, AccessControlContext context)`

- `static Object doAsPrivileged(Subject subject, PrivilegedExceptionAction action, AccessControlContext context)`

  executes the privileged action on behalf of the subject. Returns the return value of the `run` method. The `doAsPrivileged` methods execute the action in the given access control context. You can supply a "context snapshot" that you obtained earlier by calling the static method `AccessController.getContext()`, or you can supply `null` to execute the code in a new context.

API `java.security.PrivilegedAction` 1.4

- `Object run()`

  You must define this method to execute the code that you want to have executed on behalf of a subject.

API `java.security.PrivilegedExceptionAction` 1.4

- `Object run()`

  You must define this method to execute the code that you want to have executed on behalf of a subject. This method may throw any checked exceptions.

API `java.security.Principal` 1.1

- `String getName()`

  returns the identifying name of this principal.

JAAS Login Modules

In this section, we look at a JAAS example that shows you

- How to implement your own login module.
- How to implement *role-based* authentication.

Supplying your own login module is useful if you store login information in a database. Even if you are happy with the default module, studying a custom module will help you understand the JAAS configuration file options.

Role-based authentication is essential if you manage a large number of users. It would be impractical to put the names of all legitimate users into a policy file. Instead, the login module should map users to roles such as "admin" or "HR," and the permissions should be based on these roles.

One job of the login module is to populate the principal set of the subject that is being authenticated. If a login module supports roles, it adds `Principal` objects that describe roles. The Java library does not provide a class for this purpose, so we wrote our own (see Listing 9-9). The class simply stores a description/value pair, such as `role=admin`. Its `getName` method returns that pair, so we can add role-based permissions into a policy file:

```
grant principal SimplePrincipal "role=admin" { . . . }
```

Our login module looks up users, passwords, and roles in a text file that contains lines like this:

```
harry|secret|admin
carl|guessme|HR
```

Of course, in a realistic login module, you would store this information in a database or directory.

You can find the code for the `SimpleLoginModule` in Listing 9-10. The `checkLogin` method checks whether the user name and password match a user record in the password file. If so, we add two `SimplePrincipal` objects to the subject's principal set:

```
Set<Principal> principals = subject.getPrincipals();
principals.add(new SimplePrincipal("username", username));
principals.add(new SimplePrincipal("role", role));
```

The remainder of `SimpleLoginModule` is straightforward plumbing. The `initialize` method receives

- The `Subject` that is being authenticated.
- A handler to retrieve login information.
- A `sharedState` map that can be used for communication between login modules.
- An `options` map that contains name/value pairs that are set in the login configuration.

For example, we configure our module as follows:

```
SimpleLoginModule required pwfile="password.txt";
```

The login module retrieves the `pwfile` settings from the `options` map.

The login module does not gather the user name and password; that is the job of a separate handler. This separation allows you to use the same login module without worrying whether the login information comes from a GUI dialog box, a console prompt, or a configuration file.

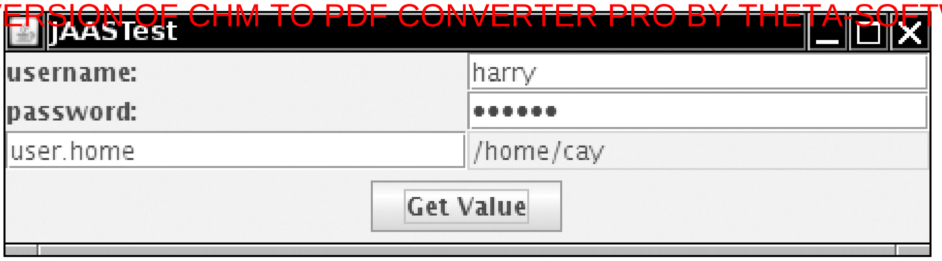
The handler is specified when you construct the `LoginContext`, for example,

```
LoginContext context = new LoginContext("Login1",
   new com.sun.security.auth.callback.DialogCallbackHandler());
```

The DialogCallbackHandler pops up a simple GUI dialog box to retrieve the user name and password. com.sun.security.auth.callback.TextCallbackHandler gets the information from the console.

However, in our application, we have our own GUI for collecting the user name and password (see Figure 9-10). We produce a simple handler that merely stores and returns that information (see Listing 9-11).

Figure 9-10. A custom login module



The handler has a single method, handle, that processes an array of Callback objects. A number of predefined classes, such as NameCallback and PasswordCallback, implement the Callback interface. You could also add your own class, such as RetinaScanCallback. The handler code is a bit unsightly because it needs to analyze the types of the callback objects:

```
public void handle(Callback[] callbacks)
{
   for (Callback callback : callbacks)
   {
      if (callback instanceof NameCallback) . . .
      else if (callback instanceof PasswordCallback) . . .
      else . . .
   }
}
```

The login module prepares an array of the callbacks that it needs for authentication:

```
NameCallback nameCall = new NameCallback("username: ");
PasswordCallback passCall = new PasswordCallback("password: ", false);
callbackHandler.handle(new Callback[] { nameCall, passCall });
```

Then it retrieves the information from the callbacks.

The program in Listing 9-12 displays a form for entering the login information and the name of a system property. If the user is authenticated, the property value is retrieved in a PrivilegedAction. As you can see from the policy file in Listing 9-13, only users with the admin role have permission to read properties.

As in the preceding section, you must separate the login and action code. Create two JAR files:

```
javac *.java
jar cvf login.jar JAAS*.class Simple*.class
jar cvf action.jar SysPropAction.class
```

Then run the program as

```
java -classpath login.jar:action.jar
   -Djava.security.policy=JAASTest.policy
   -Djava.security.auth.login.config=jaas.config
   JAASTest
```

Listing 9-14 shows the login configuration.

Note

It is possible to support a more complex two-phase protocol, whereby a login is *committed* if all modules in the login configuration were successful. For more information, see the login module developer's guide at http://java.sun.com/javase/6/docs/technotes/guides/security/jaas/JAASLMDevGuide.html.

Listing 9-9. **SimplePrincipal.java**

Code View:

```
import java.security.*;

/**
 * A principal with a named value (such as "role=HR" or "username=harry").
 * @version 1.0 2004-09-14
 * @author Cay Horstmann
 */
public class SimplePrincipal implements Principal
{
   /**
    * Constructs a SimplePrincipal to hold a description and a value.
    * @param roleName the role name
    */
   public SimplePrincipal(String descr, String value)
   {
      this.descr = descr;
      this.value = value;
   }

   /**
    * Returns the role name of this principal
    * @return the role name
    */
   public String getName()
   {
      return descr + "=" + value;
   }
```

```

   public boolean equals(Object otherObject)
   {
      if (this == otherObject) return true;
      if (otherObject == null) return false;
      if (getClass() != otherObject.getClass()) return false;
      SimplePrincipal other = (SimplePrincipal) otherObject;
      return getName().equals(other.getName());
   }

   public int hashCode()
   {
      return getName().hashCode();
   }

   private String descr;
   private String value;
}
```


Listing 9-10. **SimpleLoginModule.java**

Code View:

```
import java.io.*;
import java.security.*;
import java.util.*;
import javax.security.auth.*;
import javax.security.auth.callback.*;
import javax.security.auth.login.*;
import javax.security.auth.spi.*;

/**
 * This login module authenticates users by reading usernames, passwords, and roles from a
 * text file.
 * @version 1.0 2004-09-14
 * @author Cay Horstmann
 */
public class SimpleLoginModule implements LoginModule
{
   public void initialize(Subject subject, CallbackHandler callbackHandler,
         Map<String, ?> sharedState, Map<String, ?> options)
   {
      this.subject = subject;
      this.callbackHandler = callbackHandler;
      this.options = options;
   }

   public boolean login() throws LoginException
   {
      if (callbackHandler == null) throw new LoginException("no handler");

      NameCallback nameCall = new NameCallback("username: ");
      PasswordCallback passCall = new PasswordCallback("password: ", false);
      try
      {
```

```
         callbackHandler.handle(new Callback[] { nameCall, passCall });
      }
      catch (UnsupportedCallbackException e)
      {
         LoginException e2 = new LoginException("Unsupported callback");
         e2.initCause(e);
         throw e2;
      }
      catch (IOException e)
      {
         LoginException e2 = new LoginException("I/O exception in callback");
         e2.initCause(e);
         throw e2;
      }

      return checkLogin(nameCall.getName(), passCall.getPassword());
   }

   /**
    * Checks whether the authentication information is valid. If it is, the subject acquires
    * principals for the user name and role.
    * @param username the user name
    * @param password a character array containing the password
    * @return true if the authentication information is valid
    */
   private boolean checkLogin(String username, char[] password) throws LoginException
   {
      try
      {
         Scanner in = new Scanner(new FileReader("" + options.get("pwfile")));
         while (in.hasNextLine())
         {
            String[] inputs = in.nextLine().split("\\|");
           if (inputs[0].equals(username) && Arrays.equals(inputs[1].toCharArray(), password)
            {
               String role = inputs[2];
               Set<Principal> principals = subject.getPrincipals();
               principals.add(new SimplePrincipal("username", username));
               principals.add(new SimplePrincipal("role", role));
               return true;
            }
         }
         in.close();
         return false;
      }
      catch (IOException e)
      {
         LoginException e2 = new LoginException("Can't open password file");
         e2.initCause(e);
         throw e2;
      }
   }

   public boolean logout()
   {
      return true;
   }

   public boolean abort()
```

```
   {
      return true;
   }

   public boolean commit()
   {
      return true;
   }

   private Subject subject;
   private CallbackHandler callbackHandler;
   private Map<String, ?> options;
}
```


Listing 9-11. **SimpleCallbackHandler.java**

Code View:

```
import javax.security.auth.callback.*;

/**
 * This simple callback handler presents the given user name and password.
 * @version 1.0 2004-09-14
 * @author Cay Horstmann
 */
public class SimpleCallbackHandler implements CallbackHandler
{
   /**
    * Constructs the callback handler.
    * @param username the user name
    * @param password a character array containing the password
    */
   public SimpleCallbackHandler(String username, char[] password)
   {
      this.username = username;
      this.password = password;
   }

   public void handle(Callback[] callbacks)
   {
      for (Callback callback : callbacks)
      {
         if (callback instanceof NameCallback)
         {
            ((NameCallback) callback).setName(username);
         }
         else if (callback instanceof PasswordCallback)
         {
            ((PasswordCallback) callback).setPassword(password);
         }
      }
   }

```

```
   private String username;
   private char[] password;
}
```

Listing 9-12. **JAASTest.java**

Code View:
```
import java.awt.*;
import java.awt.event.*;
import javax.security.auth.*;
import javax.security.auth.login.*;
import javax.swing.*;

/**
 * This program authenticates a user via a custom login and then executes the SysPropAction
 * with the user's privileges.
 * @version 1.0 2004-09-14
 * @author Cay Horstmann
 */
public class JAASTest
{
   public static void main(final String[] args)
   {
      System.setSecurityManager(new SecurityManager());
      EventQueue.invokeLater(new Runnable()
         {
            public void run()
            {
               JFrame frame = new JAASFrame();
               frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
               frame.setVisible(true);
            }
         });
   }
}

/**
 * This frame has text fields for user name and password, a field for the name of the requeste
 * system property, and a field to show the property value.
 */
class JAASFrame extends JFrame
{
   public JAASFrame()
   {
      setTitle("JAASTest");

      username = new JTextField(20);
      password = new JPasswordField(20);
      propertyName = new JTextField(20);
      propertyValue = new JTextField(20);
      propertyValue.setEditable(false);

      JPanel panel = new JPanel();
      panel.setLayout(new GridLayout(0, 2));
```

```
      panel.add(new JLabel("username:"));
      panel.add(username);
      panel.add(new JLabel("password:"));
      panel.add(password);
      panel.add(propertyName);
      panel.add(propertyValue);
      add(panel, BorderLayout.CENTER);

      JButton getValueButton = new JButton("Get Value");
      getValueButton.addActionListener(new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               getValue();
            }
         });
      JPanel buttonPanel = new JPanel();
      buttonPanel.add(getValueButton);
      add(buttonPanel, BorderLayout.SOUTH);
      pack();
   }

   public void getValue()
   {
      try
      {
         LoginContext context = new LoginContext("Login1", new SimpleCallbackHandler(username
            .getText(), password.getPassword()));
         context.login();
         Subject subject = context.getSubject();
         propertyValue.setText(""
           + Subject.doAsPrivileged(subject, new SysPropAction(propertyName.getText()), null));
         context.logout();
      }
      catch (LoginException e)
      {
         JOptionPane.showMessageDialog(this, e);
      }
   }

   private JTextField username;
   private JPasswordField password;
   private JTextField propertyName;
   private JTextField propertyValue;
}
```

Listing 9-13. **JAASTest.policy**

Code View:
```
grant codebase "file:login.jar"
{
   permission java.awt.AWTPermission "showWindowWithoutWarningBanner";
   permission javax.security.auth.AuthPermission "createLoginContext.Login1";
   permission javax.security.auth.AuthPermission "doAsPrivileged";
   permission javax.security.auth.AuthPermission "modifyPrincipals";
   permission java.io.FilePermission "password.txt", "read";
};

grant principal SimplePrincipal "role=admin"
{
   permission java.util.PropertyPermission "*", "read";
};
```

Listing 9-14. **jaas.config**

```
Login1
{
   SimpleLoginModule required pwfile="password.txt";
};
```

javax.security.auth.callback.CallbackHandler 1.4

- `void handle(Callback[] callbacks)`

  handles the given callbacks, interacting with the user if desired, and stores the security information in the callback objects.

javax.security.auth.callback.NameCallback 1.4

- `NameCallback(String prompt)`
- `NameCallback(String prompt, String defaultName)`

  constructs a `NameCallback` with the given prompt and default name.

- `void setName(String name)`

- `String getName()`

  sets or gets the name gathered by this callback.

- `String getPrompt()`

  gets the prompt to use when querying this name.

- `String getDefaultName()`

  gets the default name to use when querying this name.

javax.security.auth.callback.PasswordCallback 1.4

- `PasswordCallback(String prompt, boolean echoOn)`

  constructs a `PasswordCallback` with the given prompt and echo flag.

- `void setPassword(char[] password)`
- `char[] getPassword()`

  sets or gets the password gathered by this callback.

- `String getPrompt()`

  gets the prompt to use when querying this password.

- `boolean isEchoOn()`

gets the echo flag to use when querying this password.

API

javax.security.auth.spi.LoginModule 1.4

- void initialize(Subject subject, CallbackHandler handler, Map<String,?> sharedState, Map<String,?> options)

  initializes this LoginModule for authenticating the given subject. During login processing, uses the given handler to gather login information. Use the sharedState map for communication with other login modules. The options map contains the name/value pairs specified in the login configuration for this module instance.

- boolean login()

  carries out the authentication process and populates the subject's principals. Returns true if the login was successful.

- boolean commit()

  is called after all login modules were successful, for login scenarios that require a two-phase commit. Returns true if the operation was successful.

- boolean abort()

  is called if the failure of another login module caused the login process to abort. Returns true if the operation was successful.

- boolean logout()

  logs out this subject. Returns true if the operation was successful.

Digital Signatures

As we said earlier, applets were what started the craze over the Java platform. In practice, people discovered that although they could write animated applets like the famous "nervous text" applet, applets could not do a whole lot useful stuff in the JDK 1.0 security model. For example, because applets under JDK 1.0 were so closely supervised couldn't do much good on a corporate intranet, even though relatively little risk attaches to executing an applet fro company's secure intranet. It quickly became clear to Sun that for applets to become truly useful, it was importan users to be able to assign *different* levels of security, depending on where the applet originated. If an applet come trusted supplier and it has not been tampered with, the user of that applet can then decide whether to give the ap more privileges.

To give more trust to an applet, we need to know two things:

- Where did the applet come from?
- Was the code corrupted in transit?

In the past 50 years, mathematicians and computer scientists have developed sophisticated algorithms for ensurin integrity of data and for electronic signatures. The `java.security` package contains implementations of many of t algorithms. Fortunately, you don't need to understand the underlying mathematics to use the algorithms in the `java.security` package. In the next sections, we show you how message digests can detect changes in data files digital signatures can prove the identity of the signer.

Message Digests

A message digest is a digital fingerprint of a block of data. For example, the so-called SHA1 (secure hash algorithn condenses any data block, no matter how long, into a sequence of 160 bits (20 bytes). As with real fingerprints, o that no two messages have the same SHA1 fingerprint. Of course, that cannot be true—there are only 2^{160} SHA1 fingerprints, so there must be some messages with the same fingerprint. But 2^{160} is so large that the probability o duplication occurring is negligible. How negligible? According to James Walsh in *True Odds: How Risks Affect Your Life* (Merritt Publishing 1996), the chance that you will die from being struck by lightning is about one in 30,000. N think of nine other people, for example, your nine least favorite managers or professors. The chance that you and *them* will die from lightning strikes is higher than that of a forged message having the same SHA1 fingerprint as th original. (Of course, more than ten people, none of whom you are likely to know, will die from lightning strikes. Ho we are talking about the far slimmer chance that *your particular choice* of people will be wiped out.)

A message digest has two essential properties:

- If one bit or several bits of the data are changed, then the message digest also changes.
- A forger who is in possession of a given message cannot construct a fake message that has the same messa as the original.

The second property is again a matter of probabilities, of course. Consider the following message by the billionaire

> *"Upon my death, my property shall be divided equally among my children; however, my son George shall re nothing."*

That message has an SHA1 fingerprint of

```
2D 8B 35 F3 BF 49 CD B1 94 04 E0 66 21 2B 5E 57 70 49 E1 7E
```

The distrustful father has deposited the message with one attorney and the fingerprint with another. Now, suppos can bribe the lawyer holding the message. He wants to change the message so that Bill gets nothing. Of course, t changes the fingerprint to a completely different bit pattern:

```
2A 33 0B 4B B3 FE CC 1C 9D 5C 01 A7 09 51 0B 49 AC 8F 98 92
```

Can George find some other wording that matches the fingerprint? If he had been the proud owner of a billion co from the time the Earth was formed, each computing a million messages a second, he would not yet have found a he could substitute.

A number of algorithms have been designed to compute these message digests. The two best-known are SHA1, th hash algorithm developed by the National Institute of Standards and Technology, and MD5, an algorithm invented Ronald Rivest of MIT. Both algorithms scramble the bits of a message in ingenious ways. For details about these algorithms, see, for example, *Cryptography and Network Security* , 4th ed., by William Stallings (Prentice Hall 200 that recently, subtle regularities have been discovered in both algorithms. At this point, most cryptographers reco avoiding MD5 and using SHA1 until a stronger alternative becomes available. (See http://www.rsa.com/rsalabs/nc id=2834 for more information.)

The Java programming language implements both SHA1 and MD5. The `MessageDigest` class is a *factory* for creati objects that encapsulate the fingerprinting algorithms. It has a static method, called `getInstance` , that returns a of a class that extends the `MessageDigest` class. This means the `MessageDigest` class serves double duty:

- As a factory class
- As the superclass for all message digest algorithms

For example, here is how you obtain an object that can compute SHA fingerprints:

```
MessageDigest alg = MessageDigest.getInstance("SHA-1");
```

(To get an object that can compute MD5, use the string `"MD5"` as the argument to `getInstance` .)

After you have obtained a `MessageDigest` object, you feed it all the bytes in the message by repeatedly calling the method. For example, the following code passes all bytes in a file to the `alg` object just created to do the fingerpri

```
InputStream in = . . .
int ch;
while ((ch = in.read()) != -1)
   alg.update((byte) ch);
```

Alternatively, if you have the bytes in an array, you can update the entire array at once:

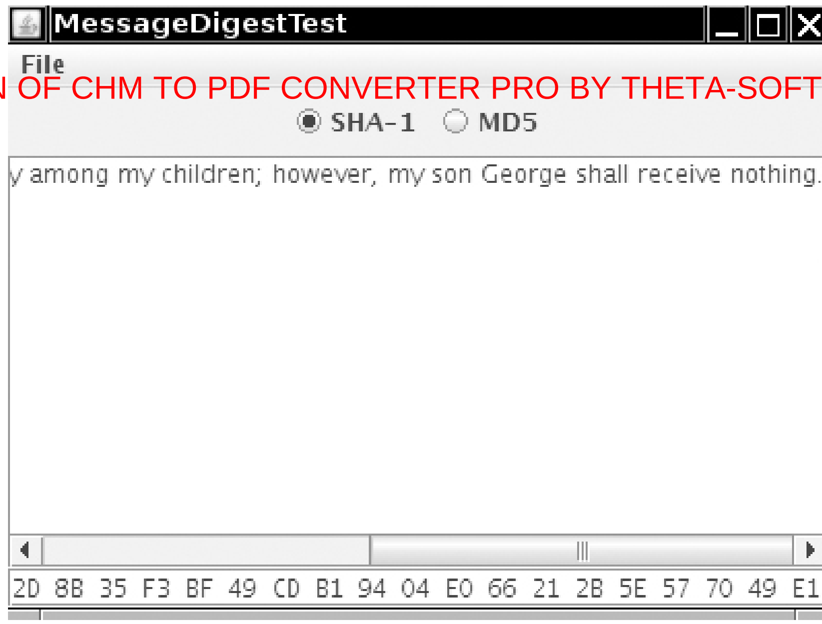
```
byte[] bytes = . . .;
alg.update(bytes);
```

When you are done, call the `digest` method. This method pads the input—as required by the fingerprinting algorithm—does the computation, and returns the digest as an array of bytes.

```
byte[] hash = alg.digest();
```

The program in Listing 9-15 computes a message digest, using either SHA or MD5. You can load the data to be dig from a file, or you can type a message in the text area. Figure 9-11 shows the application.

Figure 9-11. Computing a message digest



Listing 9-15. **MessageDigestTest.java**

Code View:
```
import java.io.*;
import java.security.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

/**
 * This program computes the message digest of a file or the contents of a text area.
 * @version 1.13 2007-10-06
 * @author Cay Horstmann
 */
public class MessageDigestTest
{
   public static void main(String[] args)
   {
      EventQueue.invokeLater(new Runnable()
         {
            public void run()
            {
               JFrame frame = new MessageDigestFrame();
```

```
               frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
               frame.setVisible(true);
            }
         });
   }
}

/**
 * This frame contains a menu for computing the message digest of a file or text area, radio
 * buttons to toggle between SHA-1 and MD5, a text area, and a text field to show the
 * messge digest.
 */
class MessageDigestFrame extends JFrame
{
   public MessageDigestFrame()
   {
      setTitle("MessageDigestTest");
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

      JPanel panel = new JPanel();
      ButtonGroup group = new ButtonGroup();
      addRadioButton(panel, "SHA-1", group);
      addRadioButton(panel, "MD5", group);

      add(panel, BorderLayout.NORTH);
      add(new JScrollPane(message), BorderLayout.CENTER);
      add(digest, BorderLayout.SOUTH);
      digest.setFont(new Font("Monospaced", Font.PLAIN, 12));

      setAlgorithm("SHA-1");

      JMenuBar menuBar = new JMenuBar();
      JMenu menu = new JMenu("File");
      JMenuItem fileDigestItem = new JMenuItem("File digest");
      fileDigestItem.addActionListener(new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               loadFile();
            }
         });
      menu.add(fileDigestItem);
      JMenuItem textDigestItem = new JMenuItem("Text area digest");
      textDigestItem.addActionListener(new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               String m = message.getText();
               computeDigest(m.getBytes());
            }
         });
      menu.add(textDigestItem);
      menuBar.add(menu);
      setJMenuBar(menuBar);
   }

   /**
    * Adds a radio button to select an algorithm.
    * @param c the container into which to place the button
```

```
    * @param name the algorithm name
    * @param g the button group
    */
   public void addRadioButton(Container c, final String name, ButtonGroup g)
   {
      ActionListener listener = new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
```

```
               setAlgorithm(name);
            }
         };
      JRadioButton b = new JRadioButton(name, g.getButtonCount() == 0);
      c.add(b);
      g.add(b);
```

```
      b.addActionListener(listener);
   }

   /**
    * Sets the algorithm used for computing the digest.
    * @param alg the algorithm name
    */
   public void setAlgorithm(String alg)
   {
      try
      {
         currentAlgorithm = MessageDigest.getInstance(alg);
         digest.setText("");
      }
      catch (NoSuchAlgorithmException e)
      {
         digest.setText("" + e);
      }
   }

   /**
    * Loads a file and computes its message digest.
    */
   public void loadFile()
   {
      JFileChooser chooser = new JFileChooser();
      chooser.setCurrentDirectory(new File("."));

      int r = chooser.showOpenDialog(this);
      if (r == JFileChooser.APPROVE_OPTION)
      {
         try
         {
            String name = chooser.getSelectedFile().getAbsolutePath();
            computeDigest(loadBytes(name));
         }
         catch (IOException e)
         {
            JOptionPane.showMessageDialog(null, e);
         }
      }
   }

   /**
```

```
    * Loads the bytes in a file.
    * @param name the file name
    * @return an array with the bytes in the file
    */
   public byte[] loadBytes(String name) throws IOException
   {
      FileInputStream in = null;

      in = new FileInputStream(name);
      try
      {
         ByteArrayOutputStream buffer = new ByteArrayOutputStream();
         int ch;
         while ((ch = in.read()) != -1)
            buffer.write(ch);
         return buffer.toByteArray();
      }
      finally
      {
         in.close();
      }
   }

   /**
    * Computes the message digest of an array of bytes and displays it in the text field.
    * @param b the bytes for which the message digest should be computed.
    */
   public void computeDigest(byte[] b)
   {
      currentAlgorithm.reset();
      currentAlgorithm.update(b);
      byte[] hash = currentAlgorithm.digest();
      String d = "";
      for (int i = 0; i < hash.length; i++)
      {
         int v = hash[i] & 0xFF;
         if (v < 16) d += "0";
         d += Integer.toString(v, 16).toUpperCase() + " ";
      }
      digest.setText(d);
   }

   private JTextArea message = new JTextArea();
   private JTextField digest = new JTextField();
   private MessageDigest currentAlgorithm;
   private static final int DEFAULT_WIDTH = 400;
   private static final int DEFAULT_HEIGHT = 300;
}
```

java.security.MessageDigest 1.1

- `static MessageDigest getInstance(String algorithmName)`

  returns a `MessageDigest` object that implements the specified algorithm. Throws `NoSuchAlgorithmException` if the algorithm is not provided.

- `void update(byte input)`

- `void update(byte[] input)`

- `void update(byte[] input, int offset, int len)`

  updates the digest, using the specified bytes.

- `byte[] digest()`

  completes the hash computation, returns the computed digest, and resets the algorithm object.

- `void reset()`

  resets the digest.

Message Signing

In the last section, you saw how to compute a message digest, a fingerprint for the original message. If the messa altered, then the fingerprint of the altered message will not match the fingerprint of the original. If the message a fingerprint are delivered separately, then the recipient can check whether the message has been tampered with. H if both the message and the fingerprint were intercepted, it is an easy matter to modify the message and then rec the fingerprint. After all, the message digest algorithms are publicly known, and they don't require secret keys. In case, the recipient of the forged message and the recomputed fingerprint would never know that the message has altered. Digital signatures solve this problem.

To help you understand how digital signatures work, we explain a few concepts from the field called *public key cryptography.* Public key cryptography is based on the notion of a *public* key and *private* key. The idea is that you everyone in the world your public key. However, only you hold the private key, and it is important that you safegu and don't release it to anyone else. The keys are matched by mathematical relationships, but the exact nature of t relationships is not important for us. (If you are interested, you can look it up in *The Handbook of Applied Cryptog* http://www.cacr.math.uwaterloo.ca/hac/ .)

The keys are quite long and complex. For example, here is a matching pair of public and private Digital Signature Algorithm (DSA) keys.

Public key:

Code View:

```
p:
fca682ce8e12caba26efccf7110e526db078b05edecbcd1eb4a208f3ae1617ae01f35b91a47e6df63413c5e12ed0899
bcd132acd50d99151bdc43ee737592e17

q: 962eddcc369cba8ebb260ee6b6a126d9346e38c5
g:678471b27a9cf44ee91a49c5147db1a9aaf244f05a434d6486931d2d14271b9e35030b71fd73da179069b32e29356
30e
1c2062354d0da20a6c416e50be794ca4

y:
c0b6e67b4ac098eb1a32c5f8c4c1f0e7e6fb9d832532e27d0bdab9ca2d2a8123ce5a8018b8161a760480fadd040b927
281ddb22cb9bc4df596d7de4d1b977d50
```

Private key:

Code View:

```
p:
fca682ce8e12caba26efccf7110e526db078b05edecbcd1eb4a208f3ae1617ae01f35b91a47e6df63413c5e12ed0899
bcd132acd50d99151bdc43ee737592e17

q: 962eddcc369cba8ebb260ee6b6a126d9346e38c5

g:
678471b27a9cf44ee91a49c5147db1a9aaf244f05a434d6486931d2d14271b9e35030b71fd73da179069b32e2935630
e1c2062354d0da20a6c416e50be794ca4

x: 146c09f881656cc6c51f27ea6c3a91b85ed1d70a
```

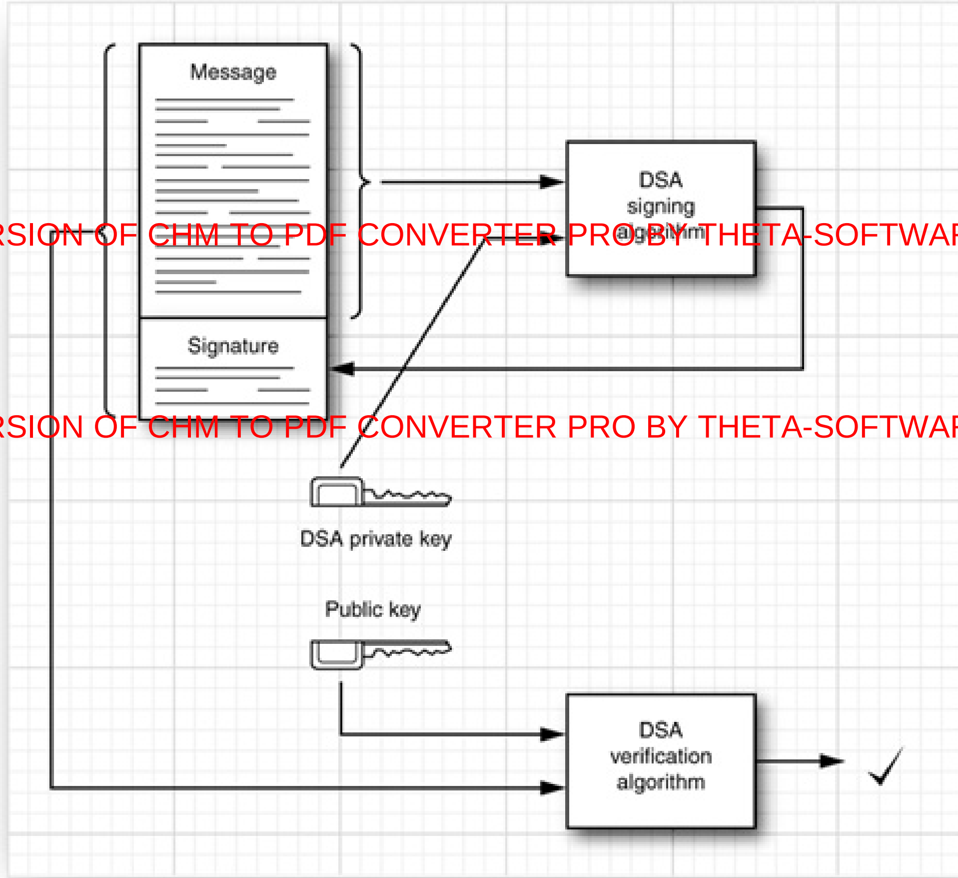
It is believed to be practically impossible to compute one key from the other. That is, even though everyone knows public key, they can't compute your private key in your lifetime, no matter how many computing resources they h available.

It might seem difficult to believe that nobody can compute the private key from the public keys, but nobody has e found an algorithm to do this for the encryption algorithms that are in common use today. If the keys are sufficien brute force—simply trying all possible keys—would require more computers than can be built from all the atoms in solar system, crunching away for thousands of years. Of course, it is possible that someone could come up with al for computing keys that are much more clever than brute force. For example, the RSA algorithm (the encryption a invented by Rivest, Shamir, and Adleman) depends on the difficulty of factoring large numbers. For the last 20 yea many of the best mathematicians have tried to come up with good factoring algorithms, but so far with no success that reason, most cryptographers believe that keys with a "modulus" of 2,000 bits or more are currently complete from any attack. DSA is believed to be similarly secure.

Figure 9-12 illustrates how the process works in practice.

Figure 9-12. Public key signature exchange with DSA

[View full size image]



Suppose Alice wants to send Bob a message, and Bob wants to know this message came from Alice and not an im
Alice writes the message and then *signs* the message digest with her private key. Bob gets a copy of her public ke
then applies the public key to *verify* the signature. If the verification passes, then Bob can be assured of two facts

- The original message has not been altered.
- The message was signed by Alice, the holder of the private key that matches the public key that Bob used fo verification.

You can see why security for private keys is all-important. If someone steals Alice's private key or if a government require her to turn it over, then she is in trouble. The thief or a government agent can impersonate her by sending messages, money transfer instructions, and so on, that others will believe came from Alice.

The X.509 Certificate Format

To take advantage of public key cryptography, the public keys must be distributed. One of the most common distr formats is called X.509. Certificates in the X.509 format are widely used by VeriSign, Microsoft, Netscape, and mai companies, for signing e-mail messages, authenticating program code, and certifying many other kinds of data. Th standard is part of the X.500 series of recommendations for a directory service by the international telephone stan body, the CCITT.

The precise structure of X.509 certificates is described in a formal notation, called "abstract syntax notation #1" o
Figure 9-13 shows the ASN.1 definition of version 3 of the X.509 format. The exact syntax is not important for us,
you can see, ASN.1 gives a precise definition of the structure of a certificate file. The *basic encoding rules,* or BER,
variation, called *distinguished encoding rules* (DER) describe precisely how to save this structure in a binary file. T
BER and DER describe how to encode integers, character strings, bit strings, and constructs such as `SEQUENCE` , `CH`
and `OPTIONAL` .

Figure 9-13. ASN.1 definition of X.509v3

Code View:

```
[Certificate  ::=  SEQUENCE  {
        tbsCertificate       TBSCertificate,
        signatureAlgorithm   AlgorithmIdentifier,
        signature            BIT STRING  }

   TBSCertificate  ::=  SEQUENCE  {
        version         [0]   EXPLICIT Version DEFAULT v1,
        serialNumber          CertificateSerialNumber,
        signature             AlgorithmIdentifier,
        issuer                Name,
        validity              Validity,
        subject               Name,
        subjectPublicKeyInfo  SubjectPublicKeyInfo,
        issuerUniqueID  [1]  IMPLICIT UniqueIdentifier OPTIONAL,
                              -- If present, version must be v2 or v3
        subjectUniqueID [2]  IMPLICIT UniqueIdentifier OPTIONAL,
                             -- If present, version must be v2 or v3
        extensions      [3]  EXPLICIT Extensions OPTIONAL
                             -- If present, version must be v3
        }

   Version  ::=  INTEGER  {  v1(0), v2(1), v3(2)  }

   CertificateSerialNumber  ::=  INTEGER

   Validity ::= SEQUENCE {
        notBefore      CertificateValidityDate,
        notAfter       CertificateValidityDate }

   CertificateValidityDate ::= CHOICE {
        utcTime        UTCTime,
        generalTime    GeneralizedTime }

   UniqueIdentifier  ::=  BIT STRING

   SubjectPublicKeyInfo  ::=  SEQUENCE  {
        algorithm            AlgorithmIdentifier,
        subjectPublicKey     BIT STRING  }

   Extensions  ::=  SEQUENCE OF Extension

   Extension  ::=  SEQUENCE  {
        extnID      OBJECT IDENTIFIER,
        critical    BOOLEAN DEFAULT FALSE,
        extnValue   OCTET STRING  }
```

Note

You can find more information on ASN.1 in *A Layman's Guide to a Subset of ASN.1, BER, and DER* by Burton S. Kaliski, Jr. (ftp://ftp.rsa.com/pub/pkcs/ps/layman.ps), *ASN.1—Communication Between Heterogeneous Systems* by Olivier Dubuisson (Academic Press 2000) (http://www.oss.com/asn1/dubuisson.html) and *ASN.1 Complete* by John Larmouth (Morgan Kaufmann Publishers 1999) (http://www.oss.com/asn1/larmouth.html).

Verifying a Signature

The JDK comes with the `keytool` program, which is a command-line tool to generate and manage a set of certifica expect that ultimately the functionality of this tool will be embedded in other, more user-friendly programs. But rig we use `keytool` to show how Alice can sign a document and send it to Bob, and how Bob can verify that the docur really was signed by Alice and not an imposter.

The `keytool` program manages *keystores,* databases of certificates and private/public key pairs. Each entry in the has an *alias* . Here is how Alice creates a keystore, `alice.certs` , and generates a key pair with alias `alice` .

```
keytool -genkeypair -keystore alice.certs -alias alice
```

When creating or opening a keystore, you are prompted for a keystore password. For this example, just use `secre` you were to use the `keytool` -generated keystore for any serious purpose, you would need to choose a good passv safeguard this file.

When generating a key, you are prompted for the following information:

Code View:
```
Enter keystore password:  secret
Reenter new password:  secret
What is your first and last name?
  [Unknown]:  Alice Lee
What is the name of your organizational unit?
  [Unknown]:  Engineering Department
What is the name of your organization?
  [Unknown]:  ACME Software
What is the name of your City or Locality?
  [Unknown]:  San Francisco
What is the name of your State or Province?
  [Unknown]:  CA
What is the two-letter country code for this unit?
  [Unknown]:  US
Is <CN=Alice Lee, OU=Engineering Department, O=ACME Software, L=San Francisco, ST=CA, C=US> cor-
rect?
  [no]:  yes
```

The keytool uses X.500 distinguished names, with components Common Name (CN), Organizational Unit (OU), Organization (O), Location (L), State (ST), and Country (C) to identify key owners and certificate issuers.

Finally, specify a key password, or press ENTER to use the keystore password as the key password.

Suppose Alice wants to give her public key to Bob. She needs to export a certificate file:

```
keytool -exportcert -keystore alice.certs -alias alice -file alice.cer
```

Now Alice can send the certificate to Bob. When Bob receives the certificate, he can print it:

```
keytool -printcert -file alice.cer
```

The printout looks like this:

Code View:
```
Owner: CN=Alice Lee, OU=Engineering Department, O=ACME Software, L=San Francisco, ST=CA, C=US
Issuer: CN=Alice Lee, OU=Engineering Department, O=ACME Software, L=San Francisco, ST=CA, C=US
Serial number: 470835ce
Valid from: Sat Oct 06 18:26:38 PDT 2007 until: Fri Jan 04 17:26:38 PST 2008
Certificate fingerprints:
         MD5:  BC:18:15:27:85:69:48:B1:5A:C3:0B:1C:C6:11:B7:81
         SHA1: 31:0A:A0:B8:C2:8B:3B:B6:85:7C:EF:C0:57:E5:94:95:61:47:6D:34
         Signature algorithm name: SHA1withDSA
         Version: 3
```

If Bob wants to check that he got the right certificate, he can call Alice and verify the certificate fingerprint over th

Note

Some certificate issuers publish certificate fingerprints on their web sites. For example, to check the VeriSign certificate in the keystore *jre* /lib/security/cacerts directory, use the -list option:

```
keytool -list -v -keystore jre/lib/security/cacerts
```

The password for this keystore is changeit . One of the certificates in this keystore is

Code View:
```
Owner: OU=VeriSign Trust Network, OU="(c) 1998 VeriSign, Inc. - For authorized use only",
OU=Class 1 Public Primary Certification Authority - G2, O="VeriSign, Inc.", C=US
Issuer: OU=VeriSign Trust Network, OU="(c) 1998 VeriSign, Inc. - For authorized
use only", OU=Class 1 Public Primary Certification Authority - G2, O="VeriSign, Inc.",
C=US
Serial number: 4cc7eaaa983e71d39310f83d3a899192
Valid from: Sun May 17 17:00:00 PDT 1998 until: Tue Aug 01 16:59:59 PDT 2028
Certificate fingerprints:
         MD5:  DB:23:3D:F9:69:FA:4B:B9:95:80:44:73:5E:7D:41:83
```

```
SHA1: 27:3E:E1:24:57:FD:C4:F9:0C:55:E8:2B:56:16:7F:62:F5:32:E5:47
```

You can check that your certificate is valid by visiting the web site http://www.verisign.com/repository/root.html .

Once Bob trusts the certificate, he can import it into his keystore.

```
keytool -importcert -keystore bob.certs -alias alice -file alice.cer
```

UNREGISTERED VERSION OF CHM TO PDF CONVERTER PRO BY THETA-SOFTWARE

Caution

Never import into a keystore a certificate that you don't fully trust. Once a certificate is added to the keystore, any program that uses the keystore assumes that the certificate can be used to verify signatures.

Now Alice can start sending signed documents to Bob. The `jarsigner` tool signs and verifies JAR files. Alice simply the document to be signed into a JAR file.

```
jar cvf document.jar document.txt
```

Then she uses the `jarsigner` tool to add the signature to the file. She needs to specify the keystore, the JAR file, alias of the key to use.

```
jarsigner -keystore alice.certs document.jar alice
```

When Bob receives the file, he uses the `-verify` option of the `jarsigner` program.

```
jarsigner -verify -keystore bob.certs document.jar
```

Bob does not need to specify the key alias. The `jarsigner` program finds the X.500 name of the key owner in the signature and looks for matching certificates in the keystore.

If the JAR file is not corrupted and the signature matches, then the `jarsigner` program prints

```
jar verified.
```

Otherwise, the program displays an error message.

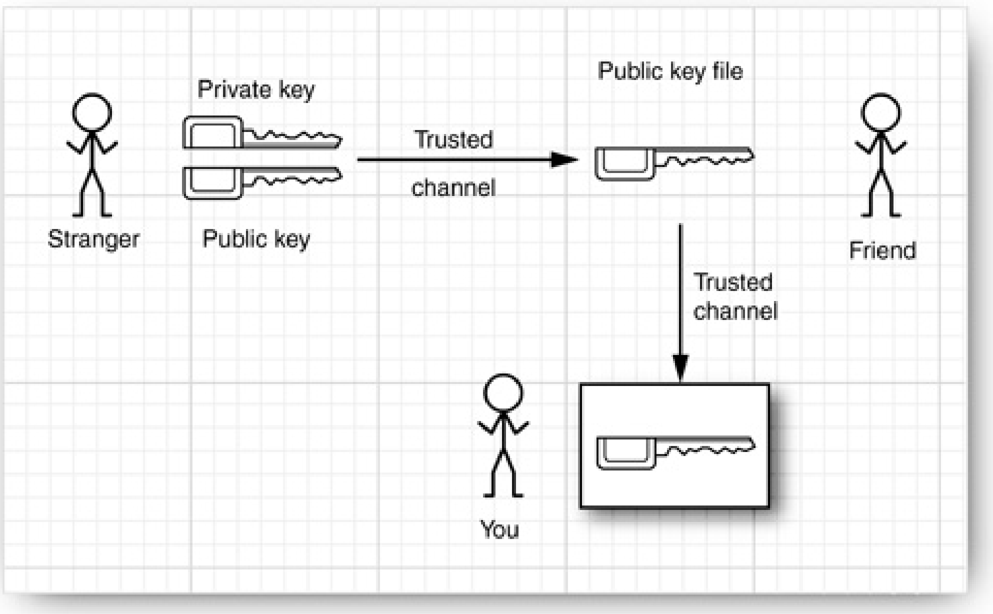
The Authentication Problem

Suppose you get a message from your friend Alice, signed with her private key, using the method we just showed might already have her public key, or you can easily get it by asking her for a copy or by getting it from her web p Then, you can verify that the message was in fact authored by Alice and has not been tampered with. Now, suppo get a message from a stranger who claims to represent a famous software company, urging you to run the progra attached to the message. The stranger even sends you a copy of his public key so you can verify that he authored message. You check that the signature is valid. This proves that the message was signed with the matching privat and that it has not been corrupted.

Be careful: *You still have no idea who wrote the message.* Anyone could have generated a pair of public and privat signed the message with the private key, and sent the signed message and the public key to you. The problem of determining the identity of the sender is called the *authentication problem.*

The usual way to solve the authentication problem is simple. Suppose the stranger and you have a common acqua you both trust. Suppose the stranger meets your acquaintance in person and hands over a disk with the public key acquaintance later meets you, assures you that he met the stranger and that the stranger indeed works for the fa software company, and then gives you the disk (see Figure 9-14). That way, your acquaintance vouches for the authenticity of the stranger.

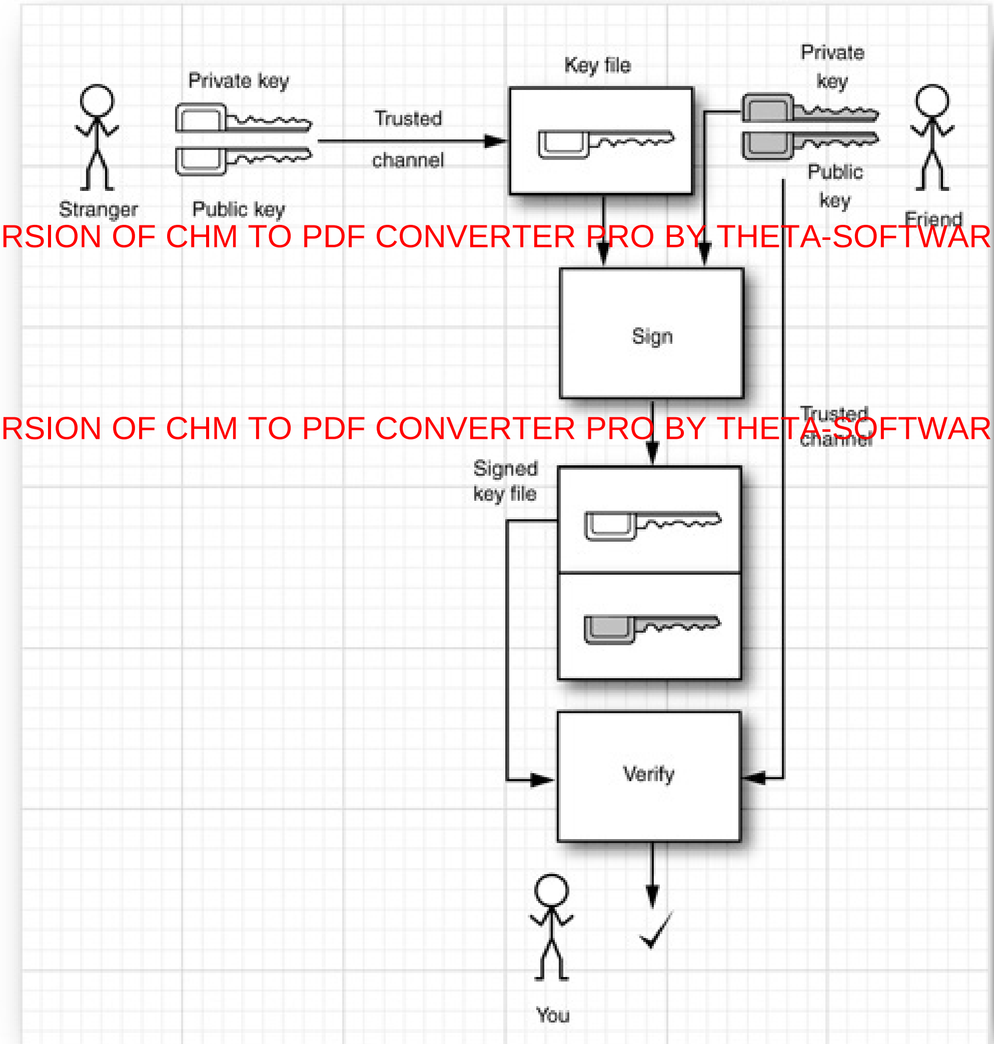
Figure 9-14. Authentication through a trusted intermediary



In fact, your acquaintance does not actually need to meet you. Instead, he can use his private key to sign the stra public key file (see Figure 9-15).

Figure 9-15. Authentication through a trusted intermediary's signature

[View full size image]



UNREGISTERED VERSION OF CHM TO PDF CONVERTER PRO BY THETA-SOFTWARE

When you get the public key file, you verify the signature of your friend, and because you trust him, you are confi he did check the stranger's credentials before applying his signature.

However, you might not have a common acquaintance. Some trust models assume that there is always a "chain o trust"—a chain of mutual acquaintances—so that you trust every member of that chain. In practice, of course, tha always true. You might trust your friend, Alice, and you know that Alice trusts Bob, but you don't know Bob and ar that you trust him. Other trust models assume that there is a benevolent big brother in whom we all trust. The be of these companies is VeriSign, Inc. (http://www.verisign.com).

You will often encounter digital signatures that are signed by one or more entities who will vouch for the authentic you will need to evaluate to what degree you trust the authenticators. You might place a great deal of trust in Veri perhaps because you saw their logo on many web pages or because you heard that they require multiple people w attach cases to come together into a secure chamber whenever new master keys are to be minted.

However, you should have realistic expectations about what is actually being authenticated. The CEO of VeriSign d personally meet every individual or company representative when authenticating a public key. You can get a "clas simply by filling out a web form and paying a small fee. The key is mailed to the e-mail address included in the ce Thus, you can be reasonably assured that the e-mail address is genuine, but the requestor could have filled in *any* and organization. There are more stringent classes of IDs. For example, with a "class 3" ID, VeriSign will require a individual requestor to appear before a notary public, and it will check the financial rating of a corporate requestor authenticators will have different procedures. Thus, when you receive an authenticated message, it is important th understand what, in fact, is being authenticated.

Certificate Signing

In the section "Verifying a Signature " on page 814 , you saw how Alice used a selfsigned certificate to distribute a key to Bob. However, Bob needed to ensure that the certificate was valid by verifying the fingerprint with Alice.

Suppose Alice wants to send her colleague Cindy a signed message, but Cindy doesn't want to bother with verifyin signature fingerprints. Now suppose that there is an entity that Cindy trusts to verify signatures. In this example, trusts the Information Resources Department at ACME Software.

That department operates a *certificate authority* (CA). Everyone at ACME has the CA's public key in their keystore installed by a system administrator who carefully checked the key fingerprint. The CA signs the keys of ACME emp When they install each other's keys, then the keystore will trust them implicitly because they are signed by a trust

Here is how you can simulate this process. Create a keystore `acmesoft.certs` . Generate a key par and export the key:

Code View:
```
keytool -genkeypair -keystore acmesoft.certs -alias acmeroot
keytool -exportcert -keystore acmesoft.certs -alias acmeroot -file acmeroot.cer
```

The public key is exported into a "self-signed" certificate. Then add it to every employee's keystore.

Code View:
```
keytool -importcert -keystore cindy.certs -alias acmeroot -file acmeroot.cer
```

For Alice to send messages to Cindy and to everyone else at ACME Software, she needs to bring her certificate to t Information Resources Department and have it signed. Unfortunately, this functionality is missing in the `keytool` p In the book's companion code, we supply a `CertificateSigner` class to fill the gap. An authorized staff member a Software would verify Alice's identity and generate a signed certificate as follows:

```
java CertificateSigner -keystore acmesoft.certs -alias acmeroot
   -infile alice.cer -outfile alice_signedby_acmeroot.cer
```

The certificate signer program must have access to the ACME Software keystore, and the staff member must know keystore password. Clearly, this is a sensitive operation.

Alice gives the file `alice_signedby_acmeroot.cer` file to Cindy and to anyone else in ACME Software. Alternativel Software can simply store the file in a company directory. Remember, this file contains Alice's public key and an as by ACME Software that this key really belongs to Alice.

Now Cindy imports the signed certificate into her keystore:

Code View:
```
keytool -importcert -keystore cindy.certs -alias alice -file alice_signedby_acmeroot.cer
```

The keystore verifies that the key was signed by a trusted root key that is already present in the keystore. Cindy i asked to verify the certificate fingerprint.

Once Cindy has added the root certificate and the certificates of the people who regularly send her documents, she has to worry about the keystore again.

Certificate Requests

In the preceding section, we simulated a CA with a keystore and the `CertificateSigner` tool. However, most CAs more sophisticated software to manage certificates, and they use slightly different formats for certificates. This sec shows the added steps that are required to interact with those software packages.

We will use the OpenSSL software package as an example. The software is preinstalled for many Linux systems ar OS X, and a Cygwin port is also available. Alternatively, you can download the software at http://www.openssl.org

To create a CA, run the `CA` script. The exact location depends on your operating system. On Ubuntu, run

```
/usr/lib/ssl/misc/CA.pl -newca
```

This script creates a subdirectory called `demoCA` in the current directory. The directory contains a root key pair and for certificates and certificate revocation lists.

You will want to import the public key into the Java keystore of all employees, but it is in the Privacy Enhanced Ma format, not the DER format that the keystore accepts easily. Copy the file `demoCA/cacert.pem` to a file `acmeroot.` open that file in a text editor. Remove everything before the line

```
-----BEGIN CERTIFICATE-----
```

and after the line

```
-----END CERTIFICATE-----
```

Now you can import `acmeroot.pem` into each keystore in the usual way:

```
keytool -importcert -keystore cindy.certs -alias alice -file acmeroot.pem
```

It seems quite incredible that the keytool cannot carry out this editing operation itself.

To sign Alice's public key, you start by generating a *certificate request* that contains the certificate in the PEM form

```
keytool -certreq -keystore alice.store -alias alice -file alice.pem
```

To sign the certificate, run

```
openssl ca -in alice.pem -out alice_signedby_acmeroot.pem
```

As before, cut out everything outside the `BEGIN CERTIFICATE` /`END CERTIFICATE` markers from `alice_signedby_acmeroot.pem` . Then import it into the keystore:

Code View:
```
keytool -importcert -keystore cindy.certs -alias alice -file alice_signedby_acmeroot.pem
```

You use the same steps to have a certificate signed by a public certificate authority such as VeriSign.

Code Signing

One of the most important uses of authentication technology is signing executable programs. If you download a program, you are naturally concerned about damage that a program can do. For example, the program could have been infected by a virus. If you know where the code comes from *and* that it has not been tampered with since it left its origin, then your comfort level will be a lot higher than without this knowledge. In fact, if the program was also written in the Java programming language, you can then use this information to make a rational decision about what privileges you will allow that program to have. You might want it to run just in a sandbox as a regular applet, or you might want to grant it a different set of rights and restrictions. For example, if you download a word processing program, you might want to grant it access to your printer and to files in a certain subdirectory. However, you might not want to give it the right to make network connections, so that the program can't try to send your files to a third party without your knowledge.

You now know how to implement this sophisticated scheme.

1. Use authentication to verify where the code came from.

2. Run the code with a security policy that enforces the permissions that you want to grant the program, depending on its origin.

JAR File Signing

In this section, we show you how to sign applets and web start applications for use with the Java Plug-in software. There are two scenarios:

- Delivery in an intranet.

- Delivery over the public Internet.

In the first scenario, a system administrator installs policy files and certificates on local machines. Whenever the Java Plug-in tool loads signed code, it consults the policy file for the permissions and the keystore for signatures. Installing the policies and certificates is straightforward and can be done once per desktop. End users can then run signed corporate code outside the sandbox. Whenever a new program is created or an existing one is updated, it must be signed and deployed on the web server. However, no desktops need to be touched as the programs evolve. We think this is a reasonable scenario that can be an attractive alternative to deploying corporate applications on every desktop.

In the second scenario, software vendors obtain certificates that are signed by CAs such as VeriSign. When an end user visits a web site that contains a signed applet, a pop-up dialog box identifies the software vendor and gives the end user two choices: to run the applet with full privileges or to confine it to the sandbox. We discuss this less desirable scenario in detail in the section "Software Developer Certificates" on page 827.

For the remainder of this section, we describe how you can build policy files that grant specific permissions to code from known sources. Building and deploying these policy files is not for casual end users. However, system administrators can carry out these tasks in preparation for distributing intranet programs.

Suppose ACME Software wants its users to run certain programs that require local file access, and it wants to deploy the programs through a browser, as applets or Web Start applications. Because these programs cannot run inside the sandbox, ACME Software needs to install policy files on employee machines.

As you saw earlier in this chapter, ACME could identify the programs by their code base. But that means that ACME would need to update the policy files each time the programs are moved to a different web server. Instead, ACME decides to *sign* the JAR files that contain the program code.

First, ACME generates a root certificate:

```
keytool -genkeypair -keystore acmesoft.certs -alias acmeroot
```

Of course, the keystore containing the private root key must be kept at a safe place. Therefore, we create a second keystore `client.certs` for the public certificates and add the public `acmeroot` certificate into it.

Code View:

```
keytool -exportcert -keystore acmesoft.certs -alias acmeroot -file acmeroot.cer
keytool -importcert -keystore client.certs -alias acmeroot -file acmeroot.cer
```

To make a signed JAR file, programmers add their class files to a JAR file in the usual way. For example,

```
javac FileReadApplet.java
jar cvf FileReadApplet.jar *.class
```

Then a trusted person at ACME runs the `jarsigner` tool, specifying the JAR file and the alias of the private key:

```
jarsigner -keystore acmesoft.certs FileReadApplet.jar acmeroot
```

The signed applet is now ready to be deployed on a web server.

Next, let us turn to the client machine configuration. A policy file must be distributed to each client machine.

To reference a keystore, a policy file starts with the line

```
keystore "keystoreURL", "keystoreType";
```

The URL can be absolute or relative. Relative URLs are relative to the location of the policy file. The type is `JKS` if the keystore was generated by `keytool`. For example,

```
keystore "client.certs", "JKS";
```

Then `grant` clauses can have suffixes `signedBy "`*alias*`"`, such as this one:

```
grant signedBy "acmeroot"
{
   . . .
};
```

Any signed code that can be verified with the public key associated with the alias is now granted the permissions inside the `grant` clause.

You can try out the code signing process with the applet in Listing 9-16. The applet tries to read from a local file. The default security policy only lets the applet read files from its code base and any subdirectories. Use `appletviewer` to run the applet and verify that you can view files from the code base directory, but not from other directories.

Now create a policy file `applet.policy` with the contents:


```
keystore "client_certs", "JKS";
grant signedBy "acmeroot"
{
   permission java.lang.RuntimePermission "usePolicy";
   permission java.io.FilePermission "/etc/*", "read";
};
```


The `usePolicy` permission overrides the default "all or nothing" permission for signed applets. Here, we say that any applets signed by `acmeroot` are allowed to read files in the `/etc` directory. (Windows users: Substitute another directory such as `C:\Windows`.)

Tell the applet viewer to use the policy file:

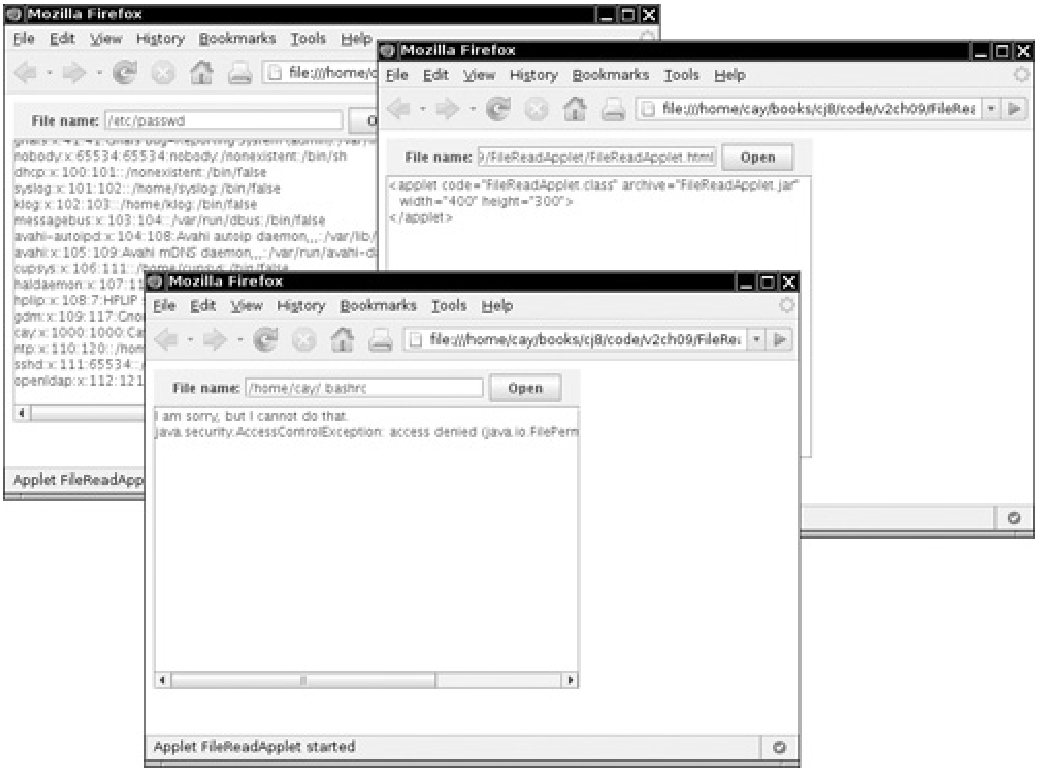
```
appletviewer -J-Djava.security.policy=applet.policy FileReadApplet.html
```

Now the applet can read files from the `/etc` directory, thus demonstrating that the signing mechanism works.

As a final test, you can run your applet inside the browser (see Figure 9-16). You need to copy the permission file and keystore inside the Java deployment directory. If you run UNIX or Linux, that directory is the `.java/deployment` subdirectory of your home directory. In Windows Vista, it is the `C:\Users\`*yourLoginName*`\AppData\Sun\Java\Deployment` directory. In the following, we refer to that directory as *deploydir*.

Figure 9-16. A signed applet can read local files

[View full size image]



Copy `applet.policy` and `client.certs` to the *deploydir*`/security` directory. In that directory, rename `applets.policy` to `java.policy`. (Double-check that you are not wiping out an existing `java.policy` file. If there is one, add the `applet.policy` contents to it.)

Tip

For more details on configuring client Java security, read the sections "Deployment Configuration File and Properties" and "Java Control Panel" in the Java deployment guide at http://java.sun.com/javase/6/docs/technotes/guides/deployment/deployment-guide/overview.html.

Restart your browser and load the `FileReadApplet.html`. You should *not* be prompted to accept any certificate. Check that you can load any file in the `/etc` directory and the directory from which the applet was loaded, but not from other directories.

When you are done, remember to clean up your *deploydir*`/security` directory. Remove the files `java.policy` and `client.certs`. Restart your browser. If you load the applet again after cleaning up, you should no longer be able to read files from the local file system. Instead, you will be prompted for a certificate. We discuss security certificates in the next section.

Listing 9-16. **FileReadApplet.java**

Code View:

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



```

/**
 * This applet can run "outside the sandbox" and read local files when it is given the right
 * permissions.
 * @version 1.11 2007-10-06
 * @author Cay Horstmann
 */
```



```
public class FileReadApplet extends JApplet
{
   public void init()
   {
      EventQueue.invokeLater(new Runnable()
         {
            public void run()
            {
               fileNameField = new JTextField(20);
               JPanel panel = new JPanel();
               panel.add(new JLabel("File name:"));
               panel.add(fileNameField);
               JButton openButton = new JButton("Open");
               panel.add(openButton);
               ActionListener listener = new ActionListener()
               {
                  public void actionPerformed(ActionEvent event)
                  {
                     loadFile(fileNameField.getText());
                  }
               };
               fileNameField.addActionListener(listener);
               openButton.addActionListener(listener);

               add(panel, "North");

               fileText = new JTextArea();
               add(new JScrollPane(fileText), "Center");
            }
         });
   }

   /**
    * Loads the contents of a file into the text area.
    * @param filename the file name
    */
   public void loadFile(String filename)
   {
      try
      {
         fileText.setText("");
         Scanner in = new Scanner(new FileReader(filename));
         while (in.hasNextLine())
            fileText.append(in.nextLine() + "\n");
```

```
            in.close();
         }
         catch (IOException e)
         {
            fileText.append(e + "\n");
         }
         catch (SecurityException e)
         {
            fileText.append("I am sorry, but I cannot do that.\n");
            fileText.append(e + "\n");
         }
      }
      private JTextField fileNameField;
      private JTextArea fileText;
   }
```

Software Developer Certificates

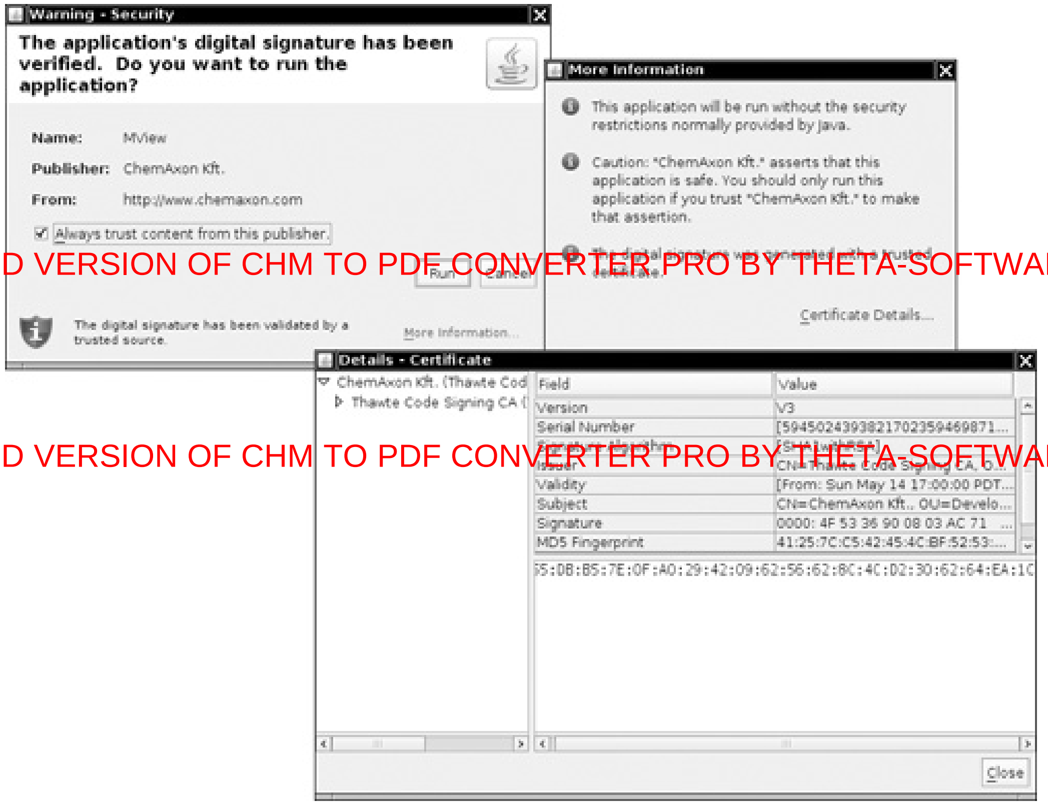
Up to now, we discussed scenarios in which programs are delivered in an intranet and for which a system administrator configures a security policy that controls the privileges of the programs. However, that strategy only works with programs from known sources.

Suppose while surfing the Internet, you encounter a web site that offers to run an applet or web start application from an unfamiliar vendor, provided you grant it the permission to do so (see Figure 9-17). Such a program is signed with a *software developer* certificate that is issued by a CA. The pop-up dialog box identifies the software developer and the certificate issuer. You now have two choices:

- Run the program with full privileges.
- Confine the program to the sandbox. (The Cancel button in the dialog box is misleading. If you click that button, the applet is not canceled. Instead, it runs in the sandbox.)

Figure 9-17. Launching a signed applet

[View full size image]



What facts do you have at your disposal that might influence your decision? Here is what you know:

- Thawte sold a certificate to the software developer.
- The program really was signed with that certificate, and it hasn't been modified in transit.
- The certificate really was signed by Thawte—it was verified by the public key in the local `cacerts` file.

Does that tell you whether the code is safe to run? Do you trust the vendor if all you know is the vendor name and the fact that Thawte sold them a software developer certificate? Presumably Thawte went to some degree of trouble to assure itself that ChemAxon Kft. is not an outright cracker. However, no certificate issuer carries out a comprehensive audit of the honesty and competence of software vendors.

In the situation of an unknown vendor, an end user is ill-equipped to make an intelligent decision whether to let this program run outside the sandbox, with all permissions of a local application. If the vendor is a well-known company, then the user can at least take the past track record of the company into account.

Note

It is possible to use very weak certificates to sign code—see http://www.dallaway.com/acad/webstart for a sobering example. Some developers even instruct users to add untrusted certificates into their certificate store—for example, http://www.agsrhichome.bnl.gov/Controls/doc/javaws/javaws_howto.html. From a security standpoint, this seems very bad.

We don't like situations in which a program demands "give me all rights, or I won't run at all." Naive users are too often cowed into granting access that can put them in danger.

Would it help if each program explained what rights it needs and requested specific permission for those rights? Unfortunately, as you have seen, that can get pretty technical. It doesn't seem reasonable for an end user to have to ponder whether a program should really have the right to inspect the AWT event queue.

We remain unenthusiastic about software developer certificates. It would be better if applets and web start applications on the public Internet tried harder to stay within their respective sandboxes, and if those sandboxes were improved. The Web Start API that we discussed in Volume I, Chapter 10 is a step in the right direction.

Encryption

So far, we have discussed one important cryptographic technique that is implemented in the Java security API, namely, authentication through digital signatures. A second important aspect of security is *encryption*. When information is authenticated, the information itself is plainly visible. The digital signature merely verifies that the information has not been changed. In contrast, when information is encrypted, it is not visible. It can only be decrypted with a matching key.

Authentication is sufficient for code signing—there is no need for hiding the code. However, encryption is necessary when applets or applications transfer confidential information, such as credit card numbers and other personal data.

Until recently, patents and export controls have prevented many companies, including Sun, from offering strong encryption. Fortunately, export controls are now much less stringent, and the patent for an important algorithm has expired. As of Java SE 1.4, good encryption support has been part of the standard library.

Symmetric Ciphers

The Java cryptographic extensions contain a class `Cipher` that is the superclass for all encryption algorithms. You get a cipher object by calling the `getInstance` method:

```
Cipher cipher = Cipher.getInstance(algorithmName);
```

or

```
Cipher cipher = Cipher.getInstance(algorithmName, providerName);
```

The JDK comes with ciphers by the provider named `"SunJCE"`. It is the default provider that is used if you don't specify another provider name. You might want another provider if you need specialized algorithms that Sun does not support.

The algorithm name is a string such as `"AES"` or `"DES/CBC/PKCS5Padding"`.

The Data Encryption Standard (DES) is a venerable block cipher with a key length of 56 bits. Nowadays, the DES algorithm is considered obsolete because it can be cracked with brute force (see, for example, http://www.eff.org/Privacy/Crypto/Crypto_misc/DESCracker/). A far better alternative is its successor, the Advanced Encryption Standard (AES). See http://www.csrc.nist.gov/publications/fips/fips197/fips-197.pdf for a detailed description of the AES algorithm. We use AES for our example.

Once you have a cipher object, you initialize it by setting the mode and the key:

```
int mode = . . .;
Key key = . . .;
cipher.init(mode, key);
```

The mode is one of

```
Cipher.ENCRYPT_MODE
Cipher.DECRYPT_MODE
Cipher.WRAP_MODE
Cipher.UNWRAP_MODE
```

The wrap and unwrap modes encrypt one key with another—see the next section for an example.

Now you can repeatedly call the update method to encrypt blocks of data:

```
int blockSize = cipher.getBlockSize();
byte[] inBytes = new byte[blockSize];
. . . // read inBytes
int outputSize= cipher.getOutputSize(blockSize);
byte[] outBytes = new byte[outputSize];
int outLength = cipher.update(inBytes, 0, outputSize, outBytes);
. . . // write outBytes
```

When you are done, you must call the doFinal method once. If a final block of input data is available (with fewer than blockSize bytes), then call

```
outBytes = cipher.doFinal(inBytes, 0, inLength);
```

If all input data have been encrypted, instead call

```
outBytes = cipher.doFinal();
```

The call to doFinal is necessary to carry out *padding* of the final block. Consider the DES cipher. It has a block size of 8 bytes. Suppose the last block of the input data has fewer than 8 bytes. Of course, we can fill the remaining bytes with 0, to obtain one final block of 8 bytes, and encrypt it. But when the blocks are decrypted, the result will have several trailing 0 bytes appended to it, and therefore it will be slightly different from the original input file. That could be a problem, and, to avoid it, we need a *padding scheme*. A commonly used padding scheme is the one described in the Public Key Cryptography Standard (PKCS) #5 by RSA Security Inc. (ftp://ftp.rsasecurity.com/pub/pkcs/pkcs-5v2/pkcs5v2-0.pdf). In this scheme, the last block is not padded with a pad value of zero, but with a pad value that equals the number of pad bytes. In other words, if L is the last (incomplete) block, then it is padded as follows:

```
L 01                                if length(L) = 7
L 02 02                             if length(L) = 6
L 03 03 03                          if length(L) = 5
. . .
L 07 07 07 07 07 07 07              if length(L) = 1
```

Finally, if the length of the input is actually divisible by 8, then one block

```
08 08 08 08 08 08 08 08
```

is appended to the input and encrypted. For decryption, the very last byte of the plaintext is a count of the padding characters to discard.

Key Generation

To encrypt, you need to generate a key. Each cipher has a different format for keys, and you need to make sure that the key generation is random. Follow these steps:

1. Get a `KeyGenerator` for your algorithm.

2. Initialize the generator with a source for randomness. If the block length of the cipher is variable, also specify the desired block length.

3. Call the `generateKey` method.

For example, here is how you generate an AES key.

```
KeyGenerator keygen = KeyGenerator.getInstance("AES");
SecureRandom random = new SecureRandom(); // see below
keygen.init(random);
Key key = keygen.generateKey();
```

UNREGISTERED VERSION OF CHM TO PDF CONVERTER PRO BY THETA-SOFTWARE

Alternatively, you can produce a key from a fixed set of raw data (perhaps derived from a password or the timing of keystrokes). Then use a `SecretKeyFactory`, like this:

```
SecretKeyFactory keyFactory = SecretKeyFactory.getInstance("AES");
byte[] keyData = . . .; // 16 bytes for AES
SecretKeySpec keySpec = new SecretKeySpec(keyData, "AES");
Key key = keyFactory.generateSecret(keySpec);
```

When generating keys, make sure you use *truly random* numbers. For example, the regular random number generator in the `Random` class, seeded by the current date and time, is not random enough. Suppose the computer clock is accurate to 1/10 of a second. Then there are at most 864,000 seeds per day. If an attacker knows the day a key was issued (as can often be deduced from a message date or certificate expiration date), then it is an easy matter to generate all possible seeds for that day.

The `SecureRandom` class generates random numbers that are far more secure than those produced by the `Random` class. You still need to provide a seed to start the number sequence at a random spot. The best method for doing this is to obtain random input from a hardware device such as a white-noise generator. Another reasonable source for random input is to ask the user to type away aimlessly on the keyboard, but each keystroke should contribute only one or two bits to the random seed. Once you gather such random bits in an array of bytes, you pass it to the `setSeed` method.

```
SecureRandom secrand = new SecureRandom();
byte[] b = new byte[20];
// fill with truly random bits
secrand.setSeed(b);
```

If you don't seed the random number generator, then it will compute its own 20-byte seed by launching threads, putting them to sleep, and measuring the exact time when they are awakened.

Note

This algorithm is *not* known to be safe. In the past, algorithms that relied on timing other components of the computer, such as hard disk access time, were later shown not to be completely random.

The sample program at the end of this section puts the AES cipher to work (see Listing 9-17). To use the program, you first generate a secret key. Run

```
java AESTest -genkey secret.key
```

The secret key is saved in the file `secret.key`.

Now you can encrypt with the command

```
java AESTest -encrypt plaintextFile encryptedFile secret.key
```

Decrypt with the command

```
java AESTest -decrypt encryptedFile decryptedFile secret.key
```

The program is straightforward. The `-genkey` option produces a new secret key and serializes it in the given file. That operation takes a long time because the initialization of the secure random generator is time consuming. The `-encrypt` and `-decrypt` options both call into the same `crypt` method that calls the `update` and `doFinal` methods of the cipher. Note how the `update` method is called as long as the input blocks have the full length, and the `doFinal` method is either called with a partial input block (which is then padded) or with no additional data (to generate one pad block).

Listing 9-17. **AESTest.java**

Code View:

```
import java.io.*;
import java.security.*;
import javax.crypto.*;

/**
 * This program tests the AES cipher. Usage:<br>
 * java AESTest -genkey keyfile<br>
 * java AESTest -encrypt plaintext encrypted keyfile<br>
 * java AESTest -decrypt encrypted decrypted keyfile<br>
 * @author Cay Horstmann
 * @version 1.0 2004-09-14
 */
public class AESTest
{
   public static void main(String[] args)
   {
      try
      {
         if (args[0].equals("-genkey"))
         {
            KeyGenerator keygen = KeyGenerator.getInstance("AES");
            SecureRandom random = new SecureRandom();
            keygen.init(random);
            SecretKey key = keygen.generateKey();
            ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream(args[1]));
            out.writeObject(key);
```

```
            out.close();
         }
         else
         {
            int mode;
            if (args[0].equals("-encrypt")) mode = Cipher.ENCRYPT_MODE;
            else mode = Cipher.DECRYPT_MODE;

            ObjectInputStream keyIn = new ObjectInputStream(new FileInputStream(args[3]));
            Key key = (Key) keyIn.readObject();
            keyIn.close();

            InputStream in = new FileInputStream(args[1]);
            OutputStream out = new FileOutputStream(args[2]);
            Cipher cipher = Cipher.getInstance("AES");
            cipher.init(mode, key);

            crypt(in, out, cipher);
            in.close();
            out.close();
         }
      }
      catch (IOException e)
      {
         e.printStackTrace();
      }
      catch (GeneralSecurityException e)
      {
         e.printStackTrace();
      }
      catch (ClassNotFoundException e)
      {
         e.printStackTrace();
      }
   }

   /**
    * Uses a cipher to transform the bytes in an input stream and sends the transformed bytes
    * to an output stream.
    * @param in the input stream
    * @param out the output stream
    * @param cipher the cipher that transforms the bytes
    */
   public static void crypt(InputStream in, OutputStream out, Cipher cipher)
         throws IOException, GeneralSecurityException
   {
      int blockSize = cipher.getBlockSize();
      int outputSize = cipher.getOutputSize(blockSize);
      byte[] inBytes = new byte[blockSize];
      byte[] outBytes = new byte[outputSize];

      int inLength = 0;
      boolean more = true;
      while (more)
      {
         inLength = in.read(inBytes);
         if (inLength == blockSize)
         {
            int outLength = cipher.update(inBytes, 0, blockSize, outBytes);
```

```
            out.write(outBytes, 0, outLength);
         }
         else more = false;
      }
      if (inLength > 0) outBytes = cipher.doFinal(inBytes, 0, inLength);
      else outBytes = cipher.doFinal();
      out.write(outBytes);
   }
}
```

API

javax.crypto.Cipher 1.4

- `static Cipher getInstance(String algorithmName)`

- `static Cipher getInstance(String algorithmName, String providerName)`

  returns a `Cipher` object that implements the specified algorithm. Throws a `NoSuchAlgorithmException` if the algorithm is not provided.

- `int getBlockSize()`

  returns the size (in bytes) of a cipher block, or 0 if the cipher is not a block cipher.

- `int getOutputSize(int inputLength)`

  returns the size of an output buffer that is needed if the next input has the given number of bytes. This method takes into account any buffered bytes in the cipher object.

- `void init(int mode, Key key)`

  initializes the cipher algorithm object. The mode is one of `ENCRYPT_MODE`, `DECRYPT_MODE`, `WRAP_MODE`, or `UNWRAP_MODE`.

- `byte[] update(byte[] in)`

- `byte[] update(byte[] in, int offset, int length)`

- `int update(byte[] in, int offset, int length, byte[] out)` transforms one block of input data. The first two methods return the output. The third method returns the number of bytes placed into `out`.

- `byte[] doFinal()`

- `byte[] doFinal(byte[] in)`
- `byte[] doFinal(byte[] in, int offset, int length)`

- `int doFinal(byte[] in, int offset, int length, byte[] out)`

  transforms the last block of input data and flushes the buffer of this algorithm object. The first three methods return the output. The fourth method returns the number of bytes placed into `out`.

API

`javax.crypto.KeyGenerator` 1.4

- `static KeyGenerator getInstance(String algorithmName)`

  returns a `KeyGenerator` object that implements the specified algorithm. Throws a `NoSuchAlgorithmException` if the algorithm is not provided.

- `void init(SecureRandom random)`
- `void init(int keySize, SecureRandom random)`

  initializes the key generator.

- `SecretKey generateKey()`

  generates a new key.

javax.crypto.SecretKeyFactory 1.4

- `static SecretKeyFactory getInstance(String algorithmName)`
- `static SecretKeyFactory getInstance(String algorithmName, String providerName)`

  returns a `SecretKeyFactory` object for the specified algorithm.

- `SecretKey generateSecret(KeySpec spec)`

  generates a new secret key from the given specification.

javax.crypto.spec.SecretKeySpec 1.4

- `SecretKeySpec(byte[] key, String algorithmName)`

  constructs a key specification.

Cipher Streams

The JCE library provides a convenient set of stream classes that automatically encrypt or decrypt stream data. For example, here is how you can encrypt data to a file:

Code View:
```
Cipher cipher = . . .;
cipher.init(Cipher.ENCRYPT_MODE, key);
CipherOutputStream out = new CipherOutputStream(new FileOutputStream(outputFileName), cipher);
byte[] bytes = new byte[BLOCKSIZE];
int inLength = getData(bytes); // get data from data source
while (inLength != -1)
{
   out.write(bytes, 0, inLength);
   inLength = getData(bytes); // get more data from data source
}
out.flush();
```

Similarly, you can use a `CipherInputStream` to read and decrypt data from a file:

Code View:
```
Cipher cipher = . . .;
```

```
cipher.init(Cipher.DECRYPT_MODE, key);
CipherInputStream in = new CipherInputStream(new FileInputStream(inputFileName), cipher);
byte[] bytes = new byte[BLOCKSIZE];
int inLength = in.read(bytes);
while (inLength != -1)
{
   putData(bytes, inLength); // put data to destination
   inLength = in.read(bytes);
}
```


The cipher stream classes transparently handle the calls to `update` and `doFinal`, which is clearly a convenience.

API

`javax.crypto.CipherInputStream` 1.4

- `CipherInputStream(InputStream in, Cipher cipher)`

  constructs an input stream that reads data from `in` and decrypts or encrypts them by using the given cipher.

- `int read()`

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

  reads data from the input stream, which is automatically decrypted or encrypted.

javax.crypto.CipherOutputStream 1.4

- `CipherOutputStream(OutputStream out, Cipher cipher)`

  constructs an output stream that writes data to `out` and encrypts or decrypts them using the given cipher.

- `void write(int ch)`

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

  writes data to the output stream, which is automatically encrypted or decrypted.

- `void flush()`

  flushes the cipher buffer and carries out padding if necessary.

Public Key Ciphers

The AES cipher that you have seen in the preceding section is a *symmetric* cipher. The same key is used for encryption and for decryption. The Achilles heel of symmetric ciphers is key distribution. If Alice sends Bob an encrypted method, then Bob needs the same key that Alice used. If Alice changes the key, then she needs to send Bob both the message and, through a secure channel, the new key. But perhaps she has no secure channel to Bob, which is why she encrypts her messages to him in the first place.

Public key cryptography solves that problem. In a public key cipher, Bob has a key pair consisting of a public key and a matching private key. Bob can publish the public key anywhere, but he must closely guard the private key. Alice simply uses the public key to encrypt her messages to Bob.

Actually, it's not quite that simple. All known public key algorithms are *much* slower than symmetric key algorithms such as DES or AES. It would not be practical to use a public key algorithm to encrypt large amounts of information. However, that problem can easily be overcome by combining a public key cipher with a fast symmetric cipher, like this:

1. Alice generates a random symmetric encryption key. She uses it to encrypt her plaintext.
2. Alice encrypts the symmetric key with Bob's public key.
3. Alice sends Bob both the encrypted symmetric key and the encrypted plaintext.
4. Bob uses his private key to decrypt the symmetric key.
5. Bob uses the decrypted symmetric key to decrypt the message.

Nobody but Bob can decrypt the symmetric key because only Bob has the private key for decryption. Thus, the expensive public key encryption is only applied to a small amount of key data.

The most commonly used public key algorithm is the RSA algorithm invented by Rivest, Shamir, and Adleman.

Until October 2000, the algorithm was protected by a patent assigned to RSA Security Inc. Licenses were not cheap—typically a 3% royalty, with a minimum payment of $50,000 per year. Now the algorithm is in the public domain. The RSA algorithm is supported in Java SE 5.0 and above.

Note

If you still use an older version of the JDK, check out the Legion of Bouncy Castle (http://www.bouncycastle.org). It supplies a cryptography provider that includes RSA as well as a number of algorithms that are not part of the SunJCE provider. The Legion of Bouncy Castle provider has been signed by Sun Microsystems so that you can combine it with the JDK.

To use the RSA algorithm, you need a public/private key pair. You use a `KeyPairGenerator` like this:

```
KeyPairGenerator pairgen = KeyPairGenerator.getInstance("RSA");
SecureRandom random = new SecureRandom();
pairgen.initialize(KEYSIZE, random);
KeyPair keyPair = pairgen.generateKeyPair();
Key publicKey = keyPair.getPublic();
Key privateKey = keyPair.getPrivate();
```

The program in Listing 9-18 has three options. The `-genkey` option produces a key pair. The `-encrypt` option generates an AES key and *wraps* it with the public key.

```
Key key = . . .; // an AES key
Key publicKey = . . .; // a public RSA key
Cipher cipher = Cipher.getInstance("RSA");
cipher.init(Cipher.WRAP_MODE, publicKey);
byte[] wrappedKey = cipher.wrap(key);
```

It then produces a file that contains

- The length of the wrapped key.
- The wrapped key bytes.
- The plaintext encrypted with the AES key.

The `-decrypt` option decrypts such a file. To try the program, first generate the RSA keys:

```
java RSATest -genkey public.key private.key
```

Then encrypt a file:

```
java RSATest -encrypt plaintextFile encryptedFile public.key
```

Finally, decrypt it and verify that the decrypted file matches the plaintext:

```
java RSATest -decrypt encryptedFile decryptedFile private.key
```

Listing 9-18. **RSATest.java**

Code View:

```
import java.io.*;
import java.security.*;
import javax.crypto.*;

/**
 * This program tests the RSA cipher. Usage:<br>
 * java RSATest -genkey public private<br>
 * java RSATest -encrypt plaintext encrypted public<br>
 * java RSATest -decrypt encrypted decrypted private<br>
 * @author Cay Horstmann
 * @version 1.0 2004-09-14
 */
public class RSATest
{
   public static void main(String[] args)
   {
      try
      {
         if (args[0].equals("-genkey"))
         {
             KeyPairGenerator pairgen = KeyPairGenerator.getInstance("RSA");
             SecureRandom random = new SecureRandom();
             pairgen.initialize(KEYSIZE, random);
             KeyPair keyPair = pairgen.generateKeyPair();
             ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream(args[1]));
             out.writeObject(keyPair.getPublic());
             out.close();
             out = new ObjectOutputStream(new FileOutputStream(args[2]));
             out.writeObject(keyPair.getPrivate());
             out.close();
         }
         else if (args[0].equals("-encrypt"))
         {
            KeyGenerator keygen = KeyGenerator.getInstance("AES");
            SecureRandom random = new SecureRandom();
            keygen.init(random);
            SecretKey key = keygen.generateKey();

            // wrap with RSA public key
            ObjectInputStream keyIn = new ObjectInputStream(new FileInputStream(args[3]));
            Key publicKey = (Key) keyIn.readObject();
            keyIn.close();

            Cipher cipher = Cipher.getInstance("RSA");
            cipher.init(Cipher.WRAP_MODE, publicKey);
            byte[] wrappedKey = cipher.wrap(key);
            DataOutputStream out = new DataOutputStream(new FileOutputStream(args[2]));
            out.writeInt(wrappedKey.length);
```

```
                out.write(wrappedKey);

                InputStream in = new FileInputStream(args[1]);
                cipher = Cipher.getInstance("AES");
                cipher.init(Cipher.ENCRYPT_MODE, key);
                crypt(in, out, cipher);
                in.close();
                out.close();
            }
```

```
            else
            {
                DataInputStream in = new DataInputStream(new FileInputStream(args[1]));
                int length = in.readInt();
                byte[] wrappedKey = new byte[length];
                in.read(wrappedKey, 0, length);
```

```

                // unwrap with RSA private key
                ObjectInputStream keyIn = new ObjectInputStream(new FileInputStream(args[3]));
                Key privateKey = (Key) keyIn.readObject();
                keyIn.close();

                Cipher cipher = Cipher.getInstance("RSA");
                cipher.init(Cipher.UNWRAP_MODE, privateKey);
                Key key = cipher.unwrap(wrappedKey, "AES", Cipher.SECRET_KEY);

                OutputStream out = new FileOutputStream(args[2]);
                cipher = Cipher.getInstance("AES");
                cipher.init(Cipher.DECRYPT_MODE, key);

                crypt(in, out, cipher);
                in.close();
                out.close();
            }
        }
        catch (IOException e)
        {
            e.printStackTrace();
        }
        catch (GeneralSecurityException e)
        {
            e.printStackTrace();
        }
        catch (ClassNotFoundException e)
        {
            e.printStackTrace();
        }
    }

    /**
     * Uses a cipher to transform the bytes in an input stream and sends the transformed bytes
     * to an output stream.
     * @param in the input stream
     * @param out the output stream
     * @param cipher the cipher that transforms the bytes
     */
    public static void crypt(InputStream in, OutputStream out, Cipher cipher)
            throws IOException, GeneralSecurityException
    {
        int blockSize = cipher.getBlockSize();
```

```
      int outputSize = cipher.getOutputSize(blockSize);
      byte[] inBytes = new byte[blockSize];
      byte[] outBytes = new byte[outputSize];

      int inLength = 0;
      ;
      boolean more = true;
      while (more)
      {
         inLength = in.read(inBytes);
         if (inLength == blockSize)
         {
            int outLength = cipher.update(inBytes, 0, blockSize, outBytes);
            out.write(outBytes, 0, outLength);
         }
         else more = false;
      }
      if (inLength > 0) outBytes = cipher.doFinal(inBytes, 0, inLength);
      else outBytes = cipher.doFinal();
      out.write(outBytes);
   }

   private static final int KEYSIZE = 512;
}
```

You have now seen how the Java security model allows the controlled execution of code, which is a unique and increasingly important aspect of the Java platform. You have also seen the services for authentication and encryption that the Java library provides. We did not cover a number of advanced and specialized issues, among them:

- The GSS-API for "generic security services" that provides support for the Kerberos protocol (and, in principle, other protocols for secure message exchange). There is a tutorial at http://java.sun.com/javase/6/docs/technotes/guides/security/jgss/tutorials/index.html.

- Support for the Simple Authentication and Security Layer (SASL), used by the Lightweight Directory Access Protocol (LDAP) and Internet Message Access Protocol (IMAP). If you need to implement SASL in your own application, look at http://java.sun.com/javase/6/docs/technotes/guides/security/sasl/sasl-refguide.html.

- Support for SSL. Using SSL over HTTP is transparent to application programmers; simply use URLs that start with `https`. If you want to add SSL to your own application, see the Java Secure Socket Extension (JSEE) reference at http://java.sun.com/javase/6/docs/technotes/guides/security/jsse/JSSERefGuide.html.

Now that we have completed our overview of Java security, we turn to distributed computing in Chapter 10.

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

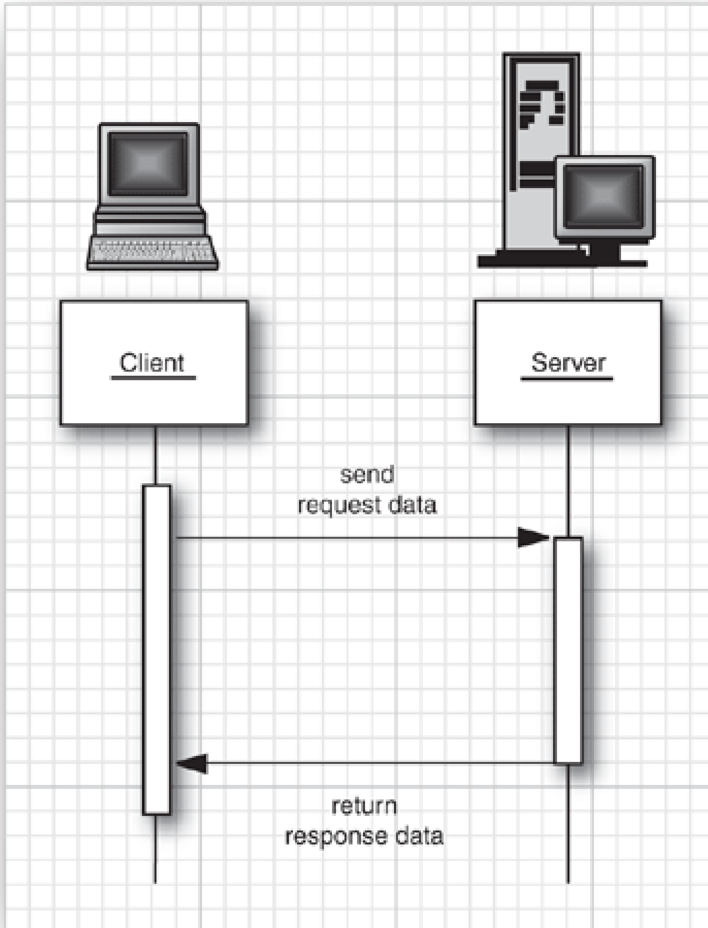
Periodically, the programming community starts thinking of "objects everywhere" as the solution to all its problems. The idea is to have a happy family of collaborating objects that can be located anywhere. When an object on one computer needs to invoke a method on an object on another computer, it sends a network message that contains the details of the request. The remote object computes a response, perhaps by accessing a database or by communicating with additional objects. Once the remote object has the answer to the client request, it sends the answer back over the network. Conceptuatlly, this process sounds quite simple, but you need to understand what goes on under the hood to use distributed objects effectively.

In this chapter, we focus on Java technologies for distributed programming, in particular the *Remote Method Invocation* (RMI) protocol for communicating between two Java virtual machines (which might run on different computers). We then briefly visit the JAX-WS technology for making remote calls to web services.

The Roles of Client and Server

The basic idea behind all distributed programming is simple. A client computer makes a request and sends the request data across a network to a server. The server processes the request and sends back a response for the client to analyze. Figure 10-1 shows the process.

Figure 10-1. Transmitting objects between client and server



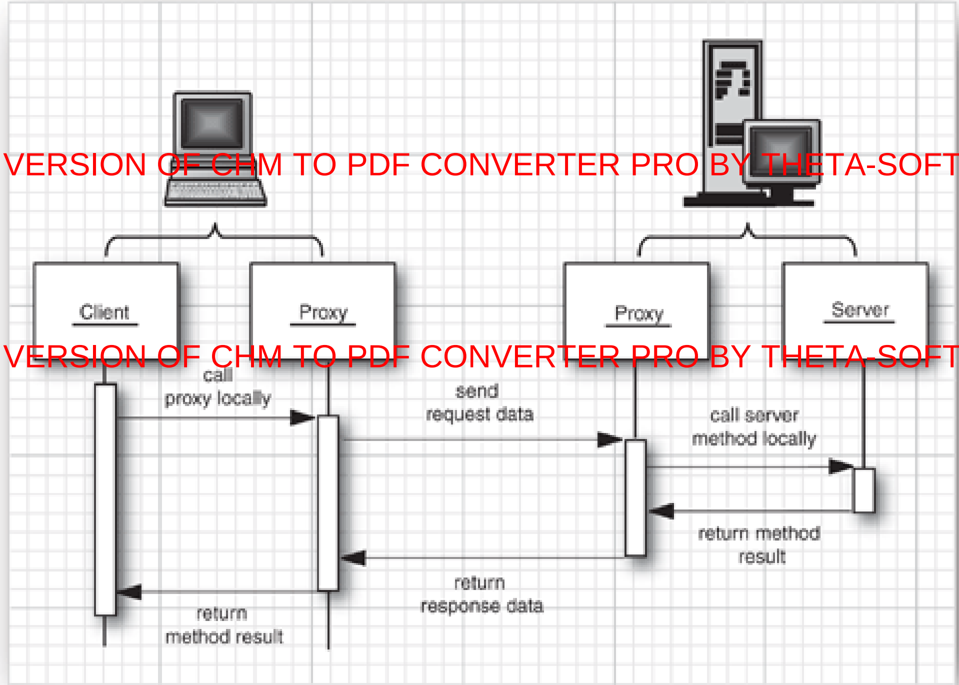
We would like to say at the outset that these requests and responses are *not* what you would see in a web application. The client is not a web browser. It can be any application that executes business rules of any complexity. The client application might or might not interact with a human user, and if it does, it can have a command-line or Swing user interface. The protocol for the request and response data allows the transfer of arbitrary objects, whereas traditional web applications are limited by using HTTP for the request and HTML for the response.

What we want is a mechanism by which the client programmer makes a regular method call, without worrying about sending data across the network or parsing the response. The solution is to install a *proxy* object on the client. The proxy is an object located in the client virtual machine that appears to the client program as if it was the remote object. The client calls the proxy, making a regular method call. The client proxy contacts the server, using a network protocol.

Similarly, the programmer who implements the service doesn't want to fuss with client communication. The solution is to install a second proxy object on the server. The server proxy communicates with the client proxy, and it makes regular method calls to the object implemeting the service (see Figure 10-2).

Figure 10-2. Remote method call with proxies

[View full size image]



How do the proxies communicate with each other? That depends on the implementation technology. There are three common choices:

- The Java RMI technology supports method calls between distributed Java objects.
- The Common Object Request Broker Architecture (CORBA) supports method calls between objects of any programming language. CORBA uses the binary Internet Inter-ORB Protocol, or IIOP, to communicate between objects.
- The web services architecture is a collection of protocols, sometimes collectively described as WS-*. It is also programming-language neutral. However, it uses XML-based communication formats. The format for transmitting objects is the Simple Object Access Protocol (SOAP).

If the communicating programs are implemented in Java code, then the full generality and complexity of CORBA or WS-* is not required. Sun developed a simple mechanism, called RMI, specifically for communication between Java applications.

It is well worth learning about RMI, even if you are not going to use it in your own programs. You will learn the mechanisms that are essential for programming distributed applications, using a straightforward architecture. Moreover, if you use enterprise Java technologies, it is very useful to have a basic understanding of RMI because that is the protocol used to communicate between enterprise Java beans (EJBs). EJBs are server-side components that are composed to make up complex applications that run on multiple servers. To make effective use of EJBs, you will want to have a good idea of the costs associated with remote calls.

Unlike RMI, CORBA and SOAP are completely language neutral. Client and server programs can be written in C, C++, C#, Java, or any other language. You supply an *interface description* to specify the signatures of the methods and the types of the data your objects can handle. These descriptions are formatted in a special language, called Interface Definition Language (IDL) for CORBA and Web Services Description Language (WSDL) for web services.

For many years, quite a few people believed that CORBA was the object model of the future. Frankly, though, CORBA has a reputation—sometimes deserved—for complex implementations and interoperability problems, and it has only reached modest success. We covered interoperability between Java and CORBA for five editions of this book, but dropped it for lack of interest. Our sentiments about CORBA are similar to those expressed by French president Charles De Gaulle about Brazil: It has a great future . . . and always will.

Web services had a similar amount of buzz when they first appeared, with the promise that they are simpler and, of course, founded in the goodness of the World Wide Web and XML. However, with the passing of time and the work of many committees, the protocol stack has become less simple, as it acquired more of the features that CORBA had all along. The XML protocol has the advantage of being (barely) human-readable, which helps with debugging. On the other hand, XML processing is a significant performance bottleneck. Recently, the WS-* stack has lost quite a bit of its luster and it too is gaining a reputation—sometimes deserved—for complex implementations and interoperability problems.

We close this chapter with an example of an application that consumes a web service. We have a look at the underlying protocol so that you can see how communication between different programming languages is implemented.

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

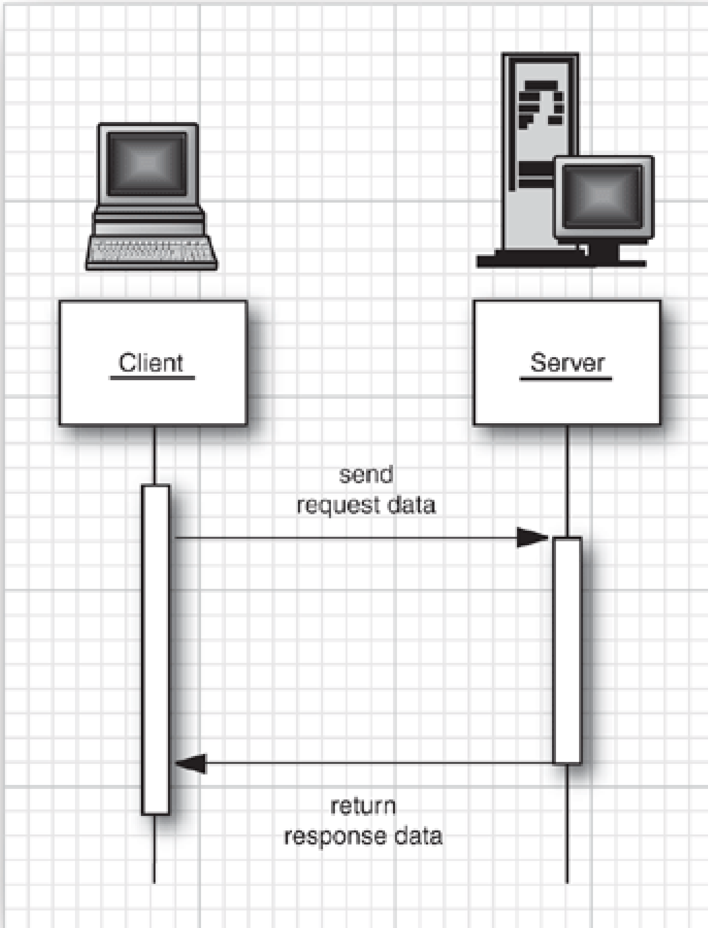
Periodically, the programming community starts thinking of "objects everywhere" as the solution to all its problems. The idea is to have a happy family of collaborating objects that can be located anywhere. When an object on one computer needs to invoke a method on an object on another computer, it sends a network message that contains the details of the request. The remote object computes a response, perhaps by accessing a database or by communicating with additional objects. Once the remote object has the answer to the client request, it sends the answer back over the network. Conceptuatlly, this process sounds quite simple, but you need to understand what goes on under the hood to use distributed objects effectively.

In this chapter, we focus on Java technologies for distributed programming, in particular the *Remote Method Invocation* (RMI) protocol for communicating between two Java virtual machines (which might run on different computers). We then briefly visit the JAX-WS technology for making remote calls to web services.

The Roles of Client and Server

The basic idea behind all distributed programming is simple. A client computer makes a request and sends the request data across a network to a server. The server processes the request and sends back a response for the client to analyze. Figure 10-1 shows the process.

Figure 10-1. Transmitting objects between client and server



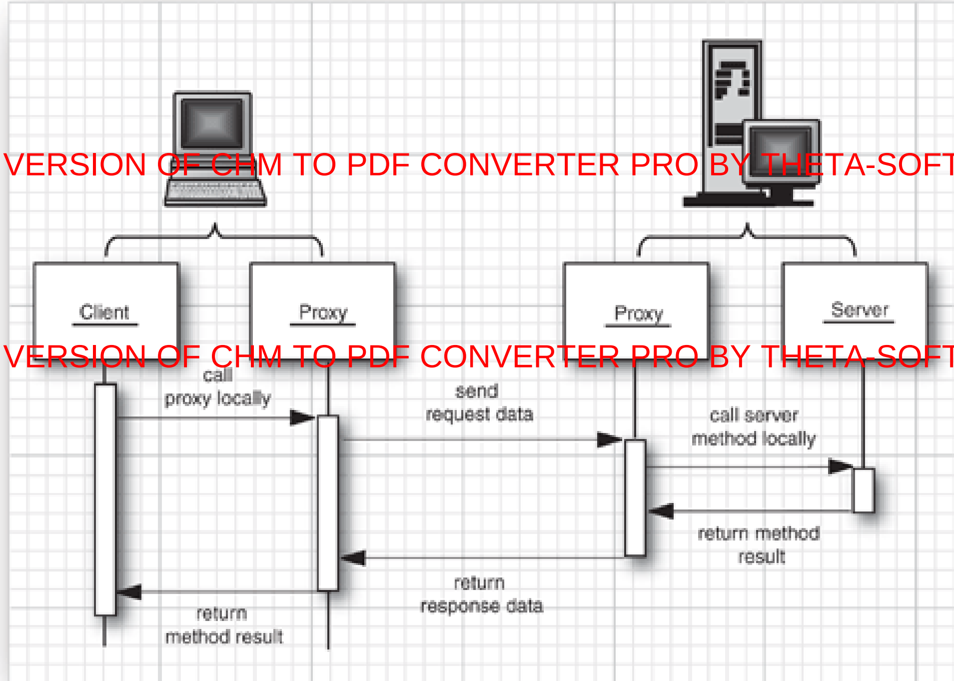
We would like to say at the outset that these requests and responses are *not* what you would see in a web application. The client is not a web browser. It can be any application that executes business rules of any complexity. The client application might or might not interact with a human user, and if it does, it can have a command-line or Swing user interface. The protocol for the request and response data allows the transfer of arbitrary objects, whereas traditional web applications are limited by using HTTP for the request and HTML for the response.

What we want is a mechanism by which the client programmer makes a regular method call, without worrying about sending data across the network or parsing the response. The solution is to install a *proxy* object on the client. The proxy is an object located in the client virtual machine that appears to the client program as if it was the remote object. The client calls the proxy, making a regular method call. The client proxy contacts the server, using a network protocol.

Similarly, the programmer who implements the service doesn't want to fuss with client communication. The solution is to install a second proxy object on the server. The server proxy communicates with the client proxy, and it makes regular method calls to the object implemeting the service (see Figure 10-2).

Figure 10-2. Remote method call with proxies

[View full size image]



How do the proxies communicate with each other? That depends on the implementation technology. There are three common choices:

- The Java RMI technology supports method calls between distributed Java objects.
- The Common Object Request Broker Architecture (CORBA) supports method calls between objects of any programming language. CORBA uses the binary Internet Inter-ORB Protocol, or IIOP, to communicate between objects.
- The web services architecture is a collection of protocols, sometimes collectively described as WS-*. It is also programming-language neutral. However, it uses XML-based communication formats. The format for transmitting objects is the Simple Object Access Protocol (SOAP).

If the communicating programs are implemented in Java code, then the full generality and complexity of CORBA or WS-* is not required. Sun developed a simple mechanism, called RMI, specifically for communication between Java applications.

It is well worth learning about RMI, even if you are not going to use it in your own programs. You will learn the mechanisms that are essential for programming distributed applications, using a straightforward architecture. Moreover, if you use enterprise Java technologies, it is very useful to have a basic understanding of RMI because that is the protocol used to communicate between enterprise Java beans (EJBs). EJBs are server-side components that are composed to make up complex applications that run on multiple servers. To make effective use of EJBs, you will want to have a good idea of the costs associated with remote calls.

Unlike RMI, CORBA and SOAP are completely language neutral. Client and server programs can be written in C, C++, C#, Java, or any other language. You supply an *interface description* to specify the signatures of the methods and the types of the data your objects can handle. These descriptions are formatted in a special language, called Interface Definition Language (IDL) for CORBA and Web Services Description Language (WSDL) for web services.

For many years, quite a few people believed that CORBA was the object model of the future. Frankly, though, CORBA has a reputation—sometimes deserved—for complex implementations and interoperability problems, and it has only reached modest success. We covered interoperability between Java and CORBA for five editions of this book, but dropped it for lack of interest. Our sentiments about CORBA are similar to those expressed by French president Charles De Gaulle about Brazil: It has a great future . . . and always will.

Web services had a similar amount of buzz when they first appeared, with the promise that they are simpler and, of course, founded in the goodness of the World Wide Web and XML. However, with the passing of time and the work of many committees, the protocol stack has become less simple, as it acquired more of the features that CORBA had all along. The XML protocol has the advantage of being (barely) human-readable, which helps with debugging. On the other hand, XML processing is a significant performance bottleneck. Recently, the WS-* stack has lost quite a bit of its luster and it too is gaining a reputation—sometimes deserved—for complex implementations and interoperability problems.

We close this chapter with an example of an application that consumes a web service. We have a look at the underlying protocol so that you can see how communication between different programming languages is implemented.

Remote Method Calls

The key to distributed computing is the *remote method call*. Some code on one machine (called the *client*) wants to invoke a method on an object on another machine (the *remote object*). To make this possible, the method parameters must somehow be shipped to the other machine, the server must be informed to locate the remote object and execute the method, and the return value must be shipped back.

Before looking at this process in detail, we want to point out that the client/server terminology applies only to a single method call. The computer that calls the remote method is the client for *that* call, and the computer hosting the object that processes the call is the server for *that* call. It is entirely possible that the roles are reversed somewhere down the road. The server of a previous call can itself become the client when it invokes a remote method on an object residing on another computer.

Stubs and Parameter Marshalling

When client code wants to invoke a method on a remote object, it actually calls an ordinary method on a proxy object called a *stub*. For example,

```
Warehouse centralWarehouse = get stub object;
double price = centralWarehouse.getPrice("Blackwell Toaster");
```

The stub resides on the client machine, not on the server. It knows how to contact the server over the network. The stub packages the parameters used in the remote method into a block of bytes. The process of encoding the parameters is called *parameter marshalling*. The purpose of parameter marshalling is to convert the parameters into a format suitable for transport from one virtual machine to another. In the RMI protocol, objects are encoded with the serialization mechanism that is described in Chapter 1. In the SOAP protocol, objects are encoded as XML.

To sum up, the stub method on the client builds an information block that consists of

- An identifier of the remote object to be used.
- A description of the method to be called.
- The parameters.

The stub then sends this information to the server. On the server side, a receiver object performs the following actions:

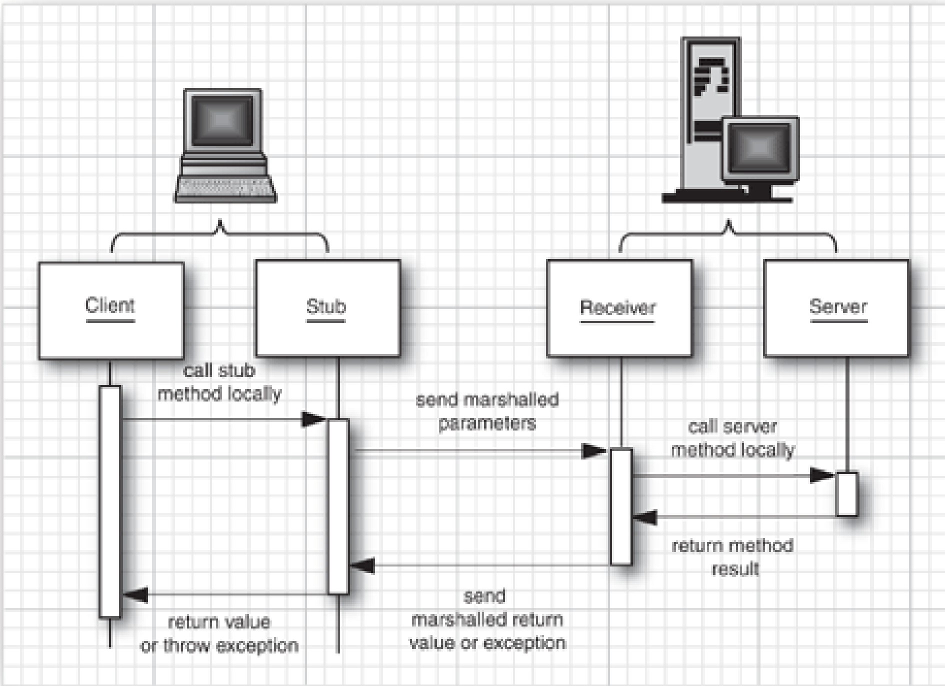
1. It locates the remote object to be called.
2. It calls the desired method, passing the supplied parameters.
3. It captures the return value or exception of the call.

4. It sends a package consisting of the marshalled return data back to the stub on the client.

The client stub unmarshals the return value or exception from the server. This value becomes the return value of the stub call. Or, if the remote method threw an exception, the stub rethrows it in the virtual machine of the caller. Figure 10-3 shows the information flow of a remote method invocation.

Figure 10-3. Parameter marshalling

[View full size image]



This process is obviously complex, but the good news is that it is completely automatic and, to a large extent, transparent for the programmer.

The details for implementing remote objects and for getting client stubs depend on the technology for distributed objects. In the following sections, we have a close look at RMI.

The RMI Programming Model

To introduce the RMI programming model, we start with a simple example. A remote object represents a warehouse. The client program asks the warehouse about the price of a product. In the following sections, you will see how to implement and launch the server and client programs.

Interfaces and Implementations

The capabilities of remote objects are expressed in interfaces that are shared between the client and server. For example, the interface in Listing 10-1 describes the service provided by a remote warehouse object:

Listing 10-1. **Warehouse.java**

```
import java.rmi.*;

/**
   The remote interface for a simple warehouse.
   @version 1.0 2007-10-09
   @author Cay Horstmann
*/
public interface Warehouse extends Remote
{
   double getPrice(String description) throws RemoteException;
}
```

Interfaces for remote objects must always extend the `Remote` interface defined in the `java.rmi` package. All the methods in those interfaces must also declare that they will throw a `RemoteException`. Remote method calls are inherently less reliable than local calls—it is always possible that a remote call will fail. For example, the server might be temporarily unavailable, or there might be a network problem. Your client code must be prepared to deal with these possibilities. For these reasons, you must handle the `RemoteException` with *every* remote method call and specify the appropriate action to take when the call does not succeed.

Next, on the server side, you must provide the class that actually carries out the work advertised in the remote interface—see Listing 10-2.

Listing 10-2. **WarehouseImpl.java**

Code View:
```
import java.rmi.*;
import java.rmi.server.*;
import java.util.*;

/**
 * This class is the implementation for the remote Warehouse interface.
 * @version 1.0 2007-10-09
 * @author Cay Horstmann
 */
public class WarehouseImpl extends UnicastRemoteObject implements Warehouse
{
   public WarehouseImpl() throws RemoteException
   {
      prices = new HashMap<String, Double>();
      prices.put("Blackwell Toaster", 24.95);
      prices.put("ZapXpress Microwave Oven", 49.95);
   }

   public double getPrice(String description) throws RemoteException
   {
      Double price = prices.get(description);
      return price == null ? 0 : price;
   }

   private Map<String, Double> prices;
}
```

Note

The `WarehouseImpl` constructor is declared to throw a `RemoteException` because the superclass constructor can throw that exception. This happens when there is a problem connecting to the network service that tracks remote objects.

You can tell that the class is the target of remote method calls because it extends `UnicastRemoteObject`. The constructor of that class makes objects remotely accessible. The "path of least resistance" is to derive from `UnicastRemoteObject`, and all service implementation classes in this chapter do so.

Occasionally, you might not want to extend the `UnicastRemoteObject` class, perhaps because your implementation class already extends another class. In that situation, you need to manually instantiate the remote objects and pass them to the static `exportObject` method. Instead of extending `UnicastRemoteObject`, call

```
UnicastRemoteObject.exportObject(this, 0);
```

in the constructor of the remote object. The second parameter is 0 to indicate that any suitable port can be used to listen to client connections.

Note

The term "unicast" refers to the fact that the remote object is located by making a call to a single IP address and port. This is the only mechanism that is supported in Java SE. More sophisticated distributed object systems (such as JINI) allow for "multicast" lookup of remote objects that might be on a number of different servers.

The RMI Registry

UNREGISTERED VERSION OF CHM TO PDF CONVERTER PRO BY THETA-SOFTWARE

To access a remote object that exists on the server, the client needs a local stub object. How can the client request such a stub? The most common method is to call a remote method of another remote object and get a stub object as a return value. There is, however, a chicken-and-egg problem here: The *first* remote object has to be located some other way. For that purpose, the JDK provides a *bootstrap registry service*.

A server program registers at least one remote object with a bootstrap registry. To register a remote object, you need a RMI URL and a reference to the implementation object.

RMI URLs start with `rmi:` and contain an optional host name, an optional port number, and the name of the remote object that is (hopefully) unique. An example is:

```
rmi://regserver.mycompany.com:99/central_warehouse
```

By default, the host name is `localhost` and the port number is 1099. The server tells the registry at the given location to associate or "bind" the name with the object.

Here is the code for registering a `WarehouseImpl` object with the RMI registry on the same server:

```
WarehouseImpl centralWarehouse = new WarehouseImpl();
Context namingContext = new InitialContext();
namingContext.bind("rmi:central_warehouse", centralWarehouse);
```

The program in Listing 10-3 simply constructs and registers a `WarehouseImpl` object.

Listing 10-3. **`WarehouseServer.java`**

Code View:
```
import java.rmi.*;
import javax.naming.*;

/**
  * This server program instantiates a remote warehouse object, registers it with the naming
  * service, and waits for clients to invoke methods.
  * @version 1.12 2007-10-09
  * @author Cay Horstmann
  */

public class WarehouseServer
{
   public static void main(String[] args) throws RemoteException, NamingException
   {
      System.out.println("Constructing server implementation...");
      WarehouseImpl centralWarehouse = new WarehouseImpl();

      System.out.println("Binding server implementation to registry...");
      Context namingContext = new InitialContext();
      namingContext.bind("rmi:central_warehouse", centralWarehouse);

      System.out.println("Waiting for invocations from clients...");
   }
}
```

Note

For security reasons, an application can bind, unbind, or rebind registry object references only if it runs on the same host as the registry. This prevents hostile clients from changing the registry information. However, any client can look up objects.

A client can enumerate all registered RMI objects by calling:

Code View:
```
Enumeration<NameClassPair> e = namingContext.list("rmi://regserver.mycompany.com");
```

`NameClassPair` is a helper class that contains both the name of the bound object and the name of its class. For example, the following code displays the names of all registered objects:

```
while (e.hasMoreElements())
   System.out.println(e.nextElement().getName());
```

A client gets a stub to access a remote object by specifying the server and the remote object name in the following way:

```
String url = "rmi://regserver.mycompany.com/central_warehouse";
Warehouse centralWarehouse = (Warehouse) namingContext.lookup(url);
```

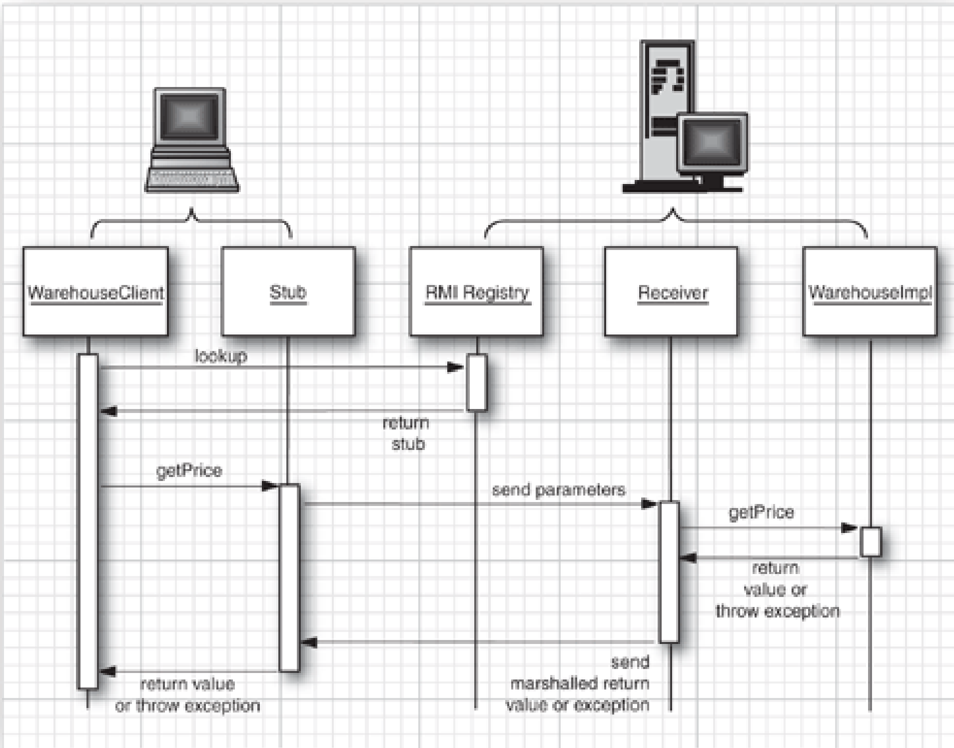

Note

Because it is notoriously difficult to keep names unique in a global registry, you should not use this technique as the general method for locating objects on the server. Instead, there should be relatively few named remote objects registered with the bootstrap service. These should be objects that can locate other objects for you.

The code in Listing 10-4 shows the client that obtains a stub to the remote warehouse object and invokes the remote `getPrice` method. Figure 10-4 shows the flow of control. The client obtains a `Warehouse` stub and invokes the `getPrice` method on it. Behind the scenes, the stub contacts the server and causes the `getPrice` method to be invoked on the `WarehouseImpl` object.

Figure 10-4. Calling the remote **getDescription** method

[View full size image]



Listing 10-4. **WarehouseClient.java**

Code View:
```
import java.rmi.*;
import java.util.*;
import javax.naming.*;

/**
 * A client that invokes a remote method.
 * @version 1.0 2007-10-09
 * @author Cay Horstmann
 */
public class WarehouseClient
{
   public static void main(String[] args) throws NamingException, RemoteExceptio
   {
      Context namingContext = new InitialContext();

      System.out.print("RMI registry bindings: ");
      Enumeration<NameClassPair> e = namingContext.list("rmi://localhost/");
      while (e.hasMoreElements())
         System.out.println(e.nextElement().getName());

      String url = "rmi://localhost/central_warehouse";
      Warehouse centralWarehouse = (Warehouse) namingContext.lookup(url);

      String descr = "Blackwell Toaster";
```

```
        double price = centralWarehouse.getPrice(descr);
        System.out.println(descr + ": " + price);
    }
}
```


javax.naming.InitialContext 1.3

- InitialContext()

  constructs a naming context that can be used for accessing the RMI registry.

javax.naming.Context 1.3

- static Object lookup(String name)

  returns the object for the given name. Throws a NamingException if the name is not currently bound.

- static void bind(String name, Object obj)

  binds name to the object obj. Throws a NameAlreadyBoundException if the object is already bound.

- static void unbind(String name)

  unbinds the name. It is legal to unbind a name that doesn't exist.

- static void rebind(String name, Object obj)

  binds name to the object obj. Replaces any existing binding.

- NamingEnumeration<NameClassPair> list(String name)

  returns an enumeration listing all matching bound objects. To list all RMI objects, call with "rmi:".

API `javax.naming.NameClassPair` 1.3

- `String getName()`

  gets the name of the named object.

- `String getClassName()`

  gets the name of the class to which the named object belongs.

API `java.rmi.Naming` 1.1

- `static Remote lookup(String url)`

  returns the remote object for the URL. Throws a `NotBoundException` if the name is not currently bound.

- `static void bind(String name, Remote obj)`

  binds `name` to the remote object `obj`. Throws an `AlreadyBoundException` if the object is already bound.

- `static void unbind(String name)`

  unbinds the name. Throws the `NotBound` exception if the name is not currently bound.

- `static void rebind(String name, Remote obj)`

  binds `name` to the remote object `obj`. Replaces any existing binding.

- `static String[] list(String url)`

  returns an array of strings of the URLs in the registry located at the given URL. The array contains a snapshot of the names present in the registry.

Deploying the Program

Deploying an application that uses RMI can be tricky because so many things can go wrong and the error messages that you get when something does go wrong are so poor. We have found that it really pays off to test the deployment under realistic conditions, separating the classes for client and server.

Make two separate directories to hold the classes for starting the server and client.

```
server/
   WarehouseServer.class
   Warehouse.class
   WarehouseImpl.class
```



```
client/
   WarehouseClient.class
   Warehouse.class
```

When deploying RMI applications, one commonly needs to dynamically deliver classes to running programs. One example is the RMI registry. Keep in mind that one instance of the registry will serve many different RMI applications. The RMI registry needs to have access to the class files of the service interfaces that are being registered. When the registry starts, however, one cannot predict all future registration requests. Therefore, the RMI registry dynamically loads the class files of any remote interfaces that it has not previously encountered.

Dynamically delivered class files are distributed through standard web servers. In our case, the server program needs to make the `Warehouse.class` file available to the RMI registry, so we put that file into a third directory that we call `download`.

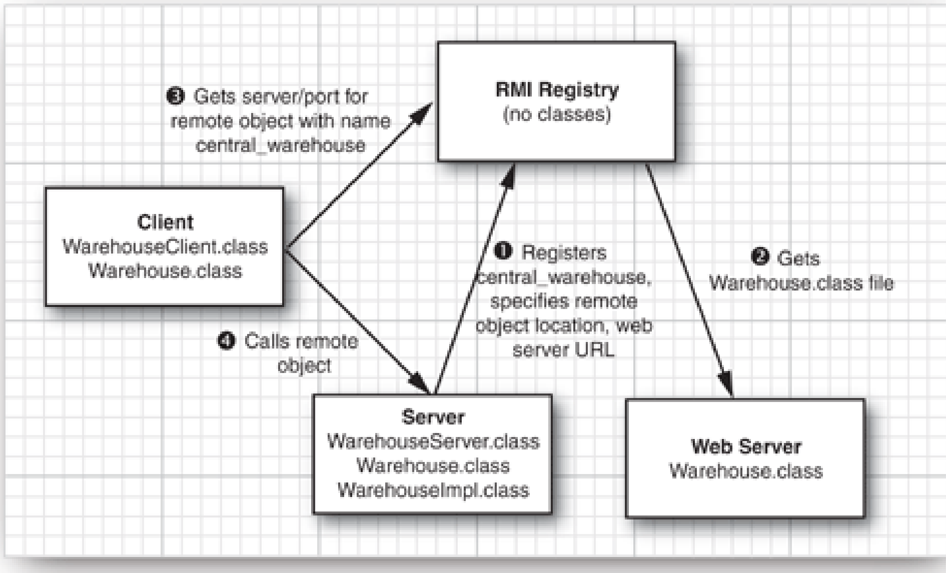
```
download/
   Warehouse.class
```

We use a web server to serve the contents of that directory.

When the application is deployed, the server, RMI registry, web server, and client can be located on four different computers—see Figure 10-5. However, for testing purposes, we will use a single computer.

Figure 10-5. Server calls in the Warehouse application

[View full size image]



Note

For security reasons, the `rmiregistry` service that is part of the JDK only allows binding calls from the same host. That is, the server and `rmiregistry` process need to be located on the same computer. However, the RMI architecture allows for a more general RMI registry implementation that supports multiple servers.

To test the sample application, use the `NanoHTTPD` web server that is available from http://elonen.iki.fi/code/nanohttpd. This tiny web server is implemented in a single Java source file. Open a new console window, change to the `download` directory, and copy `NanoHTTPD.java` to that directory. Compile the source file and start the web server, using the command

```
java NanoHTTPD 8080
```

The command-line argument is the port number. Use any other available port if port 8080 is already used on your machine.

Next, open another console window, change to a directory that *contains no class files*, and start the RMI registry:

```
rmiregistry
```

Caution

Before starting the RMI registry, make sure that the CLASSPATH environment variable is not set to anything, and double-check that the current directory contains no class files. Otherwise, the RMI registry might find spurious class files, which will confuse it when it should download additional classes from a different source. There is a reason for this behavior; see http://java.sun.com/javase/6/docs/technotes/guides/rmi/codebase.html. In a nutshell, each stub object has a *codebase* entry that specifies from where it was loaded. That codebase is used to load dependent classes. If the RMI registry finds a class locally, it will set the wrong codebase.

Now you are ready to start the server. Open a third console window, change to the `server` directory, and issue the command


```
java -Djava.rmi.server.codebase=http://localhost:8080/ WarehouseServer
```

The `java.rmi.server.codebase` property points to the URL for serving class files. The server program communicates this URL to the RMI registry.

Have a peek at the console window running `NanoHTTPD`. You will see a message that demonstrates that the `Warehouse.class` file has been served to the RMI registry.

Caution

It is very important that you make sure that the codebase URL *ends with a slash* (/).

Note that the server program does not exit. This seems strange—after all, the program just creates a `WarehouseImpl` object and registers it. Actually, the `main` method does exit immediately after registration, as you would expect. However, when you create an object of a class that extends `UnicastRemoteObject`, a separate thread that keeps the program alive indefinitely is started. Thus, the program stays around to allow clients to connect to it.

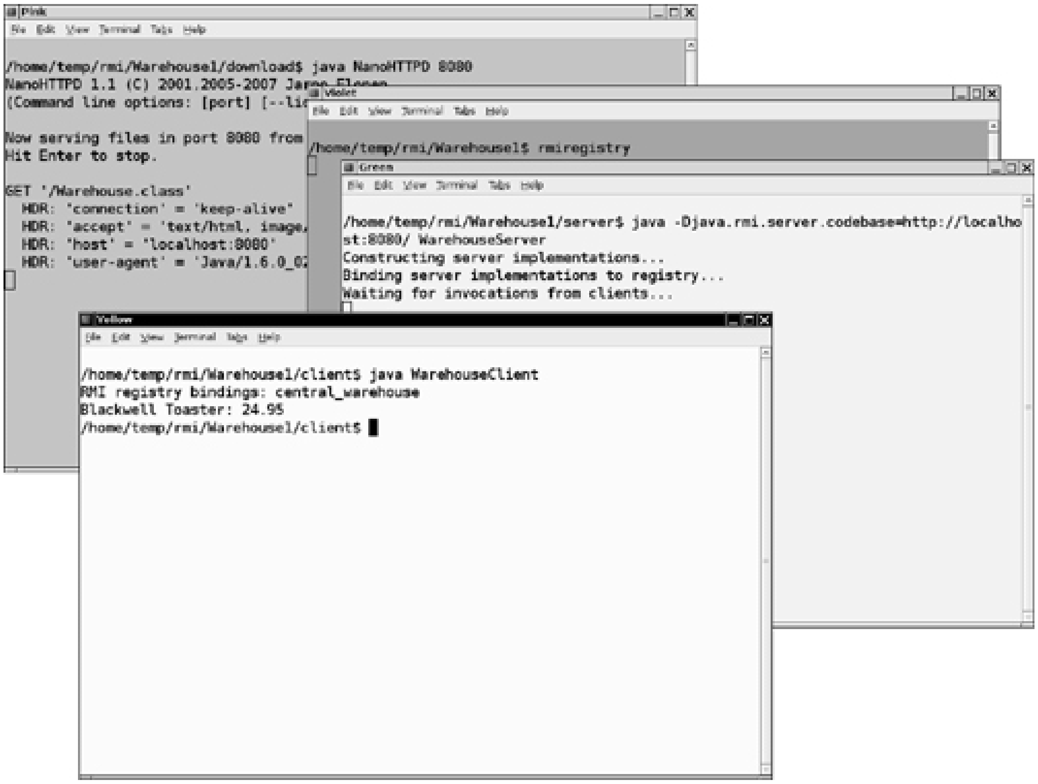
Finally, open a fourth console window, change to the `client` directory, and run

```
java WarehouseClient
```

You will see a short message, indicating that the remote method was successfully invoked (see Figure 10-6).

Figure 10-6. Testing an RMI application

[View full size image]



Note

If you just want to test out basic program logic, you can put your client and server class files into the same directory. Then you can start the RMI registry, server, and client in that directory. However, because RMI class loading is the source of much grief and confusion, we felt it best to show you the correct setup for dynamic class loading right away.

Logging RMI Activity

If you start the server with the option

```
-Djava.rmi.server.logCalls=true WarehouseServer &
```

then the server logs all remote method calls on its console. Try it—you'll get a good impression of the RMI traffic.

If you want to see additional logging messages, you have to configure RMI loggers, using the standard Java logging API. (See Volume I, Chapter 11 for more information on logging.)

Make a file `logging.properties` with the following content:

Code View:

```
handlers=java.util.logging.ConsoleHandler
.level=FINE
java.util.logging.ConsoleHandler.level=FINE
java.util.logging.ConsoleHandler.formatter=java.util.logging.SimpleFormatter
```


You can fine-tune the settings by setting individual levels for each logger rather than setting the global level. Table 10-1 lists the RMI loggers. For example, to track the class loading activity, you can set

```
sun.rmi.loader.level=FINE
```


Table 10-1. RMI Loggers

Logger Name	Logged Activity
sun.rmi.server.call	Server-side remote calls
sun.rmi.server.ref	Server-side remote references
sun.rmi.client.call	Client-side remote calls
sun.rmi.client.ref	Client-side remote references
sun.rmi.dgc	Distributed garbage collection
sun.rmi.loader	RMIClassLoader
sun.rmi.transport.misc	Transport layer
sun.rmi.transport.tcp	TCP binding and connection
sun.rmi.transport.proxy	HTTP tunneling

Start the RMI registry with the option

-J-Djava.util.logging.config.file=*directory*/logging.properties

Start the client and server with

-Djava.util.logging.config.file=*directory*/logging.properties

Here is an example of a logging message that shows a class loading problem: The RMI registry cannot find the Warehouse class because the web server has been shut down.

Code View:

```
FINE: RMI TCP Connection(1)-127.0.1.1: (port 1099) op = 80
Oct 13, 2007 4:43:30 PM sun.rmi.server.LoaderHandler loadProxyClass
FINE: RMI TCP Connection(1)-127.0.1.1: interfaces = [java.rmi.Remote, Warehouse], codebase =
"http://localhost:8080/"
Oct 13, 2007 4:43:30 PM sun.rmi.server.LoaderHandler loadProxyClass
FINE: RMI TCP Connection(1)-127.0.1.1: proxy class resolution failed
java.lang.ClassNotFoundException: Warehouse
```

Parameters and Return Values in Remote Methods

At the start of a remote method invocation, the parameters need to be moved from the virtual machine of the client to the virtual machine of the server. After the invocation has completed, the return value needs to be transferred in the other direction. When a value is passed from one virtual machine to another other, we distinguish two cases: passing remote objects and passing nonremote objects. For example, suppose that a client of the `WarehouseServer` passes a `Warehouse` reference (that is, a stub through which the remote warehouse object can be called) to another remote method. That is an example of passing a remote object. However, most method parameters will be ordinary Java objects, not stubs to remote objects. An example is the `String` parameter of the `getPrice` method in our first sample application.

Transferring Remote Objects

When a reference to a remote object is passed from one virtual machine to the other, the sender and recipient of the remote object both hold a reference to the same entity. That reference is not a memory location (which is only meaningful in a single virtual machine), but it consists of a network address and a unique identifier for the remote object. This information is encapsulated in a stub object.

Conceptually, passing a remote reference is quite similar to passing local object references within a virtual machine. However, always keep in mind that a method call on a remote reference is significantly slower and potentially less reliable than a method call on a local reference.

Transferring Nonremote Objects

Consider the `String` parameter of the `getPrice` method. The string value needs to be copied from the client to the server. It is not difficult to imagine how a copy of a string can be transported across a network. The RMI mechanism can also make copies of more complex objects, provided they are *serializable*. RMI uses the serialization mechanism described in Chapter 1 to send objects across a network connection. This means that any classes that implement the `Serializable` interface can be used as parameter or return types.

Passing parameters by serializing them has a subtle effect on the semantics of remote methods. When you pass objects into a local method, object *references* are transferred. When the method applies a mutator method to a parameter object, the caller will observe that change. But if a remote method mutates a serialized parameter, it changes the copy, and the caller will never notice.

To summarize, there are two mechanisms for transferring values between virtual machines.

- Objects of classes that implement the `Remote` interface are transferred as remote references.
- Objects of classes that implement the `Serializable` interface but not the `Remote` interface are copied using serialization.

All of this is automatic and requires no programmer intervention. Keep in mind that serialization can be slow for large objects, and that the remote method cannot mutate serialized parameters. You can, of course, avoid these issues by passing around remote references. That too comes at a cost: Invoking methods on remote references is far more expensive than calling local methods. Being aware of these costs allows you to make informed choices when designing remote services.

Note

Remote objects are garbage-collected automatically, just as local objects are. However, the distributed collector is signifcantly more complex. When the local garbage collector finds that there are further local uses of a remote reference, it notifies the distributed collector that the server is no longer referenced by this client. When a server is no longer used by any clients, it is marked as garbage.

Our next example program will illustrate the transfer of remote and serializable objects. We change the `Warehouse` interface as shown in Listing 10-5. Given a list of keywords, the warehouse returns the `Product` that is the best match.

Listing 10-5. **`Warehouse.java`**

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

/**
   The remote interface for a simple warehouse.
   @version 1.0 2007-10-09
   @author Cay Horstmann
*/
public interface Warehouse extends Remote
{
   double getPrice(String description) throws RemoteException;
   Product getProduct(List<String> keywords) throws RemoteException;
}
```

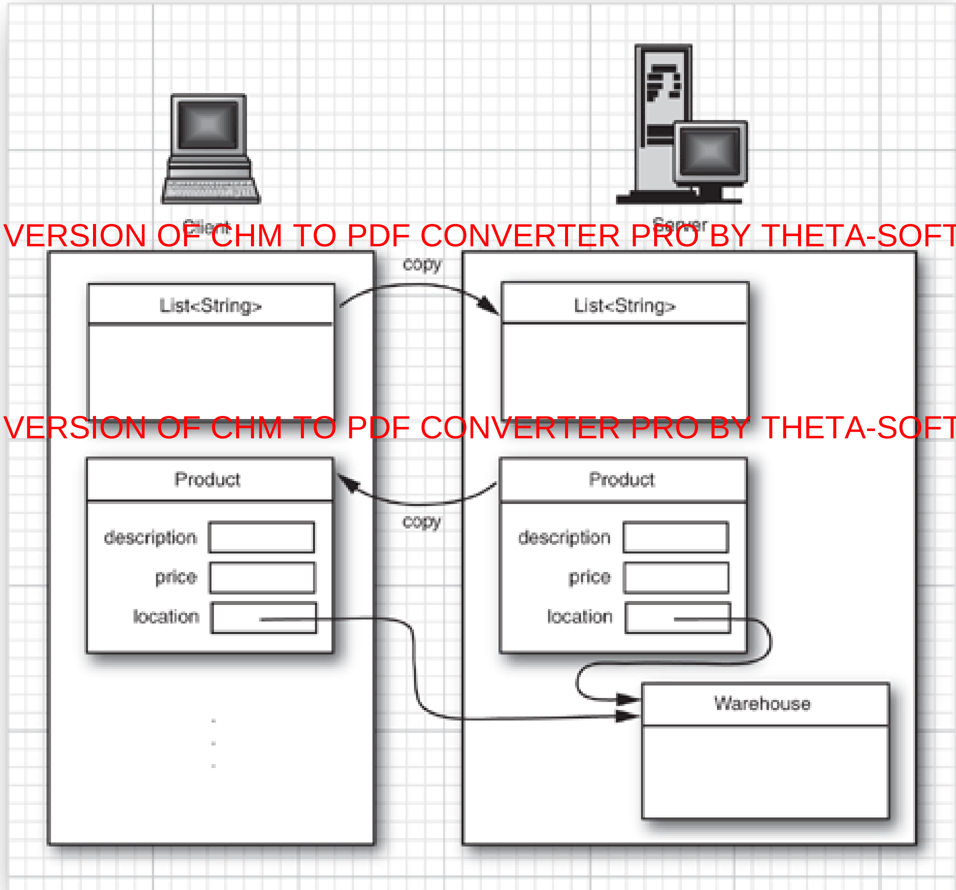
The parameter of the `getProduct` method has type `List<String>`. A parameter value must belong to a serializable class that implements the `List<String>` interface, such as `ArrayList<String>`. (Our sample client passes a value that is obtained by a call to `Arrays.asList`. Fortunately, that method is guaranteed to return a serializable list as well.)

The return type `Product` encapsulates the description, price, and location of the product—see Listing 10-6.

Note that the `Product` class is serializable. The server constructs a `Product` object, and the client gets a copy (see Figure 10-7).

Figure 10-7. Copying local parameter and result objects

[View full size image]



UNREGISTERED VERSION OF CHM TO PDF CONVERTER PRO BY THETA-SOFTWARE

Listing 10-6. **Product.java**

Code View:
```
import java.io.*;

public class Product implements Serializable
{
   public Product(String description, double price)
   {
      this.description = description;
      this.price = price;
   }

   public String getDescription()
   {
      return description;
   }

   public double getPrice()
   {
      return price;
   }

   public Warehouse getLocation()
```

```
    {
       return location;
    }

    public void setLocation(Warehouse location)
    {
       this.location = location;
    }

    private String description;
    private double price;
    private Warehouse location;
 }
```

However, there is a subtlety. The `Product` class has an instance field of type `Warehouse`, a remote interface. The warehouse object is *not* serialized, which is just as well as it might have a huge amount of state. Instead, the client receives a stub to a remote `Warehouse` object. That stub might be different from the `centralWarehouse` stub on which the `getProduct` method was called. In our implementation, we will have two kinds of products, toasters and books, that are located in different warehouses.

Dynamic Class Loading

There is another subtlety to our next sample program. A list of keyword strings is sent to the server, and the warehouse returns an instance of a class `Product`. Of course, the client program will need the class file `Product.class` to compile. However, whenever our server program cannot find a match for the keywords, it returns the one product that is sure to delight everyone: the Core Java book. That object is an instance of the `Book` class, a subclass of `Product`.

When the client was compiled, it might have never seen the `Book` class. Yet when it runs, it needs to be able to execute `Book` methods that override `Product` methods. This demonstrates that the client needs to have the capability of loading additional classes at runtime. The client uses the same mechanism as the RMI registry. Classes are served by a web server, the RMI server class communicates the URL to the client, and the client makes an HTTP request to download the class files.

Whenever a program loads new code from another network location, there is a security issue. For that reason, you need to use a *security manager* in RMI applications that dynamically load classes. (See Chapter 9 for more information on class loaders and security managers.)

Programs that use RMI should install a security manager to control the activities of the dynamically loaded classes. You install it with the instruction

```
System.setSecurityManager(new SecurityManager());
```

Note

If all classes are available locally, then you do not actually need a security manager. If you know all class files of your program at deployment time, you can deploy them all locally. However, it often happens that the client or server program evolves and new classes are added over time. Then you benefit from dynamic class loading. Any time you load code from another source, you need a security manager.

By default, the `SecurityManager` restricts all code in the program from establishing network connections. However, the program needs to make network connections to three remote locations:

- The web server that loads remote classes.
- The RMI registry.
- Remote objects.

To allow these operations, you supply a policy file. (We discussed policy files in greater detail in Chapter 9.) Here is a policy file that allows an application to make any network connection to a port with port number of at least 1024. (The RMI port is 1099 by default, and the remote objects also use ports $\geq$ 1024. We use port 8080 for dowloading classes.)

```
grant
{
   permission java.net.SocketPermission
      "*:1024-65535", "connect";
};
```

You need to instruct the security manager to read the policy file by setting the `java.security.policy` property to the file name. You can use a call such as

```
System.setProperty("java.security.policy", "rmi.policy");
```

Alternatively, you can specify the system property setting on the command line:

```
-Djava.security.policy=rmi.policy
```

To run the sample application, be sure that you have killed the RMI registry, web server, and the server program from the preceding sample. Open four console windows and follow these steps.

1. Compile the source files for the interface, implementation, client, and server classes.

   ```
   javac *.java
   ```

2. Make three directories, `client`, `server`, and `download`, and populate them as follows:

   ```
   client/
      WarehouseClient.class
      Warehouse.class
      Product.class
      client.policy
   server/
      Warehouse.class
      Product.class
      Book.class
      WarehouseImpl.class
      WarehouseServer.class
      server.policy
   download
      Warehouse.class
      Product.class
      Book.class
   ```

3. In the first console window, change to a directory that has *no* class files. Start the RMI registry.

4. In the second console window, change to the `download` directory and start `NanoHTTPD`.

5. In the third console window, change to the `server` directory and start the server.

   ```
   java -Djava.rmi.server.codebase=http://localhost:8080/ WarehouseServer
   ```

6. In the fourth console window, change to the `client` directory and run the client.

   ```
   java WarehouseClient
   ```

Listing 10-7 shows the code of the `Book` class. Note that the `getDescription` method is overridden to show the ISBN. When the client program runs, it shows the ISBN for the Core Java book, which proves that the `Book` class was loaded dynamically. Listing 10-8 shows the warehouse implementation. A warehouse has a reference to a backup warehouse. If an item cannot be found in the warehouse, the backup warehouse is searched. Listing 10-9 shows the server program. Only the central warehouse is entered into the RMI registry. Note that a remote reference to the backup warehouse can be passed to the client even though it is not included in the RMI registry. This happens whenever no keyword matches and a *Core Java* book (whose `location` field references the backup warehouse) is sent to the client.

Listing 10-7. **`Book.java`**

```
/**
 * A book is a product with an ISBN number.
 * @version 1.0 2007-10-09
 * @author Cay Horstmann
 */
public class Book extends Product
{
   public Book(String title, String isbn, double price)
   {
      super(title, price);
      this.isbn = isbn;
   }

   public String getDescription()
   {
      return super.getDescription() + " " + isbn;
   }

   private String isbn;
}
```


Listing 10-8. **WarehouseImpl.java**

Code View:

```
import java.rmi.*;
import java.rmi.server.*;
import java.util.*;

/**
 * This class is the implementation for the remote Warehouse interface.
 * @version 1.0 2007-10-09
 * @author Cay Horstmann
 */
public class WarehouseImpl extends UnicastRemoteObject implements Warehouse
{
   /**
    * Constructs a warehouse implementation.
    */
   public WarehouseImpl(Warehouse backup) throws RemoteException
   {
      products = new HashMap<String, Product>();
      this.backup = backup;
   }

   public void add(String keyword, Product product)
   {
      product.setLocation(this);
      products.put(keyword, product);
   }

   public double getPrice(String description) throws RemoteException
   {
      for (Product p : products.values())
         if (p.getDescription().equals(description)) return p.getPrice();
      if (backup == null) return 0;
      else return backup.getPrice(description);
   }

```

```
   public Product getProduct(List<String> keywords) throws RemoteException
   {
      for (String keyword : keywords)
      {
         Product p = products.get(keyword);
         if (p != null) return p;
      }
      if (backup != null)
         return backup.getProduct(keywords);
      else if (products.values().size() > 0)
         return products.values().iterator().next();
      else
         return null;
   }

   private Map<String, Product> products;
   private Warehouse backup;
}
```

Listing 10-9. **WarehouseServer.java**

Code View:

```
import java.rmi.*;
import javax.naming.*;

/**
  * This server program instantiates a remote warehouse objects, registers it with the naming
  * service, and waits for clients to invoke methods.
  * @version 1.12 2007-10-09
  * @author Cay Horstmann
  */

public class WarehouseServer
{
   public static void main(String[] args) throws RemoteException, NamingException
   {
      System.setProperty("java.security.policy", "server.policy");
      System.setSecurityManager(new SecurityManager());

      System.out.println("Constructing server implementation...");
      WarehouseImpl backupWarehouse = new WarehouseImpl(null);
      WarehouseImpl centralWarehouse = new WarehouseImpl(backupWarehouse);

      centralWarehouse.add("toaster", new Product("Blackwell Toaster", 23.95));
      backupWarehouse.add("java", new Book("Core Java vol. 2", "0132354799", 44.95));

      System.out.println("Binding server implementation to registry...");
      Context namingContext = new InitialContext();
      namingContext.bind("rmi:central_warehouse", centralWarehouse);

      System.out.println("Waiting for invocations from clients...");
   }
}
```

Remote References with Multiple Interfaces

A remote class can implement multiple interfaces. Consider a remote interface `ServiceCenter`.

```
public interface ServiceCenter extends Remote
{
   int getReturnAuthorization(Product prod) throws RemoteException;
}
```


Now suppose a `WarehouseImpl` class implements this interface as well as the `Warehouse` interface. When a remote reference to such a service center is transferred to another virtual machine, the recipient obtains a stub that has access to the remote methods in both the `ServiceCenter` and the `Warehouse` interface. You can use the `instanceof` operator to find out whether a particular remote object implements an interface. Suppose you receive a remote object through a variable of type `Warehouse`.


```
Warehouse location = product.getLocation();
```

The remote object might or might not be a service center. To find out, use the test

```
if (location instanceof ServiceCenter)
```

If the test passes, you can cast `location` to the `ServiceCenter` type and invoke the `getReturnAuthorization` method.

Remote Objects and the **equals**, **hashCode**, and **clone** Methods

Objects inserted in sets must override the `equals` method. In the case of a hash set or hash map, the `hashCode` method must be defined as well. However, there is a problem when trying to compare remote objects. To find out if two remote objects have the same contents, the call to `equals` would need to contact the servers containing the objects and compare their contents. Like any remote call, that call could fail. But the `equals` method in the class `Object` is not declared to throw a `RemoteException`, whereas all methods in a remote interface must throw that exception. Because a subclass method cannot throw more exceptions than the superclass method it replaces, you cannot define an `equals` method in a remote interface. The same holds for `hashCode`.

Instead, the `equals` and `hashCode` methods on stub objects simply look at the location of the remote objects. The `equals` method deems two stubs equal if they refer to the same remote object. Two stubs that refer to different remote objects are never equal, even if those objects have identical contents. Similarly, the hash code is computed only from the object identifier.

For the same technical reasons, remote references do not have a `clone` method. If `clone` were to make a remote call to tell the server to clone the implementation object, then the `clone` method would need to throw a `RemoteException`. However, the `clone` method in the `Object` superclass promised never to throw any exception other than `CloneNotSupportedException`.

To summarize, you can use remote references in sets and hash tables, but you must remember that equality testing and hashing do not take into account the contents of the remote objects. You simply cannot clone remote references.

Remote Object Activation

In the preceding sample programs, we used a server program to instantiate and register objects so that clients could make remote calls on them. However, in some cases, it might be wasteful to instantiate lots of remote objects and have them wait for connections, whether or not client objects use them. The *activation* mechanism lets you delay the object construction so that a remote object is only constructed when at least one client invokes a remote method on it.

To take advantage of activation, the client code is completely unchanged. The client simply requests a remote reference and makes calls through it.

However, the server program is replaced by an activation program that constructs *activation descriptors* of the objects that are to be constructed at a later time, and binds receivers for remote method calls with the naming service. When a call is made for the first time, the information in the activation descriptor is used to construct the object.

A remote object that is used in this way should extend the `Activatable` class instead of the `UnicastRemoteObject` class. Of course, it also implements one or more remote interfaces. For example,

```
class WarehouseImpl
   extends Activatable
   implements Warehouse
{
   . . .
}
```

Because the object construction is delayed until a later time, it must happen in a standardized form. Therefore, you must provide a constructor that takes two parameters:

- An activation ID (which you simply pass to the superclass constructor).
- A single object containing all construction information, wrapped in a `MarshalledObject`.

If you need multiple construction parameters, you must package them into a single object. You can always use an `Object[]` array or an `ArrayList` for this purpose.

When you build the activation descriptor, you will construct a `MarshalledObject` from the construction information like this:

```
MarshalledObject<T> param = new MarshalledObject<T>(constructionInfo);
```

In the constructor of the implementation object, use the `get` method of the `MarshalledObject` class to obtain the deserialized construction information.

```
T constructionInfo = param.get();
```

To demonstrate activation, we modify the `WarehouseImpl` class so that the construction information is a map of descriptions and prices. That information is wrapped into a `MarshalledObject` and unwrapped in the constructor:

Code View:
```
public WarehouseImpl(ActivationID id, MarshalledObject<Map<String, Double>> param)
      throws RemoteException, ClassNotFoundException, IOException
{
   super(id, 0);
   prices = param.get();
   System.out.println("Warehouse implementation constructed.");
}
```


By passing 0 as the second parameter of the superclass constructor, we indicate that the RMI library should assign a suitable port number to the listener port.

This constructor prints a message so that you can see that the warehouse object is activated on demand.

Note

Your remote objects don't actually have to extend the `Activatable` class. If they don't, then place the static method call

```
Activatable.exportObject(this, id, 0)
```

in the constructor of the server class.

Now let us turn to the activation program. First, you need to define an activation group. An activation group describes common parameters for launching the virtual machine that contains the remote objects. The most important parameter is the security policy.

Construct an activation group descriptor as follows:

```
Properties props = new Properties();
props.put("java.security.policy", "/path/to/server.policy");
ActivationGroupDesc group = new ActivationGroupDesc(props, null);
```

The second parameter describes special command options. We don't need any for this example, so we pass a `null` reference.

Next, create a group ID with the call

```
ActivationGroupID id = ActivationGroup.getSystem().registerGroup(group);
```

Now you are ready to construct activation descriptors. For each object that should be constructed on demand, you need the following:

- The activation group ID for the virtual machine in which the object should be constructed.

- The name of the class (such as "WarehouseImpl" or "com.mycompany.MyClassImpl").

- The URL string from which to load the class files. This should be the base URL, not including package paths.

- The marshalled construction information.

For example,

Code View:
```
MarshalledObject param = new MarshalledObject(constructionInfo);
ActivationDesc desc = new ActivationDesc(id, "WarehouseImpl",
                                         "http://myserver.com/download/", param);
```

Pass the descriptor to the static Activatable.register method. It returns an object of some class that implements the remote interfaces of the implementation class. You can bind that object with the naming service:

```
Warehouse centralWarehouse = (Warehouse) Activatable.register(desc);
namingContext.bind("rmi:central_warehouse", centralWarehouse);
```

Unlike the server programs of the preceding examples, the activation program exits after registering and binding the activation receivers. The remote objects are constructed only when the first remote method call occurs.

Listings 10-10 and 10-11 show the code for the activation program and the activatable warehouse implementation. The warehouse interface and the client program are unchanged.

To launch this program, follow these steps:

1. Compile all source files.

2. Distribute class files as follows:

```
client/
   WarehouseClient.class
   Warehouse.class
server/
   WarehouseActivator.class
   Warehouse.class
   WarehouseImpl.class
   server.policy
download/
   Warehouse.class
   WarehouseImpl.class
rmi/
   rmid.policy
```


3. Start the RMI registry in the `rmi` directory (which contains no class files).

4. Start the RMI activation daemon in the `rmi` directory.

```
rmid -J-Djava.security.policy=rmid.policy
```

The `rmid` program listens to activation requests and activates objects in a separate virtual machine. To launch a virtual machine, the `rmid` program needs certain permissions. These are specified in a policy file (see Listing 10-12). You use the `-J` option to pass an option to the virtual machine running the activation daemon.

5. Start the `NanoHTTPD` web server in the `download` directory.

6. Run the activation program from the `server` directory.

```
java -Djava.rmi.server.codebase=http://localhost:8080/ WarehouseActivator
```

The program exits after the activation receivers have been registered with the naming service. (You might wonder why you need to specify the codebase as it is also provided in the constructor of the activation descriptor. However, that information is only processed by the RMI activation daemon. The RMI registry still needs the codebase to load the remote interface classes.)

7. Run the client program from the `client` directory.

```
java WarehouseClient
```

The client will print the familiar product description. When you run the client for the first time, you will also see the constructor messages in the shell window of the activation daemon.

Listing 10-10. **WarehouseActivator.java**

Code View:

```
import java.io.*;
import java.rmi.*;
import java.rmi.activation.*;
import java.util.*;
import javax.naming.*;

/**
  * This server program instantiates a remote warehouse object, registers it with the naming
  * service, and waits for clients to invoke methods.
  * @version 1.12 2007-10-09
  * @author Cay Horstmann
  */

public class WarehouseActivator
{
   public static void main(String[] args) throws RemoteException, NamingException,
         ActivationException, IOException
   {
      System.out.println("Constructing activation descriptors...");

      Properties props = new Properties();
      // use the server.policy file in the current directory
      props.put("java.security.policy", new File("server.policy").getCanonicalPath());
      ActivationGroupDesc group = new ActivationGroupDesc(props, null);
      ActivationGroupID id = ActivationGroup.getSystem().registerGroup(group);

      Map<String, Double> prices = new HashMap<String, Double>();
      prices.put("Blackwell Toaster", 24.95);
      prices.put("ZapXpress Microwave Oven", 49.95);

      MarshalledObject<Map<String, Double>> param = new MarshalledObject<Map<String, Double>>(
            prices);

      String codebase = "http://localhost:8080/";

      ActivationDesc desc = new ActivationDesc(id, "WarehouseImpl", codebase, param);

      Warehouse centralWarehouse = (Warehouse) Activatable.register(desc);

      System.out.println("Binding activable implementation to registry...");
      Context namingContext = new InitialContext();
      namingContext.bind("rmi:central_warehouse", centralWarehouse);
      System.out.println("Exiting...");
   }
}
```

Listing 10-11. **WarehouseImpl.java**

Code View:

```
import java.io.*;
import java.rmi.*;
import java.rmi.activation.*;
import java.util.*;

/**
 * This class is the implementation for the remote Warehouse interface.
 * @version 1.0 2007-10-20
 * @author Cay Horstmann
 */
public class WarehouseImpl extends Activatable implements Warehouse
{
   public WarehouseImpl(ActivationID id, MarshalledObject<Map<String, Double>> param)
         throws RemoteException, ClassNotFoundException, IOException
   {
      super(id, 0);
      prices = param.get();
      System.out.println("Warehouse implementation constructed.");
   }

   public double getPrice(String description) throws RemoteException
   {
      Double price = prices.get(description);
      return price == null ? 0 : price;
   }

   private Map<String, Double> prices;
}
```


Listing 10-12. **rmid.policy**

```
grant
{
   permission com.sun.rmi.rmid.ExecPermission
      "${java.home}${/}bin${/}java";
   permission com.sun.rmi.rmid.ExecOptionPermission
      "-Djava.security.policy=*";
}
```

`java.rmi.activation.Activatable` 1.2

- `protected Activatable(ActivationID id, int port)`

  constructs the activatable object and establishes a listener on the given port. Use 0 for the port to have a port assigned automatically.

- `static Remote exportObject(Remote obj, ActivationID id, int port)`

  makes a remote object activatable. Returns the activation receiver that should be made available to remote callers. Use 0 for the port to have a port assigned automatically.

- `static Remote register(ActivationDesc desc)`

  registers the descriptor for an activatable object and prepares it for receiving remote calls. Returns the activation receiver that should be made available to remote callers.

`java.rmi.MarshalledObject` 1.2

- `MarshalledObject(Object obj)`

  constructs an object containing the serialized data of a given object.

- `Object get()`

  deserializes the stored object data and returns the object.

`java.rmi.activation.ActivationGroupDesc` 1.2

- `ActivationGroupDesc(Properties props, ActivationGroupDesc.CommandEnvironment env)`

  constructs an activation group descriptor that specifies virtual machine properties for a virtual machine that hosts activated objects. The `env` parameter contains the path to the virtual machine executable and command-line options, or it is `null` if no special settings are required.

java.rmi.activation.ActivationGroup 1.2

- `static ActivationSystem getSystem()`

  returns a reference to the activation system.

java.rmi.activation.ActivationSystem 1.2

- `ActivationGroupID registerGroup(ActivationGroupDesc group)`

  registers an activation group and returns the group ID.

java.rmi.activation.ActivationDesc 1.2

- `ActivationDesc(ActivationGroupID id, String className, String classFileURL, MarshalledObject data)`

  constructs an activation descriptor.

Web Services and JAX-WS

In recent years, *web services* have emerged as a popular technology for remote method calls. Technically, a web service has two components:

- A service that can be accessed with the SOAP transport protocol
- A description of the service in the WSDL format

SOAP is an XML protocol for invoking remote methods, similar to the protocol that RMI uses for the communicatio between clients and servers. Just as you can program RMI applications without knowing anything about the details the RMI protocol, you don't really need to know any details about SOAP to call a web service.

WSDL is an interface description language. It too is based on XML. A WSDL document describes the interface of a web service: the methods that can be called, and their parameter and return types. In this section, we generate a WSDL document from a service implemented in Java. This document contains all the information that a client program needs to invoke the service, whether it is written in Java or another programming language. In the next section, we write a Java program that invokes the Amazon e-commerce service, using the WSDL provided by Amazon. We have no idea in which language that service was implemented.

Using JAX-WS

There are several toolkits for implementing web services in Java. In this section, we discuss the JAX-WS technolog that is included in Java SE 6 and above.

With JAX-WS, you do not provide an interface for a web service. Instead, you annotate a class with `@WebService` , shown in Listing 10-13 . Note also the `@WebParam` annotation of the `description` parameter. It gives the paramete humanly readable name in the WSDL file. (This annotation is optional. By default, the parameter would be called `arg0` .)

Listing 10-13. **`Warehouse.java`**

Code View:

```
package com.horstmann.corejava;
import java.util.*;
import javax.jws.*;

/**
 * This class is the implementation for a Warehouse web service
 * @version 1.0 2007-10-09
 * @author Cay Horstmann
 */

@WebService
public class Warehouse
{
   public Warehouse()
   {
      prices = new HashMap<String, Double>();
      prices.put("Blackwell Toaster", 24.95);
      prices.put("ZapXpress Microwave Oven", 49.95);
   }

   public double getPrice(@WebParam(name="description") String description)
   {
      Double price = prices.get(description);
      return price == null ? 0 : price;
   }

   private Map<String, Double> prices;
}
```


In RMI, the stub classes were generated dynamically, but with JAX-WS, you run a tool to generate them. Change t the base directory of the `Webservices1` source and run the `wsgen` class as follows:

```
wsgen -classpath . com.horstmann.corejava.Warehouse
```

Note

The `wsgen` tool requires that the class that provides the web service is contained in a package other than the default package.

The tool generates two rather mundane classes in the `com.horstmann.corejava.jaxws` package. The first class encapsulates all parameters of the call:

Code View:

```
public class GetPrice
{
    private String description;
```

```
    public String getDescription() { return this.description; }
    public void setDescription(String description) { this.description = description; }
}
```

The second class encapsulates the return value:

```
public class GetPriceResponse
{
    private double _return;
    public double get_return() { return this._return; }
    public void set_return(double _return) { this._return = _return; }
}
```

Typically, one has a sophisticated server infrastructure for deploying web services, which we do not discuss here. T JDK contains a very simple mechanism for testing a service. Simply call the `Endpoint.publish` method. A server i started on the given URL—see Listing 10-14 .

Listing 10-14. **`WarehouseServer.java`**

Code View:

```
package com.horstmann.corejava;

import javax.xml.ws.*;

public class WarehouseServer
{
   public static void main(String[] args)
   {
      Endpoint.publish("http://localhost:8080/WebServices/warehouse", new Warehouse());
   }
}
```

At this point, you should compile the server classes, run `wsgen` , and start the server:

```
java com.horstmann.corejava.WarehouseServer
```

Now point your web browser to `http://localhost:8080/WebServices/warehouse?wsdl` . You will get this WSDL file:

Code View:

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions xmlns="http://schemas.xmlsoap.org/wsdl/" xmlns:tns="http://corejava.horstmann.com/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  targetNamespace="http://corejava.horstmann.com/" name="WarehouseService">
  <types>
    <xsd:schema>
      <xsd:import schemaLocation="http://localhost:8080/WebServices/warehouse?xsd=1"
      namespace="http://corejava.horstmann.com/"></xsd:import>
```

```
    </xsd:schema>
  </types>
  <message name="getPrice">
    <part element="tns:getPrice" name="parameters"></part>
  </message>
  <message name="getPriceResponse">
    <part element="tns:getPriceResponse" name="parameters"></part>
  </message>
```

```
  <portType name="Warehouse">
    <operation name="getPrice">
      <input message="tns:getPrice"></input>
      <output message="tns:getPriceResponse"></output>
    </operation>
  </portType>
```

```
  <binding name="WarehousePortBinding" type="tns:Warehouse">
    <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"></soap:bindi
    <operation name="getPrice">
      <soap:operation soapAction=""></soap:operation>
      <input><soap:body use="literal"></soap:body></input>
      <output><soap:body use="literal"></soap:body></output>
    </operation>
  </binding>
  <service name="WarehouseService">
    <port name="WarehousePort" binding="tns:WarehousePortBinding">
      <soap:address location="http://localhost:8080/WebServices/warehouse"></soap:address>
    </port>
  </service>
</definitions>
```

This description tells us that an operation `getPrice` is provided. Its input is a `tns:getPrice` and its output is a `tns:getPriceResponse` . (Here, `tns` is the namespace alias for the target namespace, http://corejava.horstmann.com .)

To understand these types, point your browser to `http://localhost:8080/WebServices/warehouse?xsd=1` . You will get this XSL document:

Code View:
```
<xs:schema targetNamespace="http://corejava.horstmann.com/" version="1.0">
   <xs:element name="getPrice" type="tns:getPrice"/>
   <xs:element name="getPriceResponse" type="tns:getPriceResponse"/>
   <xs:complexType name="getPrice">
      <xs:sequence><xs:element name="description" type="xs:string" minOccurs="0"/></xs:sequence>
   </xs:complexType>
   <xs:complexType name="getPriceResponse">
      <xs:sequence><xs:element name="return" type="xs:double"/></xs:sequence>
   </xs:complexType>
</xs:schema>
```

Now you can see that `getPrice` has a `description` element of type `string` , and `getPriceResponse` has a `retur` element of type `double` .

Note

The WSDL file does not specify *what* the service does. It only specifies the parameter and return types.

A Web Service Client

Let's turn to implementing the client. Keep in mind that the client knows nothing about the server except what is contained in the WSDL. To generate Java classes that can communicate with the server, you generate a set of clie classes, using the `wsimport` utility.

Code View:

```
wsimport -keep -p com.horstmann.corejava.server http://localhost:8080/WebServices/warehouse?wsdl
```

The `-keep` option keeps the source files, in case you want to look at them. The following classes and interfaces are generated:

```
GetPrice
GetPriceResponse
Warehouse
WarehouseService
ObjectFactory
```

You already saw the `GetPrice` and `GetPriceResponse` classes.

The `Warehouse` interface defines the remote `getPrice` method:

Code View:

```
public interface Warehouse
{
    @WebMethod public double getPrice(@WebParam(name = "description") String description);
}
```

You only need to know one thing about the `WarehouseService` class: its `getPort` method yields a stub of type `Warehouse` through which you invoke the service—see Listing 10-15 .

You can ignore the `ObjectFactory` class as well as the file `package-info.java` that defines a package-level annotation. (We discuss annotations in detail in Chapter 11 .)

Note

You can use any convenient package for the generated classes. If you look closely, you will notice that the `GetPrice` and `GetPriceResponse` classes are in different packages on the server and client. This is not a problem. After all, neither the server nor the client know about each other's Java implementation. They don't even know whether the other is implemented in Java.

Listing 10-15. **`WarehouseClient.java`**

Code View:
```
import java.rmi.*;
import javax.naming.*;
import com.horstmann.corejava.server.*;

/**
 * The client for the warehouse program.
 * @version 1.0 2007-10-09
 * @author Cay Horstmann
 */
public class WarehouseClient
{
   public static void main(String[] args) throws NamingException, RemoteException
   {
      WarehouseService service = new WarehouseService();
      Warehouse port = service.getPort(Warehouse.class);

      String descr = "Blackwell Toaster";
      double price = port.getPrice(descr);
      System.out.println(descr + ": " + price);
   }
}
```


Now you are ready to run the client program. Double-check that the server is still running, open another shell window, and execute

```
java WarehouseClient
```

You will get the familiar message about the price of a toaster.

Note

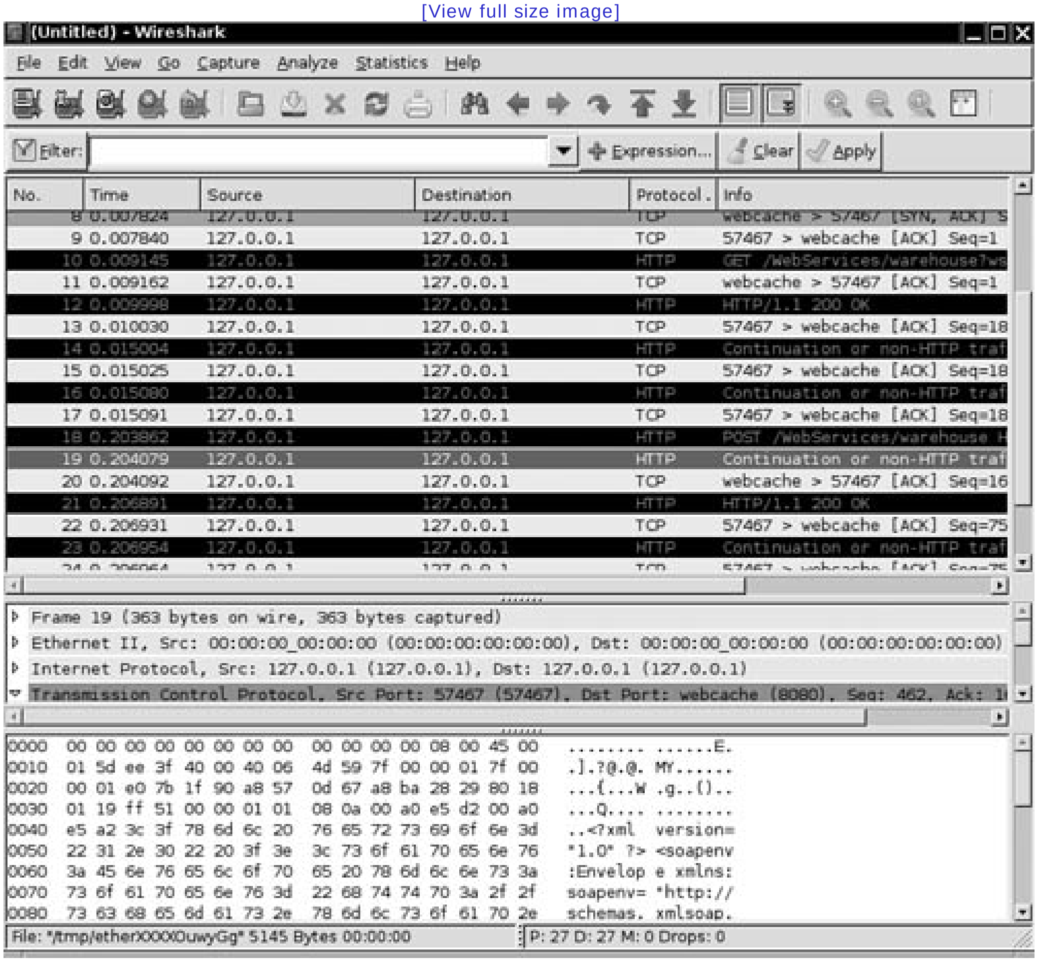
You might wonder why there is no equivalent of a RMI registry. When you locate a remote object for RMI, the client need not know on which server the object is located. It merely needs to know how to locate the registry. However, to make a web service call, the client needs the URL of the server. It is hardwired into the `WarehouseService` class.

We used a network sniffer to see how the client and server actually communicate (see Figure 10-8). The client se
the following request to the server:

Code View:
```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:ns1="http://corejava.horstmann.com/">
   <soapenv:Body>
      <ns1:getPrice><description>Blackwell Toaster</description></ns1:getPrice>
   </soapenv:Body>
</soapenv:Envelope>
```

Figure 10-8. Analyzing SOAP traffic

[View full size image]



The server responds:

Code View:
```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
```

```
   xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:ns1="http://corejava.horstmann.com/">
   <soapenv:Body>
      <ns1:getPriceResponse><return>24.95</return></ns1:getPriceResponse>
   </soapenv:Body>
</soapenv:Envelope>
```


In this section, you have seen the essentials about web services:

- The services are defined in a WSDL document, which is formatted as XML.

- The actual request and response methods use SOAP, another XML format.

- Clients and servers can be written in any language.

The Amazon E-Commerce Service

To make the discussion of web services more interesting, we look at a concrete example: the Amazon e-commerce web service, described at http://www.amazon.com/gp/aws/landing.html . The e-commerce web service allows a programmer to interact with the Amazon system for a wide variety of purposes. For example, you can get listings all books with a given author or title, or you can fill shopping carts and place orders. Amazon makes this service available for use by companies that want to sell items to their customers, using the Amazon system as a fulfillmen back end. To run our example program, you will need to sign up with Amazon and get a free developer token that lets you connect to the service.

Alternatively, you can adapt the technique described in this section to any other web service. The site http://www.xmethods.com lists many freely available web services that you can try.

Let us look more closely at the WSDL for the Amazon E-Commerce Service (located at http://webservices.amazon.com/AWSECommerceService/AWSECommerceService.wsdl). It describes an `ItemSear` operation as follows:

```
<operation name="ItemSearch">
   <input message="tns:ItemSearchRequestMsg"/>
   <output message="tns:ItemSearchResponseMsg"/>
</operation>

...

<message name="ItemSearchRequestMsg">
   <part name="body" element="tns:ItemSearch"/>
</message>
<message name="ItemSearchResponseMsg">
   <part name="body" element="tns:ItemSearchResponse"/>
</message>
```

Here are the definitions of the `ItemSearch` and `ItemSearchResponse` types:

Code View:

```
<xs:element name="ItemSearch">
```

```
   <xs:complexType>
      <xs:sequence>
         <xs:element name="MarketplaceDomain" type="xs:string" minOccurs="0"/>
         <xs:element name="AWSAccessKeyId" type="xs:string" minOccurs="0"/>
         <xs:element name="SubscriptionId" type="xs:string" minOccurs="0"/>
         <xs:element name="AssociateTag" type="xs:string" minOccurs="0"/>
         <xs:element name="XMLEscaping" type="xs:string" minOccurs="0"/>
         <xs:element name="Validate" type="xs:string" minOccurs="0"/>
         <xs:element name="Shared" type="tns:ItemSearchRequest" minOccurs="0"/>
         <xs:element name="Request" type="tns:ItemSearchRequest" minOccurs="0"
            maxOcurs="unbounded"/>
      </xs:sequence>
   </xs:complexType>
</xs:element>
<xs:element name="ItemSearchResponse">
   <xs:complexType>
      <xs:sequence>
         <xs:element ref="tns:OperationRequest" minOccurs="0"/>
         <xs:element ref="tns:Items" minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
   </xs:complexType>
</xs:element>
```

Using the JAX-WS technology, the `ItemSearch` operation becomes a method call:

Code View:

```
void itemSearch(String marketPlaceDomain, String awsAccessKeyId,
   String subscriptionId, String associateTag, String xmlEscaping, String validate,
   ItemSearchRequest shared, List<ItemSearchRequest> request,
   Holder<OperationRequest> opHolder, Holder<List<Items>> responseHolder)
```

The `ItemSearchRequest` parameter type is defined as

Code View:

```
<xs:complexType name="ItemSearchRequest">
   <xs:sequence>
      <xs:element name="Actor" type="xs:string" minOccurs="0"/>
      <xs:element name="Artist" type="xs:string" minOccurs="0"/>
      . . .
      <xs:element name="Author" type="xs:string" minOccurs="0"/>
      . . .
      <xs:element name="ResponseGroup" type="xs:string" minOccurs="0" maxOccurs="unbounded"/>
      . . .
      <xs:element name="SearchIndex" type="xs:string" minOccurs="0"/>
      . . .
</xs:complexType>
```

This description is translated into a class.

```
public class ItemSearchRequest
{
   public ItemSearchRequest() { ... }
   public String getActor() { ... }
   public void setActor(String newValue) { ... }
   public String getArtist() { ... }
   public void setArtist(String newValue) { ... }
   ...
   public String getAuthor() { ... }
   public void setAuthor(String newValue) { ... }
   ...
   public List<String> getResponseGroup() { ... }
   ...
   public void setSearchIndex(String newValue) { ... }
   ...
}
```


To invoke the search service, construct an `ItemSearchRequest` object and call the `itemSearch` method of the "por object.

Code View:

```
ItemSearchRequest request = new ItemSearchRequest();
request.getResponseGroup().add("ItemAttributes");
request.setSearchIndex("Books");
Holder<List<Items>> responseHolder = new Holder<List<Items>>();
request.setAuthor(name);
port.itemSearch("", accessKey, "", "", "", "", request, null, null, responseHolder);
```

The port object translates the Java object into a SOAP message, passes it to the Amazon server, translates the returned message into a `ItemSearchResponse` object, and places the response in the "holder" object.

Note

The Amazon documentation about the parameters and return values is extremely sketchy. However, you can fill out forms at http://awszone.com/scratchpads/index.aws to see the SOAP requests and responses. Those help you guess what parameter values you need to supply and what return values you can expect.

Our sample application (in Listing 10-16) is straightforward. The user specifies an author name and clicks the Sea button. We simply show the first page of the response (see Figure 10-9). This shows that the web service is successful. We leave it as the proverbial exercise for the reader to extend the functionality of the application.

Figure 10-9. Connecting to a web service

To run this program, you first generate the client-side artifact classes:

Code View:
```
wsimport -p com.horstmann.amazon
   http://webservices.amazon.com/AWSECommerceService/AWSECommerceService.wsdl
```

Then edit the `AmazonTest.java` file to include your Amazon key, compile, and run:

```
javac AmazonTest.java
java AmazonTest
```

Listing 10-16. **AmazonTest.java**

Code View:
```
import com.horstmann.amazon.*;
import java.awt.*;
import java.awt.event.*;
import java.util.List;
import javax.swing.*;
import javax.xml.ws.*;

/**
  * The client for the Amazon e-commerce test program.
  * @version 1.10 2007-10-20
  * @author Cay Horstmann
  */

public class AmazonTest
```

```
{
   public static void main(String[] args)
   {
      JFrame frame = new AmazonTestFrame();
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.setVisible(true);
   }
}

```



```
/**
 * A frame to select the book author and to display the server response.
 */
class AmazonTestFrame extends JFrame
{
   public AmazonTestFrame()
```



```
   {
      setTitle("AmazonTest");
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

      JPanel panel = new JPanel();

      panel.add(new JLabel("Author:"));
      author = new JTextField(20);
      panel.add(author);

      JButton searchButton = new JButton("Search");
      panel.add(searchButton);
      searchButton.addActionListener(new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               result.setText("Please wait...");
               new SwingWorker<Void, Void>()
                  {
                     @Override
                     protected Void doInBackground() throws Exception
                     {
                        String name = author.getText();
                        String books = searchByAuthor(name);
                        result.setText(books);
                        return null;
                     }
                  }.execute();
            }
         });

      result = new JTextArea();
      result.setLineWrap(true);
      result.setEditable(false);

      if (accessKey.equals("your key here"))
      {
         result.setText("You need to edit the Amazon access key.");
         searchButton.setEnabled(false);
      }

      add(panel, BorderLayout.NORTH);
      add(new JScrollPane(result), BorderLayout.CENTER);
   }
```

```

   /**
    * Calls the Amazon web service to find titles that match the author.
    * @param name the author name
    * @return a description of the matching titles
    */
   private String searchByAuthor(String name)
   {
      AWSECommerceService service = new AWSECommerceService();
      AWSECommerceServicePortType port = service.getPort(AWSECommerceServicePortType.class);
      ItemSearchRequest request = new ItemSearchRequest();
      request.getResponseGroup().add("ItemAttributes");
      request.setSearchIndex("Books");

      Holder<List<Items>> responseHolder = new Holder<List<Items>>();
      request.setAuthor(name);
      port.itemSearch("", accessKey, "", "", "", "", request, null, null, responseHolder);

      List<Item> response = responseHolder.value.get(0).getItem();

      StringBuilder r = new StringBuilder();
      for (Item item : response)
      {
         r.append("authors=");
         List<String> authors = item.getItemAttributes().getAuthor();
         r.append(authors);
         r.append(",title=");
         r.append(item.getItemAttributes().getTitle());
         r.append(",publisher=");
         r.append(item.getItemAttributes().getPublisher());
         r.append(",pubdate=");
         r.append(item.getItemAttributes().getPublicationDate());
         r.append("\n");
      }
      return r.toString();
   }

   private static final int DEFAULT_WIDTH = 450;
   private static final int DEFAULT_HEIGHT = 300;

   private static final String accessKey = "12Y1EEATQ8DDYJCVQYR2";

   private JTextField author;
   private JTextArea result;
}
```

This example shows that calling a web service is fundamentally the same as making any other remote method call The programmer calls a local method on a proxy object, and the proxy connects to a server. Because web services are springing up everywhere, this is clearly an interesting technology for application programmers.

You have now seen the RMI mechanism, a sophisticated distributed programming model for Java programs that is used extensively in the Java EE architecture. You have also had an introduction into web services, which allow you connect clients and servers, independent of the programming language. In the next chapter, we turn to a different aspect of Java programming: interacting with "native" code in a different programming language on the same machine.

Chapter 11. Scripting, Compiling, and Annotation Processing

- Scripting for the Java Platform
- The Compiler API
- Using Annotations
- Annotation Syntax
- Standard Annotations
- Source-Level Annotation Processing
- Bytecode Engineering

This chapter introduces three techniques for processing code. The scripting API lets you invoke code in a scripting language such as JavaScript or Groovy. You use the compiler API when you want to compile Java code inside your application. Annotation processors operate on Java source or class files that contain annotations. As you will see, there are many applications for annotation processing, ranging from simple diagnostics to "bytecode engineering," the insertion of byte codes into class files or even running programs.

Scripting for the Java Platform

A scripting language is a language that avoids the usual edit/compile/link/run cycle by interpreting program text at runtime. Scripting languages have a number of advantages:

- Rapid turnaround, encouraging experimentation.
- Changing the behavior of a running program.
- Enabling customization by program users.

On the other hand, most scripting languages lack features that are beneficial for programming complex applications, such as strong typing, encapsulation, and modularity.

It is therefore tempting to combine the advantages of scripting and traditional languages. The scripting API lets you do just that for the Java platform. It enables you to invoke scripts written in JavaScript, Groovy, Ruby, and even exotic languages such as Scheme and Haskell, from a Java program. (The other direction, accessing Java from the scripting language, is the responsibility of the scripting language provider. Most scripting languages that run on the Java virtual machine have this capability.)

In the following sections, we show you how to select an engine for a particular language, how to execute scripts, and how to take advantage of advanced features that some scripting engines offer.

Getting a Scripting Engine

A scripting engine is a library that can execute scripts in a particular language. When the virtual machine starts, it discovers the available scripting engines. To enumerate them, construct a `ScriptEngineManager` and invoke the `getEngineFactories` method. You can ask each engine factory for the supported engine names, MIME types, and file extensions. Table 11-1 shows typical values.

Table 11-1. Properties of Scripting Engine Factories

Engine	Names	MIME types	Extensions
Rhino (included in Java SE 6)	js, rhino, JavaScript, javascript, ECMAScript, ecmascript	application/javascript, application/ecmascript, text/javascript, text/ecmascript]	js
Groovy	groovy	None	groovy
SISC Scheme	scheme, sisc	None	scc, sce, scm, shp

Usually, you know which engine you need, and you can simply request it by name, MIME type, or extension. For example,

```
ScriptEngine engine = manager.getEngineByName("JavaScript");
```

Java SE 6 includes a version of Rhino, a JavaScript interpreter developed by the Mozilla foundation. You can add additional languages by providing the necessary JAR files on the class path. You will generally need two sets of JAR files. The scripting language itself is implemented by a single JAR file or a set of JARs. The engine that adapts the language to the scripting API usually requires an additional JAR. The site http://scripting.dev.java.net provides engines for a wide range of scripting languages. For example, to add support for Groovy, the class path should contain *groovy/lib/** (from http://groovy.codehaus.org) and `groovy-engine.jar` (from http://scripting.dev.java.net).

javax.script.ScriptEngineManager 6

- List<ScriptEngineFactory> getEngineFactories()

  gets a list of all discovered engine factories.

- ScriptEngine getEngineByName(String name)

- ScriptEngine getEngineByExtension(String extension)

- ScriptEngine getEngineByMimeType(String mimeType)

  gets the script engine with the given name, script file extension, or MIME type.

javax.script.ScriptEngineFactory 6

- List<String> getNames()

- List<String> getExtensions()

- List<String> getMimeTypes()

  gets the names, script file extensions, and MIME types under which this factory is known.

Script Evaluation and Bindings

Once you have an engine, you can call a script simply by invoking

```
Object result = engine.eval(scriptString);
```

If the script is stored in a file, then open a Reader and call

```
Object result = engine.eval(reader);
```

You can invoke multiple scripts on the same engine. If one script defines variables, functions, or classes, most scripting engines retain the definitions for later use. For example,

```
engine.eval("n = 1728");
```

```
Object result = engine.eval("n + 1");
```

will return 1729.

Note

To find out whether it is safe to concurrently execute scripts in multiple threads, call

```
Object param = factory.getParameter("THREADING");
```


The returned value is one of the following:

- `null`: Concurrent execution is not safe
- `"MULTITHREADED"`: Concurrent execution is safe. Effects from one thread might be visible from another thread.
- `"THREAD-ISOLATED"`: In addition to `"MULTITHREADED"`, different variable bindings are maintained for each thread.
- `"STATELESS"`: In addition to `"THREAD-ISOLATED"`, scripts do not alter variable bindings.

You often want to add variable bindings to the engine. A binding consists of a name and an associated Java object. For example, consider these statements:

```
engine.put(k, 1728);
Object result = engine.eval("k + 1");
```

The script code reads the definition of `k` from the bindings in the "engine scope." This is particularly important because most scripting languages can access Java objects, often with a syntax that is simpler than the Java syntax. For example,

```
engine.put(b, new JButton());
engine.eval("f.text = 'Ok'");
```

Conversely, you can retrieve variables that were bound by scripting statements:

```
engine.eval("n = 1728");
Object result = engine.get("n");
```

In addition to the engine scope, there is also a global scope. Any bindings that you add to the `ScriptEngineManager` are visible to all engines.

Instead of adding bindings to the engine or global scope, you can collect them in an object of type `Bindings` and pass them to the `eval` method:

```
Bindings scope = engine.createBindings();
scope.put(b, new JButton());
engine.eval(scriptString, scope);
```

This is useful if a set of bindings should not persist for future calls to the `eval` method.

Note

You might want to have scopes other than the engine and global scopes. For example, a web container might need request and session scopes. However, then you are on your own. You need to implement a class that implements the `ScriptContext` interface, managing a collection of scopes. Each scope is identified by an integer number, and scopes with lower numbers should be searched first. (The standard library provides a `SimpleScriptContext` class, but it only holds global and engine scopes.)

API

`javax.script.ScriptEngine` 6

- `Object eval(String script)`
- `Object eval(Reader reader)`
- `Object eval(String script, Bindings bindings)`
- `Object eval(Reader reader, Bindings bindings)`

  evaluates the script given by the string or reader, subject to the given bindings.

- `Object get(String key)`
- `void put(String key, Object value)`

  gets or puts a binding in the engine scope.

- `Bindings createBindings()`

  creates an empty `Bindings` object suitable for this engine.

API `javax.script.ScriptEngineManager` 6

- `Object get(String key)`

- `void put(String key, Object value)`

  gets or puts a binding in the global scope.

API `javax.script.Bindings` 6

- `Object get(String key)`
- `void put(String key, Object value)`

  gets or puts a binding into the scope represented by this `Bindings` object.

Redirecting Input and Output

You can redirect the standard input and output of a script by calling the `setReader` and `setWriter` method of the script context. For example,

```
StringWriter writer = new StringWriter();
engine.getContext().setWriter(new PrintWriter(writer, true));
```

Any output written with the JavaScript `print` or `println` functions is sent to `writer`.

Caution

You can pass any `Writer` to the `setWriter` method, but the Rhino engine throws an exception if it is not a `PrintWriter`.

The `setReader` and `setWriter` methods only affect the scripting engine's standard input and output sources. For example, if you execute the JavaScript code

```
println("Hello");
java.lang.System.out.println("World");
```

only the first output is redirected.

The Rhino engine does not have the notion of a standard input source. Calling `setReader` has no effect.

`javax.script.ScriptEngine` 6

- `ScriptContext getContext()`

  gets the default script context for this engine.

`javax.script.ScriptContext` 6

- `Reader getReader()`
- `void setReader(Reader reader)`
- `Writer getWriter()`
- `void setWriter(Writer writer)`
- `Writer getErrorWriter()`
- `void setErrorWriter(Writer writer)`

  gets or sets the reader for input or writer for normal or error output.

Calling Scripting Functions and Methods

With many script engines, you can invoke a function in the scripting language without having to evaluate the actual script code. This is useful if you allow users to implement a service in a scripting language of their choice.

The script engines that offer this functionality implement the `Invocable` interface. In particular, the Rhino engine implements `Invocable`.

To call a function, call the `invokeFunction` method with the function name, followed by the function parameters:

```
if (engine implements Invocable)
   ((Invocable) engine).invokeFunction("aFunction", param1, param2);
```


If the scripting language is object oriented, you call can a method like this:

Code View:
```
((Invocable) engine).invokeMethod(implicitParam, "aMethod", explicitParam1, explicitParam2);
```

Here, the `implicitParam` object is a proxy to an object in the scripting language. It must be the result of a prior call to the scripting engine.

Note

If the script engine does not implement the `Invocable` interface, you might still be able to call a method in a language-independent way. The `getMethodCallSyntax` method of the `ScriptEngineFactory` class produces a string that you can pass to the `eval` method. However, all method parameters must be bound to names, whereas `invokeMethod` can be called with arbitrary values.

You can go a step further and ask the scripting engine to implement a Java interface. Then you can call scripting functions and methods with the Java method call syntax.

The details depend on the scripting engine, but typically you need to supply a function for each method of the interface. For example, consider a Java interface

```
public interface Greeter
{
   String greet(String whom);
}
```

In Rhino, you provide a function

```
function greet(x) { return "Hello, " + x + "!"; }
```

This code must be evaluated first. Then you can call

```
Greeter g = ((Invocable) engine).getInterface(Greeter.class);
```

Now you can make a plain Java method call

```
String result = g.greet("World");
```

Behind the scenes, the JavaScript `greet` method is invoked. This approach is similar to making a remote method call, as discussed in Chapter 10.

In an object-oriented scripting language, you can access a script class through a matching Java interface. For example, consider this JavaScript code, which defines a `SimpleGreeter` class.

Code View:
```
function SimpleGreeter(salutation) { this.salutation = salutation; }
SimpleGreeter.prototype.greet = function(whom) { return this.salutation + ", " + whom + "!"; }
```

You can use this class to construct greeters with different salutations (such as Hello, Goodbye, and so on).

Note

For more information on how to define classes in JavaScript, see *JavaScript—The Definitive Guide*, 5th ed., by David Flanagan (O'Reilly 2006).

After evaluating the JavaScript class definition, call

Code View:
```
Object goodbyeGreeter = engine.eval("new SimpleGreeter('Goodbye')");
Greeter g = ((Invocable) engine).getInterface(goodbyeGreeter, Greeter.class);
```

When you call `g.greet("World")`, the `greet` method is invoked on the JavaScript object `goodbyeGreeter`. The result is a string `"Goodbye, World!"`.

In summary, the `Invocable` interface is useful if you want to call scripting code from Java without worrying about the scripting language syntax.

javax.script.Invocable 6

- Object invokeFunction(String name, Object... parameters)

- Object invokeMethod(Object implicitParameter, String name, Object... explicitParameters)

  invokes the function or method with the given name, passing the given parameters.

- <T> T getInterface(Class<T> iface)

  returns an implementation of the given interface, implementing the methods with functions in the scripting engine.

- <T> T getInterface(Object implicitParameter, Class<T> iface)

  returns an implementation of the given interface, implementing the methods with methods of the given object.

Compiling a Script

Some scripting engines can compile scripting code into an intermediate form for efficient execution. Those engines implement the Compilable interface. The following example shows how to compile and evaluate code that is contained in a script file:

```
Reader reader = new FileReader("myscript.js");
CompiledScript script = null;
if (engine implements Compilable)
   CompiledScript script = ((Compilable) engine).compile(reader);
```

Once the script is compiled, you can execute it. The following code executes the compiled script if compilation was successful, or the original script if the engine didn't support compilation.

```
if (script != null)
   script.eval();
else
   engine.eval(reader);
```

Of course, you only want to compile a script if you need to execute it repeatedly.

javax.script.Compilable 6

- `CompiledScript compile(String script)`
- `CompiledScript compile(Reader reader)`

  compiles the script given by a string or reader.

javax.script.CompiledScript 6

- `Object eval()`
- `Object eval(Bindings bindings)`

  evaluates this script.

An Example: Scripting GUI Events

To illustrate the scripting API, we will develop a sample program that allows users to specify event handlers in a scripting language of their choice.

Have a look at the program in Listing 11-1. The `ButtonFrame` class is similar to the event handling demo in Volume I, with two differences:

- Each component has its `name` property set.
- There are no event handlers.

The event handlers are defined in a properties file. Each property definition has the form

componentName . *eventName* = *scriptCode*

For example, if you choose to use JavaScript, you supply the event handlers in a file `js.properties`, like this:

```
yellowButton.action=panel.background = java.awt.Color.YELLOW
blueButton.action=panel.background = java.awt.Color.BLUE
redButton.action=panel.background = java.awt.Color.RED
```

The companion code also has files for Groovy and SISC Scheme.

The program starts by loading an engine for the language that is specified on the command line. If no language is specified, we use JavaScript.

We then process a script `init.`*language* if it is present. This seems like a good idea in general. Moreover, the Scheme interpreter needs some cumbersome initializations that we did not want to include in every event handler script.

Next, we recursively traverse all child components and add the bindings (*name*, *object*) into the engine scope.

Then we read the file *language*`.properties`. For each property, we synthesize an event handler proxy that causes the script code to be executed. The details are a bit technical. You might want to read the section on proxies in Volume I, Chapter 6, together with the section on JavaBeans events in Chapter 8 of this volume, if you want follow the implementation in detail. The essential part, however, is that each event handler calls

```
engine.eval(scriptCode);
```


Let us look at the `yellowButton` in more detail. When the line

```
yellowButton.action=panel.background = java.awt.Color.YELLOW
```

is processed, we find the `JButton` component with the name `"yellowButton"`. We then attach an `ActionListener` with an `actionPerformed` method that executes the script

```
panel.background = java.awt.Color.YELLOW
```

The engine contains a binding that binds the name `"panel"` to the `JPanel` object. When the event occurs, the `setBackground` method of the panel is executed, and the color changes.

You can run this program with the JavaScript event handlers, simply by executing

```
java ScriptTest
```

For the Groovy handlers, use

Code View:
```
java -classpath .:groovy/lib/*:jsr223-engines/groovy/build/groovy-engine.jar ScriptTest groovy
```

Here, *groovy* is the directory into which you installed Groovy, and *jsr223-engines* is the directory that contains the engine adapters from http://scripting.dev.java.net.

To try out Scheme, download SISC Scheme from http://sisc-scheme.org/ and run

Code View:
```
java -classpath .:sisc/*:jsr223-engines/scheme/build/scheme-engine.jar ScriptTest scheme
```

This application demonstrates how to use scripting for Java GUI programming. One could go one step further and describe the GUI with an XML file, as you have seen in Chapter 2. Then our program would become an interpreter for GUIs that have visual presentation defined by XML and behavior defined by a scripting language. Note the similarity to a dynamic HTML page or a dynamic server-side scripting environment.

Listing 11-1. **ScriptTest.java**

Code View:

```
import java.awt.*;
import java.beans.*;
import java.io.*;
import java.lang.reflect.*;
import java.util.*;
import javax.script.*;
import javax.swing.*;

/**
 * @version 1.00 2007-10-28
 * @author Cay Horstmann
 */
public class ScriptTest
{
   public static void main(final String[] args)
   {
      EventQueue.invokeLater(new Runnable()
         {
            public void run()
            {
               String language;
               if (args.length == 0) language = "js";
               else language = args[0];

               ScriptEngineManager manager = new ScriptEngineManager();
               System.out.println("Available factories: ");
               for (ScriptEngineFactory factory : manager.getEngineFactories())
                  System.out.println(factory.getEngineName());
               final ScriptEngine engine = manager.getEngineByName(language);

               if (engine == null)
               {
                  System.err.println("No engine for " + language);
                  System.exit(1);
               }

               ButtonFrame frame = new ButtonFrame();

               try
               {
                  File initFile = new File("init." + language);
                  if (initFile.exists())
                  {
                     engine.eval(new FileReader(initFile));
                  }

                  getComponentBindings(frame, engine);

                  final Properties events = new Properties();
                  events.load(new FileReader(language + ".properties"));
                  for (final Object e : events.keySet())
                  {
                     String[] s = ((String) e).split("\\.");
```

```
                     addListener(s[0], s[1], (String) events.get(e), engine);
                  }
               }
               catch (Exception e)
               {
                  e.printStackTrace();
               }

               frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```



```
               frame.setTitle("ScriptTest");
               frame.setVisible(true);
            }
         });
   }

   /**
```



```
    * Gathers all named components in a container.
    * @param c the component
    * @param namedComponents
    */
   private static void getComponentBindings(Component c, ScriptEngine engine)
   {
      String name = c.getName();
      if (name != null) engine.put(name, c);
      if (c instanceof Container)
      {
         for (Component child : ((Container) c).getComponents())
            getComponentBindings(child, engine);
      }
   }

   /**
    * Adds a listener to an object whose listener method executes a script.
    * @param beanName the name of the bean to which the listener should be added
    * @param eventName the name of the listener type, such as "action" or "change"
    * @param scriptCode the script code to be executed
    * @param engine the engine that executes the code
    * @param bindings the bindings for the execution
    */
   private static void addListener(String beanName, String eventName, final String scriptCode,
         final ScriptEngine engine) throws IllegalArgumentException, IntrospectionException,
         IllegalAccessException, InvocationTargetException
   {
      Object bean = engine.get(beanName);
      EventSetDescriptor descriptor = getEventSetDescriptor(bean, eventName);
      if (descriptor == null) return;
      descriptor.getAddListenerMethod().invoke(
            bean,
            Proxy.newProxyInstance(null, new Class[] { descriptor.getListenerType() },
                  new InvocationHandler()
                     {
                        public Object invoke(Object proxy, Method method, Object[] args)
                              throws Throwable
                        {
                           engine.eval(scriptCode);
                           return null;
                        }
                     }));

```

```
   }

   private static EventSetDescriptor getEventSetDescriptor(Object bean, String eventName)
         throws IntrospectionException
   {
      for (EventSetDescriptor descriptor : Introspector.getBeanInfo(bean.getClass())
            .getEventSetDescriptors())
         if (descriptor.getName().equals(eventName)) return descriptor;
      return null;
   }
}

class ButtonFrame extends JFrame
{
   public ButtonFrame()
   {
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

      panel = new JPanel();
      panel.setName("panel");
      add(panel);

      yellowButton = new JButton("Yellow");
      yellowButton.setName("yellowButton");
      blueButton = new JButton("Blue");
      blueButton.setName("blueButton");
      redButton = new JButton("Red");
      redButton.setName("redButton");

      panel.add(yellowButton);
      panel.add(blueButton);
      panel.add(redButton);
   }

   public static final int DEFAULT_WIDTH = 300;
   public static final int DEFAULT_HEIGHT = 200;

   private JPanel panel;
   private JButton yellowButton;
   private JButton blueButton;
   private JButton redButton;
}
```

Chapter 11. Scripting, Compiling, and Annotation Processing

- Scripting for the Java Platform
- The Compiler API
- Using Annotations
- Annotation Syntax
- Standard Annotations
- Source-Level Annotation Processing
- Bytecode Engineering

UNREGISTERED VERSION OF CHM TO PDF CONVERTER PRO BY THETA-SOFTWARE

This chapter introduces three techniques for processing code. The scripting API lets you invoke code in a scripting language such as JavaScript or Groovy. You use the compiler API when you want to compile Java code inside your application. Annotation processors operate on Java source or class files that contain annotations. As you will see, there are many applications for annotation processing, ranging from simple diagnostics to "bytecode engineering," the insertion of byte codes into class files or even running programs.

Scripting for the Java Platform

A scripting language is a language that avoids the usual edit/compile/link/run cycle by interpreting program text at runtime. Scripting languages have a number of advantages:

- Rapid turnaround, encouraging experimentation.
- Changing the behavior of a running program.
- Enabling customization by program users.

On the other hand, most scripting languages lack features that are beneficial for programming complex applications, such as strong typing, encapsulation, and modularity.

It is therefore tempting to combine the advantages of scripting and traditional languages. The scripting API lets you do just that for the Java platform. It enables you to invoke scripts written in JavaScript, Groovy, Ruby, and even exotic languages such as Scheme and Haskell, from a Java program. (The other direction, accessing Java from the scripting language, is the responsibility of the scripting language provider. Most scripting languages that run on the Java virtual machine have this capability.)

In the following sections, we show you how to select an engine for a particular language, how to execute scripts, and how to take advantage of advanced features that some scripting engines offer.

Getting a Scripting Engine

A scripting engine is a library that can execute scripts in a particular language. When the virtual machine starts, it discovers the available scripting engines. To enumerate them, construct a `ScriptEngineManager` and invoke the `getEngineFactories` method. You can ask each engine factory for the supported engine names, MIME types, and file extensions. Table 11-1 shows typical values.

Table 11-1. Properties of Scripting Engine Factories

Engine	Names	MIME types	Extensions
Rhino (included in Java SE 6)	js, rhino, JavaScript, javascript, ECMAScript, ecmascript	application/javascript, application/ecmascript, text/javascript, text/ecmascript]	js
Groovy	groovy	None	groovy
SISC Scheme	scheme, sisc	None	scc, sce, scm, shp

Usually, you know which engine you need, and you can simply request it by name, MIME type, or extension. For example,

```
ScriptEngine engine = manager.getEngineByName("JavaScript");
```

Java SE 6 includes a version of Rhino, a JavaScript interpreter developed by the Mozilla foundation. You can add additional languages by providing the necessary JAR files on the class path. You will generally need two sets of JAR files. The scripting language itself is implemented by a single JAR file or a set of JARs. The engine that adapts the language to the scripting API usually requires an additional JAR. The site http://scripting.dev.java.net provides engines for a wide range of scripting languages. For example, to add support for Groovy, the class path should contain *groovy/lib/** (from http://groovy.codehaus.org) and `groovy-engine.jar` (from http://scripting.dev.java.net).

javax.script.ScriptEngineManager 6

- List<ScriptEngineFactory> getEngineFactories()

  gets a list of all discovered engine factories.

- ScriptEngine getEngineByName(String name)
- ScriptEngine getEngineByExtension(String extension)

- ScriptEngine getEngineByMimeType(String mimeType)

  gets the script engine with the given name, script file extension, or MIME type.

javax.script.ScriptEngineFactory 6

- List<String> getNames()
- List<String> getExtensions()
- List<String> getMimeTypes()

  gets the names, script file extensions, and MIME types under which this factory is known.

Script Evaluation and Bindings

Once you have an engine, you can call a script simply by invoking

```
Object result = engine.eval(scriptString);
```

If the script is stored in a file, then open a `Reader` and call

```
Object result = engine.eval(reader);
```

You can invoke multiple scripts on the same engine. If one script defines variables, functions, or classes, most scripting engines retain the definitions for later use. For example,

```
engine.eval("n = 1728");
```

```
Object result = engine.eval("n + 1");
```

will return 1729.

Note

To find out whether it is safe to concurrently execute scripts in multiple threads, call

```
Object param = factory.getParameter("THREADING");
```

The returned value is one of the following:

- `null`: Concurrent execution is not safe
- `"MULTITHREADED"`: Concurrent execution is safe. Effects from one thread might be visible from another thread.
- `"THREAD-ISOLATED"`: In addition to `"MULTITHREADED"`, different variable bindings are maintained for each thread.
- `"STATELESS"`: In addition to `"THREAD-ISOLATED"`, scripts do not alter variable bindings.

You often want to add variable bindings to the engine. A binding consists of a name and an associated Java object. For example, consider these statements:

```
engine.put(k, 1728);
Object result = engine.eval("k + 1");
```

The script code reads the definition of `k` from the bindings in the "engine scope." This is particularly important because most scripting languages can access Java objects, often with a syntax that is simpler than the Java syntax. For example,

```
engine.put(b, new JButton());
engine.eval("f.text = 'Ok'");
```

Conversely, you can retrieve variables that were bound by scripting statements:

```
engine.eval("n = 1728");
Object result = engine.get("n");
```

In addition to the engine scope, there is also a global scope. Any bindings that you add to the `ScriptEngineManager` are visible to all engines.

Instead of adding bindings to the engine or global scope, you can collect them in an object of type `Bindings` and pass them to the `eval` method:

```
Bindings scope = engine.createBindings();
scope.put(b, new JButton());
engine.eval(scriptString, scope);
```


This is useful if a set of bindings should not persist for future calls to the `eval` method.

Note

You might want to have scopes other than the engine and global scopes. For example, a web container might need request and session scopes. However, then you are on your own. You need to implement a class that implements the `ScriptContext` interface, managing a collection of scopes. Each scope is identified by an integer number, and scopes with lower numbers should be searched first. (The standard library provides a `SimpleScriptContext` class, but it only holds global and engine scopes.)

API

`javax.script.ScriptEngine` 6

- `Object eval(String script)`
- `Object eval(Reader reader)`
- `Object eval(String script, Bindings bindings)`
- `Object eval(Reader reader, Bindings bindings)`

  evaluates the script given by the string or reader, subject to the given bindings.

- `Object get(String key)`
- `void put(String key, Object value)`

  gets or puts a binding in the engine scope.

- `Bindings createBindings()`

  creates an empty `Bindings` object suitable for this engine.

API `javax.script.ScriptEngineManager` **6**

- `Object get(String key)`
- `void put(String key, Object value)`

  gets or puts a binding in the global scope.

API `javax.script.Bindings` **6**

- `Object get(String key)`
- `void put(String key, Object value)`

  gets or puts a binding into the scope represented by this `Bindings` object.

Redirecting Input and Output

You can redirect the standard input and output of a script by calling the `setReader` and `setWriter` method of the script context. For example,

```
StringWriter writer = new StringWriter();
engine.getContext().setWriter(new PrintWriter(writer, true));
```

Any output written with the JavaScript `print` or `println` functions is sent to `writer`.

Caution

You can pass any `Writer` to the `setWriter` method, but the Rhino engine throws an exception if it is not a `PrintWriter`.

The `setReader` and `setWriter` methods only affect the scripting engine's standard input and output sources. For example, if you execute the JavaScript code

```
println("Hello");
java.lang.System.out.println("World");
```


only the first output is redirected.

The Rhino engine does not have the notion of a standard input source. Calling `setReader` has no effect.

API `javax.script.ScriptEngine` 6

- `ScriptContext getContext()`

  gets the default script context for this engine.

API `javax.script.ScriptContext` 6

- `Reader getReader()`
- `void setReader(Reader reader)`
- `Writer getWriter()`
- `void setWriter(Writer writer)`
- `Writer getErrorWriter()`
- `void setErrorWriter(Writer writer)`

  gets or sets the reader for input or writer for normal or error output.

Calling Scripting Functions and Methods

With many script engines, you can invoke a function in the scripting language without having to evaluate the actual script code. This is useful if you allow users to implement a service in a scripting language of their choice.

The script engines that offer this functionality implement the `Invocable` interface. In particular, the Rhino engine implements `Invocable`.

To call a function, call the `invokeFunction` method with the function name, followed by the function parameters:

```
if (engine implements Invocable)
   ((Invocable) engine).invokeFunction("aFunction", param1, param2);
```

If the scripting language is object oriented, you call can a method like this:

Code View:
```
((Invocable) engine).invokeMethod(implicitParam, "aMethod", explicitParam1, explicitParam2);
```

Here, the `implicitParam` object is a proxy to an object in the scripting language. It must be the result of a prior call to the scripting engine.

Note

If the script engine does not implement the `Invocable` interface, you might still be able to call a method in a language-independent way. The `getMethodCallSyntax` method of the `ScriptEngineFactory` class produces a string that you can pass to the `eval` method. However, all method parameters must be bound to names, whereas `invokeMethod` can be called with arbitrary values.

You can go a step further and ask the scripting engine to implement a Java interface. Then you can call scripting functions and methods with the Java method call syntax.

The details depend on the scripting engine, but typically you need to supply a function for each method of the interface. For example, consider a Java interface

```
public interface Greeter
{
   String greet(String whom);
}
```

In Rhino, you provide a function

```
function greet(x) { return "Hello, " + x + "!"; }
```

This code must be evaluated first. Then you can call

```
Greeter g = ((Invocable) engine).getInterface(Greeter.class);
```

Now you can make a plain Java method call

```
String result = g.greet("World");
```


Behind the scenes, the JavaScript `greet` method is invoked. This approach is similar to making a remote method call, as discussed in Chapter 10.

In an object-oriented scripting language, you can access a script class through a matching Java interface. For example, consider this JavaScript code, which defines a `SimpleGreeter` class.

Code View:
```
function SimpleGreeter(salutation) { this.salutation = salutation; }
SimpleGreeter.prototype.greet = function(whom) { return this.salutation + ", " + whom + "!"; }
```

You can use this class to construct greeters with different salutations (such as Hello, Goodbye, and so on).

Note

For more information on how to define classes in JavaScript, see *JavaScript—The Definitive Guide*, 5th ed., by David Flanagan (O'Reilly 2006).

After evaluating the JavaScript class definition, call

Code View:
```
Object goodbyeGreeter = engine.eval("new SimpleGreeter('Goodbye')");
Greeter g = ((Invocable) engine).getInterface(goodbyeGreeter, Greeter.class);
```

When you call `g.greet("World")`, the `greet` method is invoked on the JavaScript object `goodbyeGreeter`. The result is a string `"Goodbye, World!"`.

In summary, the `Invocable` interface is useful if you want to call scripting code from Java without worrying about the scripting language syntax.

javax.script.Invocable 6

- Object invokeFunction(String name, Object... parameters)

- Object invokeMethod(Object implicitParameter, String name, Object... explicitParameters)

  invokes the function or method with the given name, passing the given parameters.

- <T> T getInterface(Class<T> iface)

  returns an implementation of the given interface, implementing the methods with functions in the scripting engine.

- <T> T getInterface(Object implicitParameter, Class<T> iface)

  returns an implementation of the given interface, implementing the methods with methods of the given object.

Compiling a Script

Some scripting engines can compile scripting code into an intermediate form for efficient execution. Those engines implement the Compilable interface. The following example shows how to compile and evaluate code that is contained in a script file:

```
Reader reader = new FileReader("myscript.js");
CompiledScript script = null;
if (engine implements Compilable)
   CompiledScript script = ((Compilable) engine).compile(reader);
```

Once the script is compiled, you can execute it. The following code executes the compiled script if compilation was successful, or the original script if the engine didn't support compilation.

```
if (script != null)
   script.eval();
else
   engine.eval(reader);
```

Of course, you only want to compile a script if you need to execute it repeatedly.

API `javax.script.Compilable` 6

- `CompiledScript compile(String script)`
- `CompiledScript compile(Reader reader)`

  compiles the script given by a string or reader.

API `javax.script.CompiledScript` 6

- `Object eval()`
- `Object eval(Bindings bindings)`

  evaluates this script.

An Example: Scripting GUI Events

To illustrate the scripting API, we will develop a sample program that allows users to specify event handlers in a scripting language of their choice.

Have a look at the program in Listing 11-1. The `ButtonFrame` class is similar to the event handling demo in Volume I, with two differences:

- Each component has its `name` property set.
- There are no event handlers.

The event handlers are defined in a properties file. Each property definition has the form

componentName . *eventName* = *scriptCode*

For example, if you choose to use JavaScript, you supply the event handlers in a file `js.properties`, like this:

```
yellowButton.action=panel.background = java.awt.Color.YELLOW
blueButton.action=panel.background = java.awt.Color.BLUE
redButton.action=panel.background = java.awt.Color.RED
```

The companion code also has files for Groovy and SISC Scheme.

The program starts by loading an engine for the language that is specified on the command line. If no language is specified, we use JavaScript.

We then process a script `init.`*language* if it is present. This seems like a good idea in general. Moreover, the Scheme interpreter needs some cumbersome initializations that we did not want to include in every event handler script.

Next, we recursively traverse all child components and add the bindings (*name*, *object*) into the engine scope.

Then we read the file *language*`.properties`. For each property, we synthesize an event handler proxy that causes the script code to be executed. The details are a bit technical. You might want to read the section on proxies in Volume I, Chapter 6, together with the section on JavaBeans events in Chapter 8 of this volume, if you want follow the implementation in detail. The essential part, however, is that each event handler calls

```
engine.eval(scriptCode);
```

Let us look at the `yellowButton` in more detail. When the line

```
yellowButton.action=panel.background = java.awt.Color.YELLOW
```

is processed, we find the `JButton` component with the name `"yellowButton"`. We then attach an `ActionListener` with an `actionPerformed` method that executes the script

```
panel.background = java.awt.Color.YELLOW
```

The engine contains a binding that binds the name `"panel"` to the `JPanel` object. When the event occurs, the `setBackground` method of the panel is executed, and the color changes.

You can run this program with the JavaScript event handlers, simply by executing

```
java ScriptTest
```

For the Groovy handlers, use

Code View:

```
java -classpath .:groovy/lib/*:jsr223-engines/groovy/build/groovy-engine.jar ScriptTest groovy
```

Here, *groovy* is the directory into which you installed Groovy, and *jsr223-engines* is the directory that contains the engine adapters from http://scripting.dev.java.net.

To try out Scheme, download SISC Scheme from http://sisc-scheme.org/ and run

Code View:

```
java -classpath .:sisc/*:jsr223-engines/scheme/build/scheme-engine.jar ScriptTest scheme
```

This application demonstrates how to use scripting for Java GUI programming. One could go one step further and describe the GUI with an XML file, as you have seen in Chapter 2. Then our program would become an interpreter for GUIs that have visual presentation defined by XML and behavior defined by a scripting language. Note the similarity to a dynamic HTML page or a dynamic server-side scripting environment.

Listing 11-1. **ScriptTest.java**

Code View:

```
import java.awt.*;
import java.beans.*;
```



```
import java.io.*;
import java.lang.reflect.*;
import java.util.*;
import javax.script.*;
import javax.swing.*;

/**
```



```
 * @version 1.00 2007-10-28
 * @author Cay Horstmann
 */
public class ScriptTest
{
   public static void main(final String[] args)
   {
      EventQueue.invokeLater(new Runnable()
         {
            public void run()
            {
               String language;
               if (args.length == 0) language = "js";
               else language = args[0];

               ScriptEngineManager manager = new ScriptEngineManager();
               System.out.println("Available factories: ");
               for (ScriptEngineFactory factory : manager.getEngineFactories())
                  System.out.println(factory.getEngineName());
               final ScriptEngine engine = manager.getEngineByName(language);

               if (engine == null)
               {
                  System.err.println("No engine for " + language);
                  System.exit(1);
               }

               ButtonFrame frame = new ButtonFrame();

               try
               {
                  File initFile = new File("init." + language);
                  if (initFile.exists())
                  {
                     engine.eval(new FileReader(initFile));
                  }

                  getComponentBindings(frame, engine);

                  final Properties events = new Properties();
                  events.load(new FileReader(language + ".properties"));
                  for (final Object e : events.keySet())
                  {
                     String[] s = ((String) e).split("\\.");
```

```
                     addListener(s[0], s[1], (String) events.get(e), engine);
                  }
               }
               catch (Exception e)
               {
                  e.printStackTrace();
               }

               frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
               frame.setTitle("ScriptTest");
               frame.setVisible(true);
            }
         });
   }

   /**
    * Gathers all named components in a container.
    * @param c the component
    * @param namedComponents
    */
   private static void getComponentBindings(Component c, ScriptEngine engine)
   {
      String name = c.getName();
      if (name != null) engine.put(name, c);
      if (c instanceof Container)
      {
         for (Component child : ((Container) c).getComponents())
            getComponentBindings(child, engine);
      }
   }

   /**
    * Adds a listener to an object whose listener method executes a script.
    * @param beanName the name of the bean to which the listener should be added
    * @param eventName the name of the listener type, such as "action" or "change"
    * @param scriptCode the script code to be executed
    * @param engine the engine that executes the code
    * @param bindings the bindings for the execution
    */
   private static void addListener(String beanName, String eventName, final String scriptCode,
         final ScriptEngine engine) throws IllegalArgumentException, IntrospectionException,
         IllegalAccessException, InvocationTargetException
   {
      Object bean = engine.get(beanName);
      EventSetDescriptor descriptor = getEventSetDescriptor(bean, eventName);
      if (descriptor == null) return;
      descriptor.getAddListenerMethod().invoke(
            bean,
            Proxy.newProxyInstance(null, new Class[] { descriptor.getListenerType() },
                  new InvocationHandler()
                     {
                        public Object invoke(Object proxy, Method method, Object[] args)
                              throws Throwable
                        {
                           engine.eval(scriptCode);
                           return null;
                        }
                     }));

```

```
   }

   private static EventSetDescriptor getEventSetDescriptor(Object bean, String eventName)
         throws IntrospectionException
   {
      for (EventSetDescriptor descriptor : Introspector.getBeanInfo(bean.getClass())
            .getEventSetDescriptors())
         if (descriptor.getName().equals(eventName)) return descriptor;
      return null;
   }
}

class ButtonFrame extends JFrame
{
   public ButtonFrame()
   {
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

      panel = new JPanel();
      panel.setName("panel");
      add(panel);

      yellowButton = new JButton("Yellow");
      yellowButton.setName("yellowButton");
      blueButton = new JButton("Blue");
      blueButton.setName("blueButton");
      redButton = new JButton("Red");
      redButton.setName("redButton");

      panel.add(yellowButton);
      panel.add(blueButton);
      panel.add(redButton);
   }

   public static final int DEFAULT_WIDTH = 300;
   public static final int DEFAULT_HEIGHT = 200;

   private JPanel panel;
   private JButton yellowButton;
   private JButton blueButton;
   private JButton redButton;
}
```


UNREGISTERED VERSION OF CHM TO PDF CONVERTER PRO BY THETA-SOFTWARE

The Compiler API

In the preceding sections, you saw how to interact with code in a scripting language. Now we turn to a different scenario: Java programs that compile Java code. There are quite a few tools that need to invoke the Java compiler, such as:

- Development environments.
- Java teaching and tutoring programs.
- Build and test automation tools.
- Templating tools that process snippets of Java code, such as JavaServer Pages (JSP).

In the past, applications invoked the Java compiler by calling undocumented classes in the *jdk/*`lib/tools.jar` library. As of Java SE 6, a public API for compilation is a part of the Java platform, and it is no longer necessary to use `tools.jar`. This section explains the compiler API.

Compiling the Easy Way

It is very easy to invoke the compiler. Here is a sample call:

Code View:
```
JavaCompiler compiler = ToolProvider.getSystemJavaCompiler();
OutputStream outStream = ..., errStream = ...;
int result = compiler.run(null, outStream, errStream, "-sourcepath", "src", "Test.java");
```

A result value of 0 indicates successful compilation.

The compiler sends output and error messages to the provided streams. You can set these parameters to `null`, in which case `System.out` and `System.err` are used. The first parameter of the `run` method is an input stream. Because the compiler takes no console input, you always leave it as `null`. (The `run` method is inherited from a generic `Tool` interface, which allows for tools that read input.)

The remaining parameters of the `run` method are simply the arguments that you would pass to `javac` if you invoked it on the command line. These can be options or file names.

Using Compilation Tasks

You can have even more control over the compilation process with a `CompilationTask` object. In particular, you can

- Control the source of program code, for example, by providing code in a string builder instead of a file.

- Control the placement of class files, for example, by storing them in a database.

- Listen to error and warning messages as they occur during compilation.

- Run the compiler in the background.

The location of source and class files is controlled by a `JavaFileManager`. It is responsible for determining `JavaFileObject` instances for source and class files. A `JavaFileObject` can correspond to a disk file, or it can provide another mechanism for reading and writing its contents.

To listen to error messages, you install a `DiagnosticListener`. The listener receives a `Diagnostic` object whenever the compiler reports a warning or error message. The `DiagnosticCollector` class implements this interface. It simply collects all diagnostics so that you can iterate through them after the compilation is complete.

A `Diagnostic` object contains information about the problem location (including the file name, line number, and column number) as well as a human-readable description.

You obtain a `CompilationTask` object by calling the `getTask` method of the `JavaCompiler` class. You need to specify:

- A `Writer` for any compiler output that is not reported as a `Diagnostic`, or `null` to use `System.err`.

- A `JavaFileManager`, or `null` to use the compiler's standard file manager.

- A `DiagnosticListener`.

- Option strings, or `null` for no options.

- Class names for annotation processing, or `null` if none are specified. (We discuss annotation processing later in this chapter.)

- `JavaFileObject` instances for source files.

You need to provide the last three arguments as `Iterable` objects. For example, a sequence of options might be specified as

```
Iterable<String> options = Arrays.asList("-g", "-d", "classes");
```

Alternatively, you can use any collection class.

If you want the compiler to read source files from disk, then you can ask the `StandardJavaFileManager` to translate file name strings or `File` objects to `JavaFileObject` instances. For example,

Code View:

```
StandardJavaFileManager fileManager = compiler.getStandardFileManager(null, null, null);
Iterable<JavaFileObject> fileObjects = fileManager.getJavaFileObjectsFromStrings(fileNames);
```

However, if you want the compiler to read source code from somewhere other than a disk file, then you supply your own `JavaFileObject` subclass. Listing 11-2 shows the code for a source file object with data that are contained in a `StringBuilder`. The class extends the `SimpleJava FileObject` convenience class and overrides the `getCharContent` method to return the content of the string builder. We use this class in our example program in which we dynamically produce the code for a Java class and then compile it.

The `CompilationTask` class implements the `Callable<Boolean>` interface. You can pass it to an `Executor` for execution in another thread, or you can simply invoke the `call` method. A return value of `Boolean.FALSE` indicates failure.

Code View:

```
Callable<Boolean> task = new JavaCompiler.CompilationTask(null, fileManager, diagnostics,
   options, null, fileObjects);
if (!task.call())
   System.out.println("Compilation failed");
```

If you simply want the compiler to produce class files on disk, you need not customize the `JavaFileManager`. However, our sample application will generate class files in byte arrays and later read them from memory, using a special class loader. Listing 11-3 defines a class that implements the `JavaFileObject` interface. Its `openOutputStream` method returns the `ByteArrayOutputStream` into which the compiler will deposit the byte codes.

It turns out a bit tricky to tell the compiler's file manager to uses these file objects. The library doesn't supply a class that implements the `StandardJavaFileManager` interface. Instead, you subclass the `ForwardingJavaFileManager` class that delegates all calls to a given file manager. In our situation, we only want to change the `getJavaFileForOutput` method. We achieve this with the following outline:

Code View:

```
JavaFileManager fileManager = compiler.getStandardFileManager(diagnostics, null, null);
fileManager = new ForwardingJavaFileManager<JavaFileManager>(fileManager)
   {
      public JavaFileObject getJavaFileForOutput(Location location, final String className,
            Kind kind, FileObject sibling) throws IOException
      {
         return custom file object
      }
   };
```

In summary, you call the `run` method of the `JavaCompiler` task if you simply want to invoke the compiler in the usual way, reading and writing disk files. You can capture the output and error messages, but you need to parse them yourself.

If you want more control over file handling or error reporting, you use the `CompilationTask` class instead. Its

API is quite complex, but you can control every aspect of the compilation process.

Listing 11-2. **StringBuilderJavaSource.java**

Code View:

```
import java.net.*;
import javax.tools.*;

/**
 * A Java source that holds the code in a string builder.
 * @version 1.00 2007-11-02
 * @author Cay Horstmann
 */
public class StringBuilderJavaSource extends SimpleJavaFileObject
{
   /**
    * Constructs a new StringBuilderJavaSource
    * @param name the name of the source file represented by this file object
    */
   public StringBuilderJavaSource(String name)
   {
      super(URI.create("string:///" + name.replace('.', '/') + Kind.SOURCE.extension),
               Kind.SOURCE);
      code = new StringBuilder();
   }

   public CharSequence getCharContent(boolean ignoreEncodingErrors)
   {
      return code;
   }

   public void append(String str)
   {
      code.append(str);
      code.append('\n');
   }

   private StringBuilder code;
}
```

Listing 11-3. **ByteArrayJavaClass.java**

Code View:

```
import java.io.*;
import java.net.*;
import javax.tools.*;

/**
 * A Java class that holds the bytecodes in a byte array.
 * @version 1.00 2007-11-02
 * @author Cay Horstmann
 */
public class ByteArrayJavaClass extends SimpleJavaFileObject
{
   /**
    * Constructs a new ByteArrayJavaClass
    * @param name the name of the class file represented by this file object
    */
   public ByteArrayJavaClass(String name)
   {
      super(URI.create("bytes:///" + name), Kind.CLASS);
      stream = new ByteArrayOutputStream();
   }

   public OutputStream openOutputStream() throws IOException
   {
      return stream;
   }

   public byte[] getBytes()
   {
      return stream.toByteArray();
   }

   private ByteArrayOutputStream stream;
}
```

API

`javax.tools.Tool` 6

- `int run(InputStream in, OutputStream out, OutputStream err, String... arguments)`

  runs the tool with the given input, output, and error streams, and the given arguments. Returns 0 for success, a nonzero value for failure.

javax.tools.JavaCompiler 6

- StandardJavaFileManager getStandardFileManager(DiagnosticListener<? super JavaFileObject> diagnosticListener, Locale locale, Charset charset)

  gets the standard file manager for this compiler. You can supply null for default error reporting, locale, and character set.

- JavaCompiler.CompilationTask getTask(Writer out, JavaFileManager fileManager, DiagnosticListener<? super JavaFileObject> diagnosticListener, Iterable<String> options, Iterable<String> classesForAnnotationProcessing, Iterable<? extends JavaFileObject> sourceFiles)

  gets a compilation task that, when called, will compile the given source files. See the discussion in the preceding section for details.

javax.tools.StandardJavaFileManager 6

- Iterable<? extends JavaFileObject> getJavaFileObjectsFromStrings(Iterable<String> fileNames)

- Iterable<? extends JavaFileObject> getJavaFileObjectsFromFiles(Iterable<? extends File> files)

  translates a sequence of file names or files into a sequence of JavaFileObject instances.

javax.tools.JavaCompiler.CompilationTask 6

- Boolean call()

  performs the compilation task.

`javax.tools.DiagnosticCollector<S>` 6

- `DiagnosticCollector()`

  constructs an empty collector.

- `List<Diagnostic<? extends S>> getDiagnostics()`

  gets the collected diagnostics.

`javax.tools.Diagnostic<S>` 6

- `S getSource()`

  gets the source object associated with this diagnostic.

- `Diagnostic.Kind getKind()`

  gets the type of this diagnostic, one of `ERROR`, `WARNING`, `MANDATORY_WARNING`, `NOTE`, or `OTHER`.

- `String getMessage(Locale locale)`

  gets the message describing the issue raised in this diagnostic. Pass `null` for the default locale.

- `long getLineNumber()`

- `long getColumnNumber()`

  gets the position of the issue raised in this diagnostic.

javax.tools.SimpleJavaFileObject 6

- `CharSequence getCharContent(boolean ignoreEncodingErrors)`

  override this method for a file object that represents a source file, and produce the source code.

- `OutputStream openOutputStream()`

  override this method for a file object that represents a class file, and produce a stream to which the byte codes can be written.

javax.tools.ForwardingJavaFileManager<M extends JavaFileManager> 6

- `protected ForwardingJavaFileManager(M fileManager)`

  constructs a JavaFileManager that delegates all calls to the given file manager.

- `FileObject getFileForOutput(JavaFileManager.Location location, String className, JavaFileObject.Kind kind, FileObject sibling)`

  intercept this call if you want to substitute a file object for writing class files. kind is one of SOURCE, CLASS, HTML, or OTHER.

An Example: Dynamic Java Code Generation

In JSP technology for dynamic web pages, you can mix HTML with snippets of Java code, such as

```
<p>The current date and time is <b><%= new java.util.Date() %></b>.</p>
```

The JSP engine dynamically compiles the Java code into a servlet. In our sample application, we use a simpler example and generate dynamic Swing code instead. The idea is that you use a GUI builder to lay out the components in a frame and specify the behavior of the components in an external file. Listing 11-4 shows a very simple example of a frame class, and Listing 11-5 shows the code for the button actions. Note that the constructor of the frame class calls an abstract method addEventHandlers. Our code generator will produce a subclass that implements the addEventHandlers method, adding an action listener for each line in the action.properties class. (We leave it as the proverbial exercise to the reader to extend the code generation to other event types.)

We place the subclass into a package with the name x, which we hope is not used anywhere else in the program. The generated code has the form

Code View:

```
package x;
public class Frame extends SuperclassName {
   protected void addEventHandlers() {
      componentName1.addActionListener(new java.awt.event.ActionListener() {
         public void actionPerformed(java.awt.event.ActionEvent) {
            code for event handler1
         } } );
      // repeat for the other event handlers ...
   } }
```

The buildSource method in the program of Listing 11-6 builds up this code and places it into a StringBuilderJavaSource object. That object is passed to the Java compiler.

We use a ForwardingJavaFileManager with a getJavaFileForOutput method that constructs a ByteArrayJavaClass object for every class in the x package. These objects capture the class files that are generated when the x.Frame class is compiled. The method adds each file object to a list before returning it so that we can locate the byte codes later. Note that compiling the x.Frame class produces a class file for the main class and one class file per listener class.

After compilation, we build a map that associates class names with bytecode arrays. A simple class loader (shown in Listing 11-7) loads the classes stored in this map.

We ask the class loader to load the class that we just compiled, and then we construct and display the application's frame class.

```
ClassLoader loader = new MapClassLoader(byteCodeMap);
Class<?> cl = loader.loadClass("x.Frame");
Frame frame = (JFrame) cl.newInstance();
frame.setVisible(true);
```

When you click the buttons, the background color changes in the usual way. To see that the actions are dynamically compiled, change one of the lines in action.properties, for example like this:

Code View:

```
yellowButton=panel.setBackground(java.awt.Color.YELLOW); yellowButton.setEnabled(false);
```

Run the program again. Now the Yellow button is disabled after you click it. Also have a look at the code directories. You will not find any source or class files for the classes in the x package. This example demonstrates how you can use dynamic compilation with in-memory source and class files.

Listing 11-4. **ButtonFrame.java**

Code View:

```
package com.horstmann.corejava;
import javax.swing.*;

/**
 * @version 1.00 2007-11-02
 * @author Cay Horstmann
 */
public abstract class ButtonFrame extends JFrame
{
   public ButtonFrame()
   {
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

      panel = new JPanel();
      add(panel);

      yellowButton = new JButton("Yellow");
      blueButton = new JButton("Blue");
      redButton = new JButton("Red");

      panel.add(yellowButton);
      panel.add(blueButton);
      panel.add(redButton);

      addEventHandlers();
   }

   protected abstract void addEventHandlers();

   public static final int DEFAULT_WIDTH = 300;
   public static final int DEFAULT_HEIGHT = 200;

   protected JPanel panel;
   protected JButton yellowButton;
   protected JButton blueButton;
   protected JButton redButton;
}
```


Listing 11-5. **action.properties**

```
yellowButton=panel.setBackground(java.awt.Color.YELLOW);
blueButton=panel.setBackground(java.awt.Color.BLUE);
redButton=panel.setBackground(java.awt.Color.RED);
```

Listing 11-6. **CompilerTest.java**

Code View:

```
import java.awt.*;
import java.io.*;
import java.util.*;
import java.util.List;
import javax.swing.*;
```

```
import javax.tools.*;
import javax.tools.JavaFileObject.*;

/**
 * @version 1.00 2007-11-02
 * @author Cay Horstmann
 */
public class CompilerTest
{
   public static void main(final String[] args) throws IOException
   {
      JavaCompiler compiler = ToolProvider.getSystemJavaCompiler();

      final List<ByteArrayJavaClass> classFileObjects = new ArrayList<ByteArrayJavaClass>();

      DiagnosticCollector<JavaFileObject> diagnostics = new DiagnosticCollector<JavaFileObject>()

      JavaFileManager fileManager = compiler.getStandardFileManager(diagnostics, null, null);
      fileManager = new ForwardingJavaFileManager<JavaFileManager>(fileManager)
         {
            public JavaFileObject getJavaFileForOutput(Location location,
                  final String className, Kind kind, FileObject sibling) throws IOException
            {
               if (className.startsWith("x."))
               {
                  ByteArrayJavaClass fileObject = new ByteArrayJavaClass(className);
                  classFileObjects.add(fileObject);
                  return fileObject;
               }
               else return super.getJavaFileForOutput(location, className, kind, sibling);
            }
         };

      JavaFileObject source = buildSource("com.horstmann.corejava.ButtonFrame");
      JavaCompiler.CompilationTask task = compiler.getTask(null, fileManager, diagnostics,
            null, null, Arrays.asList(source));
      Boolean result = task.call();

      for (Diagnostic<? extends JavaFileObject> d : diagnostics.getDiagnostics())
         System.out.println(d.getKind() + ": " + d.getMessage(null));
      fileManager.close();
      if (!result)
      {
         System.out.println("Compilation failed.");
         System.exit(1);
      }

      EventQueue.invokeLater(new Runnable()
         {
            public void run()
            {
               try
               {
                  Map<String, byte[]> byteCodeMap = new HashMap<String, byte[]>();
                  for (ByteArrayJavaClass cl : classFileObjects)
                     byteCodeMap.put(cl.getName().substring(1), cl.getBytes());
                  ClassLoader loader = new MapClassLoader(byteCodeMap);
                  Class<?> cl = loader.loadClass("x.Frame");
                  JFrame frame = (JFrame) cl.newInstance();
```

```
                  frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                  frame.setTitle("CompilerTest");
                  frame.setVisible(true);
               }
               catch (Exception ex)
               {
                  ex.printStackTrace();
               }
```



```
            }
         });
   }

   /*
    * Builds the source for the subclass that implements the addEventHandlers method.
    * @return a file object containing the source in a string builder
```



```
    */
   static JavaFileObject buildSource(String superclassName) throws IOException
   {
      StringBuilderJavaSource source = new StringBuilderJavaSource("x.Frame");
      source.append("package x;\n");
      source.append("public class Frame extends " + superclassName + " {");
      source.append("protected void addEventHandlers() {");
      Properties props = new Properties();
      props.load(new FileReader("action.properties"));
      for (Map.Entry<Object, Object> e : props.entrySet())
      {
         String beanName = (String) e.getKey();
         String eventCode = (String) e.getValue();
         source.append(beanName + ".addActionListener(new java.awt.event.ActionListener() {");
         source.append("public void actionPerformed(java.awt.event.ActionEvent event) {");
         source.append(eventCode);
         source.append("} } );");
      }
      source.append("} }");
      return source;
   }
}
```

Listing 11-7. **MapClassLoader.java**

Code View:

```
import java.util.*;

/**
 * A class loader that loads classes from a map whose keys are class names and whose
 * values are byte code arrays.
 * @version 1.00 2007-11-02
 * @author Cay Horstmann
 */
public class MapClassLoader extends ClassLoader
{
   public MapClassLoader(Map<String, byte[]> classes)
   {
      this.classes = classes;
   }

   protected Class<?> findClass(String name) throws ClassNotFoundException
   {
      byte[] classBytes = classes.get(name);
      if (classBytes == null) throw new ClassNotFoundException(name);
      Class<?> cl = defineClass(name, classBytes, 0, classBytes.length);
      if (cl == null) throw new ClassNotFoundException(name);
      return cl;
   }

   private Map<String, byte[]> classes;
}
```

Using Annotations

Annotations are tags that you insert into your source code so that some tool can process them. The tools can operate on the source level, or they can process class files into which the compiler has placed annotations.

Annotations do not change the way in which your programs are compiled. The Java compiler generates the same virtual machine instructions with or without the annotations.

To benefit from annotations, you need to select a *processing tool*. You insert annotations into your code that your processing tool understands, and then apply the processing tool.

There is a wide range of uses for annotations, and that generality can be initially confusing. Here are some uses for annotations:

- Automatic generation of auxiliary files, such as deployment descriptors or bean information classes.

- Automatic generation of code for testing, logging, transaction semantics, and so on.

We start our discussion of annotations with the basic concepts and put them to use in a concrete example: We mark methods as event listeners for AWT components, and show you an annotation processor that analyzes the annotations and hooks up the listeners. We then discuss the syntax rules in detail. We finish the chapter with two advanced examples for annotation processing. One of them processes source-level annotations. The other uses the Apache Bytecode Engineering Library to process class files, injecting additional bytecodes into annotated methods.

Here is an example of a simple annotation:

```
public class MyClass
{
   . . .
   @Test public void checkRandomInsertions()
}
```

The annotation `@Test` annotates the `checkRandomInsertions` method.

In Java, an annotation is used like a *modifier*, and it is placed before the annotated item, *without a semicolon*. (A modifier is a keyword such as `public` or `static`.) The name of each annotation is preceded by an `@` symbol, similar to Javadoc comments. However, Javadoc comments occur inside `/** . . . */` delimiters, whereas annotations are part of the code.

By itself, the `@Test` annotation does not do anything. It needs a tool to be useful. For example, the JUnit 4 testing tool (available at http://junit.org) calls all methods that are labeled as `@Test` when testing a class. Another tool might remove all test methods from a class file so that they are not shipped with the program after it has been tested.

Annotations can be defined to have *elements*, such as

```
@Test(timeout="10000")
```

These elements can be processed by the tools that read the annotations. Other forms of elements are possible; we discuss them later in this chapter.

Besides methods, you can annotate classes, fields, and local variables—an annotation can be anywhere you could put a modifier such as `public` or `static`.

Each annotation must be defined by an *annotation interface*. The methods of the interface correspond to the elements of the annotation. For example, the JUnit `Test` annotation is defined by the following interface:

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface Test
{
   long timeout() default 0L;
   . . .
}
```

The `@interface` declaration creates an actual Java interface. Tools that process annotations receive objects that implement the annotation interface. A tool would call the `timeout` method to retrieve the `timeout` element of a particular `Test` annotation.

The `Target` and `Retention` annotations are *meta-annotations*. They annotate the `Test` annotation, marking it as an annotation that can be applied to methods only and that is retained when the class file is loaded into the virtual machine. We discuss them in detail in the section "Meta-Annotations" on page 917.

You have now seen the basic concepts of program metadata and annotations. In the next section, we walk through a concrete example of annotation processing.

An Example: Annotating Event Handlers

One of the more boring tasks in user interface programming is the wiring of listeners to event sources. Many listeners are of the form

```
myButton.addActionListener(new
   ActionListener()
   {
      public void actionPerformed(ActionEvent event)
      {
         doSomething();
      }
   });
```

In this section, we design an annotation to avoid this drudgery. The annotation has the form

```
@ActionListenerFor(source="myButton") void doSomething() { . . . }
```

The programmer no longer has to make calls to `addActionListener`. Instead, each method is simply tagged with an annotation. Listing 11-8 shows the `ButtonFrame` class from Volume I, Chapter 8, reimplemented with these annotations.

We also need to define an annotation interface. The code is in Listing 11-9.

Listing 11-8. **`ButtonFrame.java`**

Code View:

```
import java.awt.*;
import javax.swing.*;

/**
 * A frame with a button panel
 * @version 1.00 2004-08-17
 * @author Cay Horstmann
 */
public class ButtonFrame extends JFrame
{
   public ButtonFrame()
   {
      setTitle("ButtonTest");
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

      panel = new JPanel();
      add(panel);

      yellowButton = new JButton("Yellow");
      blueButton = new JButton("Blue");
      redButton = new JButton("Red");

      panel.add(yellowButton);
      panel.add(blueButton);
      panel.add(redButton);

      ActionListenerInstaller.processAnnotations(this);
   }

   @ActionListenerFor(source = "yellowButton")
   public void yellowBackground()
   {
      panel.setBackground(Color.YELLOW);
   }

   @ActionListenerFor(source = "blueButton")
   public void blueBackground()
   {
      panel.setBackground(Color.BLUE);
   }

   @ActionListenerFor(source = "redButton")
   public void redBackground()
   {
      panel.setBackground(Color.RED);
   }

   public static final int DEFAULT_WIDTH = 300;
   public static final int DEFAULT_HEIGHT = 200;

   private JPanel panel;
   private JButton yellowButton;
   private JButton blueButton;
   private JButton redButton;
}
```

Listing 11-9. **ActionListenerFor.java**

```
import java.lang.annotation.*;

/**
 * @version 1.00 2004-08-17
 * @author Cay Horstmann
 */

@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface ActionListenerFor
{
   String source();
}
```

Of course, the annotations don't do anything by themselves. They sit in the source file. The compiler places them in the class file, and the virtual machine loads them. We now need a mechanism to analyze them and install action listeners. That is the job of the `ActionListenerInstaller` class. The `ButtonFrame` constructor calls

```
ActionListenerInstaller.processAnnotations(this);
```

The static `processAnnotations` method enumerates all methods of the object that it received. For each method, it gets the `ActionListenerFor` annotation object and processes it.

```
Class<?> cl = obj.getClass();
for (Method m : cl.getDeclaredMethods())
{
   ActionListenerFor a = m.getAnnotation(ActionListenerFor.class);
   if (a != null) . . .
}
```

Here, we use the `getAnnotation` method that is defined in the `AnnotatedElement` interface. The classes `Method`, `Constructor`, `Field`, `Class`, and `Package` implement this interface.

The name of the source field is stored in the annotation object. We retrieve it by calling the `source` method, and then look up the matching field.

```
String fieldName = a.source();
Field f = cl.getDeclaredField(fieldName);
```

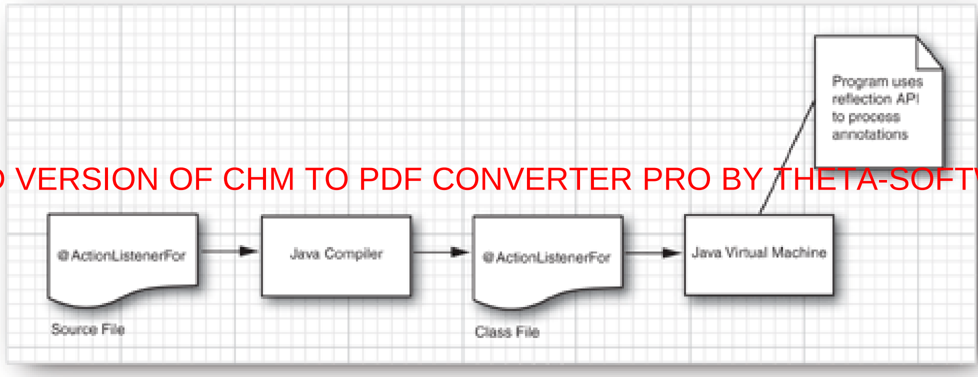
This shows a limitation of our annotation. The source element must be the name of a field. It cannot be a local variable.

The remainder of the code is rather technical. For each annotated method, we construct a proxy object that implements the `ActionListener` interface and with an `actionPerformed` method that calls the annotated method. (For more information about proxies, see Volume I, Chapter 6.) The details are not important. The key observation is that the functionality of the annotations was established by the `processAnnotations` method.

Figure 11-1 shows how annotations are handled in this example.

Figure 11-1. Processing annotations at runtime

[View full size image]



In this example, the annotations were processed at runtime. It would also have been possible to process them at the source level. A source code generator might have produced the code for adding the listeners. Alternatively, the annotations might have been processed at the bytecode level. A bytecode editor might have injected the calls to `addActionListener` into the frame constructor. This sounds complex, but libraries are available to make this task relatively straightforward. You can see an example in the section "Bytecode Engineering" on page 926.

Our example was not intended as a serious tool for user interface programmers. A utility method for adding a listener could be just as convenient for the programmer as the annotation. (In fact, the `java.beans.EventHandler` class tries to do just that. You could easily refine the class to be truly useful by supplying a method that adds the event handler instead of just constructing it.)

However, this example shows the mechanics of annotating a program and of analyzing the annotations. Having seen a concrete example, you are now more prepared (we hope) for the following sections that describe the annotation syntax in complete detail.

Listing 11-10. **ActionListenerInstaller.java**

Code View:
```
import java.awt.event.*;
import java.lang.reflect.*;

/**
 * @version 1.00 2004-08-17
 * @author Cay Horstmann
 */
public class ActionListenerInstaller
{
   /**
    * Processes all ActionListenerFor annotations in the given object.
    * @param obj an object whose methods may have ActionListenerFor annotations
    */
   public static void processAnnotations(Object obj)
```

```
   {
      try
      {
         Class<?> cl = obj.getClass();
         for (Method m : cl.getDeclaredMethods())
         {
            ActionListenerFor a = m.getAnnotation(ActionListenerFor.class);
            if (a != null)
            {
               Field f = cl.getDeclaredField(a.source());
               f.setAccessible(true);
               addListener(f.get(obj), obj, m);
            }
         }
      }
      catch (Exception e)
      {
         e.printStackTrace();
      }
   }

   /**
    * Adds an action listener that calls a given method.
    * @param source the event source to which an action listener is added
    * @param param the implicit parameter of the method that the listener calls
    * @param m the method that the listener calls
    */
   public static void addListener(Object source, final Object param, final Method m)
         throws NoSuchMethodException, IllegalAccessException, InvocationTargetException
   {
      InvocationHandler handler = new InvocationHandler()
         {
            public Object invoke(Object proxy, Method mm, Object[] args) throws Throwable
            {
               return m.invoke(param);
            }
         };

      Object listener = Proxy.newProxyInstance(null,
            new Class[] { java.awt.event.ActionListener.class }, handler);
      Method adder = source.getClass().getMethod("addActionListener", ActionListener.class);
      adder.invoke(source, listener);
   }
}
```

`java.lang.AnnotatedElement` 5.0

- `boolean isAnnotationPresent(Class<? extends Annotation> annotationType)`

  returns `true` if this item has an annotation of the given type.

- `<T extends Annotation> T getAnnotation(Class<T> annotationType)`

  gets the annotation of the given type, or `null` if this item has no such annotation.

- `Annotation[] getAnnotations()`

  gets all annotations that are present for this item, including inherited annotations. If no annotations are present, an array of length 0 is returned.

- `Annotation[] getDeclaredAnnotations()`

  gets all annotations that are declared for this item, excluding inherited annotations. If no annotations are present, an array of length 0 is returned.

Annotation Syntax

In this section, we cover everything you need to know about the annotation syntax.

An annotation is defined by an annotation interface:

modifiers `@interface` *AnnotationName*
`{`
    element declaration$_1$
    element declaration$_2$
    `. . .`
`}`

Each element declaration has the form

type elementName`();`

or

type elementName`() default` *value*`;`

For example, the following annotation has two elements, `assignedTo` and `severity`.

```
public @interface BugReport
{
   String assignedTo() default "[none]";
   int severity() = 0;
}
```

Each annotation has the format

@AnnotationName`(`*elementName*$_1$=*value*$_1$, *elementName*$_2$=*value*$_2$, `. . .)`

For example,

```
@BugReport(assignedTo="Harry", severity=10)
```

The order of the elements does not matter. The annotation

```
@BugReport(severity=10, assignedTo="Harry")
```

is identical to the preceding one.

The default value of the declaration is used if an element value is not specified. For example, consider the annotation

```
@BugReport(severity=10)
```

The value of the `assignedTo` element is the string `"[none]"`.

Caution

Defaults are not stored with the annotation; instead, they are dynamically computed. For example, if you change the default for the `assignedTo` element to `"[]"` and recompile the `BugReport` interface, then the annotation `@BugReport(severity=10)` uses the new default, even in class files that have been compiled before the default changed.

Two special shortcuts can simplify annotations.

If no elements are specified, either because the annotation doesn't have any or because all of them use the default value, then you don't need to use parentheses. For example,

```
@BugReport
```

is the same as

```
@BugReport(assignedTo="[none]", severity=0)
```

Such an annotation is called a *marker annotation*.

The other shortcut is the *single value annotation*. If an element has the special name `value`, and no other element is specified, then you can omit the element name and the `=` symbol. For example, had we defined the `ActionListenerFor` annotation interface of the preceding section as

```
public @interface ActionListenerFor
{
   String value();
}
```

then we could have written the annotations as

```
@ActionListenerFor("yellowButton")
```

instead of

```
@ActionListenerFor(value="yellowButton")
```

All annotation interfaces implicitly extend the interface `java.lang.annotation.Annotation`. That interface is a regular interface, *not* an annotation interface. See the API notes at the end of this section for the methods provided by this interface.

You cannot extend annotation interfaces. In other words, all annotation interfaces directly extend `java.lang.annotation.Annotation`.

You never supply classes that implement annotation interfaces. Instead, the virtual machine generates proxy classes and objects when needed. For example, when requesting an `ActionListenerFor` annotation, the virtual machine carries out an operation similar to the following:

Code View:
```
return Proxy.newProxyInstance(classLoader, ActionListenerFor.class,
   new
      InvocationHandler()
      {
         public Object invoke(Object proxy, Method m, Object[] args) throws Throwable
         {
            if (m.getName().equals("source")) return value of source annotation;
             . . .
         }
      });
```

The element declarations in the annotation interface are actually method declarations. The methods of an annotation interface can have no parameters and no `throws` clauses, and they cannot be generic.

The type of an annotation element is one of the following:

- A primitive type (`int`, `short`, `long`, `byte`, `char`, `double`, `float`, or `boolean`)
- `String`
- `Class` (with an optional type parameter such as `Class<? extends MyClass>`)
- An `enum` type
- An annotation type
- An array of the preceding types (an array of arrays is not a legal element type)

Here are examples for valid element declarations:

```
public @interface BugReport
{
   enum Status { UNCONFIRMED, CONFIRMED, FIXED, NOTABUG };
   boolean showStopper() default false;
   String assignedTo() default "[none]";
   Class<?> testCase() default Void.class;
   Status status() default Status.UNCONFIRMED;
```

```
   Reference ref() default @Reference(); // an annotation type
   String[] reportedBy();
}
```

Because annotations are evaluated by the compiler, all element values must be compile-time constants. For example,


```
@BugReport(showStopper=true, assignedTo="Harry", testCase=MyTestCase.class,
   status=BugReport.Status.CONFIRMED, . . .)
```

Caution

An annotation element can never be set to `null`. Not even a `default` of `null` is permissible. This can be rather inconvenient in practice. You will need to find other defaults, such as `""` or `Void.class`.

If an element value is an array, you enclose its values in braces, like this:

```
@BugReport(. . ., reportedBy={"Harry", "Carl"})
```

You can omit the braces if the element has a single value:

```
@BugReport(. . ., reportedBy="Joe") // OK, same as {"Joe"}
```

Because an annotation element can be another annotation, you can build arbitrarily complex annotations. For example,

```
@BugReport(ref=@Reference(id="3352627"), . . .)
```

Note

It is an error to introduce circular dependencies in annotations. For example, because `BugReport` has an element of the annotation type `Reference`, then `Reference` can't have an element of type `BugReport`.

You can add annotations to the following items:

- Packages

- Classes (including `enum`)
- Interfaces (including annotation interfaces)
- Methods
- Constructors
- Instance fields (including `enum` constants)
- Local variables
- Parameter variables

However, annotations for local variables can only be processed at the source level. Class files do not describe local variables. Therefore, all local variable annotations are discarded when a class is compiled. Similarly, annotations for packages are not retained beyond the source level.

Note

You annotate a package in a file `package-info.java` that contains only the package statement, preceded by annotations.

An item can have multiple annotations, provided they belong to different types. You cannot use the same annotation type more than once when annotating a particular item. For example,

```
@BugReport(showStopper=true, reportedBy="Joe")
@BugReport(reportedBy={"Harry", "Carl"})
void myMethod()
```

is a compile-time error. If this is a problem, you can design an annotation that has a value of an array of simpler annotations:

```
@BugReports({
   @BugReport(showStopper=true, reportedBy="Joe"),
   @BugReport(reportedBy={"Harry", "Carl"}))
void myMethod()
```

API

java.lang.annotation.Annotation 5.0

- Class<? extends Annotation> annotationType()

  returns the Class object that represents the annotation interface of this annotation object. Note that calling getClass on an annotation object would return the actual class, not the interface.

- boolean equals(Object other)

  returns true if other is an object that implements the same annotation interface as this annotation object and if all elements of this object and other are equal to another.

- int hashCode()

  returns a hash code that is compatible with the equals method, derived from the name of the annotation interface and the element values.

- String toString()

  returns a string representation that contains the annotation interface name and the element values, for example, @BugReport(assignedTo=[none], severity=0)

Standard Annotations

Java SE defines a number of annotation interfaces in the `java.lang`, `java.lang.annotation`, and `javax.annotation` packages. Four of them are meta-annotations that describe the behavior of annotation interfaces. The others are regular annotations that you can use to annotate items in your source code. Table 11-2 shows these annotations. We discuss them in detail in the following two sections.

Table 11-2. The Standard Annotations

Annotation Interface	Applicable To	Purpose
`Deprecated`	All	Marks item as deprecated
`SuppressWarnings`	All but packages and annotations	Suppresses warnings of the given type
`Override`	Methods	Checks that this method overrides a superclass method
`PostConstruct` `PreDestroy`	Methods	The marked method should be invoked immediately after construction or before removal
`Resource`	Classes, interfaces, methods, fields	On a class or interface: marks it as a resource to be used elsewhere. On a method or field: marks it for "injection"
`Resources`	Classes, interfaces	An array of resources
`Generated`	All	Marks item as source code that has been generated by a tool
`Target`	Annotations	Specifies the items to which this annotation can be applied
`Retention`	Annotations	Specifies how long this annotation is retained
`Documented`	Annotations	Specifies that this annotation should be included in the documentation of annotated items
`Inherited`	Annotations	Specifies that this annotation, when applied to a class, is automatically inherited by its subclasses

Annotations for Compilation

The `@Deprecated` annotation can be attached to any items for which use is no longer encouraged. The compiler will warn when you use a deprecated item. This annotation has the same role as the `@deprecated` Javadoc tag.

The `@SuppressWarnings` annotation tells the compiler to suppress warnings of a particular type, for example,

```
@SuppressWarnings("unchecked")
```

The `@Override` annotation applies only to methods. The compiler checks that a method with this annotation really overrides a method from the superclass. For example, if you declare

```
public MyClass
{
   @Override public boolean equals(MyClass other);
   . . .
}
```


then the compiler will report an error. After all, the `equals` method does *not* override the `equals` method of the `Object` class. That method has a parameter of type `Object`, not `MyClass`.

The `@Generated` annotation is intended for use by code generator tools. Any generated source code can be annotated to differentiate it from programmer-provided code. For example, a code editor can hide the generated code, or a code generator can remove older versions of generated code. Each annotation must contain a unique identifier for the code generator. A date string (in ISO8601 format) and a comment string are optional. For example,


```
@Generated("com.horstmann.beanproperty", "2008-01-04T12:08:56.235-0700");
```

Annotations for Managing Resources

The `@PostConstruct` and `@PreDestroy` annotations are used in environments that control the lifecycle of objects, such as web containers and application servers. Methods tagged with these annotations should be invoked immediately after an object has been constructed or immediately before it is being removed.

The `@Resource` annotation is intended for resource injection. For example, consider a web application that accesses a database. Of course, the database access information should not be hardwired into the web application. Instead, the web container has some user interface for setting connection parameters and a JNDI name for a data source. In the web application, you can reference the data source like this:

```
@Resource(name="jdbc/mydb")
private DataSource source;
```

When an object containing this field is constructed, the container "injects" a reference to the data source.

Meta-Annotations

The `@Target` meta-annotation is applied to an annotation, restricting the items to which the annotation applies. For example,

```
@Target({ElementType.TYPE, ElementType.METHOD})
public @interface BugReport
```

Table 11-3 shows all possible values. They belong to the enumerated type `ElementType`. You can specify any number of element types, enclosed in braces.

Table 11-3. Element Types for the **@Target** Annotation

Element Type	Annotation Applies To
`ANNOTATION_TYPE`	Annotation type declarations

Element Type	Annotation Applies To
PACKAGE	Packages
TYPE	Classes (including enum) and interfaces (including annotation types)
METHOD	Methods
CONSTRUCTOR	Constructors
FIELD	Fields (including enum constants)
PARAMETER	Method or constructor parameters
LOCAL_VARIABLE	Local variables

An annotation without an @Target restriction can be applied to any item. The compiler checks that you apply an annotation only to a permitted item. For example, if you apply @BugReport to a field, a compile-time error results.

The @Retention meta-annotation specifies how long an annotation is retained. You specify at most one of the values in Table 11-4. The default is RetentionPolicy.CLASS.

Table 11-4. Retention Policies for the **@Retention** Annotation

Retention Policy	Description
SOURCE	Annotations are not included in class files.
CLASS	Annotations are included in class files, but the virtual machine need not load them.
RUNTIME	Annotations are included in class files and loaded by the virtual machine. They are available through the reflection API.

In Listing 11-9, the @ActionListenerFor annotation was declared with RetentionPolicy.RUNTIME because we used reflection to process annotations. In the following two sections, you will see examples of processing annotations at the source and class file levels.

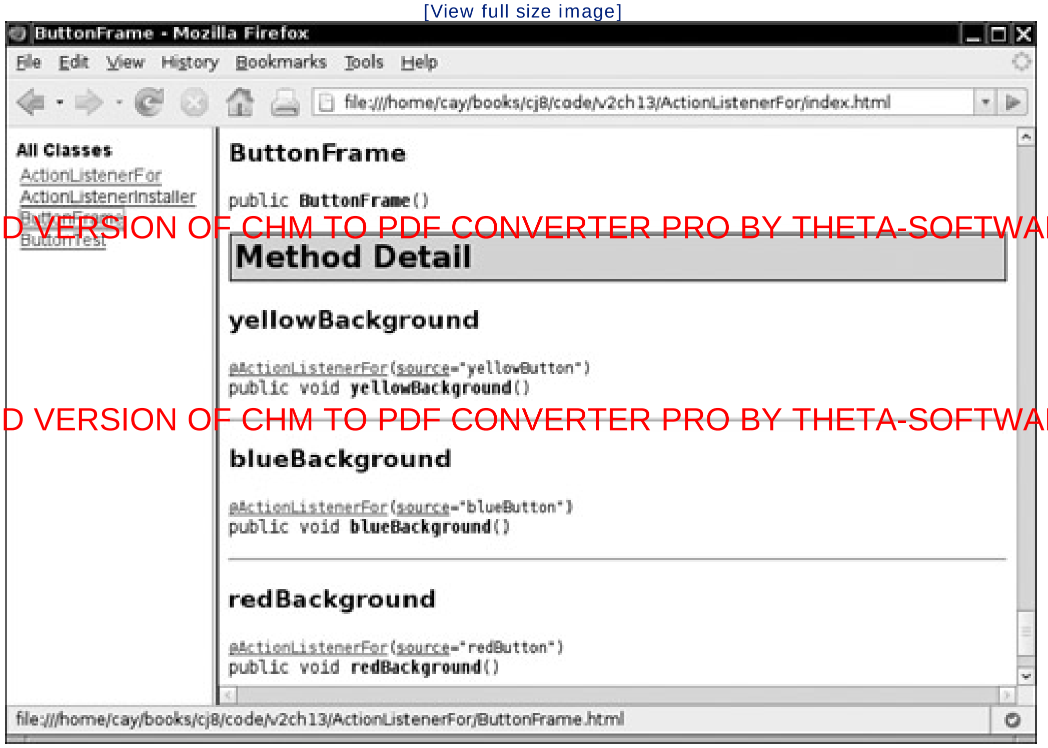
The @Documented meta-annotation gives a hint to documentation tools such as Javadoc. Documented annotations should be treated just like other modifiers such as protected or static for documentation purposes. The use of other annotations is not included in the documentation. For example, suppose we declare @ActionListenerFor as a documented annotation:

```
@Documented
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface ActionListenerFor
```

Now the documentation of each annotated method contains the annotation, as shown in Figure 11-2.

Figure 11-2. Documented annotations

[View full size image]



If an annotation is transient (such as `@BugReport`), you should probably not document its use.

Note

It is legal to apply an annotation to itself. For example, the `@Documented` annotation is itself annotated as `@Documented`. Therefore, the Javadoc documentation for annotations shows whether they are documented.

The `@Inherited` meta-annotation applies only to annotations for classes. When a class has an inherited annotation, then all of its subclasses automatically have the same annotation. This makes it easy to create annotations that work in the same way as marker interfaces such as `Serializable`.

In fact, an annotation `@Serializable` would be more appropriate than the `Serializable` marker interfaces with no methods. A class is serializable because there is runtime support for reading and writing its fields, not because of any principles of object-oriented design. An annotation describes this fact better than does interface inheritance. Of course, the `Serializable` interface was created in JDK 1.1, long before annotations existed.

Suppose you define an inherited annotation `@Persistent` to indicate that objects of a class can be saved in a database. Then the subclasses of persistent classes are automatically annotated as persistent.

```
@Inherited @Persistent { }
@Persistent class Employee { . . . }
```

```
class Manager extends Employee { . . . } // also @Persistent
```

When the persistence mechanism searches for objects to store in the database, it will detect both `Employee` and `Manager` objects.

Source-Level Annotation Processing

One use for annotation is the automatic generation of "side files" that contain additional information about programs. In the past, the Enterprise Edition of Java was notorious for making programmers fuss with lots of boilerplate code. Java EE 5 uses annotations to greatly simplify the programming model.

In this section, we demonstrate this technique with a simpler example. We write a program that automatically produces bean info classes. You tag bean properties with an annotation and then run a tool that parses the source file, analyzes the annotations, and writes out the source file of the bean info class.

Recall from Chapter 8 that a bean info class describes a bean more precisely than the automatic introspection process can. The bean info class lists all of the properties of the bean. Properties can have optional property editor

The `ChartBeanBeanInfo` class in Chapter 8 is a typical example.

To eliminate the drudgery of writing bean info classes, we supply an `@Property` annotation. You can tag either the property getter or setter, like this:

```
@Property String getTitle() { return title; }
```

or

```
@Property(editor="TitlePositionEditor")
public void setTitlePosition(int p) { titlePosition = p; }
```

Listing 11-11 contains the definition of the `@Property` annotation. Note that the annotation has a retention policy `SOURCE`. We analyze the annotation at the source level only. It is not included in class files and not available durin reflection.

Listing 11-11. **Property.java**

```
package com.horstmann.annotations;
import java.lang.annotation.*;

@Documented
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.SOURCE)
public @interface Property
{
   String editor() default "";
}
```

Note

It would have made sense to declare the `editor` element to have type `Class`. However, the annotation processor cannot retrieve annotations of type `Class` because the meaning of a class can depend on external factors (such as the class path or class loaders). Therefore, we use a string to specify the editor class name.

To automatically generate the bean info class of a class with name *BeanClass* , we carry out the following tasks:

1. Write a source file *BeanClass* `BeanInfo.java` . Declare the *BeanClass* `BeanInfo` class to extend `SimpleBeanInfo` , and override the `getPropertyDescriptors` method.
2. For each annotated method, recover the property name by stripping off the `get` or `set` prefix and "decapitalizing" the remainder.
3. For each property, write a statement for constructing a `PropertyDescriptor` .
4. If the property has an editor, write a method call to `setPropertyEditorClass` .
5. Write code for returning an array of all property descriptors.

For example, the annotation

```
@Property(editor="TitlePositionEditor")
public void setTitlePosition(int p) { titlePosition = p; }
```

in the `ChartBean` class is translated into

Code View:
```
public class ChartBeanBeanInfo extends java.beans.SimpleBeanInfo
{
   public java.beans.PropertyDescriptor[] getProperties()
   {
      java.beans.PropertyDescriptor titlePositionDescriptor
         = new java.beans.PropertyDescriptor("titlePosition", ChartBean.class);
      titlePositionDescriptor.setPropertyEditorClass(TitlePositionEditor.class)
      . . .
      return new java.beans.PropertyDescriptor[]
      {
         titlePositionDescriptor,
         . . .
      }
   }
}
```

(The boilerplate code is printed in the lighter gray.)

All this is easy enough to do, provided we can locate all methods that have been tagged with the `@Property` annotation.

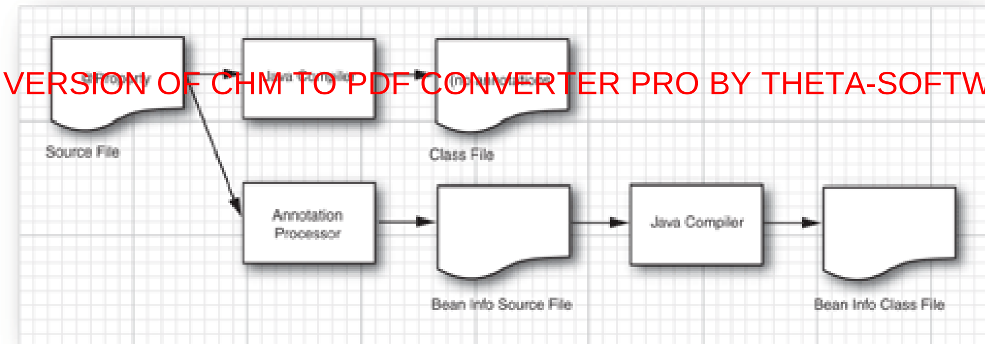
As of Java SE 6, you can add *annotation processors* to the Java compiler. (In Java SE 5, a stand-alone tool, called `apt` , was used for the same purpose.) To invoke annotation processing, run

`javac -processor` $ProcessorClassName_1$, $ProcessorClassName_2$, ... *sourceFiles*

The compiler locates the annotations of the source files. It then selects the annotation processors that should be applied. Each annotation processor is executed in turn. If an annotation processor creates a new source file, then the process is repeated. If a processing round yields no further source files, then all source files are compiled. Figure 11-3 shows how the `@Property` annotations are processed.

Figure 11-3. Processing source-level annotations

[View full size image]



We do not discuss the annotation processing API in detail, but the program in Listing 11-12 will give you a flavor o its capabilities.

An annotation processor implements the `Processor` interface, generally by extending the `AbstractProcessor` clas You need to specify which annotations your processor supports. Because the designers of the API love annotations they use an annotation for this purpose:

```
@SupportedAnnotationTypes("com.horstmann.annotations.Property")
public class BeanInfoAnnotationProcessor extends AbstractProcessor
```

A processor can claim specific annotation types, wildcards such as `"com.horstmann.*"` (all annotations in the `com.horstmann` package or any subpackage), or even `"*"` (all annotations).

The `BeanInfoAnnotationProcessor` has a single public method, `process` , that is called for each file. The `process` method has two parameters, the set of annotations that is being processed in this round, and a `RoundEnv` referenc that contains information about the current processing round.

In the `process` method, we iterate through all annotated methods. For each method, we get the property name by stripping off the `get` , `set` , or `is` prefix and changing the next letter to lower case. Here is the outline of the code

Code View:

```
public boolean process(Set<? extends TypeElement> annotations, RoundEnvironment roundEnv)
  {
     for (TypeElement t : annotations)
     {
        Map<String, Property> props = new LinkedHashMap<String, Property>();
        for (Element e : roundEnv.getElementsAnnotatedWith(t))
        {
           props.put(property name, e.getAnnotation(Property.class));
```

```
        }
      }
      write bean info source file
    }
    return true;
}
```

The `process` method should return `true` if it *claims* all the annotations presented to it; that is, if those annotations should not be passed on to other processors.

The code for writing the source file is straightforward, just a sequence of `out.print` statements. Note that we create the output writer as follows:

Code View:

```
JavaFileObject sourceFile = processingEnv.getFiler().createSourceFile(beanClassName + "BeanInfo"
PrintWriter out = new PrintWriter(sourceFile.openWriter());
```

The `AbstractProcessor` class has a protected field `processingEnv` for accessing various processing services. The `Filer` interface is responsible for creating new files and tracking them so that they can be processed in subsequen processing rounds.

When an annotation processor detects an error, it uses the `Messager` to communicate with the user. For example, we issue an error message if a method has been annotated with `@Property` but its name doesn't start with `get` , `set` , or `is` :

```
if (!found) processingEnv.getMessager().printMessage(Kind.ERROR,
   "@Property must be applied to getXxx, setXxx, or isXxx method", e);
```

In the companion code for this book, we supply you with an annotated file, `ChartBean.java` . Compile the annotation processor:

```
javac BeanInfoAnnotationProcessor.java
```

Then run

Code View:

```
javac -processor BeanInfoAnnotationProcessor com/horstmann/corejava/ChartBean.java
```

and have a look at the automatically generated file `ChartBeanBeanInfo.java` .

To see the annotation processing in action, add the command-line option `XprintRounds` to the `javac` command. You will get this output:

```
Round 1:
        input files: {com.horstmann.corejava.ChartBean}
```

```
        annotations: [com.horstmann.annotations.Property]
        last round: false
Round 2:
        input files: {com.horstmann.corejava.ChartBeanBeanInfo}
        annotations: []
        last round: false
Round 3:
        input files: {}
        annotations: []
        last round: true
```


This example demonstrates how tools can harvest source file annotations to produce other files. The generated files don't have to be source files. Annotation processors may choose to generate XML descriptors, property files, shell scripts, HTML documentation, and so on.

Note

Some people have suggested using annotations to remove an even bigger drudgery. Wouldn't it be nice if trivial getters and setters were generated automatically? For example, the annotation

```
@Property private String title;
```

could produce the methods

```
public String getTitle() { return title; }
public void setTitle(String title) { this.title = title; }
```

However, those methods need to be added to the *same class* . This requires editing a source file, not just generating another file, and is beyond the capabilities of annotation processors. It would be possible to build another tool for this purpose, but such a tool would go beyond the mission of annotations. An annotation is intended as a description *about* a code item, not a directive for adding or changing code.

Listing 11-12. **BeanInfoAnnotationFactory.java**

Code View:

```
import java.beans.*;
import java.io.*;
import java.util.*;
import javax.annotation.processing.*;
import javax.lang.model.*;
import javax.lang.model.element.*;
import javax.tools.*;
import javax.tools.Diagnostic.*;
import com.horstmann.annotations.*;

/**
 * This class is the processor that analyzes Property annotations.
```

```
 * @version 1.10 2007-10-27
 * @author Cay Horstmann
 */

@SupportedAnnotationTypes("com.horstmann.annotations.Property")
@SupportedSourceVersion(SourceVersion.RELEASE_6)
public class BeanInfoAnnotationProcessor extends AbstractProcessor
{
   @Override
   public boolean process(Set<? extends TypeElement> annotations, RoundEnvironment roundEnv
   {
      for (TypeElement t : annotations)
      {
         Map<String, Property> props = new LinkedHashMap<String, Property>();
         String beanClassName = null;
         for (Element e : roundEnv.getElementsAnnotatedWith(t))
         {
            String mname = e.getSimpleName().toString();
            String[] prefixes = { "get", "set", "is" };
            boolean found = false;
            for (int i = 0; !found && i < prefixes.length; i++)
               if (mname.startsWith(prefixes[i]))
               {
                  found = true;
                  int start = prefixes[i].length();
                  String name = Introspector.decapitalize(mname.substring(start));
                  props.put(name, e.getAnnotation(Property.class));
               }

            if (!found) processingEnv.getMessager().printMessage(Kind.ERROR,
                  "@Property must be applied to getXxx, setXxx, or isXxx method", e);
            else if (beanClassName == null)
               beanClassName = ((TypeElement) e.getEnclosingElement()).getQualifiedName()
                     .toString();
         }
         try
         {
            if (beanClassName != null) writeBeanInfoFile(beanClassName, props);
         }
         catch (IOException e)
         {
            e.printStackTrace();
         }
      }
      return true;
   }

   /**
    * Writes the source file for the BeanInfo class.
    * @param beanClassName the name of the bean class
    * @param props a map of property names and their annotations
    */
   private void writeBeanInfoFile(String beanClassName, Map<String, Property> props)
         throws IOException
   {
      JavaFileObject sourceFile = processingEnv.getFiler().createSourceFile(
            beanClassName + "BeanInfo");
      PrintWriter out = new PrintWriter(sourceFile.openWriter());
      int i = beanClassName.lastIndexOf(".");
```

```
        if (i > 0)
        {
            out.print("package ");
            out.print(beanClassName.substring(0, i));
            out.println(";");
        }
        out.print("public class ");
        out.print(beanClassName.substring(i + 1));
        out.println("BeanInfo extends java.beans.SimpleBeanInfo");
        out.println("{");
        out.println("   public java.beans.PropertyDescriptor[] getPropertyDescriptors()");
        out.println("   {");
        out.println("      try");
        out.println("      {");
        for (Map.Entry<String, Property> e : props.entrySet())
        {
            out.print("         java.beans.PropertyDescriptor ");
            out.print(e.getKey());
            out.println("Descriptor");
            out.print("            = new java.beans.PropertyDescriptor(\"");
            out.print(e.getKey());
            out.print("\", ");
            out.print(beanClassName);
            out.println(".class);");
            String ed = e.getValue().editor().toString();
            if (!ed.equals(""))
            {
                out.print("         ");
                out.print(e.getKey());
                out.print("Descriptor.setPropertyEditorClass(");
                out.print(ed);
                out.println(".class);");
            }
        }
        out.println("         return new java.beans.PropertyDescriptor[]");
        out.print("         {");
        boolean first = true;
        for (String p : props.keySet())
        {
            if (first) first = false;
            else out.print(",");
            out.println();
            out.print("            ");
            out.print(p);
            out.print("Descriptor");
        }
        out.println();
        out.println("         };");
        out.println("      }");
        out.println("      catch (java.beans.IntrospectionException e)");
        out.println("      {");
        out.println("         e.printStackTrace();");
        out.println("         return null;");
        out.println("      }");
        out.println("   }");
        out.println("}");
        out.close();
    }
}
```

Bytecode Engineering

You have seen how annotations can be processed at runtime or at the source code level. There is a third possibility bytecode level. Unless annotations are removed at the source level, they are present in the class files. The class fi (see http://java.sun.com/docs/books/vmspec). The format is rather complex, and it would be challenging to proc special libraries. One such library is the Bytecode Engineering Library (BCEL), available at http://jakarta.apache.o

In this section, we use BCEL to add logging messages to annotated methods. If a method is annotated with

```
@LogEntry(logger=loggerName)
```


then we add the bytecodes for the following statement at the beginning of the method:

```
Logger.getLogger(loggerName).entering(className, methodName);
```

For example, if you annotate the `hashCode` method of the `Item` class as

```
@LogEntry(logger="global") public int hashCode()
```

then a message similar to the following is printed whenever the method is called:

```
Aug 17, 2004 9:32:59 PM Item hashCode
FINER: ENTRY
```

To achieve this task, we do the following:

1. Load the bytecodes in the class file.
2. Locate all methods.
3. For each method, check whether it has a `LogEntry` annotation.
4. If it does, add the bytecodes for the following instructions at the beginning of the method:

   Code View:
   ```
   ldc loggerName
   invokestatic  java/util/logging/Logger.getLogger:(Ljava/lang/String;)Ljava/util/loggi
   ldc className
   ldc methodName
   invokevirtual  java/util/logging/Logger.entering:(Ljava/lang/String;Ljava/lang/String
   ```

Inserting these bytecodes sounds tricky, but BCEL makes it fairly straightforward. We don't describe the process o bytecodes in detail. The important point is that the program in Listing 11-13 edits a class file and inserts a logging the methods that are annotated with the `LogEntry` annotation.

Note

If you are interested in the details of bytecode engineering, we suggest that you read through the BC
manual at http://jakarta.apache.org/bcel/manual.html .

You need version 5.3 or later of the BCEL library to compile and run the EntryLogger program. (As this chapter wa
still a work in progress. If it isn't finished when you read this, check out the trunk from the Subversion repository.)

For example, here is how you add the logging instructions to Item.java in Listing 11-14 :

```
javac Item.java
javac -classpath .:bcel-version.jar EntryLogger.java
java -classpath .:bcel-version.jar EntryLogger Item
```

Try running

```
javap -c Item
```

before and after modifying the Item class file. You can see the inserted instructions at the beginning of the hashCo
compareTo methods.

Code View:

```
public int hashCode();
  Code:
   0:   ldc     #85; //String global
   2:   invokestatic    #80; //Method java/util/logging/Logger.getLogger:(Ljava/lang/String;)Ljav
   5:   ldc     #86; //String Item
   7:   ldc     #88; //String hashCode
   9:   invokevirtual   #84; //Method java/util/logging/Logger.entering:(Ljava/lang/String;Ljava/
   12:  bipush  13
   14:  aload_0
   15:  getfield        #2; //Field description:Ljava/lang/String;
   18:  invokevirtual   #15; //Method java/lang/String.hashCode:()I
   21:  imul
   22:  bipush  17
   24:  aload_0
   25:  getfield        #3; //Field partNumber:I
   28:  imul
   29:  iadd
   30:  ireturn
```

The SetTest program in Listing 11-15 inserts Item objects into a hash set. When you run it with the modified clas
logging messages.

Code View:

```
Aug 18, 2004 10:57:59 AM Item hashCode
```

```
FINER: ENTRY
Aug 18, 2004 10:57:59 AM Item hashCode
FINER: ENTRY
Aug 18, 2004 10:57:59 AM Item hashCode
FINER: ENTRY
Aug 18, 2004 10:57:59 AM Item equals
FINER: ENTRY
[[descripion=Toaster, partNumber=1729], [descripion=Microwave, partNumber=4104]]
```


Note the call to `equals` when we insert the same item twice.

This example shows the power of bytecode engineering. Annotations are used to add directives to a program. A by up the directives and modifies the virtual machine instructions.

Listing 11-13. **`EntryLogger.java`**

Code View:

```
import java.io.*;
import org.apache.bcel.*;
import org.apache.bcel.classfile.*;
import org.apache.bcel.generic.*;

/**
 * Adds "entering" logs to all methods of a class that have the LogEntry annotation.
 * @version 1.10 2007-10-27
 * @author Cay Horstmann
 */
public class EntryLogger
{
   /**
    * Adds entry logging code to the given class
    * @param args the name of the class file to patch
    */
   public static void main(String[] args)
   {
      try
      {
         if (args.length == 0) System.out.println("USAGE: java EntryLogger classname");
         else
         {
            JavaClass jc = Repository.lookupClass(args[0]);
            ClassGen cg = new ClassGen(jc);
            EntryLogger el = new EntryLogger(cg);
            el.convert();
            File f = new File(Repository.lookupClassFile(cg.getClassName()).getPath());
            cg.getJavaClass().dump(f.getPath());
         }
      }
      catch (Exception e)
      {
         e.printStackTrace();
      }
   }

   /**
```

```
    * Constructs an EntryLogger that inserts logging into annotated methods of a given class
    * @param cg the class
    */
   public EntryLogger(ClassGen cg)
   {
      this.cg = cg;
      cpg = cg.getConstantPool();
   }

   /**
    * converts the class by inserting the logging calls.
    */
   public void convert() throws IOException
   {
      for (Method m : cg.getMethods())
      {
         AnnotationEntry[] annotations = m.getAnnotationEntries();
         for (AnnotationEntry a : annotations)
         {
            if (a.getAnnotationType().equals("LLogEntry;"))
            {
               for (ElementValuePair p : a.getElementValuePairs())
               {
                  if (p.getNameString().equals("logger"))
                  {
                     String loggerName = p.getValue().stringifyValue();
                     cg.replaceMethod(m, insertLogEntry(m, loggerName));
                  }
               }
            }
         }
      }
   }

   /**
    * Adds an "entering" call to the beginning of a method.
    * @param m the method
    * @param loggerName the name of the logger to call
    */
   private Method insertLogEntry(Method m, String loggerName)
   {
      MethodGen mg = new MethodGen(m, cg.getClassName(), cpg);
      String className = cg.getClassName();
      String methodName = mg.getMethod().getName();
      System.out.printf("Adding logging instructions to %s.%s%n", className, methodName);

      int getLoggerIndex = cpg.addMethodref("java.util.logging.Logger", "getLogger",
            "(Ljava/lang/String;)Ljava/util/logging/Logger;");
      int enteringIndex = cpg.addMethodref("java.util.logging.Logger", "entering",
            "(Ljava/lang/String;Ljava/lang/String;)V");

      InstructionList il = mg.getInstructionList();
      InstructionList patch = new InstructionList();
      patch.append(new PUSH(cpg, loggerName));
      patch.append(new INVOKESTATIC(getLoggerIndex));
      patch.append(new PUSH(cpg, className));
      patch.append(new PUSH(cpg, methodName));
      patch.append(new INVOKEVIRTUAL(enteringIndex));
      InstructionHandle[] ihs = il.getInstructionHandles();
```

```
      il.insert(ihs[0], patch);

      mg.setMaxStack();
      return mg.getMethod();
   }

   private ClassGen cg;
   private ConstantPoolGen cpg;
}
```


Listing 11-14. **Item.java**

Code View:

```
/**
 * An item with a description and a part number.
 * @version 1.00 2004-08-17
 * @author Cay Horstmann
 */
public class Item
{
   /**
    * Constructs an item.
    * @param aDescription the item's description
    * @param aPartNumber the item's part number
    */
   public Item(String aDescription, int aPartNumber)
   {
      description = aDescription;
      partNumber = aPartNumber;
   }

   /**
    * Gets the description of this item.
    * @return the description
    */
   public String getDescription()
   {
      return description;
   }

   public String toString()
   {
      return "[description=" + description + ", partNumber=" + partNumber + "]";
   }

   @LogEntry(logger = "global")
   public boolean equals(Object otherObject)
   {
      if (this == otherObject) return true;
      if (otherObject == null) return false;
      if (getClass() != otherObject.getClass()) return false;
      Item other = (Item) otherObject;
```

```
      return description.equals(other.description) && partNumber == other.partNumber;
   }

   @LogEntry(logger = "global")
   public int hashCode()
   {
      return 13 * description.hashCode() + 17 * partNumber;
   }

   private String description;
   private int partNumber;
}
```

Listing 11-15. **SetTest.java**

Code View:

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

/**
 * @version 1.01 2007-10-27
 * @author Cay Horstmann
 */
public class SetTest
{
   public static void main(String[] args)
   {
      Logger.getLogger(Logger.GLOBAL_LOGGER_NAME).setLevel(Level.FINEST);
      Handler handler = new ConsoleHandler();
      handler.setLevel(Level.FINEST);
      Logger.getLogger(Logger.GLOBAL_LOGGER_NAME).addHandler(handler);

      Set<Item> parts = new HashSet<Item>();
      parts.add(new Item("Toaster", 1279));
      parts.add(new Item("Microwave", 4104));
      parts.add(new Item("Toaster", 1279));
      System.out.println(parts);
   }
}
```

Modifying Bytecodes at Load Time

In the preceding section, you saw a tool that edits class files. However, it can be cumbersome to add yet another t
An attractive alternative is to defer the bytecode engineering until *load time* , when the class loader loads the clas

Before Java SE 5.0, you had to write a custom classloader to achieve this task. Now, the *instrumentation API* has a
bytecode transformer. The transformer must be installed before the `main` method of the program is called. You ha
defining an *agent* , a library that is loaded to monitor a program in some way. The agent code can carry out initiali
method.

Here are the steps required to build an agent:

1. Implement a class with a method

   ```
   public static void premain(String arg, Instrumentation instr)
   ```

   This method is called when the agent is loaded. The agent can get a single command-line argument, which is
   parameter. The `instr` parameter can be used to install various hooks.

2. Make a manifest file that sets the `Premain-Class` attribute, for example:

   ```
   Premain-Class: EntryLoggingAgent
   ```

3. Package the agent code and the manifest into a JAR file, for example:


   ```
   javac -classpath .:bcel-version.jar EntryLoggingAgent
   jar cvfm EntryLoggingAgent.jar EntryLoggingAgent.mf Entry*.class
   ```

To launch a Java program together with the agent, use the following command-line options:

```
java -javaagent:AgentJARFile=agentArgument . . .
```

For example, to run the `SetTest` program with the entry logging agent, call

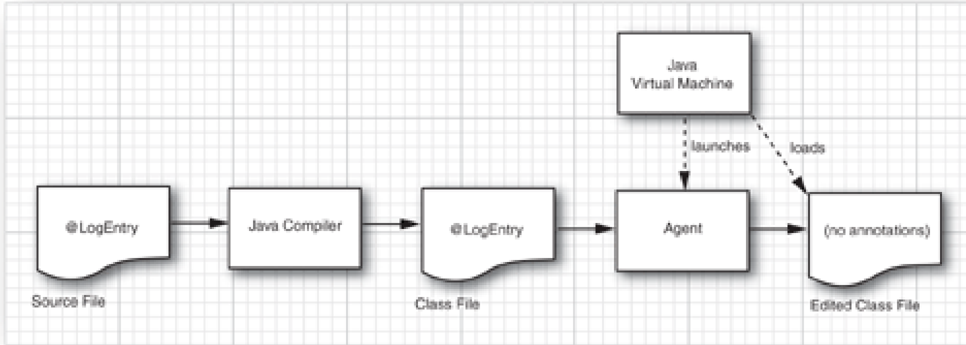
Code View:
```
javac SetTest.java
java -javaagent:EntryLoggingAgent.jar=Item -classpath .:bcel-version.jar SetTest
```

The `Item` argument is the name of the class that the agent should modify.

Listing 11-16 shows the agent code. The agent installs a class file transformer. The transformer first checks wheth
the agent argument. If so, it uses the `EntryLogger` class from the preceding section to modify the bytecodes. How
bytecodes are not saved to a file. Instead, the transformer returns them for loading into the virtual machine (see F
words, this technique carries out "just in time" modification of the bytecodes.

Figure 11-4. Modifying classes at load time

[View full size image]



Listing 11-16. **EntryLoggingAgent.java**

Code View:

```
import java.lang.instrument.*;
import java.io.*;
import java.security.*;
import org.apache.bcel.classfile.*;
import org.apache.bcel.generic.*;

/**
 * @version 1.00 2004-08-17
 * @author Cay Horstmann
 */
public class EntryLoggingAgent
{
   public static void premain(final String arg, Instrumentation instr)
   {
      instr.addTransformer(new ClassFileTransformer()
         {
            public byte[] transform(ClassLoader loader, String className, Class<?> cl,
                  ProtectionDomain pd, byte[] data)
            {
               if (!className.equals(arg)) return null;
               try
               {
                  ClassParser parser = new ClassParser(new ByteArrayInputStream(data),
                        className + ".java");
                  JavaClass jc = parser.parse();
                  ClassGen cg = new ClassGen(jc);
                  EntryLogger el = new EntryLogger(cg);
                  el.convert();
                  return cg.getJavaClass().getBytes();
               }
               catch (Exception e)
               {
                  e.printStackTrace();
                  return null;
               }
            }
         });
   }
}
```

In this chapter, you have learned how to

- Add annotations to Java programs.

- Design your own annotation interfaces.

- Implement tools that make use of the annotations.

You have seen three technologies for processing code: scripting, compiling Java programs, and processing annotat quite straightforward. On the other hand, building annotation tools is undeniably complex and not something that to tackle. This chapter gave you the background knowledge for understanding the inner workings of the annotatio encounter, and perhaps piqued your interest in developing your own tools.

In the final chapter of this book, we tackle the API for native methods. That API allows you to mix Java and C/C++

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

While a "100% Pure Java" solution is nice in principle, there are situations in which you will want to write (or use) code written in another language. (Such code is usually called *native* code.)

Particularly in the early days of Java, many people assumed that it would be a good idea to use C or C++ to speed up critical parts of a Java application. However, in practice, this was rarely useful. A presentation at the 1996 JavaOne conference showed this clearly. The implementors of the cryptography library at Sun Microsystems reported that a pure Java platform implementation of their cryptographic functions was more than adequate. It was true that the code was not as fast as a C implementation would have been, but it turned out not to matter. The Java platform implementation was far faster than the network I/O. This turned out to be the real bottleneck.

Of course, there are drawbacks to going native. If a part of your application is written in another language, you must supply a separate native library for every platform you want to support. Code written in C or C++ offers no protection against overwriting memory through invalid pointer usage. It is easy to write native methods that corrupt your program or infect the operating system.

Thus, we suggest using native code only when you need to. In particular, there are three reasons why native code might be the right choice:

- Your application requires access to system features or devices that are not accessible through the Java platform.
- You have substantial amounts of tested and debugged code in another language, and you know how to port it to all desired target platforms.

- You have found, through benchmarking, that the Java code is much slower than the equivalent code in another language.

The Java platform has an API for interoperating with native C code called the Java Native Interface (JNI). We discuss JNI programming in this chapter.

C++ Note

You can also use C++ instead of C to write native methods. There are a few advantages—type checking is slightly stricter, and accessing the JNI functions is a bit more convenient. However, JNI does not support any mapping between Java and C++ classes.

Calling a C Function from a Java Program

Suppose you have a C function that does something you like and, for one reason or another, you don't want to bother reimplementing it in Java. For the sake of illustration, we start with a simple C function that prints a greeting.

The Java programming language uses the keyword `native` for a native method, and you will obviously need to place a method in a class. The result is shown in Listing 12-1.

The `native` keyword alerts the compiler that the method will be defined externally. Of course, native methods will contain no code in the Java programming language, and the method header is followed immediately by a terminating semicolon. Therefore, native method declarations look similar to abstract method declarations.

Listing 12-1. **`HelloNative.java`**

```
/**
 * @version 1.11 2007-10-26
 * @author Cay Horstmann
 */
class HelloNative
{
   public static native void greeting();
}
```

In this particular example, the native method is also declared as `static`. Native methods can be both static and nonstatic. We start with a static method because we do not yet want to deal with parameter passing.

You actually can compile this class, but when you go to use it in a program, then the virtual machine will tell you it doesn't know how to find `greeting`—reporting an `UnsatisfiedLinkError`. To implement the native code, write a corresponding C function. You must name that function *exactly* the way the Java virtual machine expects. Here are the rules:

1. Use the full Java method name, such as `HelloNative.greeting`. If the class is in a package, then prepend the package name, such as `com.horstmann.HelloNative.greeting`.
2. Replace every period with an underscore, and append the prefix `Java_`. For example, `Java_HelloNative_greeting` or `Java_com_horstmann_HelloNative_greeting`.
3. If the class name contains characters that are not ASCII letters or digits—that is, `'_'`, `'$'`, or Unicode characters with code greater than `'\u007F'`—replace them with `_0`*xxxx*, where *xxxx* is the sequence of four hexadecimal digits of the character's Unicode value.

Note

If you *overload* native methods, that is, if you provide multiple native methods with the same name, then you must append a double underscore followed by the encoded argument types. (We describe the encoding of the argument types later in this chapter.) For example, if you have a native method, `greeting`, and another native method, `greeting(int repeat)`, then the first one is called `Java_HelloNative_greeting__`, and the second, `Java_HelloNative_greeting__I`.

Actually, nobody does this by hand; instead, you run the `javah` utility, which automatically generates the function names. To use `javah`, first, compile the source file in Listing 12-1:

```
javac HelloNative.java
```

Next, call the `javah` utility, which produces a C header file from the class file. The `javah` executable can be found in the *jdk*/`bin` directory. You invoke it with the name of the class, just as you would invoke the Java compiler. For example,

```
javah HelloNative
```

This command creates a header file, `HelloNative.h`, which is shown in Listing 12-2.

Listing 12-2. **`HelloNative.h`**


```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class HelloNative */

#ifndef _Included_HelloNative
#define _Included_HelloNative
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     HelloNative
 * Method:    greeting
 * Signature: ()V
 */
JNIEXPORT void JNICALL Java_HelloNative_greeting
  (JNIEnv *, jclass);

#ifdef __cplusplus
}
#endif
#endif
```

As you can see, this file contains the declaration of a function `Java_HelloNative_greeting`. (The macros `JNIEXPORT` and `JNICALL` are defined in the header file `jni.h`. They denote compiler-dependent specifiers for exported functions that come from a dynamically loaded library.)

Now, you simply copy the function prototype from the header file into the source file and give the implementation code for the function, as shown in Listing 12-3.

Listing 12-3. **`HelloNative.c`**

Code View:
```
/*
   @version 1.10 1997-07-01
   @author Cay Horstmann
*/

#include "HelloNative.h"
#include <stdio.h>

JNIEXPORT void JNICALL Java_HelloNative_greeting(JNIEnv* env, jclass cl)
{
   printf("Hello Native World!\n");
}
```

In this simple function, ignore the `env` and `cl` arguments. You'll see their use later.

C++ Note

You can use C++ to implement native methods. However, you must then declare the functions that implement the native methods as `extern "C"`. (This stops the C++ compiler from "mangling" the method name.) For example,

```
extern "C"
JNIEXPORT void JNICALL Java_HelloNative_greeting(JNIEnv* env, jclass cl)
{
   cout << "Hello, Native World!" << endl;
}
```

You compile the native C code into a dynamically loaded library. The details depend on your compiler.

For example, with the Gnu C compiler on Linux, use these commands:

Code View:
```
gcc -fPIC -I jdk/include -I jdk/include/linux -shared -o libHelloNative.so HelloNative.c
```

With the Sun compiler under the Solaris Operating System, the command is

Code View:
```
cc -G -I jdk/include -I jdk/include/solaris -o libHelloNative.so HelloNative.c
```

With the Microsoft compiler under Windows, the command is

```
cl -I jdk\include -I jdk\include\win32 -LD HelloNative.c -FeHelloNative.dll
```

Here, *jdk* is the directory that contains the JDK.

Tip

If you use the Microsoft compiler from a command shell, first run the batch file `vcvars32.bat` or `vsvars32.bat`. That batch file sets up the path and the

environment variables needed by the compiler. You can find it in the directory `c:\Program Files\Microsoft Visual Studio .NET 2003\Common7\tools`, `c:\Program Files\Microsoft Visual Studio 8\VC`, or a similar monstrosity.

You can also use the freely available Cygwin programming environment, available from http://www.cygwin.com. It contains the Gnu C compiler and libraries for UNIX-style programming on Windows. With Cygwin, use the command

```
gcc -mno-cygwin -D __int64="long long" -I jdk/include/ -I jdk/include/win32
   -shared -Wl,--add-stdcall-alias -o HelloNative.dll HelloNative.c
```

Type the entire command on a single line.

Note

The Windows version of the header file `jni_md.h` contains the type declaration

```
typedef __int64 jlong;
```

which is specific to the Microsoft compiler. If you use the Gnu compiler, you might want to edit that file, for example,

```
#ifdef __GNUC__
   typedef long long jlong;
#else
   typedef __int64 jlong;
#endif
```

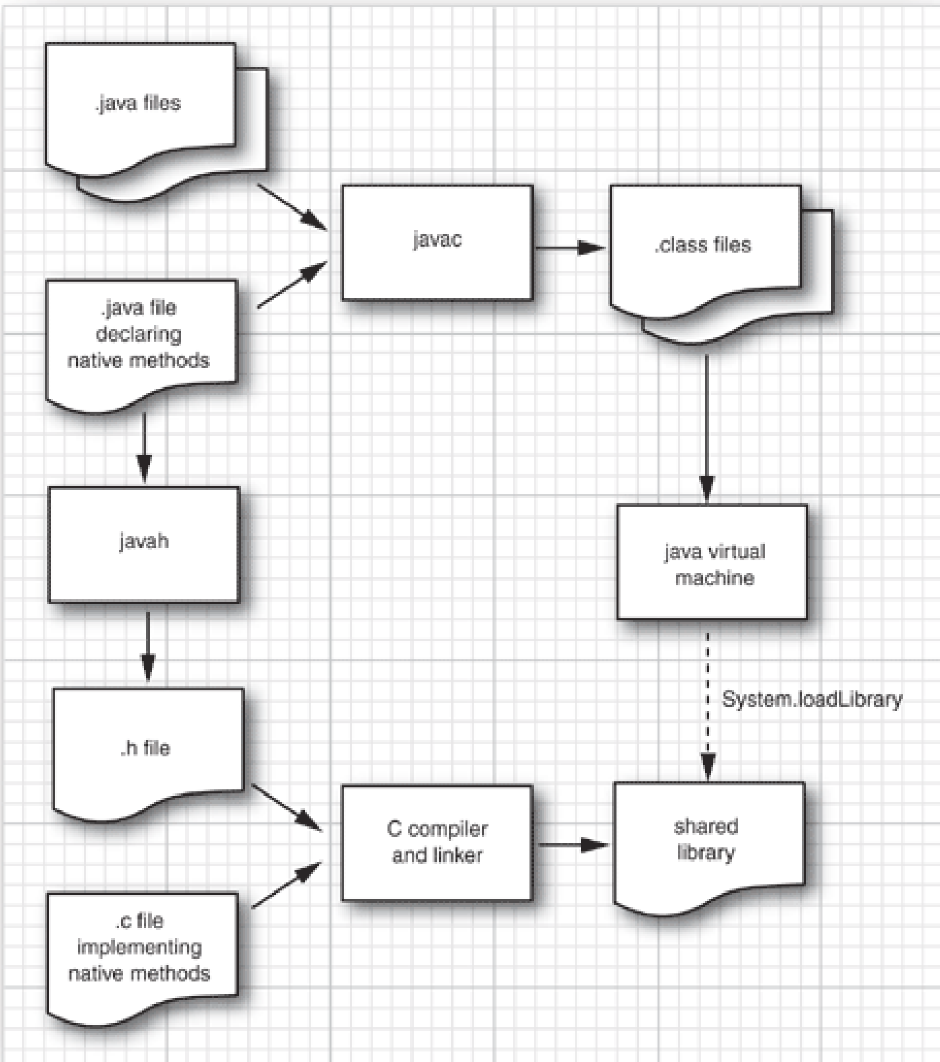
Alternatively, compile with `-D __int64="long long"`, as shown in the sample compiler invocation.

Finally, add a call to the `System.loadLibrary` method in your program. To ensure that the virtual machine will load the library before the first use of the class, use a static initialization block, as in Listing 12-4.

Figure 12-1 gives a summary of the native code processing.

Figure 12-1. Processing native code

[View full size image]



Listing 12-4. **`HelloNativeTest.java`**

```
/**
 * @version 1.11 2007-10-26
 * @author Cay Horstmann
 */
class HelloNativeTest
{
   public static void main(String[] args)
   {
      HelloNative.greeting();
   }

   static
   {
      System.loadLibrary("HelloNative");
   }
}
```


UNREGISTERED VERSION OF CHM TO PDF CONVERTER PRO BY THETA-SOFTWARE

If you compile and run this program, the message "Hello, Native World!" is displayed in a terminal window.

Note

If you run Linux, you must add the current directory to the library path. Either set the `LD_LIBRARY_PATH` environment variable,

```
export LD_LIBRARY_PATH=.:$LD_LIBRARY_PATH
```

or set the `java.library.path` system property:

```
java -Djava.library.path=. HelloNativeTest
```

Of course, this is not particularly impressive by itself. However, if you keep in mind that this message is generated by the C `printf` command and not by any Java programming language code, you will see that we have taken the first steps toward bridging the gap between the two languages!

In summary, you follow these steps to link a native method to a Java program:

1. Declare a native method in a Java class.
2. Run `javah` to get a header file with a C declaration for the method.
3. Implement the native method in C.
4. Place the code in a shared library.
5. Load that library in your Java program.

java.lang.System 1.0

- `void loadLibrary(String libname)`

  loads the library with the given name. The library is located in the library search path. The exact method for locating the library is operating-system dependent.

Note

Some shared libraries for native code must run initialization code. You can place any initialization code into a `JNI_OnLoad` method. Similarly, when the virtual machine (VM) shuts down, it will call the `JNI_OnUnload` method if you provide it. The prototypes are

```
jint JNI_OnLoad(JavaVM* vm, void* reserved);
void JNI_OnUnload(JavaVM* vm, void* reserved);
```

The `JNI_OnLoad` method needs to return the minimum version of the VM that it requires, such as `JNI_VERSION_1_2`.

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

While a "100% Pure Java" solution is nice in principle, there are situations in which you will want to write (or use) code written in another language. (Such code is usually called *native* code.)

Particularly in the early days of Java, many people assumed that it would be a good idea to use C or C++ to speed up critical parts of a Java application. However, in practice, this was rarely useful. A presentation at the 1996 JavaOne conference showed this clearly. The implementors of the cryptography library at Sun Microsystems reported that a pure Java platform implementation of their cryptographic functions was more than adequate. It was true that the code was not as fast as a C implementation would have been, but it turned out not to matter. The Java platform implementation was far faster than the network I/O. This turned out to be the real bottleneck.

Of course, there are drawbacks to going native. If a part of your application is written in another language, you must supply a separate native library for every platform you want to support. Code written in C or C++ offers no protection against overwriting memory through invalid pointer usage. It is easy to write native methods that corrupt your program or infect the operating system.

Thus, we suggest using native code only when you need to. In particular, there are three reasons why native code might be the right choice:

- Your application requires access to system features or devices that are not accessible through the Java platform.
- You have substantial amounts of tested and debugged code in another language, and you know how to port it to all desired target platforms.
- You have found, through benchmarking, that the Java code is much slower than the equivalent code in another language.

The Java platform has an API for interoperating with native C code called the Java Native Interface (JNI). We discuss JNI programming in this chapter.

C++ Note

You can also use C++ instead of C to write native methods. There are a few advantages—type checking is slightly stricter, and accessing the JNI functions is a bit more convenient. However, JNI does not support any mapping between Java and C++ classes.

Calling a C Function from a Java Program

Suppose you have a C function that does something you like and, for one reason or another, you don't want to bother reimplementing it in Java. For the sake of illustration, we start with a simple C function that prints a greeting.

The Java programming language uses the keyword `native` for a native method, and you will obviously need to place a method in a class. The result is shown in Listing 12-1.

The `native` keyword alerts the compiler that the method will be defined externally. Of course, native methods will contain no code in the Java programming language, and the method header is followed immediately by a terminating semicolon. Therefore, native method declarations look similar to abstract method declarations.

Listing 12-1. **HelloNative.java**

```
/**
 * @version 1.11 2007-10-26
 * @author Cay Horstmann
 */
class HelloNative
{
   public static native void greeting();
}
```


In this particular example, the native method is also declared as `static`. Native methods can be both static and nonstatic. We start with a static method because we do not yet want to deal with parameter passing.

You actually can compile this class, but when you go to use it in a program, then the virtual machine will tell you it doesn't know how to find `greeting`—reporting an `UnsatisfiedLinkError`. To implement the native code, write a corresponding C function. You must name that function *exactly* the way the Java virtual machine expects. Here are the rules:

1. Use the full Java method name, such as `HelloNative.greeting`. If the class is in a package, then prepend the package name, such as `com.horstmann.HelloNative.greeting`.

2. Replace every period with an underscore, and append the prefix `Java_`. For example, `Java_HelloNative_greeting` or `Java_com_horstmann_HelloNative_greeting`.

3. If the class name contains characters that are not ASCII letters or digits—that is, `'_'`, `'$'`, or Unicode characters with code greater than `'\u007F'`—replace them with `_0`*xxxx*, where *xxxx* is the sequence of four hexadecimal digits of the character's Unicode value.

Note

If you *overload* native methods, that is, if you provide multiple native methods with the same name, then you must append a double underscore followed by the encoded argument types. (We describe the encoding of the argument types later in this chapter.) For example, if you have a native method, `greeting`, and another native method, `greeting(int repeat)`, then the first one is called `Java_HelloNative_greeting__`, and the second, `Java_HelloNative_greeting__I`.

Actually, nobody does this by hand; instead, you run the `javah` utility, which automatically generates the function names. To use `javah`, first, compile the source file in Listing 12-1:

```
javac HelloNative.java
```

Next, call the `javah` utility, which produces a C header file from the class file. The `javah` executable can be found in the *jdk*/bin directory. You invoke it with the name of the class, just as you would invoke the Java compiler. For example,

```
javah HelloNative
```

This command creates a header file, HelloNative.h, which is shown in Listing 12-2.

Listing 12-2. **HelloNative.h**

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class HelloNative */

#ifndef _Included_HelloNative
#define _Included_HelloNative
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     HelloNative
 * Method:    greeting
 * Signature: ()V
 */
JNIEXPORT void JNICALL Java_HelloNative_greeting
  (JNIEnv *, jclass);

#ifdef __cplusplus
}
#endif
#endif
```

As you can see, this file contains the declaration of a function Java_HelloNative_greeting. (The macros JNIEXPORT and JNICALL are defined in the header file jni.h. They denote compiler-dependent specifiers for exported functions that come from a dynamically loaded library.)

Now, you simply copy the function prototype from the header file into the source file and give the implementation code for the function, as shown in Listing 12-3.

Listing 12-3. **HelloNative.c**

Code View:

```
/*
   @version 1.10 1997-07-01
   @author Cay Horstmann
*/

#include "HelloNative.h"
#include <stdio.h>

JNIEXPORT void JNICALL Java_HelloNative_greeting(JNIEnv* env, jclass cl)
{
   printf("Hello Native World!\n");
}
```


In this simple function, ignore the `env` and `cl` arguments. You'll see their use later.

C++ Note

You can use C++ to implement native methods. However, you must then declare the functions that implement the native methods as `extern "C"`. (This stops the C++ compiler from "mangling" the method name.) For example,

```
extern "C"
JNIEXPORT void JNICALL Java_HelloNative_greeting(JNIEnv* env, jclass cl)
{
   cout << "Hello, Native World!" << endl;
}
```

You compile the native C code into a dynamically loaded library. The details depend on your compiler.

For example, with the Gnu C compiler on Linux, use these commands:

Code View:

```
gcc -fPIC -I jdk/include -I jdk/include/linux -shared -o libHelloNative.so HelloNative.c
```

With the Sun compiler under the Solaris Operating System, the command is

Code View:

```
cc -G -I jdk/include -I jdk/include/solaris -o libHelloNative.so HelloNative.c
```

With the Microsoft compiler under Windows, the command is

```
cl -I jdk\include -I jdk\include\win32 -LD HelloNative.c -FeHelloNative.dll
```

Here, *jdk* is the directory that contains the JDK.

Tip

If you use the Microsoft compiler from a command shell, first run the batch file `vcvars32.bat` or `vsvars32.bat`. That batch file sets up the path and the environment variables needed by the compiler. You can find it in the directory `c:\Program Files\Microsoft Visual Studio .NET 2003\Common7\tools`, `c:\Program Files\Microsoft Visual Studio 8\VC,` or a similar monstrosity.

You can also use the freely available Cygwin programming environment, available from http://www.cygwin.com. It contains the Gnu C compiler and libraries for UNIX-style programming on Windows. With Cygwin, use the command

```
gcc -mno-cygwin -D __int64="long long" -I jdk/include/ -I jdk/include/win32
   -shared -Wl,--add-stdcall-alias -o HelloNative.dll HelloNative.c
```

Type the entire command on a single line.

Note

The Windows version of the header file `jni_md.h` contains the type declaration

```
typedef __int64 jlong;
```

which is specific to the Microsoft compiler. If you use the Gnu compiler, you might want to edit that file, for example,

```
#ifdef __GNUC__
   typedef long long jlong;
#else
   typedef __int64 jlong;
#endif
```

Alternatively, compile with `-D __int64="long long"`, as shown in the sample compiler invocation.

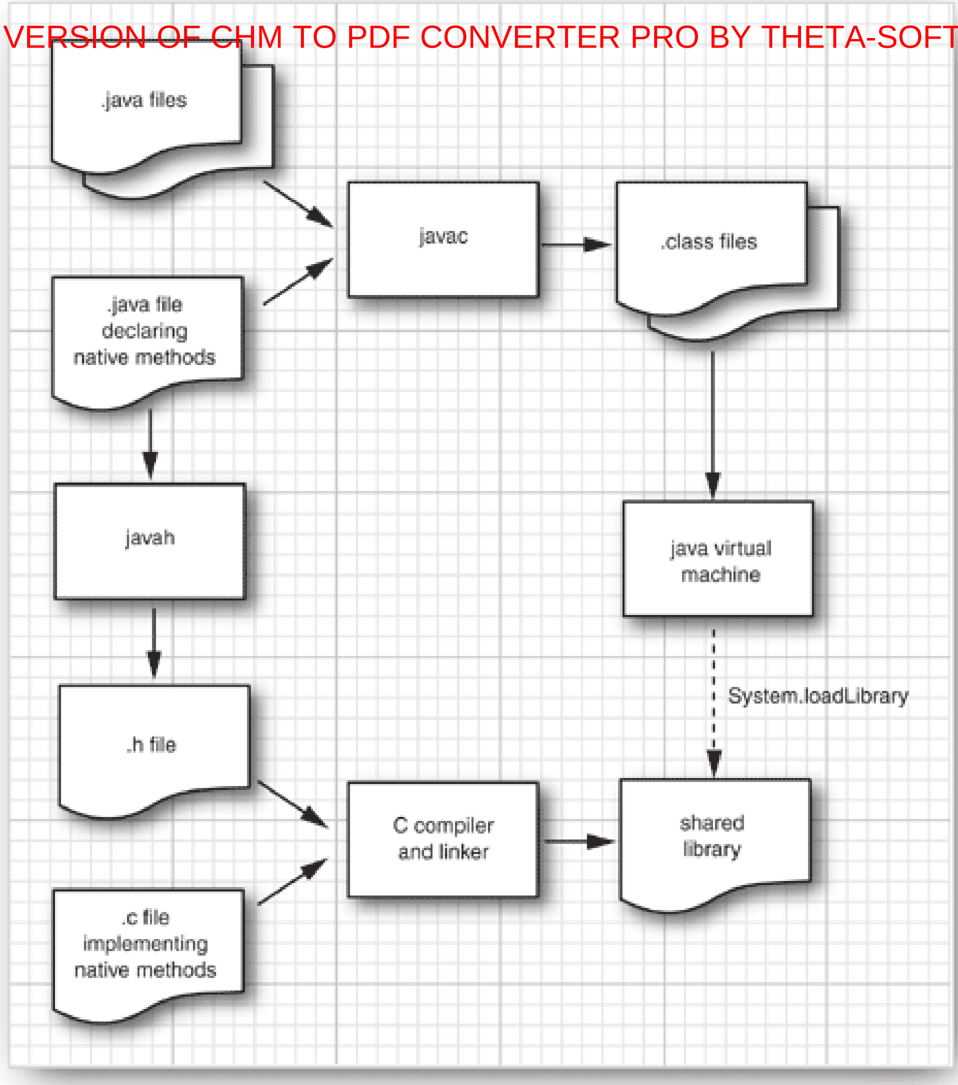
Finally, add a call to the System.loadLibrary method in your program. To ensure that the virtual machine will load the library before the first use of the class, use a static initialization block, as in Listing 12-4.

Figure 12-1 gives a summary of the native code processing.

Figure 12-1. Processing native code

[View full size image]

UNREGISTERED VERSION OF CHM TO PDF CONVERTER PRO BY THETA-SOFTWARE



Listing 12-4. **HelloNativeTest.java**

```
/**
 * @version 1.11 2007-10-26
 * @author Cay Horstmann
 */
class HelloNativeTest
{
   public static void main(String[] args)
   {
      HelloNative.greeting();
   }

   static
   {
      System.loadLibrary("HelloNative");
   }
}
```

If you compile and run this program, the message "Hello, Native World!" is displayed in a terminal window.

Note

If you run Linux, you must add the current directory to the library path. Either set the `LD_LIBRARY_PATH` environment variable,

```
export LD_LIBRARY_PATH=.:$LD_LIBRARY_PATH
```

or set the `java.library.path` system property:

```
java -Djava.library.path=. HelloNativeTest
```

Of course, this is not particularly impressive by itself. However, if you keep in mind that this message is generated by the C `printf` command and not by any Java programming language code, you will see that we have taken the first steps toward bridging the gap between the two languages!

In summary, you follow these steps to link a native method to a Java program:

1. Declare a native method in a Java class.
2. Run `javah` to get a header file with a C declaration for the method.
3. Implement the native method in C.
4. Place the code in a shared library.
5. Load that library in your Java program.

`java.lang.System` 1.0

- `void loadLibrary(String libname)`

  loads the library with the given name. The library is located in the library search path. The exact method for locating the library is operating system dependent.

Note

Some shared libraries for native code must run initialization code. You can place any initialization code into a `JNI_OnLoad` method. Similarly, when the virtual machine (VM) shuts down, it will call the `JNI_OnUnload` method if you provide it. The prototypes are

```
jint JNI_OnLoad(JavaVM* vm, void* reserved);
void JNI_OnUnload(JavaVM* vm, void* reserved);
```

The `JNI_OnLoad` method needs to return the minimum version of the VM that it requires, such as `JNI_VERSION_1_2`.

Numeric Parameters and Return Values

When passing numbers between C and Java, you should understand which types correspond to each other. For example, although C does have data types called `int` and `long`, their implementation is platform dependent. On some platforms, `ints` are 16-bit quantities, and on others they are 32-bit quantities. In the Java platform, of course, an `int` is *always* a 32-bit integer. For that reason, JNI defines types `jint`, `jlong`, and so on.

Table 12-1 shows the correspondence between Java types and C types.

Table 12-1. Java Types and C Types

Java Programming Language	C Programming Language	Bytes
`boolean`	`jboolean`	1
`byte`	`jbyte`	1
`char`	`jchar`	2
`short`	`jshort`	2
`int`	`jint`	4
`long`	`jlong`	8
`float`	`jfloat`	4
`double`	`jdouble`	8

In the header file `jni.h`, these types are declared with `typedef` statements as the equivalent types on the target platform. That header file also defines the constants `JNI_FALSE = 0` and `JNI_TRUE = 1`.

Using **`printf`** for Formatting Numbers

Until Java SE 5.0, Java had no direct analog to the C `printf` function. In the following examples, we will suppose you are stuck with an ancient JDK release and decide to implement the same functionality by calling the C `printf` function in a native method.

Listing 12-5 shows a class called `Printf1` that uses a native method to print a floating-point number with a given field width and precision.

Listing 12-5. **`Printf1.java`**

```
/**
 * @version 1.10 1997-07-01
 * @author Cay Horstmann
 */
class Printf1
{
   public static native int print(int width, int precision, double x);

   static
   {
      System.loadLibrary("Printf1");
   }
}
```


Notice that when the method is implemented in C, all `int` and `double` parameters are changed to `jint` and `jdouble`, as shown in Listing 12-6.

Listing 12-6. **Printf1.c**

```
/**
   @version 1.10 1997-07-01
   @author Cay Horstmann
*/

#include "Printf1.h"
#include <stdio.h>

JNIEXPORT jint JNICALL Java_Printf1_print(JNIEnv* env, jclass cl,
   jint width, jint precision, jdouble x)
{
   char fmt[30];
   jint ret;
   sprintf(fmt, "%%%d.%df", width, precision);
   ret = printf(fmt, x);
   fflush(stdout);
   return ret;
}
```

The function simply assembles a format string `"%w.pf"` in the variable `fmt`, then calls `printf`. It then returns the number of characters printed.

Listing 12-7 shows the test program that demonstrates the `Printf1` class.

Listing 12-7. **Printf1Test.java**

```java
/**
 * @version 1.10 1997-07-01
 * @author Cay Horstmann
 */
class Printf1Test
{
   public static void main(String[] args)
   {
      int count = Printf1.print(8, 4, 3.14);
      count += Printf1.print(8, 4, count);
      System.out.println();
      for (int i = 0; i < count; i++)
         System.out.print("-");
      System.out.println();
   }
}
```

String Parameters

Next, we want to consider how to transfer strings to and from native methods. As you know, strings in the Java programming language are sequences of UTF-16 code points whereas C strings are null-terminated sequences of bytes, so strings are quite different in the two languages. JNI has two sets of functions for manipulating strings, one that converts Java strings to "modified UTF-8" byte sequences and one that converts them to arrays of UTF-16 values, that is, to `jchar` arrays. (The UTF-8, "modified UTF-8", and UTF-16 formats were discussed in Volume I, Chapter 12. Recall that the "modified UTF-8" encoding leaves ASCII characters unchanged, but all other Unicode characters are encoded as multibyte sequences.)

Note

The standard UTF-8 encoding and the "modified UTF-8" encoding differ only for "supplementary" characters with code higher than 0xFFFF. In the standard UTF-8 encoding, these characters are encoded as a 4-byte sequence. However, in the "modified" encoding, the character is first encoded as a pair of "surrogates" in the UTF-16 encoding, and then each surrogate is encoded with UTF-8, yielding a total of 6 bytes. This is clumsy, but it is a historical accident—the JVM specification was written when Unicode was still limited to 16 bits.

If your C code already uses Unicode, you'll want to use the second set of conversion functions. On the other hand, if all your strings are restricted to ASCII characters, you can use the "modified UTF-8" conversion functions.

A native method with a `String` parameter actually receives a value of an opaque type called `jstring`. A native method with a return value of type `String` must return a value of type `jstring`. JNI functions read and construct these `jstring` objects. For example, the `NewStringUTF` function makes a new `jstring` object out of a `char` array that contains ASCII characters or, more generally, "modified UTF-8"-encoded byte sequences.

JNI functions have a somewhat odd calling convention. Here is a call to the `NewStringUTF` function.

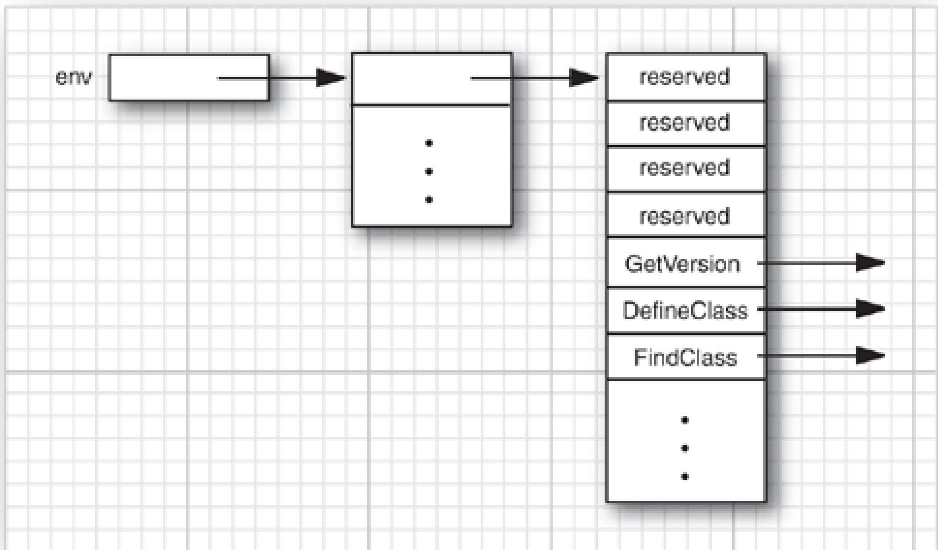
Code View:
```
JNIEXPORT jstring JNICALL Java_HelloNative_getGreeting(JNIEnv* env, jclass cl)
{
   jstring jstr;
   char greeting[] = "Hello, Native World\n";
   jstr = (*env)->NewStringUTF(env, greeting);
   return jstr;
}
```

Note

Unless explicitly mentioned otherwise, all code in this chapter is C code.

All calls to JNI functions use the `env` pointer that is the first argument of every native method. The `env` pointer is a pointer to a table of function pointers (see Figure 12-2). Therefore, you must prefix every JNI call with `(*env)->` to actually dereference the function pointer. Furthermore, `env` is the first parameter of every JNI function.

Figure 12-2. The **env** pointer



C++ Note

It is simpler to access JNI functions in C++. The C++ version of the `JNIEnv` class has inline member functions that take care of the function pointer lookup for you. For example, you can call the `NewStringUTF` function as

```
jstr = env->NewStringUTF(greeting);
```

Note that you omit the `JNIEnv` pointer from the parameter list of the call.

The `NewStringUTF` function lets you construct a new `jstring`. To read the contents of an existing `jstring` object, use the GetStringUTFChars function. This function returns a const jbyte* pointer to the "modified UTF-8" characters that describe the character string. Note that a specific virtual machine is free to choose this character encoding for its internal string representation, so you might get a character pointer into the actual Java string. Because Java strings are meant to be immutable, it is *very* important that you treat the const seriously and do not try to write into this character array. On the other hand, if the virtual machine uses UTF-16 or UTF-32 characters for its internal string representation, then this function call allocates a new memory block that will be filled with the "modified UTF-8" equivalents.

The virtual machine must know when you are finished using the string so that it can garbage-collect it. (The garbage collector runs in a separate thread, and it can interrupt the execution of native methods.) For that reason, you must call the ReleaseStringUTFChars function.

Alternatively, you can supply your own buffer to hold the string characters by calling the GetStringRegion or GetStringUTFRegion methods.

Finally, the GetStringUTFLength function returns the number of characters needed for the "modified UTF-8" encoding of the string.

Note

You can find the JNI API at http://java.sun.com/javase/6/docs/technotes/guides/jni/index.html.

Accessing Java Strings from C Code

- `jstring NewStringUTF(JNIEnv* env, const char bytes[])`

  returns a new Java string object from a zero byte-terminated "modified UTF-8" byte sequence, or NULL if the string cannot be constructed.

- `jsize GetStringUTFLength(JNIEnv* env, jstring string)`

  returns the number of bytes required for the "modified UTF-8" encoding (not counting a zero byte terminator).

- `const jbyte* GetStringUTFChars(JNIEnv* env, jstring string, jboolean* isCopy)`

  returns a pointer to the "modified UTF-8" encoding of a string, or NULL if the character array cannot be constructed. The pointer is valid until ReleaseStringUTFChars is called. isCopy points to a jboolean that is filled with JNI_TRUE if a copy is made; with JNI_FALSE otherwise.

- `void ReleaseStringUTFChars(JNIEnv* env, jstring string, const` `jbyte bytes[])`

informs the virtual machine that the native code no longer needs access to the Java string through `bytes` (a pointer returned by `GetStringUTFChars`).

- `void GetStringRegion(JNIEnv *env, jstring string, jsize start, jsize length, jchar *buffer)`

  copies a sequence of UTF-16 double-bytes from a string to a user-supplied buffer of size at least 2 x `length`.

- `void GetStringUTFRegion(JNIEnv *env, jstring string, jsize start, jsize length, jbyte *buffer)`

  copies a sequence of "modified UTF-8" bytes from a string to a user-supplied buffer. The buffer must be long enough to hold the bytes. In the worst case, 3 x `length` bytes are copied.

- `jstring NewString(JNIEnv* env, const jchar chars[], jsize length)`

  returns a new Java string object from a Unicode string, or `NULL` if the string cannot be constructed.

Parameters:	`env`	The JNI interface pointer
	`chars`	The null-terminated UTF16 string
	`length`	The number of characters in the string

- `jsize GetStringLength(JNIEnv* env, jstring string)`

  returns the number of characters in the string.

- `const jchar* GetStringChars(JNIEnv* env, jstring string, jboolean* isCopy)`

  returns a pointer to the Unicode encoding of a string, or `NULL` if the character array cannot be constructed. The pointer is valid until `ReleaseStringChars` is called. `isCopy` is either `NULL` or points to a `jboolean` that is filled with `JNI_TRUE` if a copy is made; with `JNI_FALSE` otherwise.

- `void ReleaseStringChars(JNIEnv* env, jstring string, const jchar chars[])`

  informs the virtual machine that the native code no longer needs access to the Java string through `chars` (a pointer returned by `GetStringChars`).

Let us put these functions to work and write a class that calls the C function `sprintf`. We would like to call the function as shown in Listing 12-8.

Listing 12-8. **Printf2Test.java**


```
/**
 * @version 1.10 1997-07-01
 * @author Cay Horstmann
 */
class Printf2Test
{
   public static void main(String[] args)
   {
      double price = 44.95;
      double tax = 7.75;
      double amountDue = price * (1 + tax / 100);

      String s = Printf2.sprint("Amount due = %8.2f", amountDue);
      System.out.println(s);
   }
}
```


Listing 12-9 shows the class with the native `sprint` method.

Listing 12-9. **Printf2.java**

```
/**
 * @version 1.10 1997-07-01
 * @author Cay Horstmann
 */
class Printf2
{
   public static native String sprint(String format, double x);

   static
   {
      System.loadLibrary("Printf2");
   }
}
```

Therefore, the C function that formats a floating-point number has the prototype

Code View:

```
JNIEXPORT jstring JNICALL Java_Printf2_sprint(JNIEnv* env, jclass cl, jstring format, jdouble x)
```

Listing 12-10 shows the code for the C implementation. Note the calls to `GetStringUTFChars` to read the format argument, `NewStringUTF` to generate the return value, and `ReleaseStringUTFChars` to inform the virtual machine that access to the string is no longer required.

Listing 12-10. **Printf2.c**

Code View:

```
/**
   @version 1.10 1997-07-01
   @author Cay Horstmann
*/

#include "Printf2.h"
#include <string.h>
#include <stdlib.h>
#include <float.h>

/**
   @param format a string containing a printf format specifier
   (such as "%8.2f"). Substrings "%%" are skipped.
   @return a pointer to the format specifier (skipping the '%')
   or NULL if there wasn't a unique format specifier
*/
char* find_format(const char format[])
{
   char* p;
   char* q;

      p = strchr(p + 2, '%');
   if (p == NULL) return NULL;
   /* now check that % is unique */
   p = strchr(format, '%');
   while (p != NULL && *(p + 1) == '%') /* skip %% */
   p++;
   q = strchr(p, '%');
   while (q != NULL && *(q + 1) == '%') /* skip %% */
      q = strchr(q + 2, '%');
   if (q != NULL) return NULL; /* % not unique */
   q = p + strspn(p, " -0+#"); /* skip past flags */
   q += strspn(q, "0123456789"); /* skip past field width */
   if (*q == '.') { q++; q += strspn(q, "0123456789"); }
      /* skip past precision */
   if (strchr("eEfFgG", *q) == NULL) return NULL;
      /* not a floating-point format */
   return p;
}

JNIEXPORT jstring JNICALL Java_Printf2_sprint(JNIEnv* env, jclass cl,
   jstring format, jdouble x)
{
   const char* cformat;
   char* fmt;
   jstring ret;

   cformat = (*env)->GetStringUTFChars(env, format, NULL);
   fmt = find_format(cformat);
   if (fmt == NULL)
      ret = format;
   else
   {
      char* cret;
      int width = atoi(fmt);
      if (width == 0) width = DBL_DIG + 10;
      cret = (char*) malloc(strlen(cformat) + width);
      sprintf(cret, cformat, x);
```

```
        ret = (*env)->NewStringUTF(env, cret);
        free(cret);
    }
    (*env)->ReleaseStringUTFChars(env, format, cformat);
    return ret;
}
```


In this function, we chose to keep the error handling simple. If the format code to print a floating-point number is not of the form `%w.pc`, where `c` is one of the characters `e`, `E`, `f`, `g`, or `G`, then we simply do not format the number. We show you later how to make a native method throw an exception.

Accessing Fields

All the native methods that you saw so far were static methods with number and string parameters. We next consider native methods that operate on objects. As an exercise, we implement a method of the `Employee` class that was introduced in Volume I, Chapter 4, using a native method. Again, this is not something you would norma want to do, but it does illustrate how to access fields from a native method when you need to do so.

Accessing Instance Fields

To see how to access instance fields from a native method, we will reimplement the `raiseSalary` method. Here is the code in Java:

```
public void raiseSalary(double byPercent)
{
   salary *= 1 + byPercent / 100;
}
```

Let us rewrite this as a native method. Unlike the previous examples of native methods, this is not a static methoc Running `javah` gives the following prototype:

Code View:

```
JNIEXPORT void JNICALL Java_Employee_raiseSalary(JNIEnv *, jobject, jdouble);
```

Note the second argument. It is no longer of type `jclass` but of type `jobject` . In fact, it is the equivalent of the `this` reference. Static methods obtain a reference to the class, whereas nonstatic methods obtain a reference to tl implicit `this` argument object.

Now we access the `salary` field of the implicit argument. In the "raw" Java-to-C binding of Java 1.0, this was easy—a programmer could directly access object data fields. However, direct access requires all virtual machines t expose their internal data layout. For that reason, the JNI requires programmers to get and set the values of data fields by calling special JNI functions.

In our case, we need to use the `GetDoubleField` and `SetDoubleField` functions because the type of `salary` is a `double` . There are other functions—`GetIntField/SetIntField` , `GetObjectField/SetObjectField` , and so on—for other field types. The general syntax is:

```
x = (*env)->GetXxxField(env, this_obj, fieldID);
(*env)->SetXxxField(env, this_obj, fieldID, x);
```

Here, `class` is a value that represents a Java object of type `Class` , `fieldID` is a value of a special type, `jfieldID` that identifies a field in a structure, and *Xxx* represents a Java data type (`Object` , `Boolean` , `Byte` , and so on). There are two ways to obtain the `class` object. The `GetObjectClass` function returns the class of any object. For example:

```
jclass class_Employee = (*env)->GetObjectClass(env, this_obj);
```

The `FindClass` function lets you specify the class name as a string (curiously, with `/` instead of periods as package name separators).

```
jclass class_String = (*env)->FindClass(env, "java/lang/String");
```

Use the `GetFieldID` function to obtain the `fieldID` . You must supply the name of the field and its *signature* , an encoding of its type. For example, here is the code to obtain the field ID of the `salary` field.

Code View:


```
jfieldID id_salary = (*env)->GetFieldID(env, class_Employee, "salary", "D");
```

The string `"D"` denotes the type `double` . You learn the complete rules for encoding signatures in the next section.

You might be thinking that accessing a data field seems quite convoluted. The designers of the JNI did not want to expose the data fields directly, so they had to supply functions for getting and setting field values. To minimize the cost of these functions, computing the field ID from the field name—which is the most expensive step—is factored out into a separate step. That is, if you repeatedly get and set the value of a particular field, you incur the cost of computing the field identifier only once.

Let us put all the pieces together. The following code reimplements the `raiseSalary` method as a native method.

Code View:

```
JNIEXPORT void JNICALL Java_Employee_raiseSalary(JNIEnv* env, jobject this_obj, jdouble byPercen
{
   /* get the class */
   jclass class_Employee = (*env)->GetObjectClass(env, this_obj);

   /* get the field ID */
   jfieldID id_salary = (*env)->GetFieldID(env, class_Employee, "salary", "D");

   /* get the field value */
   jdouble salary = (*env)->GetDoubleField(env, this_obj, id_salary);

   salary *= 1 + byPercent / 100;

   /* set the field value */
   (*env)->SetDoubleField(env, this_obj, id_salary, salary);
}
```

Caution

Class references are only valid until the native method returns. Thus, you cannot cache the return values of GetObjectClass in your code. Do *not* store away a class reference for reuse in a later method call. You must call GetObjectClass every time the native method executes. If this is intolerable, you can lock the reference with a call to NewGlobalRef :

```
static jclass class_X = 0;
static jfieldID id_a;
. . .
if (class_X == 0)
{
   jclass cx = (*env)->GetObjectClass(env, obj);
   class_X = (*env)->NewGlobalRef(env, cx);
   id_a = (*env)->GetFieldID(env, cls, "a", ". . .");
}
```

Now you can use the class reference and field IDs in subsequent calls. When you are done using the class, make sure to call

```
(*env)->DeleteGlobalRef(env, class_X);
```

Listings 12-11 and 12-12 show the Java code for a test program and the Employee class. Listing 12-13 contains th C code for the native raiseSalary method.

Listing 12-11. **EmployeeTest.java**

```
/**
 * @version 1.10 1999-11-13
 * @author Cay Horstmann
 */

public class EmployeeTest
{
   public static void main(String[] args)
   {
      Employee[] staff = new Employee[3];

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

      for (Employee e : staff)
         e.raiseSalary(5);
      for (Employee e : staff)
         e.print();
   }
}
```

Listing 12-12. **Employee.java**

Code View:
```
/**
 * @version 1.10 1999-11-13
 * @author Cay Horstmann
 */

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

   public native void raiseSalary(double byPercent);

   public void print()
   {
      System.out.println(name + " " + salary);
   }

   private String name;
   private double salary;

   static
   {
      System.loadLibrary("Employee");
   }
}
```


Listing 12-13. **Employee.c**

Code View:
```
/**
   @version 1.10 1999-11-13
   @author Cay Horstmann
*/

#include "Employee.h"

#include <stdio.h>

JNIEXPORT void JNICALL Java_Employee_raiseSalary(JNIEnv* env, jobject this_obj,
   jdouble byPercent)
{
   /* get the class */
   jclass class_Employee = (*env)->GetObjectClass(env, this_obj);

   /* get the field ID */
   jfieldID id_salary = (*env)->GetFieldID(env, class_Employee, "salary", "D");

   /* get the field value */
   jdouble salary = (*env)->GetDoubleField(env, this_obj, id_salary);

```

```
    salary *= 1 + byPercent / 100;

    /* set the field value */
    (*env)->SetDoubleField(env, this_obj, id_salary, salary);
}
```

Accessing Static Fields

Accessing static fields is similar to accessing nonstatic fields. You use the `GetStaticFieldID` and `GetStatic` *Xxx* `Field/SetStatic` *Xxx* `Field` functions. They work almost identically to their nonstatic counterpart, with two differences:

- Because you have no object, you must use `FindClass` instead of `GetObjectClass` to obtain the class reference.

- You supply the class, not the instance object, when accessing the field.

For example, here is how you can get a reference to `System.out` .

Code View:

```
/* get the class */
jclass class_System = (*env)->FindClass(env, "java/lang/System");

/* get the field ID */
jfieldID id_out = (*env)->GetStaticFieldID(env, class_System, "out",
   "Ljava/io/PrintStream;");

/* get the field value */
jobject obj_out = (*env)->GetStaticObjectField(env, class_System, id_out);
```

Accessing Fields

- `jfieldID GetFieldID(JNIEnv *env, jclass cl, const char name[], const char fieldSignature[])`

  returns the identifier of a field in a class.

- *Xxx* `Get` *Xxx* `Field(JNIEnv *env, jobject obj, jfieldID id)`

  returns the value of a field. The field type *Xxx* is one of `Object` , `Boolean` , `Byte` , `Char` , `Short` , `Int` , `Long` , `Float` , or `Double` .

- `void Set` *Xxx* `Field(JNIEnv *env, jobject obj, jfieldID id,` *Xxx* `value)`

  sets a field to a new value. The field type *Xxx* is one of `Object` , `Boolean` , `Byte` , `Char` , `Short` , `Int` , `Long` , `Float` , or `Double` .

- `jfieldID GetStaticFieldID(JNIEnv *env, jclass cl, const char name[], const char fieldSignature[])`

  returns the identifier of a static field in a class.

- *Xxx* `GetStatic` *Xxx* `Field(JNIEnv *env, jclass cl, jfieldID id)`

  returns the value of a static field. The field type *Xxx* is one of `Object` , `Boolean` , `Byte` , `Char` , `Short` , `Int` , `Long` , `Float` , or `Double` .

- `void SetStatic` *Xxx* `Field(JNIEnv *env, jclass cl, jfieldID id,` *Xxx* `value)`

  sets a static field to a new value. The field type *Xxx* is one of `Object` , `Boolean` , `Byte` , `Char` , `Short` , `Int` , `Long` , `Float` , or `Double` .

Encoding Signatures

To access instance fields and call methods that are defined in the Java programming language, you need to learn the rules for "mangling" the names of data types and method signatures. (A method signature describes the parameters and return type of the method.) Here is the encoding scheme:

`B`	`byte`
`C`	`char`
`D`	`double`
`F`	`float`
`I`	`int`
`J`	`long`
`L`*classname*`;`	a class type
`S`	`short`
`V`	`void`
`Z`	`boolean`

To describe an array type, use a `[`. For example, an array of strings is

```
[Ljava/lang/String;
```

A `float[][]` is mangled into

```
[[F
```

For the complete signature of a method, you list the parameter types inside a pair of parentheses and then list the return type. For example, a method receiving two integers and returning an integer is encoded as

```
(II)I
```

The print method that we used in the preceding example has a mangled signature of

```
(Ljava/lang/String;)V
```

That is, the method receives a string and returns `void`.

Note that the semicolon at the end of the `L` expression is the terminator of the type expression, not a separator between parameters. For example, the constructor

```
Employee(java.lang.String, double, java.util.Date)
```

has a signature

```
"(Ljava/lang/String;DLjava/util/Date;)V"
```

Note that there is no separator between the `D` and `Ljava/util/Date;`. Also note that in this encoding scheme, you must use `/` instead of `.` to separate the package and class names. The `V` at the end denotes a return type of `void`. Even though you don't specify a return type for constructors in Java, you need to add a `V` to the virtual machine signature.

Tip

You can use the `javap` command with option `-s` to generate the method signatures from class files. For example, run

```
javap -s -private Employee
```

You get the following output, displaying the signatures of all fields and methods.

```
Compiled from "Employee.java"
public class Employee extends java.lang.Object{
private java.lang.String name;
  Signature: Ljava/lang/String;
private double salary;
  Signature: D
public Employee(java.lang.String, double);
  Signature: (Ljava/lang/String;D)V
public native void raiseSalary(double);
  Signature: (D)V
public void print();
  Signature: ()V
static {};
  Signature: ()V
}
```

Note

There is no rationale whatsoever for forcing programmers to use this mangling scheme for describing signatures. The designers of the native calling mechanism could have just as easily written a function that reads signatures in the Java programming language style, such as `void(int,java.lang.String)`, and encodes them into whatever internal representation they prefer. Then again, using the mangled signatures lets you partake in the mystique of programming close to the virtual machine.

Calling Java Methods

Of course, Java programming language functions can call C functions—that is what native methods are for. Can we go the other way? Why would we want to do this anyway? The answer is that it often happens that a native method needs to request a service from an object that was passed to it. We first show you how to do it for instance methods, and then we show you how to do it for static methods.

Instance Methods

As an example of calling a Java method from native code, let's enhance the `Printf` class and add a method that works similarly to the C function `fprintf`. That is, it should be able to print a string on an arbitrary `PrintWriter` object. Here is the definition of the method in Java:


```
class Printf3
{
   public native static void fprint(PrintWriter out, String s, double x);
      . . .
}
```

We first assemble the string to be printed into a `String` object `str`, as in the `sprint` method that we already implemented. Then, we call the `print` method of the `PrintWriter` class from the C function that implements the native method.

You can call any Java method from C by using the function call

Code View:
```
(*env)->CallXxxMethod(env, implicit parameter, methodID, explicit parameters)
```

Replace *Xxx* with `Void`, `Int`, `Object`, and so on, depending on the return type of the method. Just as you need a `fieldID` to access a field of an object, you need a method ID to call a method. You obtain a method ID by calling the JNI function `GetMethodID` and supplying the class, the name of the method, and the method signature.

In our example, we want to obtain the ID of the `print` method of the `PrintWriter` class. As you saw in Volume I, Chapter 12, the `PrintWriter` class has several overloaded methods called `print`. For that reason, you must also supply a string describing the parameters and return the value of the specific function that you want to use. For example, we want to use `void print(java.lang.String)`. As described in the preceding section, we must now "mangle" the signature into the string `"(Ljava/lang/String;)V"`.

Here is the complete code to make the method call, by

1. Obtaining the class of the implicit parameter.

2. Obtaining the method ID.

3. Making the call.

   Code View:
```
/* get the class */
class_PrintWriter = (*env)->GetObjectClass(env, out);

/* get the method ID */
id_print = (*env)->GetMethodID(env, class_PrintWriter, "print", "(Ljava/lang/String;)V");

/* call the method */
(*env)->CallVoidMethod(env, out, id_print, str);
```

Listings 12-14 and 12-15 show the Java code for a test program and the `Printf3` class. Listing 12-16 contains the C code for the native `fprint` method.

Note

The numerical method IDs and field IDs are conceptually similar to `Method` and `Field` objects in the reflection API. You can convert between them with the following functions:

```
jobject ToReflectedMethod(JNIEnv* env, jclass class, jmethodID methodID);
   // returns Method object
methodID FromReflectedMethod(JNIEnv* env, jobject method);
jobject ToReflectedField(JNIEnv* env, jclass class, jfieldID fieldID);
   // returns Field object
fieldID FromReflectedField(JNIEnv* env, jobject field);
```

Static Methods

Calling static methods from native methods is similar to calling instance methods. There are two differences.

- You use the `GetStaticMethodID` and `CallStatic`*`Xxx`*`Method` functions.
- You supply a class object, not an implicit parameter object, when invoking the method.

As an example of this, let's make the call to the static method

```
System.getProperty("java.class.path")
```

from a native method. The return value of this call is a string that gives the current class path.

First, we have to find the class to use. Because we have no object of the class `System` readily available, we use `FindClass` rather than `GetObjectClass`.

```
jclass class_System = (*env)->FindClass(env, "java/lang/System");
```

Next, we need the ID of the static `getProperty` method. The encoded signature of that method is

```
"(Ljava/lang/String;)Ljava/lang/String;"
```


because both the parameter and the return value are a string. Hence, we obtain the method ID as follows:

Code View:
```
jmethodID id_getProperty = (*env)->GetStaticMethodID(env, class_System, "getProperty",
   "(Ljava/lang/String;)Ljava/lang/String;");
```


Finally, we can make the call. Note that the class object is passed to the `CallStaticObjectMethod` function.

Code View:
```
jobject obj_ret = (*env)->CallStaticObjectMethod(env, class_System, id_getProperty,
   (*env)->NewStringUTF(env, "java.class.path"));
```

The return value of this method is of type `jobject`. If we want to manipulate it as a string, we must cast it to `jstring`:

```
jstring str_ret = (jstring) obj_ret;
```

C++ Note

In C, the types `jstring` and `jclass`, as well as the array types that are introduced later, are all type equivalent to `jobject`. The cast of the preceding example is therefore not strictly necessary in C. But in C++, these types are defined as pointers to "dummy classes" that have the correct inheritance hierarchy. For example, the assignment of a `jstring` to a `jobject` is legal without a cast in C++, but the assignment from a `jobject` to a `jstring` requires a cast.

Constructors

A native method can create a new Java object by invoking its constructor. You invoke the constructor by calling the `NewObject` function.

Code View:
```
jobject obj_new = (*env)->NewObject(env, class, methodID, construction parameters);
```

You obtain the method ID needed for this call from the GetMethodID function by specifying the method name as "<init>" and the encoded signature of the constructor (with return type void). For example, here is how a native method can create a FileOutputStream object.

Code View:

```
const char[] fileName = ". . .";
jstring str_fileName = (*env)->NewStringUTF(env, fileName);
jclass class_FileOutputStream = (*env)->FindClass(env, "java/io/FileOutputStream");
jmethodID id_FileOutputStream
   = (*env)->GetMethodID(env, class_FileOutputStream, "<init>", "(Ljava/lang/String;)V");
jobject obj_stream
   = (*env)->NewObject(env, class_FileOutputStream, id_FileOutputStream, str_fileName);
```

Note that the signature of the constructor takes a parameter of type java.lang.String and has a return type of void.

Alternative Method Invocations

Several variants of the JNI functions call a Java method from native code. These are not as important as the functions that we already discussed, but they are occasionally useful.

The CallNonvirtual*Xxx*Method functions receive an implicit argument, a method ID, a class object (which must correspond to a superclass of the implicit argument), and explicit arguments. The function calls the version of the method in the specified class, bypassing the normal dynamic dispatch mechanism.

All call functions have versions with suffixes "A" and "V" that receive the explicit parameters in an array or a va_list (as defined in the C header stdarg.h).

Listing 12-14. **Printf3Test.java**

```
import java.io.*;

/**
 * @version 1.10 1997-07-01
 * @author Cay Horstmann
 */
class Printf3Test
{
   public static void main(String[] args)
   {
      double price = 44.95;
      double tax = 7.75;
      double amountDue = price * (1 + tax / 100);
      PrintWriter out = new PrintWriter(System.out);
      Printf3.fprint(out, "Amount due = %8.2f\n", amountDue);
      out.flush();
   }
}
```

Listing 12-15. **Printf3.java**

Code View:

```
import java.io.*;

/**
 * @version 1.10 1997-07-01
 * @author Cay Horstmann
 */
```



```
class Printf3
{
   public static native void fprint(PrintWriter out, String format, double x);

   static
   {
      System.loadLibrary("Printf3");
```



```
   }
}
```

Listing 12-16. **Printf3.c**

Code View:

```
/**
   @version 1.10 1997-07-01
   @author Cay Horstmann
*/

#include "Printf3.h"
#include <string.h>
#include <stdlib.h>
#include <float.h>

/**
   @param format a string containing a printf format specifier
   (such as "%8.2f"). Substrings "%%" are skipped.
   @return a pointer to the format specifier (skipping the '%')
   or NULL if there wasn't a unique format specifier
*/
char* find_format(const char format[])
{
   char* p;
   char* q;

   p = strchr(format, '%');
   while (p != NULL && *(p + 1) == '%') /* skip %% */
      p = strchr(p + 2, '%');
   if (p == NULL) return NULL;
   /* now check that % is unique */
   p++;
   q = strchr(p, '%');
   while (q != NULL && *(q + 1) == '%') /* skip %% */
      q = strchr(q + 2, '%');
   if (q != NULL) return NULL; /* % not unique */
   q = p + strspn(p, " -0+#"); /* skip past flags */
   q += strspn(q, "0123456789"); /* skip past field width */
   if (*q == '.') { q++; q += strspn(q, "0123456789"); }
      /* skip past precision */
```

```
   if (strchr("eEfFgG", *q) == NULL) return NULL;
      /* not a floating-point format */
   return p;
}

JNIEXPORT void JNICALL Java_Printf3_fprint(JNIEnv* env, jclass cl,
   jobject out, jstring format, jdouble x)
{
   const char* cformat;
   char* fmt;
   jstring str;
   jclass class_PrintWriter;
   jmethodID id_print;

   cformat = (*env)->GetStringUTFChars(env, format, NULL);
   fmt = find_format(cformat);
   if (fmt == NULL)
      str = format;
   else
   {
      char* cstr;
      int width = atoi(fmt);
      if (width == 0) width = DBL_DIG + 10;
      cstr = (char*) malloc(strlen(cformat) + width);
      sprintf(cstr, cformat, x);
      str = (*env)->NewStringUTF(env, cstr);
      free(cstr);
   }
   (*env)->ReleaseStringUTFChars(env, format, cformat);

   /* now call ps.print(str) */

   /* get the class */
   class_PrintWriter = (*env)->GetObjectClass(env, out);

   /* get the method ID */
   id_print = (*env)->GetMethodID(env, class_PrintWriter, "print", "(Ljava/lang/String;)V");

   /* call the method */
   (*env)->CallVoidMethod(env, out, id_print, str);
}
```

Executing Java Methods

- `jmethodID GetMethodID(JNIEnv *env, jclass cl, const char name[], const char methodSignature[])`

  returns the identifier of a method in a class.

- *Xxx* Call*Xxx*Method(JNIEnv *env, jobject obj, jmethodID id, args)

- *Xxx* Call*Xxx*MethodA(JNIEnv *env, jobject obj, jmethodID id, jvalue args[])

- *Xxx* Call*Xxx*MethodV(JNIEnv *env, jobject obj, jmethodID id, va_list args)

  calls a method. The return type *Xxx* is one of Object, Boolean, Byte, Char, Short, Int, Long, Float, or Double. The first function has a variable number of arguments—simply append the method parameters after the method ID. The second function receives the method arguments in an array of jvalue, where jvalue is a union defined as


```
typedef union jvalue
{
  jboolean z;
  jbyte b;
  jchar c;
  jshort s;
  jint i;
  jlong j;
  jfloat f;
  jdouble d;
  jobject l;
} jvalue;
```

  The third function receives the method parameters in a va_list, as defined in the C header stdarg.h.

- *Xxx* CallNonvirtual*Xxx*Method(JNIEnv *env, jobject obj, jclass cl, jmethodID id, args)

- *Xxx* CallNonvirtual*Xxx*MethodA(JNIEnv *env, jobject obj, jclass cl, jmethodID id, jvalue args[])

- *Xxx* CallNonvirtual*Xxx*MethodV(JNIEnv *env, jobject obj, jclass cl, jmethodID id, va_list args)

  calls a method, bypassing dynamic dispatch. The return type *Xxx* is one of Object, Boolean, Byte, Char, Short, Int, Long, Float, or Double. The first function has a variable number of arguments—simply append the method parameters after the method ID. The second function receives the method arguments in an array of jvalue. The third function receives the method parameters in a va_list, as defined in the C header stdarg.h.

- jmethodID GetStaticMethodID(JNIEnv *env, jclass cl, const char name[], const char methodSignature[])

returns the identifier of a static method in a class.

- *Xxx* `CallStatic`*Xxx*`Method(JNIEnv *env, jclass cl, jmethodID id, args)`
- *Xxx* `CallStatic`*Xxx*`MethodA(JNIEnv *env, jclass cl, jmethodID id, jvalue args[])`
- *Xxx* `CallStatic`*Xxx*`MethodV(JNIEnv *env, jclass cl, jmethodID id, va_list args)`

  calls a static method. The return type *Xxx* is one of `Object`, `Boolean`, `Byte`, `Char`, `Short`, `Int`, `Long`, `Float`, or `Double`. The first function has a variable number of arguments—simply append the method parameters after the method ID. The second function receives the method arguments in an array of `jvalue`. The third function receives the method parameters in a `va_list`, as defined in the C header `stdarg.h`.

- `jobject NewObject(JNIEnv *env, jclass cl, jmethodID id, args)`
- `jobject NewObjectA(JNIEnv *env, jclass cl, jmethodID id, jvalue args[])`
- `jobject NewObjectV(JNIEnv *env, jclass cl, jmethodID id, va_list args)`

  calls a constructor. The method ID is obtained from `GetMethodID` with a method name of `"<init>"` and a return type of `void`. The first function has a variable number of arguments—simply append the method parameters after the method ID. The second function receives the method arguments in an array of `jvalue`. The third function receives the method parameters in a `va_list`, as defined in the C header `stdarg.h`.

Accessing Array Elements

All array types of the Java programming language have corresponding C types, as shown in Table 12-2.

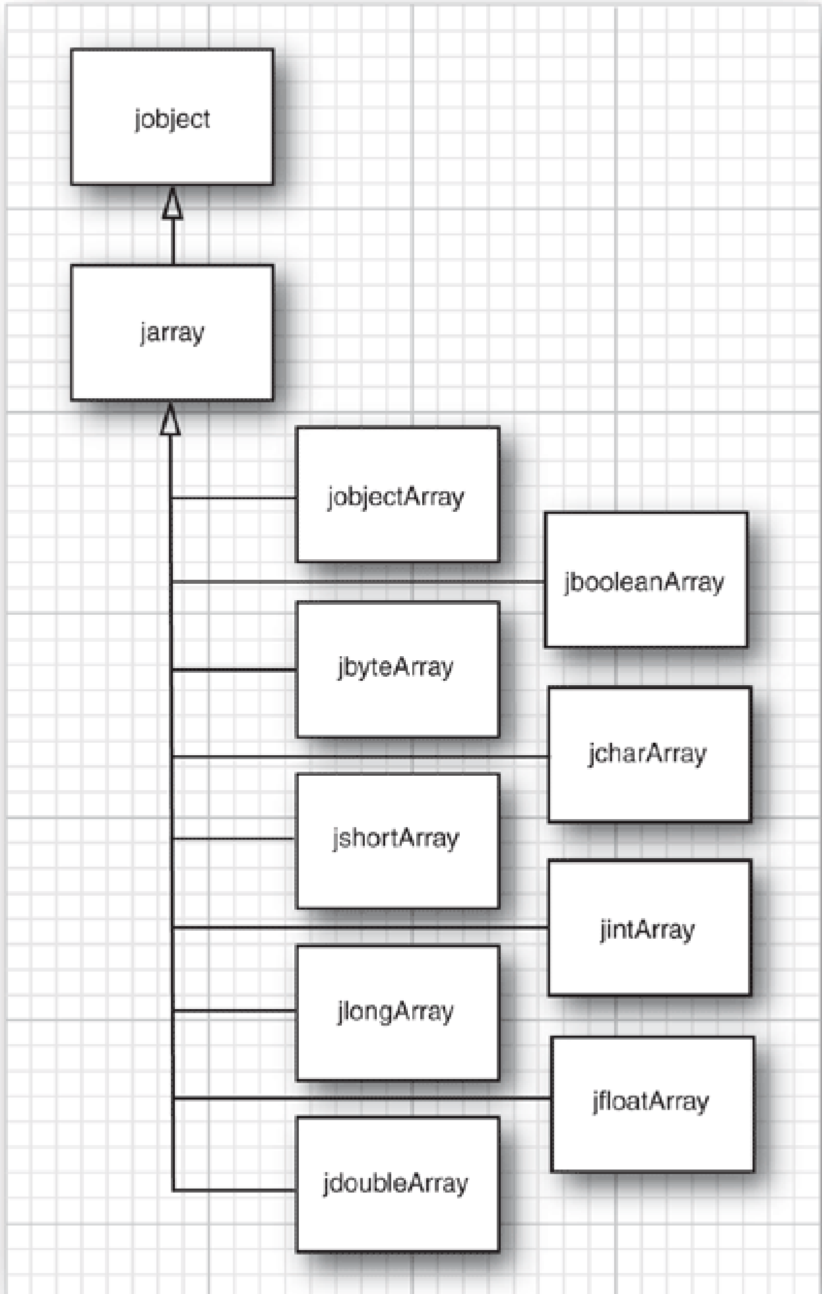
Table 12-2. Correspondence Between Java Array Types and C Types

Java Type	C Type
boolean[]	jbooleanArray
byte[]	jbyteArray
char[]	jcharArray
int[]	jintArray
short[]	jshortArray
long[]	jlongArray
float[]	jfloatArray
double[]	jdoubleArray
Object[]	jobjectArray

C++ Note

In C, all these array types are actually type synonyms of `jobject`. In C++, however, they are arranged in the inheritance hierarchy shown in Figure 12-3. The type `jarray` denotes a generic array.

Figure 12-3. Inheritance hierarchy of array types



The GetArrayLength function returns the length of an array.

```
jarray array = . . .;
jsize length = (*env)->GetArrayLength(env, array);
```

How you access elements in the array depends on whether the array stores objects or a primitive type (bool, char, or a numeric type). You access elements in an object array with the GetObjectArrayElement and SetObjectArrayElement methods.

```
jobjectArray array = . . .;
int i, j;
jobject x = (*env)->GetObjectArrayElement(env, array, i);
(*env)->SetObjectArrayElement(env, array, j, x);
```


Although simple, this approach is also clearly inefficient; you want to be able to access array elements directly, especially when doing vector and matrix computations.

The Get*Xxx*ArrayElements function returns a C pointer to the starting element of the array. As with ordinary strings, you must remember to call the corresponding Release*Xxx*ArrayElements function to tell the virtual machine when you no longer need that pointer. Here, the type *Xxx* must be a primitive type; that is, not Object. You can then read and write the array elements directly. However, because the pointer *might point to a copy*, any changes that you make are guaranteed to be reflected in the original array only when you call the corresponding Release*Xxx*ArrayElements function!

Note

You can find out if an array is a copy by passing a pointer to a jboolean variable as the third parameter to a Get*Xxx*ArrayElements method. The variable is filled with JNI_TRUE if the array is a copy. If you aren't interested in that information, just pass a NULL pointer.

Here is a code sample that multiplies all elements in an array of double values by a constant. We obtain a C pointer a into the Java array and then access individual elements as a[i].

```
jdoubleArray array_a = . . .;
double scaleFactor = . . .;
double* a = (*env)->GetDoubleArrayElements(env, array_a, NULL);
for (i = 0; i < (*env)->GetArrayLength(env, array_a); i++)
   a[i] = a[i] * scaleFactor;
(*env)->ReleaseDoubleArrayElements(env, array_a, a, 0);
```

Whether the virtual machine actually copies the array depends on how it allocates arrays and does its garbage collection. Some "copying" garbage collectors routinely move objects around and update object references. That strategy is not compatible with "pinning" an array to a particular location, because the collector cannot update the pointer values in native code.

Note

In the Sun JVM implementation, `boolean` arrays are represented as packed arrays of 32-bit words. The `GetBooleanArrayElements` method copies them into unpacked arrays of `jboolean` values.

To access just a few elements of a large array, use the `Get`*Xxx*`ArrayRegion` and `Set`*Xxx*`ArrayRegion` methods that copy a range of elements from the Java array into a C array and back.

You can create new Java arrays in native methods with the `New`*Xxx*`Array` function. To create a new array of objects, you specify the length, the type of the array elements, and an initial element for all entries (typically, `NULL`). Here is an example.

Code View:

```
jclass class_Employee = (*env)->FindClass(env, "Employee");
jobjectArray array_e = (*env)->NewObjectArray(env, 100, class_Employee, NULL);
```

Arrays of primitive types are simpler. You just supply the length of the array.

```
jdoubleArray array_d = (*env)->NewDoubleArray(env, 100);
```

The array is then filled with zeroes.

Note

Java SE 1.4 added three methods to the JNI API:

```
jobject NewDirectByteBuffer(JNIEnv* env, void* address, jlong capacity)
void* GetDirectBufferAddress(JNIEnv* env, jobject buf)
jlong GetDirectBufferCapacity(JNIEnv* env, jobject buf)
```

Direct buffers are used in the `java.nio` package to support more efficient input/output operations and to minimize the copying of data between native and Java arrays.

API

Manipulating Java Arrays

- `jsize GetArrayLength(JNIEnv *env, jarray array)`

  returns the number of elements in the array.

- `jobject GetObjectArrayElement(JNIEnv *env, jobjectArray array, jsize index)`

  returns the value of an array element.

- `void SetObjectArrayElement(JNIEnv *env, jobjectArray array, jsize index, jobject value)`

  sets an array element to a new value.

- *Xxx*`* Get`*Xxx*`ArrayElements(JNIEnv *env, jarray array, jboolean* isCopy)`

  yields a C pointer to the elements of a Java array. The field type *Xxx* is one of `Boolean`, `Byte`, `Char`, `Short`, `Int`, `Long`, `Float`, or `Double`. The pointer must be passed to `Release`*Xxx*`ArrayElements` when it is no longer needed.`isCopy` is either `NULL` or points to a `jboolean` that is filled with `JNI_TRUE` if a copy is made; with `JNI_FALSE` otherwise.

- `void Release`*Xxx*`ArrayElements(JNIEnv *env, jarray array, `*Xxx*` elems[], jint mode)`

  notifies the virtual machine that a pointer obtained by `Get`*Xxx*`ArrayElements` is no longer needed. `mode` is one of 0 (free the `elems` buffer after updating the array elements), `JNI_COMMIT` (do not free the `elems` buffer after updating the array elements), or `JNI_ABORT` (free the `elems` buffer without updating the array elements)

- `void Get`*Xxx*`ArrayRegion(JNIEnv *env, jarray array, jint start, jint length, `*Xxx*` elems[])`

  copies elements from a Java array to a C array. The field type *Xxx* is one of `Boolean`, `Byte`, `Char`, `Short`, `Int`, `Long`, `Float`, or `Double`.

- `void Set`*Xxx*`ArrayRegion(JNIEnv *env, jarray array, jint start, jint length, `*Xxx*` elems[])`

  copies elements from a C array to a Java array. The field type *Xxx* is one of `Boolean`, `Byte`, `Char`, `Short`, `Int`, `Long`, `Float`, or `Double`.

Handling Errors

Native methods are a significant security risk to programs in the Java programming language. The C runtime system has no protection against array bounds errors, indirection through bad pointers, and so on. It is particularly important that programmers of native methods handle all error conditions to preserve the integrity of

the Java platform. In particular, when your native method diagnoses a problem that it cannot handle, it should report this problem to the Java virtual machine. Then, you would naturally throw an exception in this situation. However, C has no exceptions. Instead, you must call the `Throw` or `ThrowNew` function to create a new exception object. When the native method exits, the Java virtual machine throws that exception.

To use the `Throw` function, call `NewObject` to create an object of a subtype of `Throwable`. For example, here we allocate an `EOFException` object and throw it.

Code View:

```
jclass class_EOFException = (*env)->FindClass(env, "java/io/EOFException");
jmethodID id_EOFException = (*env)->GetMethodID(env, class_EOFException, "<init>", "()V");
   /* ID of default constructor */
jthrowable obj_exc = (*env)->NewObject(env, class_EOFException, id_EOFException);
(*env)->Throw(env, obj_exc);
```

It is usually more convenient to call `ThrowNew`, which constructs an exception object, given a class and a "modified UTF-8" byte sequence.

Code View:

```
(*env)->ThrowNew(env, (*env)->FindClass(env, "java/io/EOFException"), "Unexpected end of file");
```

Both `Throw` and `ThrowNew` merely *post* the exception; they do not interrupt the control flow of the native method. Only when the method returns does the Java virtual machine throw the exception. Therefore, every call to `Throw` and `ThrowNew` should always immediately be followed by a `return` statement.

C++ Note

If you implement native methods in C++, you cannot throw a Java exception object in your C++ code. In a C++ binding, it would be possible to implement a translation between exceptions in the C++ and Java programming languages—however, this is not currently implemented. Use `Throw` or `ThrowNew` to throw a Java exception in a native C++ method, and make sure that your native methods throw no C++ exceptions.

Normally, native code need not be concerned with catching Java exceptions. However, when a native method calls a Java method, that method might throw an exception. Moreover, a number of the JNI functions throw exceptions as well. For example, SetObjectArrayElement throws an ArrayIndexOutOfBoundsException if the index is out of bounds, and an ArrayStoreException if the class of the stored object is not a subclass of the element class of the array. In situations like these, a native method should call the ExceptionOccurred method to determine whether an exception has been thrown. The call

```
jthrowable obj_exc = (*env)->ExceptionOccurred(env);
```

returns NULL if no exception is pending, or it returns a reference to the current exception object. If you just want to check whether an exception has been thrown, without obtaining a reference to the exception object, use

```
jboolean occurred = (*env)->ExceptionCheck(env);
```

Normally, a native method should simply return when an exception has occurred so that the virtual machine can propagate it to the Java code. However, a native method *may* analyze the exception object to determine if it can handle the exception. If it can, then the function

```
(*env)->ExceptionClear(env);
```

must be called to turn off the exception.

In our next example, we implement the fprint native method with the paranoia that is appropriate for a native method. Here are the exceptions that we throw:

- A NullPointerException if the format string is NULL.
- An IllegalArgumentException if the format string doesn't contain a % specifier that is appropriate for printing a double.
- An OutOfMemoryError if the call to malloc fails.

Finally, to demonstrate how to check for an exception when calling a Java method from a native method, we send the string to the stream, a character at a time, and call ExceptionOccurred after each call. Listing 12-17 shows the code for the native method, and Listing 12-18 contains the definition of the class containing the native method. Notice that the native method does not immediately terminate when an exception occurs in the call to PrintWriter.print—it first frees the cstr buffer. When the native method returns, the virtual machine again raises the exception. The test program in Listing 12-19 demonstrates how the native method throws an exception when the formatting string is not valid.

Listing 12-17. **Printf4.c**

Code View:

```
/**
   @version 1.10 1997-07-01
   @author Cay Horstmann
*/

#include "Printf4.h"
#include <string.h>
#include <stdlib.h>
#include <float.h>
```

```

/**
   @param format a string containing a printf format specifier
   (such as "%8.2f"). Substrings "%%" are skipped.
   @return a pointer to the format specifier (skipping the '%')
   or NULL if there wasn't a unique format specifier
 */
char* find_format(const char format[])
{
```



```
   char* p;
   char* q;

   p = strchr(format, '%');
   while (p != NULL && *(p + 1) == '%') /* skip %% */
      p = strchr(p + 2, '%');
   if (p == NULL) return NULL;
```



```
   /* now check that % is unique */
   p++;
   q = strchr(p, '%');
   while (q != NULL && *(q + 1) == '%') /* skip %% */
      q = strchr(q + 2, '%');
   if (q != NULL) return NULL; /* % not unique */
   q = p + strspn(p, " -0+#"); /* skip past flags */
   q += strspn(q, "0123456789"); /* skip past field width */
   if (*q == '.') { q++; q += strspn(q, "0123456789"); }
      /* skip past precision */
   if (strchr("eEfFgG", *q) == NULL) return NULL;
      /* not a floating-point format */
   return p;
}

JNIEXPORT void JNICALL Java_Printf4_fprint(JNIEnv* env, jclass cl,
   jobject out, jstring format, jdouble x)
{
   const char* cformat;
   char* fmt;
   jclass class_PrintWriter;
   jmethodID id_print;
   char* cstr;
   int width;
   int i;

   if (format == NULL)
   {
      (*env)->ThrowNew(env,
         (*env)->FindClass(env,
         "java/lang/NullPointerException"),
         "Printf4.fprint: format is null");
      return;
   }

   cformat = (*env)->GetStringUTFChars(env, format, NULL);
   fmt = find_format(cformat);

   if (fmt == NULL)
   {
      (*env)->ThrowNew(env,
         (*env)->FindClass(env,
         "java/lang/IllegalArgumentException"),
```

```
         "Printf4.fprint: format is invalid");
      return;
   }

   width = atoi(fmt);
   if (width == 0) width = DBL_DIG + 10;
   cstr = (char*)malloc(strlen(cformat) + width);

   if (cstr == NULL)
   {
      (*env)->ThrowNew(env,
         (*env)->FindClass(env, "java/lang/OutOfMemoryError"),
         "Printf4.fprint: malloc failed");
      return;
   }

   sprintf(cstr, cformat, x);

   (*env)->ReleaseStringUTFChars(env, format, cformat);

   /* now call ps.print(str) */

   /* get the class */
   class_PrintWriter = (*env)->GetObjectClass(env, out);

   /* get the method ID */
   id_print = (*env)->GetMethodID(env, class_PrintWriter, "print", "(C)V");

   /* call the method */
   for (i = 0; cstr[i] != 0 && !(*env)->ExceptionOccurred(env); i++)
      (*env)->CallVoidMethod(env, out, id_print, cstr[i]);

   free(cstr);
}
```

Listing 12-18. **Printf4.java**

Code View:

```
import java.io.*;

/**
 * @version 1.10 1997-07-01
 * @author Cay Horstmann
 */
class Printf4
{
   public static native void fprint(PrintWriter ps, String format, double x);

   static
   {
      System.loadLibrary("Printf4");
   }
}
```


Listing 12-19. **Printf4Test.java**

```
import java.io.*;

/**
 * @version 1.10 1997-07-01
 * @author Cay Horstmann
 */
class Printf4Test
{
   public static void main(String[] args)
   {
      double price = 44.95;
      double tax = 7.75;
      double amountDue = price * (1 + tax / 100);
      PrintWriter out = new PrintWriter(System.out);
      /* This call will throw an exception--note the %% */
      Printf4.fprint(out, "Amount due = %%8.2f\n", amountDue);
      out.flush();
   }
}
```

`Handling Java Exceptions`

- `jint Throw(JNIEnv *env, jthrowable obj)`

  prepares an exception to be thrown upon exiting from the native code. Returns `0` on success, a negative value on failure.

- `jint ThrowNew(JNIEnv *env, jclass cl, const char msg[])`

  prepares an exception of type `cl` to be thrown upon exiting from the native code. Returns `0` on success, a negative value on failure. `msg` is a "modified UTF-8" byte sequence denoting the `String` construction argument of the exception object.

- `jthrowable ExceptionOccurred(JNIEnv *env)`

  returns the exception object if an exception is pending, or `NULL` otherwise.

- `jboolean ExceptionCheck(JNIEnv *env)`

  returns `true` if an exception is pending.

- `void ExceptionClear(JNIEnv *env)`

  clears any pending exceptions.

Using the Invocation API

Up to now, we have considered programs in the Java programming language that made a few C calls, presumably because C was faster or allowed access to functionality that was inaccessible from the Java platform. Suppose you are in the opposite situation. You have a C or C++ program and would like to make calls to Java code. The *invocation API* enables you to embed the Java virtual machine into a C or C++ program. Here is the minimal code that you need to initialize a virtual machine:


```
JavaVMOption options[1];
JavaVMInitArgs vm_args;
JavaVM *jvm;
JNIEnv *env;
```



```
options[0].optionString = "-Djava.class.path=.";

memset(&vm_args, 0, sizeof(vm_args));
vm_args.version = JNI_VERSION_1_2;
vm_args.nOptions = 1;
vm_args.options = options;

JNI_CreateJavaVM(&jvm, (void**) &env, &vm_args);
```

The call to `JNI_CreateJavaVM` creates the virtual machine and fills in a pointer, `jvm`, to the virtual machine and a pointer, `env`, to the execution environment.

You can supply any number of options to the virtual machine. Simply increase the size of the `options` array and the value of `vm_args.nOptions`. For example,

```
options[i].optionString = "-Djava.compiler=NONE";
```

deactivates the just-in-time compiler.

Tip

When you run into trouble and your program crashes, refuses to initialize the JVM, or can't load your classes, then turn on the JNI debugging mode. Set an option to

```
options[i].optionString = "-verbose:jni";
```

You will see a flurry of messages that indicate the progress in initializing the JVM. If you don't see your classes loaded, check both your path and your class path settings.

Once you have set up the virtual machine, you can call Java methods in the way described in the preceding sections: Simply use the env pointer in the usual way.

You need the jvm pointer only to call other functions in the invocation API. Currently, there are only four such functions. The most important one is the function to terminate the virtual machine:

```
(*jvm)->DestroyJavaVM(jvm);
```

Unfortunately, under Windows, it has become difficult to dynamically link to the JNI_CreateJavaVM function in the jre/bin/client/jvm.dll library, due to changed linking rules in Vista and Sun's reliance on an older C runtime library. Our sample program overcomes this problem by loading the library manually. This is the same approach used by the java program—see the file launcher/java_md.c in the src.jar file that is a part of the JDK.

The C program in Listing 12-20 sets up a virtual machine and then calls the main method of the Welcome class, which was discussed in Volume I, Chapter 2. (Make sure to compile the Welcome.java file before starting the invocation test program.)

Listing 12-20. **InvocationTest.c**

Code View:

```
/**
   @version 1.20 2007-10-26
   @author Cay Horstmann
*/

#include <jni.h>
#include <stdlib.h>

#ifdef _WINDOWS

#include <windows.h>
static HINSTANCE loadJVMLibrary(void);
typedef jint (JNICALL *CreateJavaVM_t)(JavaVM **, void **, JavaVMInitArgs *);

#endif

int main()
{
   JavaVMOption options[2];
   JavaVMInitArgs vm_args;
   JavaVM *jvm;
   JNIEnv *env;
   long status;

   jclass class_Welcome;
   jclass class_String;
   jobjectArray args;
   jmethodID id_main;

#ifdef _WINDOWS
   HINSTANCE hjvmlib;
   CreateJavaVM_t createJavaVM;
#endif

   options[0].optionString = "-Djava.class.path=.";

```

```
    memset(&vm_args, 0, sizeof(vm_args));
    vm_args.version = JNI_VERSION_1_2;
    vm_args.nOptions = 1;
    vm_args.options = options;


#ifdef _WINDOWS
    hjvmlib = loadJVMLibrary();
    createJavaVM = (CreateJavaVM_t) GetProcAddress(hjvmlib, "JNI_CreateJavaVM");
```



```
    status = (*createJavaVM)(&jvm, (void **) &env, &vm_args);
#else
    status = JNI_CreateJavaVM(&jvm, (void **) &env, &vm_args);
#endif

    if (status == JNI_ERR)
    {
```



```
       fprintf(stderr, "Error creating VM\n");
       return 1;
    }

    class_Welcome = (*env)->FindClass(env, "Welcome");
    id_main = (*env)->GetStaticMethodID(env, class_Welcome, "main", "([Ljava/lang/String;)V");

    class_String = (*env)->FindClass(env, "java/lang/String");
    args = (*env)->NewObjectArray(env, 0, class_String, NULL);
    (*env)->CallStaticVoidMethod(env, class_Welcome, id_main, args);

    (*jvm)->DestroyJavaVM(jvm);

    return 0;
}

#ifdef _WINDOWS

static int GetStringFromRegistry(HKEY key, const char *name, char *buf, jint bufsize)
{
    DWORD type, size;

    return RegQueryValueEx(key, name, 0, &type, 0, &size) == 0
       && type == REG_SZ
       && size < (unsigned int) bufsize
       && RegQueryValueEx(key, name, 0, 0, buf, &size) == 0;
}

static void GetPublicJREHome(char *buf, jint bufsize)
{
    HKEY key, subkey;
    char version[MAX_PATH];

    /* Find the current version of the JRE */
    char *JRE_KEY = "Software\\JavaSoft\\Java Runtime Environment";
    if (RegOpenKeyEx(HKEY_LOCAL_MACHINE, JRE_KEY, 0, KEY_READ, &key) != 0)
    {
       fprintf(stderr, "Error opening registry key '%s'\n", JRE_KEY);
       exit(1);
    }

    if (!GetStringFromRegistry(key, "CurrentVersion", version, sizeof(version)))
    {
```

```
      fprintf(stderr, "Failed reading value of registry key:\n\t%s\\CurrentVersion\n", JRE_KEY)
      RegCloseKey(key);
      exit(1);
   }

   /* Find directory where the current version is installed. */
   if (RegOpenKeyEx(key, version, 0, KEY_READ, &subkey) != 0)
   {
     fprintf(stderr, "Error opening registry key '%s\\%s'\n", JRE_KEY, version);
      RegCloseKey(key);
      exit(1);
   }

   if (!GetStringFromRegistry(subkey, "JavaHome", buf, bufsize))
   {
      fprintf(stderr, "Failed reading value of registry key:\n\t%s\\%s\\JavaHome\n",
         JRE_KEY, version);
      RegCloseKey(key);
      RegCloseKey(subkey);
      exit(1);
   }

   RegCloseKey(key);
   RegCloseKey(subkey);
}

static HINSTANCE loadJVMLibrary(void)
{
   HINSTANCE h1, h2;
   char msvcdll[MAX_PATH];
   char javadll[MAX_PATH];
   GetPublicJREHome(msvcdll, MAX_PATH);
   strcpy(javadll, msvcdll);
   strncat(msvcdll, "\\bin\\msvcr71.dll", MAX_PATH - strlen(msvcdll));
   msvcdll[MAX_PATH - 1] = '\0';
   strncat(javadll, "\\bin\\client\\jvm.dll", MAX_PATH - strlen(javadll));
   javadll[MAX_PATH - 1] = '\0';

   h1 = LoadLibrary(msvcdll);
   if (h1 == NULL)
   {
      fprintf(stderr, "Can't load library msvcr71.dll\n");
      exit(1);
   }

   h2 = LoadLibrary(javadll);
   if (h2 == NULL)
   {
      fprintf(stderr, "Can't load library jvm.dll\n");
      exit(1);
   }
   return h2;
}

#endif
```

To compile this program under Linux, use

```
gcc -I jdk/include -I jdk/include/linux -o InvocationTest
   -L jdk/jre/lib/i386/client -ljvm InvocationTest.c
```


Under Solaris, use

```
cc -I jdk/include -I jdk/include/solaris -o InvocationTest
   -L jdk/jre/lib/sparc -ljvm InvocationTest.c
```

When compiling in Windows with the Microsoft compiler, use the command line

Code View:

```
cl -D_WINDOWS -I jdk\include -I jdk\include\win32 InvocationTest.c jdk\lib\jvm.lib advapi32.lib
```

You will need to make sure that the `INCLUDE` and `LIB` environment variables include the paths to the Windows API header and library files.

With Cygwin, you compile with

Code View:

```
gcc -D_WINDOWS -mno-cygwin -I jdk\include -I jdk\include\win32 -D__int64="long long"
   -I c:\cygwin\usr\include\w32api -o InvocationTest
```

Before you run the program under Linux/UNIX, make sure that the `LD_LIBRARY_PATH` contains the directories for the shared libraries. For example, if you use the `bash` shell on Linux, issue the following command:

```
export LD_LIBRARY_PATH=jdk/jre/lib/i386/client:$LD_LIBRARY_PATH
```

Invocation API Functions

- `jint JNI_CreateJavaVM(JavaVM** p_jvm, void** p_env, JavaVMInitArgs* vm_args)`

  initializes the Java virtual machine. The function returns `0` if successful, `JNI_ERR` on failure.

Parameters:	`p_jvm`	Filled with a pointer to the invocation API function table
	`p_env`	Filled with a pointer to the JNI function table
	`vm_args`	The virtual machine arguments

- `jint DestroyJavaVM(JavaVM* jvm)`

  destroys the virtual machine. Returns `0` on success, a negative number on failure. This function must be called through a virtual machine pointer, i.e., `(*jvm)->DestroyJavaVM(jvm)`.

A Complete Example: Accessing the Windows Registry

In this section, we describe a full, working example that covers everything we discussed in this chapter: using native methods with strings, arrays, objects, constructor calls, and error handling. We show you how to put a Java platform wrapper around a subset of the ordinary C-based API used to work with the Windows registry. Of course, being a Windows-specific feature, a program using the Windows registry is inherently nonportable. For that reason, the standard Java library has no support for the registry, and it makes sense to use native methods to gain access to it.

Overview of the Windows Registry

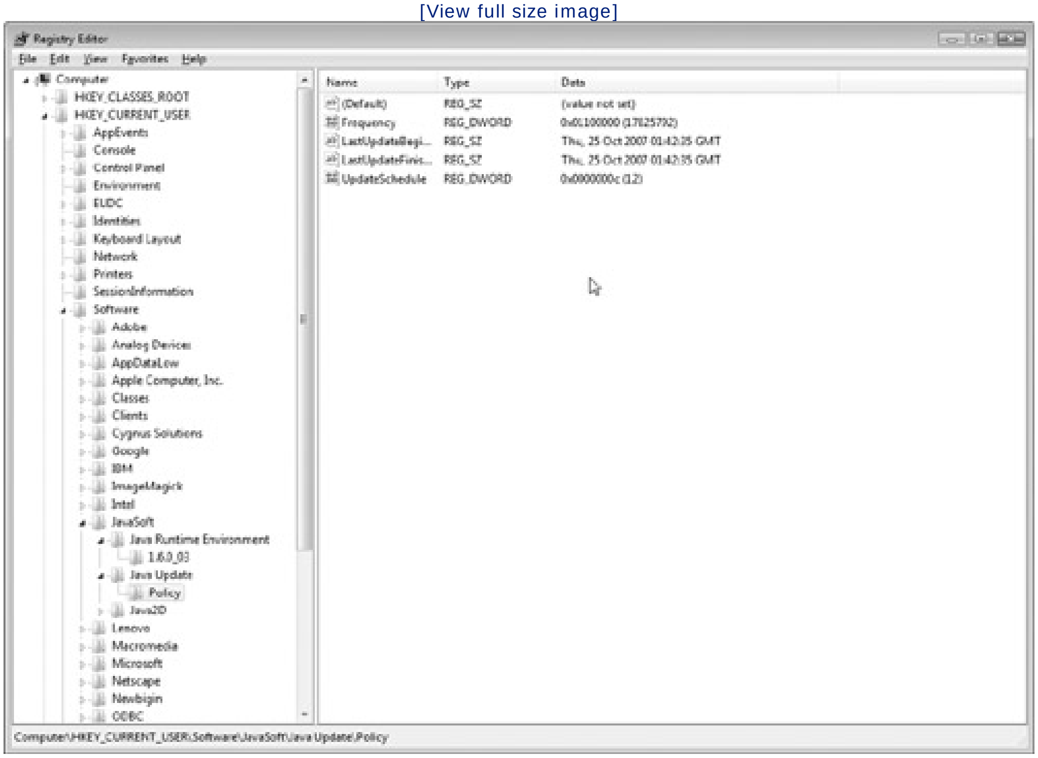
The Windows registry is a data depository that holds configuration information for the Windows operating system and application programs. It provides a single point for administration and backup of system and application preferences. On the downside, the registry is also a single point of failure—if you mess up the registry, your computer could malfunction or even fail to boot!

We don't suggest that you use the registry to store configuration parameters for your Java programs. The Java preferences API is a better solution—see Volume I, Chapter 10 for more information. We simply use the registry to demonstrate how to wrap a nontrivial native API into a Java class.

The principal tool for inspecting the registry is the *registry editor.* Because of the potential for error by naive but enthusiastic users, there is no icon for launching the registry editor. Instead, start a DOS shell (or open the Start -> Run dialog box) and type `regedit`. Figure 12-4 shows the registry editor in action.

Figure 12-4. The registry editor

[View full size image]



The left side shows the keys, which are arranged in a tree structure. Note that each key starts with one of the `HKEY` nodes like

```
HKEY_CLASSES_ROOT
HKEY_CURRENT_USER
HKEY_LOCAL_MACHINE
. . .
```

The right side shows the name/value pairs that are associated with a particular key. For example, if you installed Java SE 6, the key

```
HKEY_LOCAL_MACHINE\Software\JavaSoft\Java Runtime Environment
```

contains a name/value pair such as

```
CurrentVersion="1.6.0_03"
```

In this case, the value is a string. The values can also be integers or arrays of bytes.

A Java Platform Interface for Accessing the Registry

We implement a simple interface to access the registry from Java code, and then implement this interface with native code. Our interface allows only a few registry operations; to keep the code size down, we omitted other important operations such as adding, deleting, and enumerating keys. (It would be easy to add the remaining registry API functions.)

Even with the limited subset that we supply, you can

- Enumerate all names stored in a key.
- Read the value stored with a name.
- Set the value stored with a name.

Here is the Java class that encapsulates a registry key.

```
public class Win32RegKey
{
   public Win32RegKey(int theRoot, String thePath) { . . . }
   public Enumeration names() { . . . }
   public native Object getValue(String name);
   public native void setValue(String name, Object value);

   public static final int HKEY_CLASSES_ROOT = 0x80000000;
   public static final int HKEY_CURRENT_USER = 0x80000001;
   public static final int HKEY_LOCAL_MACHINE = 0x80000002;
   . . .
```

```
}
```

The `names` method returns an enumeration that holds all the names stored with the key. You can get at them with the familiar `hasMoreElements/nextElement` methods. The `getValue` method returns an object that is either a string, an `Integer` object, or a byte array. The value parameter of the `setValue` method must also be of one of these three types.

Implementation of Registry Access Functions as Native Methods

We need to implement three actions:

- Get the value of a key.

- Set the value of a key.
- Iterate through the names of a key.

Fortunately, you have seen essentially all the tools that are required, such as the conversion between Java strings and arrays and those of C. You also saw how to raise a Java exception in case something goes wrong.

Two issues make these native methods more complex than the preceding examples. The `getValue` and `setValue` methods deal with the type `Object`, which can be one of `String`, `Integer`, or `byte[]`. The enumeration object stores the state between successive calls to `hasMoreElements` and `nextElement`.

Let us first look at the `getValue` method. The method (which is shown in Listing 12-22) goes through the following steps:

1. Opens the registry key. To read their values, the registry API requires that keys be open.
2. Queries the type and size of the value that is associated with the name.
3. Reads the data into a buffer.
4. Calls `NewStringUTF` to create a new string with the value data if the type is `REG_SZ` (a string).
5. Invokes the `Integer` constructor if the type is `REG_DWORD` (a 32-bit integer).
6. Calls `NewByteArray` to create a new byte array, then `SetByteArrayRegion` to copy the value data into the byte array, if the type is `REG_BINARY`.
7. If the type is none of these or if an error occurred when an API function was called, throws an exception and releases all resources that had been acquired up to that point.
8. Closes the key and returns the object (`String`, `Integer`, or `byte[]`) that had been created.

As you can see, this example illustrates quite nicely how to generate Java objects of different types.

In this native method, coping with the generic return type is not difficult. The `jstring`, `jobject`, or `jarray` reference is simply returned as a `jobject`. However, the `setValue` method receives a reference to an `Object` and must determine the `Object`'s exact type to save the `Object` as a string, integer, or byte array. We can make this determination by querying the class of the `value` object, finding the class references for `java.lang.String`, `java.lang.Integer`, and `byte[]`, and comparing them with the `IsAssignableFrom` function.

If `class1` and `class2` are two class references, then the call

```
(*env)->IsAssignableFrom(env, class1, class2)
```

returns `JNI_TRUE` when `class1` and `class2` are the same class or when `class1` is a subclass of `class2`. In either case, references to objects of `class1` can be cast to `class2`. For example, when

Code View:
```
(*env)->IsAssignableFrom(env, (*env)->GetObjectClass(env, value), (*env)->FindClass(env, "[B"))
```

is `true`, then we know that `value` is a byte array.

Here is an overview of the steps in the `setValue` method.

1. Opens the registry key for writing.
2. Finds the type of the value to write.
3. Calls `GetStringUTFChars` to get a pointer to the characters if the type is `String`.
4. Calls the `intValue` method to get the integer stored in the wrapper object if the type is `Integer`.
5. Calls `GetByteArrayElements` to get a pointer to the bytes if the type is `byte[]`.
6. Passes the data and length to the registry.
7. Closes the key
8. Releases the pointer to the data if the type is `String` or `byte[]`.

Finally, let us turn to the native methods that enumerate keys. These are methods of the `Win32RegKeyNameEnumeration` class (see Listing 12-21). When the enumeration process starts, we must open the key. For the duration of the enumeration, we must retain the key handle. That is, the key handle must be stored with the enumeration object. The key handle is of type `DWORD`, a 32-bit quantity, and, hence, can be stored in a Java integer. It is stored in the `hkey` field of the enumeration class. When the enumeration starts, the field is initialized with `SetIntField`. Subsequent calls read the value with `GetIntField`.

In this example, we store three other data items with the enumeration object. When the enumeration first starts, we can query the registry for the count of name/value pairs and the length of the longest name, which we need so we can allocate C character arrays to hold the names. These values are stored in the `count` and `maxsize` fields of the enumeration object. Finally, the `index` field is initialized with -1 to indicate the start of the enumeration, is set to `0` once the other instance fields are initialized, and is incremented after every enumeration step.

Let's walk through the native methods that support the enumeration. The `hasMoreElements` method is simple:

1. Retrieves the `index` and `count` fields.

2. If the index is -1, calls the `startNameEnumeration` function, which opens the key, queries the count and maximum length, and initializes the `hkey`, `count`, `maxsize`, and `index` fields.

3. Returns `JNI_TRUE` if `index` is less than `count`; `JNI_FALSE` otherwise.

The `nextElement` method needs to work a little harder:

1. Retrieves the `index` and `count` fields.

2. If the index is -1, calls the `startNameEnumeration` function, which opens the key, queries the count and maximum length, and initializes the `hkey`, `count`, `maxsize`, and `index` fields.

3. If `index` equals `count`, throws a `NoSuchElementException`.

4. Reads the next name from the registry.

5. Increments `index`.

6. If `index` equals `count`, closes the key.

Before compiling, remember to run `javah` on both `Win32RegKey` and `Win32RegKeyNameEnumeration`. The complete command line for the Microsoft compiler is

Code View:
```
cl -I jdk\include -I jdk\include\win32 -LD Win32RegKey.c advapi32.lib -FeWin32RegKey.dll
```

With Cygwin, use

Code View:
```
gcc -mno-cygwin -D __int64="long long" -I jdk\include -I jdk\include\win32
   -I c:\cygwin\usr\include\w32api -shared -Wl,--add-stdcall-alias -o Win32RegKey.dll
Win32RegKey.c
```

Because the registry API is specific to Windows, this program will not work on other operating systems.

Listing 12-23 shows a program to test our new registry functions. We add three name/value pairs, a string, an integer, and a byte array to the key.

```
HKEY_CURRENT_USER\Software\JavaSoft\Java Runtime Environment
```

We then enumerate all names of that key and retrieve their values. The program will print

```
Default user=Harry Hacker
Lucky number=13
Small primes=2 3 5 7 11 13
```

Although adding these name/value pairs to that key probably does no harm, you might want to use the registry editor to remove them after running this program.

Listing 12-21. **Win32RegKey.java**

Code View:

```
import java.util.*;

/**
 * A Win32RegKey object can be used to get and set values of a registry key in the Windows
 * registry.
 * @version 1.00 1997-07-01
 * @author Cay Horstmann
 */
public class Win32RegKey
{
   /**
    * Construct a registry key object.
    * @param theRoot one of HKEY_CLASSES_ROOT, HKEY_CURRENT_USER, HKEY_LOCAL_MACHINE,
    * HKEY_USERS, HKEY_CURRENT_CONFIG, HKEY_DYN_DATA
    * @param thePath the registry key path
    */
   public Win32RegKey(int theRoot, String thePath)
   {
      root = theRoot;
      path = thePath;
   }

   /**
    * Enumerates all names of registry entries under the path that this object describes.
    * @return an enumeration listing all entry names
    */
   public Enumeration<String> names()
   {
      return new Win32RegKeyNameEnumeration(root, path);
   }

   /**
    * Gets the value of a registry entry.
    * @param name the entry name
    * @return the associated value
    */
   public native Object getValue(String name);
   /**
    * Sets the value of a registry entry.
    * @param name the entry name
    * @param value the new value
    */

   public native void setValue(String name, Object value);

   public static final int HKEY_CLASSES_ROOT = 0x80000000;
   public static final int HKEY_CURRENT_USER = 0x80000001;
```

```
   public static final int HKEY_LOCAL_MACHINE = 0x80000002;
   public static final int HKEY_USERS = 0x80000003;
   public static final int HKEY_CURRENT_CONFIG = 0x80000005;
   public static final int HKEY_DYN_DATA = 0x80000006;

   private int root;
   private String path;

   static
   {
      System.loadLibrary("Win32RegKey");
   }
}

class Win32RegKeyNameEnumeration implements Enumeration<String>
{
   Win32RegKeyNameEnumeration(int theRoot, String thePath)
   {
      root = theRoot;
      path = thePath;
   }

   public native String nextElement();

   public native boolean hasMoreElements();

   private int root;
   private String path;
   private int index = -1;
   private int hkey = 0;
   private int maxsize;
   private int count;
}

class Win32RegKeyException extends RuntimeException
{
   public Win32RegKeyException()
   {
   }

   public Win32RegKeyException(String why)
   {
      super(why);
   }
}
```


Listing 12-22. **Win32RegKey.c**

Code View:

```
/**
   @version 1.00 1997-07-01
   @author Cay Horstmann
```

```
*/

#include "Win32RegKey.h"
#include "Win32RegKeyNameEnumeration.h"
#include <string.h>
#include <stdlib.h>
#include <windows.h>

JNIEXPORT jobject JNICALL Java_Win32RegKey_getValue(JNIEnv* env, jobject this_obj, jobject name
{
   const char* cname;
   jstring path;
   const char* cpath;
   HKEY hkey;
   DWORD type;
   DWORD size;
   jclass this_class;
   jfieldID id_root;
   jfieldID id_path;
   HKEY root;
   jobject ret;
   char* cret;

   /* get the class */
   this_class = (*env)->GetObjectClass(env, this_obj);

   /* get the field IDs */
   id_root = (*env)->GetFieldID(env, this_class, "root", "I");
   id_path = (*env)->GetFieldID(env, this_class, "path", "Ljava/lang/String;");

   /* get the fields */
   root = (HKEY) (*env)->GetIntField(env, this_obj, id_root);
   path = (jstring)(*env)->GetObjectField(env, this_obj, id_path);
   cpath = (*env)->GetStringUTFChars(env, path, NULL);

   /* open the registry key */
   if (RegOpenKeyEx(root, cpath, 0, KEY_READ, &hkey) != ERROR_SUCCESS)
   {
      (*env)->ThrowNew(env, (*env)->FindClass(env, "Win32RegKeyException"),
         "Open key failed");
      (*env)->ReleaseStringUTFChars(env, path, cpath);
      return NULL;
   }

   (*env)->ReleaseStringUTFChars(env, path, cpath);
   cname = (*env)->GetStringUTFChars(env, name, NULL);

   /* find the type and size of the value */
   if (RegQueryValueEx(hkey, cname, NULL, &type, NULL, &size) != ERROR_SUCCESS)
   {
      (*env)->ThrowNew(env, (*env)->FindClass(env, "Win32RegKeyException"),
         "Query value key failed");
      RegCloseKey(hkey);
      (*env)->ReleaseStringUTFChars(env, name, cname);
      return NULL;
   }

   /* get memory to hold the value */
   cret = (char*)malloc(size);
```

```

   /* read the value */
   if (RegQueryValueEx(hkey, cname, NULL, &type, cret, &size) != ERROR_SUCCESS)
   {
      (*env)->ThrowNew(env, (*env)->FindClass(env, "Win32RegKeyException"),
         "Query value key failed");
      free(cret);
      RegCloseKey(hkey);
      (*env)->ReleaseStringUTFChars(env, name, cname);
```

```
      return NULL;
   }

   /* depending on the type, store the value in a string,
      integer or byte array */
   if (type == REG_SZ)
```

```
   {
      ret = (*env)->NewStringUTF(env, cret);
   }
   else if (type == REG_DWORD)
   {
      jclass class_Integer = (*env)->FindClass(env, "java/lang/Integer");
      /* get the method ID of the constructor */
      jmethodID id_Integer = (*env)->GetMethodID(env, class_Integer, "<init>", "(I)V");
      int value = *(int*) cret;
      /* invoke the constructor */
      ret = (*env)->NewObject(env, class_Integer, id_Integer, value);
   }
   else if (type == REG_BINARY)
   {
      ret = (*env)->NewByteArray(env, size);
      (*env)->SetByteArrayRegion(env, (jarray) ret, 0, size, cret);
   }
   else
   {
      (*env)->ThrowNew(env, (*env)->FindClass(env, "Win32RegKeyException"),
         "Unsupported value type");
      ret = NULL;
   }

   free(cret);
   RegCloseKey(hkey);
   (*env)->ReleaseStringUTFChars(env, name, cname);

   return ret;
}

JNIEXPORT void JNICALL Java_Win32RegKey_setValue(JNIEnv* env, jobject this_obj,
   jstring name, jobject value)
{
   const char* cname;
   jstring path;
   const char* cpath;
   HKEY hkey;
   DWORD type;
   DWORD size;
   jclass this_class;
   jclass class_value;
   jclass class_Integer;
   jfieldID id_root;
```

```
   jfieldID id_path;
   HKEY root;
   const char* cvalue;
   int ivalue;

   /* get the class */
   this_class = (*env)->GetObjectClass(env, this_obj);

   /* get the field IDs */
   id_root = (*env)->GetFieldID(env, this_class, "root", "I");
   id_path = (*env)->GetFieldID(env, this_class, "path", "Ljava/lang/String;");

   /* get the fields */
   root = (HKEY)(*env)->GetIntField(env, this_obj, id_root);
   path = (jstring)(*env)->GetObjectField(env, this_obj, id_path);
   cpath = (*env)->GetStringUTFChars(env, path, NULL);

   /* open the registry key */
   if (RegOpenKeyEx(root, cpath, 0, KEY_WRITE, &hkey) != ERROR_SUCCESS)
   {
      (*env)->ThrowNew(env, (*env)->FindClass(env, "Win32RegKeyException"),
         "Open key failed");
      (*env)->ReleaseStringUTFChars(env, path, cpath);
      return;
   }

   (*env)->ReleaseStringUTFChars(env, path, cpath);
   cname = (*env)->GetStringUTFChars(env, name, NULL);

   class_value = (*env)->GetObjectClass(env, value);
   class_Integer = (*env)->FindClass(env, "java/lang/Integer");
   /* determine the type of the value object */
   if ((*env)->IsAssignableFrom(env, class_value, (*env)->FindClass(env, "java/lang/String")))
   {
      /* it is a string--get a pointer to the characters */
      cvalue = (*env)->GetStringUTFChars(env, (jstring) value, NULL);
      type = REG_SZ;
      size = (*env)->GetStringLength(env, (jstring) value) + 1;
   }
   else if ((*env)->IsAssignableFrom(env, class_value, class_Integer))
   {
      /* it is an integer--call intValue to get the value */
      jmethodID id_intValue = (*env)->GetMethodID(env, class_Integer, "intValue", "()I");
      ivalue = (*env)->CallIntMethod(env, value, id_intValue);
      type = REG_DWORD;
      cvalue = (char*)&ivalue;
      size = 4;
   }
   else if ((*env)->IsAssignableFrom(env, class_value, (*env)->FindClass(env, "[B")))
   {
      /* it is a byte array--get a pointer to the bytes */
      type = REG_BINARY;
      cvalue = (char*)(*env)->GetByteArrayElements(env, (jarray) value, NULL);
      size = (*env)->GetArrayLength(env, (jarray) value);
   }
   else
   {
      /* we don't know how to handle this type */
      (*env)->ThrowNew(env, (*env)->FindClass(env, "Win32RegKeyException"),
```

```
         "Unsupported value type");
      RegCloseKey(hkey);
      (*env)->ReleaseStringUTFChars(env, name, cname);
      return;
   }

   /* set the value */
   if (RegSetValueEx(hkey, cname, 0, type, cvalue, size) != ERROR_SUCCESS)
   {
```

```
      (*env)->ThrowNew(env, (*env)->FindClass(env, "Win32RegKeyException"),
         "Set value failed");
   }

   RegCloseKey(hkey);
   (*env)->ReleaseStringUTFChars(env, name, cname);
```

```

   /* if the value was a string or byte array, release the pointer */
   if (type == REG_SZ)
   {
      (*env)->ReleaseStringUTFChars(env, (jstring) value, cvalue);
   }
   else if (type == REG_BINARY)
   {
      (*env)->ReleaseByteArrayElements(env, (jarray) value, (jbyte*) cvalue, 0);
   }
}

/* helper function to start enumeration of names */
static int startNameEnumeration(JNIEnv* env, jobject this_obj, jclass this_class)
{
   jfieldID id_index;
   jfieldID id_count;
   jfieldID id_root;
   jfieldID id_path;
   jfieldID id_hkey;
   jfieldID id_maxsize;

   HKEY root;
   jstring path;
   const char* cpath;
   HKEY hkey;
   DWORD maxsize = 0;
   DWORD count = 0;

   /* get the field IDs */
   id_root = (*env)->GetFieldID(env, this_class, "root", "I");
   id_path = (*env)->GetFieldID(env, this_class, "path", "Ljava/lang/String;");
   id_hkey = (*env)->GetFieldID(env, this_class, "hkey", "I");
   id_maxsize = (*env)->GetFieldID(env, this_class, "maxsize", "I");
   id_index = (*env)->GetFieldID(env, this_class, "index", "I");
   id_count = (*env)->GetFieldID(env, this_class, "count", "I");

   /* get the field values */
   root = (HKEY)(*env)->GetIntField(env, this_obj, id_root);
   path = (jstring)(*env)->GetObjectField(env, this_obj, id_path);
   cpath = (*env)->GetStringUTFChars(env, path, NULL);

   /* open the registry key */
   if (RegOpenKeyEx(root, cpath, 0, KEY_READ, &hkey) != ERROR_SUCCESS)
```

```
   {
      (*env)->ThrowNew(env, (*env)->FindClass(env, "Win32RegKeyException"),
         "Open key failed");
      (*env)->ReleaseStringUTFChars(env, path, cpath);
      return -1;
   }
   (*env)->ReleaseStringUTFChars(env, path, cpath);

   /* query count and max length of names */
   if (RegQueryInfoKey(hkey, NULL, NULL, NULL, NULL, NULL, NULL, &count, &maxsize,
          NULL, NULL, NULL) != ERROR_SUCCESS)
   {
      (*env)->ThrowNew(env, (*env)->FindClass(env, "Win32RegKeyException"),
         "Query info key failed");
      RegCloseKey(hkey);
      return -1;
   }

   /* set the field values */
   (*env)->SetIntField(env, this_obj, id_hkey, (DWORD) hkey);
   (*env)->SetIntField(env, this_obj, id_maxsize, maxsize + 1);
   (*env)->SetIntField(env, this_obj, id_index, 0);
   (*env)->SetIntField(env, this_obj, id_count, count);
   return count;
}

JNIEXPORT jboolean JNICALL Java_Win32RegKeyNameEnumeration_hasMoreElements(JNIEnv* env,
   jobject this_obj)
{  jclass this_class;
   jfieldID id_index;
   jfieldID id_count;
   int index;
   int count;
   /* get the class */
   this_class = (*env)->GetObjectClass(env, this_obj);

   /* get the field IDs */
   id_index = (*env)->GetFieldID(env, this_class, "index", "I");
   id_count = (*env)->GetFieldID(env, this_class, "count", "I");

   index = (*env)->GetIntField(env, this_obj, id_index);
   if (index == -1) /* first time */
   {
      count = startNameEnumeration(env, this_obj, this_class);
      index = 0;
   }
   else
      count = (*env)->GetIntField(env, this_obj, id_count);
   return index < count;
}

JNIEXPORT jobject JNICALL Java_Win32RegKeyNameEnumeration_nextElement(JNIEnv* env,
   jobject this_obj)
{
   jclass this_class;
   jfieldID id_index;
   jfieldID id_hkey;
   jfieldID id_count;
   jfieldID id_maxsize;
```

```

   HKEY hkey;
   int index;
   int count;
   DWORD maxsize;

   char* cret;
   jstring ret;

```



```
   /* get the class */
   this_class = (*env)->GetObjectClass(env, this_obj);

   /* get the field IDs */
   id_index = (*env)->GetFieldID(env, this_class, "index", "I");
   id_count = (*env)->GetFieldID(env, this_class, "count", "I");
```



```
   id_hkey = (*env)->GetFieldID(env, this_class, "hkey", "I");
   id_maxsize = (*env)->GetFieldID(env, this_class, "maxsize", "I");

   index = (*env)->GetIntField(env, this_obj, id_index);
   if (index == -1) /* first time */
   {
      count = startNameEnumeration(env, this_obj, this_class);
      index = 0;
   }
   else
      count = (*env)->GetIntField(env, this_obj, id_count);

   if (index >= count) /* already at end */
   {
      (*env)->ThrowNew(env, (*env)->FindClass(env, "java/util/NoSuchElementException"),
         "past end of enumeration");
      return NULL;
   }

   maxsize = (*env)->GetIntField(env, this_obj, id_maxsize);
   hkey = (HKEY)(*env)->GetIntField(env, this_obj, id_hkey);
   cret = (char*)malloc(maxsize);

   /* find the next name */
   if (RegEnumValue(hkey, index, cret, &maxsize, NULL, NULL, NULL, NULL) != ERROR_SUCCESS)
   {
      (*env)->ThrowNew(env, (*env)->FindClass(env, "Win32RegKeyException"),
         "Enum value failed");
      free(cret);
      RegCloseKey(hkey);
      (*env)->SetIntField(env, this_obj, id_index, count);
      return NULL;
   }

   ret = (*env)->NewStringUTF(env, cret);
   free(cret);

   /* increment index */
   index++;
   (*env)->SetIntField(env, this_obj, id_index, index);

   if (index == count) /* at end */
   {
      RegCloseKey(hkey);
```

```
   }

   return ret;
}
```

Listing 12-23. **Win32RegKeyTest.java**

Code View:

```
import java.util.*;

/**
   @version 1.02 2007-10-26
   @author Cay Horstmann
*/
public class Win32RegKeyTest
{
   public static void main(String[] args)
   {
      Win32RegKey key = new Win32RegKey(
         Win32RegKey.HKEY_CURRENT_USER, "Software\\JavaSoft\\Java Runtime Environment");

      key.setValue("Default user", "Harry Hacker");
      key.setValue("Lucky number", new Integer(13));
      key.setValue("Small primes", new byte[] { 2, 3, 5, 7, 11 });

      Enumeration<String> e = key.names();

      while (e.hasMoreElements())
      {
         String name = e.nextElement();
         System.out.print(name + "=");

         Object value = key.getValue(name);

         if (value instanceof byte[])
            for (byte b : (byte[]) value) System.out.print((b & 0xFF) + " ");
         else
            System.out.print(value);

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

Type Inquiry Functions

- `jboolean IsAssignableFrom(JNIEnv *env, jclass cl1, jclass cl2)`

  returns `JNI_TRUE` if objects of the first class can be assigned to objects of the second class; `JNI_FALSE` otherwise. This is the case in which the classes are the same, `cl1` is a subclass of `cl2`, or `cl2` represents an interface that is implemented by `cl1` or one of its superclasses.

- `jclass GetSuperclass(JNIEnv *env, jclass cl)`

  returns the superclass of a class. If `cl` represents the class `Object` or an interface, returns `NULL`.

You have now reached the end of the second volume of *Core Java*, completing a long journey in which you encountered many advanced APIs. We started out with topics that every Java programmer needs to know: streams, XML, networking, datatabases, and internationalization. Three long chapters covered graphics and GUI programming. We concluded with very technical chapters on security, remote methods, annotation processing, and native methods. We hope that you enjoyed your tour through the the vast breadth of the Java APIs, and that you can apply your newly gained knowledge in your projects.

Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Z]

Index

character, in a choice format
$ (dollar sign), matching beginning and end of a line
% character, in a LIKE clause
@ operator, in XPath
@ symbol, preceding the name of each annotation
[] operator, in XPath
\ (backslash)
    as an escape character
    in a Windows environment
\\ (backslashes), for Windows-style path names
\\ escape sequence, in a Windows file name
"\\|" expression
/ (forward slash) [See Forward slash (/).]
]]> string
^, matching beginning and end of a line
| characters, in a choice format
+ (possessive or greedy match)
< symbol, in a choice format
<= symbol, in a choice format
<> operator, in SQL
= operator, in SQL
= = operator, testing for object equality
? (question mark)
    in a prepared query
    in date output
? (reluctant or stingy match)
; (semicolon), annotation placed without
- character, in a LIKE clause
. symbol, matching any character
"2D", classes with a name ending in
2D graphics, printing
3D rectangle
8-bit Unicode Transformation Format
32-bit cyclic redundancy checksum [See CRC32 checksum.]

Index

Absolute identifiers
Absolute nonopaque URIs
Absolute path name
Absolute URI
Abstract method declarations
ABSTRACT modifier
Abstract syntax notation #1 [See ASN.1.]
AbstractCellEditor class 2nd
AbstractFormatter class
AbstractListModel class
AbstractProcessor class
AbstractSpinnerModel class 2nd
AbstractTableModel class 2nd
accept method
acceptChanges method
Access control mechanism
Accessor methods
Action event listener
Action listeners, installing
ActionListener interface
ActionListenerFor.java
ActionListenerInstaller class
ActionListenerInstaller.java
Actions lists, for permissions
Activatable class 2nd 3rd
Activatable warehouse implementation 2nd
ACTIVATED value, for getEventType
Activation, of remote objects
Activation descriptors, constructing 2nd
Activation group
Activation ID
Activation program 2nd
ActivationDesc class
ActivationGroup class
ActivationGroupDesc class
ActivationSystem class
add method, of the SystemTray class
add operation 2nd
addBatch method
addChangeListener method
addColumn method
addEventHandlers method
addPropertyChangeListener method 2nd
addTab method
addTreeSelectionListener method
addVetoableChangeListener method
addWindowListener method
AES (Advanced Encryption Standard) algorithm
AES key 2nd
AESTest.java
Affine transformation
Affine transforms, constructing
AffineTransform class 2nd

- AffineTransform object
- AffineTransformOp class 2nd
- Agent
- Aliases
  - for ISO-8859-1
  - iterating through
  - for namespaces in XML
- aliases method
- Allows children node property

- AllPermission permission
- Alnum character class
- Alpha channel
- Alpha character class
- Alpha composites
- AlphaComposite class 2nd
- AlphaComposite object 2nd

- AlreadyBoundException
- Altered class files, constructing
- Amazon e-commerce web service
- AmazonTest.java
- Anchor rectangle
- andFilter method
- Angle swept out, for an arc
- AnnotatedElement class
- Annotation(s)
  - circular dependencies for
  - for compilation
  - defined
  - for event handlers
  - example of simple
  - for managing resources
  - passing at runtime
  - processing source-level
  - shortcuts simplifying
  - using
- Annotation elements 2nd
- Annotation interfaces 2nd
  - defined by Java SE
  - defining an annotation 2nd
  - extending
- Annotation objects, source fields locked in
- Annotation processors
- Annotation syntax
- Anonymous type definition
- Antialiasing technique 2nd
- Apache Batik viewer 2nd
- Apache Derby database [See Derby database.]
- append methods 2nd
- Appendable interface 2nd 3rd
- Applet class 2nd
- Applet viewer, security policy
- Applets
  - executing safely
  - JDBC in
  - not exiting the virtual machine
- Application(s) [See also Java applications.]
  - building in Visual Basic
  - deploying RMI
  - managing frames
  - using beans to build
- Application class loader [See System class loader.]
- Application classes, loading
- Application data, storing

Application programs, file locking in
Application servers, structure for
apt stand-alone tool
Arbitrary data, using JavaBeans persistence
Arbitrary sequences, building
Arc(s) 2nd
Arc angles 2nd 3rd
Arc2D class
Arc2D.CHORD arc type
Arc2D.Double class
Arc2D.OPEN arc type
Arc2D.PIE arc type
ArcMaker class 2nd
Area class
Areas
ARGB color value 2nd 3rd
Array(s)
    creating Java in native methods
    element values as
    manipulating Java
    multiplying elements in by a constant
    properties specifying
    saving in object serialization format
ARRAY data type, in SQL
Array elements, accessing
Array types 2nd
Array values, fetching
ArrayIndexOutOfBoundsException
ArrayStoreException
ASCII (American Standard Code for Information Exchange)
ASCII character class
ASCII encoding, using plain
ASCII files, storing properties
ASN.1 2nd
ASN.1 - Communication Between Heterogeneous Systems (Dubuisson)
ASN.1 Complete (Larmouth)
Asymmetry, of the Swing table
Attribute(s) [See also Printing attributes.]
    advantage for enumerated types
    checking the value of
    compared to elements
    enumerating all in LDAP
    for grid bag constraints
    groups of
    LDAP 2nd
    retrieving
    in SVG
    in XML
    in XML elements
    in XML Schema
Attribute class 2nd
Attribute hierarchy, class diagram of
Attribute interface
Attribute names, in HTML
Attribute set(s)
    constructing
    hierarchy
    interfaces and classes for
    as a specialized kind of map
Attribute types
Attribute values
    copying with XSLT
    in XML

`Attributes` class 2nd
`AttributeSet` superinterface 2nd
`AttributesImpl` class
`AudioPermission` permission
Authentication
    to SMTP
    of users
Authentication problem
`AuthenticationException`

`authority` part, of server-based URIs
Authorization, of users
`AuthPermission` permission
`AuthTest.java`
Autoboxing
Autocommit mode
Autoflush mode

Autogenerated keys
Automatic registration
Automatic resizing, of table columns
Auto-numbering rows, in a database
Auxiliary files, automatic generation of
`available` method 2nd
`availableCharsets` method
Average value, replacement of each pixel with
`AWTPermission` permission

Index

Background color, of a cell
Backslash (\) [See \ (backslash).]
Bad words, not allowing into a text area
Banding, in dot-matrix and inkjet printers
Banner, printing 2nd
Base URI
`BASE64Encoder` class
Basic encoding rules (BER)
`BasicAttributes` class
`BasicAttributes` constructor
`BasicAttributes` object
`BasicPermission` class
`BasicStroke` class 2nd
`BasicStroke` constructer 2nd
Batch updates
BCEL (Bytecode Engineering Library) 2nd
Bean Builder, experimental
Bean descriptor
Bean info classes 2nd 3rd
`BeanDescriptor` class
`BeanInfo` classes
    API notes 2nd 3rd
    setting a property using
    supplying 2nd
`BeanInfoAnnotationFactory.java`
`BeanInfoAnnotationProcessor`
Beans [See also JavaBeans.]
    composing in a builder environment
    defined
    packaging in JAR files
    property types
    rules for designing
    saving to a stream
    using to build an application
    writing
`Beans` class
BER (basic encoding rules)
Bevel join 2nd
`BIG_ENDIAN` constant
Big-endian method
Bilinear interpolation
Binary data
    from a `Blob`
    reading and writing
    reading from a file
    writing
Binary format, for saving data
Binary values, reading
Bindings
`Bindings` class
Biometric login modules
`BitSet` object, re-creating
`Blank` character class
Blending, of source and destination

Blob class
BLOB data type, in SQL 2nd
BLOBs (binary large objects)
Blocking, by read and write methods
Blur filter
Book class 2nd
Book.java
Books table, view of 2nd
BooksAuthors table

BookTest.java
boolean arrays
BOOLEAN data type, in SQL 2nd
Boolean valued properties
Bootstrap class loader 2nd
Bootstrap registry service
Bound properties

Boundary matchers
Bounding box, for an arc
Breadth-first enumeration 2nd
Breadth-first search algorithm
Breadth-first traversal
Browsers 2nd
Buffer(s) 2nd 3rd
Buffer class 2nd 3rd
Buffer data structure
Buffer objects
Buffered image, obtaining
Buffered stream, creating 2nd
BufferedImage class 2nd 3rd
BufferedImage object
BufferedImageOp class
BufferedImageOp interface 2nd
BufferedInputStream
BufferedOutputStream
BufferedReader class
Builder environments 2nd
Builder tools
buildSource method 2nd
Bundle classes
Business logic 2nd
Butt cap
ButtonFrame class
ButtonFrame.java 2nd
bypass methods
Byte(s)
Byte array, saving data into
Byte sequences, decoding
byte values, converting
BYTE_ARRAY data source
ByteArrayJavaClass object
ByteArrayJavaClass.java
ByteBuffer class 2nd 3rd
Bytecode engineering
Bytecode Engineering Library [See BCEL (Bytecode Engineering Library).]
Bytecode level 2nd
Bytecode verification
Bytecodes, modifying 2nd
ByteLookupTable subclass 2nd
Byte-oriented streams, Unicode and

Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Z]

C code
    accessing Java strings from
    calling any Java method from
    making calls to Java code
    for the native `fprint` method 2nd
C functions
    calling from Java programs
    calling Java methods
    naming
C header file, producing
C strings
C types, compared to Java types
C#
C++
    accessing JNI functions in
    implementing native methods
    inheritance hierarchy of array types
    making calls to Java code
CA (certificate authority) 2nd
CA script, running
Cached row sets
`CachedRowSet` class
`CachedRowSet` interface
`CachedRowSet` object
Caching, prepared statements
Caesar cipher
`Caesar.java` 2nd
Calendar display, locating dates in 2nd
`CalendarBean` 2nd
`call` escape
Call functions, versions of
`call` method, invoking
Call methods, accessing
Call stack, during permission checking
Call transitional event
`Callback` interface
`CallbackHandler` class
`CallNonvirtual`*Xxx*`Method` functions
`CallStaticObjectMethod` function
`CallStatic`*Xxx*`Method` function
Cancel button, in a progress monitor dialog box
`cancelCellEditing` method 2nd
Cancellation requests
`cancelRowUpdates` method
`canImport` method
`canInsertImage` method
`CANON_EQ` flag
Canonical path name
`CANONICAL_DECOMPOSITION` collator value
Capacity, of a buffer
Cascading windows
Case sensitivity, of XML
`CASE_INSENSITIVE` flag
Catalog, describing schemas

Category, of an attribute
Category character class
CDATA attribute value
CDATA sections, in XML documents
Cell(s) 2nd
Cell color
Cell editing 2nd
Cell renderers 2nd
Cell selection

CellEditor class
Certificate authority [See CA (certificate authority).]
Certificates
  importing into keystores
  set of
  signing
  in the X.509 format

CertificateSigner class
Chain of trust, assuming
ChangeListener
ChangeTrackingTest.java
changeUpdate method, of DocumentListener
Channel(s)
  avoiding multiple on the same locked file
  from a file
  read and write methods of
  turning into an output stream
Channels class
char arrays, converting strings to
CHAR_ARRAY data source
Character(s) 2nd
Character classes
  predefined 2nd 3rd
  predefined names
  in regular expressions
Character data, getting
CHARACTER data type, in SQL 2nd
Character encoding 2nd 3rd
Character outlines
Character references, in XML documents
Character sets
CharacterData class
CharBuffer class 2nd 3rd 4th
CharSequence interface 2nd
Charset class 2nd
Chart bean 2nd
ChartBean2Customizer.java
ChartBeanBeanInfo class
ChartBeanBeanInfo.java
Checkbox editor, installed by JTable
checkError method 2nd
checkExit method
checkPermission method 2nd
checkRandomInsertions method
Child elements
  inheriting namespace of parent
  in an XML document
Child nodes 2nd 3rd
Children
  adding to the root node
  analyzing in XML documents
Chinese characters and messages
Choice formats

CHORD arc type
CIE (Commission Internationale de l'Eclairage)
Cipher class 2nd
Cipher object, initializing
Cipher streams, in the JCE library
Circular dependencies, in annotations
Class(es)
    loading different with the same name
    with the same class and package name
    separating from different web pages
    undocumented
Class browser, example
Class class 2nd
Class descriptors 2nd
Class files
    controlling the placement of
    names of
    producing unsafe
    program loading encrypted
Class fingerprint
Class identifier
Class IDs
Class loader hierarchy
Class loaders
    described
    in every Java program
    as namespaces
    simple 2nd
    specifying
    writing for specialized purposes
class object, obtaining
CLASS retention policy, for annotations
Class tree program
ClassLoader class 2nd
Classloader inversion
ClassLoaderTest.java
CLASSPATH environment variable
ClassTree.java
clear method, calling
CLEAR rule 2nd
Client(s)
    configuration of
    configuring Java security
    connecting to a server port
    enumerating all registered RMI objects
    getting a stub to access a remote object
    implementing for a web service
    installing proxy objects on
    invoking a method on another machine
    loading additional classes at runtime
    role in distributed programming
    serving multiple
Client classes, generating
Client program, running for a web service
Client/server application, traditional
Client-side artifact classes
Clip area, restoring
clip method 2nd
Clipboard [See also Local clipboard; System clipboard.]
    reading a string from
    transferring images into
Clipboard class 2nd 3rd 4th
Clipboard services

ClipboardOwner interface 2nd
Clipping, shapes
Clipping area 2nd
Clipping region, setting
Clipping shape
Clob class
CLOB data type, in SQL 2nd
Clob object, retrieving
CLOBs (character large objects)

Clone method, remote references not having
Cloneable interface
CloneNotSupportedException
Cloning, using serialization for
Close box, adding
close method
    calling immediately

    for streams 2nd
Close property, user vetoing
Closeable interface 2nd
Closed nonleaf icon
closed property, of the JInternalFrame class
closeEntry method
closePath method
Closure type, for an arc
Cntrl character class
Code [See also Java code.]
    automatic generation of
    techniques for processing
Code base 2nd 3rd
Code generator tools, annotations used by
Code Page 437, for file names
Code signing 2nd
Code sources
codebase entry
Codebase URL, ending with a slash (/)
The Codebreakers (Kahn)
CodeSource class
Collation, localizing
Collation key object
Collation order
CollationKey class
CollationTest.java
Collator, default
Collator class
Collator object
Collators, cutting the strength of
Color, dragging into a text field
Color chooser
Color class 2nd
Color constructor
Color model
Color rendering
Color space conversions
Color type, cells of
Color values 2nd
ColorConvertOp operation
Colored rectangles, expressing a set of
ColorModel class
Color-model-specific description
Column classes, in Swing
Column names
    changing

    prefixing with table names
    for a table
Columns
    accessing
    in a database
    determining which are selected
    hiding and displaying in tables
    rearranging
    resizing
    selecting
    selection and filtering of
    setting in a text field
    specifying comparators for
Combo box
Combo box editor
Command-line arguments
Commands
    in comments
    terminating in SQL
Comma-separated data file, script sending back
Comments, in XML documents
Commit behavior, with `setFocusLostBehavior` method
`commit` method, calling for transactions
Commit or revert behavior 2nd
Commited text string
Committed transactions 2nd
Common Dialog control, in Visual Basic
Common Gateway Interface (CGI) scripts
Common Name (CN) component
Common Object Request Broker Architecture (CORBA)
Comparator, installing for each column
`Comparator` interface
`compareTo` method
Compatibility characters, decomposing
`Compilable` interface 2nd
Compilation, annotations for
Compilation tasks
`CompilationTask` class 2nd 3rd
`CompilationTask` objects 2nd
`CompiledScript` class
Compiler [See also Microsoft compiler.]
Compiler API
`CompilerTest.java`
Compiling, scripts
Completion percentage, progress bar computing
Complex area, constructing
Complex types 2nd
`Component` class 2nd
Component organizers
Composing, transformations 2nd
`Composite` interface
`CompositeTest.java`
Composition
Composition rules
    designing 2nd
    program exploring
    selecting
    setting
Compressed format, storing files in
Compression method, setting
Computer Graphics: Principles and Practice, Second Edition in C (Foley/Dam/Feiner) 2nd 3rd
Concurrency setting, of a result set
Concurrency values, for result sets

Concurrent connections
Confidential information, transferring
Configuration file
connect method
Connection class
  API notes 2nd 3rd 4th 5th 6th
  close method of
Connection management
Connection object

Connection pool
Connections
  managing
  pooling
  starting new threads
Constrained properties
Construction parameters, packaging

Constructive area geometry operations
Constructor(s) [See also *specific constructors*.]
  constructing trees out of a collection of elements
  native methods invoking
  specifying for the InputStreamReader
@ConstructorProperties annotation
Content handlers 2nd
ContentHandler class
ContentHandler interface
Context, closing
Context class
Context class loader
Context interface
Contexts, beans usable in a variety of
CONTIGUOUS_TREE_SELECTION
Control points 2nd
Controls, in Visual Basic
convertColumnIndexToModel method
convertRowIndexToModel method
Convolution, mathematical
Convolution operator
ConvolveOp object
ConvolveOp operation 2nd
Coordinate system, translating
Coordinate transformations
Copies attribute
Copies class
CORBA (Common Object Request Broker Architecture)
Core Java Foundation Classes (Topley) 2nd
Core Swing: Advanced Programming (Topley) 2nd
COREJAVA database
Corner area, for a RoundRectangle2D
Country (C) component
Country code, ISO codes for
CRC32 checksum 2nd 3rd 4th
CRC32 class
CREATE TABLE statement, in SQL
createBlob method
createClob method
createElement method
CreateJavaVM
createNewFile method
createSubcontext method
createTextNode method
createTransferable method
Cross-platform print dialog box

Cryptographic algorithms
Cryptography and Network Security (Stallings)
CTRL key, dragging and
CTRL+V keystroke
Cubic curves 2nd
`CubicCurve2D.Double` class
Currencies, formatting
`Currency` class 2nd
Currency identifiers
Cursor, moving by a number of rows
`curveTo` method
Custom cell editor
Custom editor dialog box
Custom editors
Custom formatters
Custom permissions
Custom tree models
`Customizer` class, writing
`Customizer` interface 2nd
Customizers
Cut and paste
Cyclic gradient paint
`cyclic` parameter, of `GradientPaint`
Cygwin programming environment 2nd

Index

DamageReport objects

DamageReporter.java
DamageReport.java
Dash pattern
Dashed lines, program specifying
Data
    avoiding duplication of
    changing in a database 2nd

    digital fingerprint of a block of
    encrypting to a file
    posting to a script
    reading in text format
    sending back to web servers and programs
Data Definition Language (DDL) statements
Data Encryption Standard (DES)
Data field descriptors
Data fields 2nd 3rd
Data file [See also File(s).]
Data sources
    defined
    for JDBC
    for print services
Data transfer
    API
    capabilities of the clipboard
    classes and interfaces for
    support in Swing
Data types
    Java
    for print services
    print services for
    in SQL
Database
    combining queries
    connecting to 2nd
    creating for experimental use
    driver reporting nonfatal conditions
    example for this book
    integrity
    populating
    programs
    starting
    URLs
    vendors
Database configuration
Database connections
    cost of establishing
    keeping in a queue
    opening in Java
Database server, starting and stopping
Database-independent protocol
DatabaseMetaData class
    API notes 2nd 3rd
    giving data about the database
    methods inquiring about the database

DatabaseMetaData method
DatabaseMetaData type
DataFlavor class 2nd
DataFormat class
Datagrams
DataIO helper class
DataInput interface
DataInputStream methods
DataInputStream subclass
DataOutput interface 2nd
DataOutputStream subclass
DataSource interface
DataTruncation class
Date(s)
- convenient way of entering 2nd
- display of
- incrementing or decrementing in a spinner

Date and time
- formatting
- literals, embedding

Date class
DATE data type, in SQL 2nd
Date editor, for a spinner
Date filter
Date format, as lenient
Date models, for spinners
DateEditor class
dateFilter method
DateFormat class 2nd
DateFormatTest.java
DDL (Data Definition Language) statement
Decapitalization
DECIMAL data type, in SQL 2nd
decode method
Decomposition mode
Decryption key
Default(s), not stored with an annotation
Default cell editor
Default collator
Default constructor, for a bean
Default mutable tree node
Default rendering actions
Default tree model
Default value, for integer input
DefaultCellEditor class
- API notes
- variations of

DefaultFormatter class 2nd 3rd
DefaultHandler class
DefaultListModel class
DefaultMutableTreeNode class 2nd 3rd 4th
DefaultPersistenceDelegate class
defaultReadObject method
DefaultRowSorter class
DefaultTableCellRenderer class
DefaultTableModel
DefaultTreeCellRenderer class 2nd 3rd
DefaultTreeModel class
- API notes 2nd
- automatic notification by
- constructing

example not using
defaultWriteObject method
defineClass method
Degree, of normalization
Delayed formatting, of complex data
DELETE query, in SQL
deleteRow method
Delimiters, separating instance fields
@Deprecated annotation 2nd

@Deprecated Javadoc tag
Depth-first enumeration
Depth-first traversal
depthFirstEnumeration method
DER (distinguished encoding rules)
Derby database 2nd 3rd 4th
derbyclient.jar file

DES algorithm
Design patterns
DeskTop, populating
Desktop applications, launching
Desktop class 2nd
Desktop pane
DesktopAppTest.java
DesktopManager class
Destination pixel
DestroyJavaVM function 2nd
destroySubcontext method
Device coordinates 2nd [See also Pixels.]
Diagnostic class
Diagnostic objects
DiagnosticCollector class
DiagnosticListener, installing
DialogCallbackHandler
DianosticCollector class
digest method
Digit character class
Digital Signature Algorithm keys [See DSA (Digital Signature Algorithm) keys.]
Digital signatures
described
verifying 2nd
DirContext class
Direct buffers
Directory 2nd
Directory context 2nd
Directory tree, in LDAP 2nd 3rd
DISCONTIGUOUS_TREE_SELECTION
Disk files, as random access
displayMessage method
Distinguished encoding rules (DER)
Distinguished name 2nd
Distributed collector
Distributed programming
Dithering
doAsPrivileged method
Doc attributes
Doc interface
DocAttribute interface 2nd
DocFlavor class
DocPrintJob class 2nd
DOCTYPE declaration, in a DTD
DOCTYPE node, including in output
Document(s), XML files called

Document class 2nd 3rd
Document filter 2nd
Document flavors, for print services
Document interface
Document listener, installing
Document object
Document Object Model parser [See DOM parser.]
Document structure
Document type definitions [See DTDs (Document Type Definitions).]
DocumentBuilder class 2nd 3rd
DocumentBuilder object
DocumentBuilderFactory class 2nd 3rd
@Documented meta-annotation
DocumentEvent class
DocumentFilter class 2nd
DocumentListener, attaching to a text field
DocumentListener class
DocumentListener methods
doFinal method, calling once
DOM (Document Oject Model) approach
DOM parser 2nd 3rd
DOM tree 2nd 3rd
DOMResult class 2nd
DOMSource class
DOMTreeModel class
DOMTreeTest.java
doPost method
DOTALL flag, in a pattern
DOUBLE data type, in SQL 2nd
Double underscores, in native method names
DRAFT constant
Drag and drop 2nd
Drag sources, configuring
Drag-and-drop user interface
Dragging, activating
draw method 2nd
draw operation
draw3DRect method
Drawing, shapes
Drawing operations, constraining
Driver class, registering
DriverManager 2nd
Drivers, types of JDBC
drivers property
Drop actions
Drop cursor shapes
Drop location, obtaining
Drop modes, supported by Swing components
Drop targets 2nd
DropLocation classes
DSA (Digital Signature Algorithm) keys 2nd
DST rules 2nd 3rd
DTDs (Document Type Definitions) 2nd 3rd
Dynamic class loading

Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Z]

Echo server, accessing

EchoServer.java
e-commerce web service
Edge detection
EDGE_NO_OP edge condition
EDGE_ZERO_FILL edge condition
Edit dialog box
Edited value, for a cell

Editor pane 2nd
EditorPaneTest.java
Editors, custom
EJBs (Enterprise JavaBeans) 2nd 3rd
Element(s)
    of annotations
    of attributes
    compared to attributes 2nd
    constructing for documents
    describing data
    legal attributes of
Element attributes
Element class 2nd
Element content
    rules for
    whitespace
Element declarations, for an annotation
ELEMENT rule, in a DTD
Ellipse2D class
Elliptical arc 2nd
E-mail 2nd
Employee records, storing 2nd
Employee.java
EmployeeTest.java
Encoder class
-encoding flag
-encoding option
Encoding process
Encoding schemes
Encryption
End cap styles 2nd
End points, of quadratic and cubic curves
End tags, in XML and HTML
End-of-line character
Engine [See Scripting, engine.]
English, retirement calculator in
ENTERED value, for getEventType
Enterprise JavaBeans (EJBs) 2nd 3rd
Entities, defined by DTDs
ENTITY attribute value
Entity references 2nd
Entity resolver, installing
EntityResolver interface 2nd
Entry class 2nd
EntryLogger.java
EntryLoggingAgent.java

enum construct
EnumCombo helper class 2nd
EnumCombo.java
Enumerated type
Enumeration, native methods supporting
Enumeration objects 2nd 3rd
Enumeration values, for attributes
EnumSyntax class
env pointer
EOFException object
Equals comparison, in SQL
equals method
  of the File class
  looking at the location of remote objects
  remote objects overriding
  of a set class
Error handler, installing
Error handling, in batch mode
ErrorHandler interface 2nd
Errors, handling in native methods
Escape hatch mechanism
Escapes
  in regular expressions
  in SQL
Euro symbol
evaluate method
Event firing 2nd
Event handlers 2nd 3rd
Event listeners, adding
EventHandler class
EventListenerList convenience class
EventObject
Events 2nd
ExceptionListener class
ExceptionOccurred method
exclusive flag, locking a file
Exclusive lock
exclusiveOr operation 2nd
ExecSQL.java
Executable applets, delivering
Executable programs, signing
execute method
execute statement
EXECUTE_FAILED value
executeQuery method
executeQuery object
executeUpdate method 2nd
exists method
exit method
EXITED value
exitInternal method
exportDone method
exportObject method
Expression class
Extensible Stylesheet Language Transformations [See XSLT (XSL Transformations).]
Extension class loader 2nd
extern "C", native methods as
Externalizable classes
Externalizable interface 2nd

Index

Factoring algorithms

Factory methods 2nd 3rd 4th
FeatureDescriptor class
Field(s)
    accessing from native methods
    marking as transient
    preventing from being serialized
    in a variable

Field identifier, cost of computing
Field IDs, compared to Field objects
fieldID, obtaining
File(s)
    counting lines in
    creating from a File object
    determining the total number of bytes in
    locking a portion of
    memory-mapped
    with multiple images
    reading numbers from
File class 2nd
File extensions, indexed property for
File formats
    for object serialization
    supported
File locking
File management
File names, specifying
File object 2nd 3rd
File objects, substituting
File operations, timing data for
File output stream
File permission targets
File pointer
File separator character
File suffixes 2nd
file URLs
FileChannel class 2nd 3rd
FileInputStream 2nd 3rd
FileInputStream class 2nd
FileLock class
fileName property
FileNameBean component 2nd
FileNameBean.java
FilenameFilter 2nd 3rd
FileOutputStream 2nd 3rd 4th
FilePermission permission
Filer interface
FileReadApplet.java
FileReader class
FileWriter class
FileWriter constructor
fill methods 2nd
Filling, shapes

Filter(s)
combining
image processing operations
implementing
nesting
predefined
for user input
Filter classes
FilteredRowSet interface
Filtering
images
rows
FilterInputStream class
FilterOutputStream class
fin object, reading
finally block
find method
FindClass function 2nd 3rd
findClass method
FindDirectories.java
Fingerprint 2nd 3rd
fireIndexedPropertyChange method
firePropertyChange method
fireVetoableChange method
Fixed cell size
Fixed-size record
Flag byte
FlavorListener 2nd
flip method
float coordinates
FLOAT data type, in SQL 2nd
Floating-point numbers, storing
flush method 2nd
Flushable interface 2nd
Flushing, the buffer
Focus, text field losing
Focus listener
Folder icons
Font(s), antialiasing 2nd
Font choices, displaying
Font dialog
Font name, showing its own font
Font render context
fontdialog.xml
Forest 2nd 3rd
Form data, posting
Form view, creating
Format class
format method, using the current locale
Format names
Format string, in a choice format
Formatter objects
Formatters
custom
supported by JFormattedTextField
FormatTest example program 2nd
FormatTest.java
Forms, filled out by users
forName method
Fortune cookie icon
Forward slash (/)
as a directory separator in Windows

    ending the codebase URL with
    as a file separator
    in a UNIX environment
ForwardingJavaFileManager class 2nd 3rd
fprint native method
Fractals
Fractional character dimensions
Frame(s)
    applications managing
    closing
    dragging across the desktop
    making visible
    setting to be resizable
    tiling
    with two nested split panes
Frame class 2nd
Frame icon
Frame state
Frame window
FROM clause, in SQL
FULL OUTER JOIN
Functions, built-in to SQL

Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Z]

Garbage collectors 2nd
Gasp table, of a font
Gawor, Jarek
GeneralPath class 2nd 3rd
GeneralPath object
@Generated annotation
German, retirement calculator in
Gesture, initiating a drag operation
get methods
    for beans
    in ByteBuffer
    calling
    for reading and writing
    of ResultSet
    of URI
GET response command
getAbsolutePath method
getAllByName method
getAllFrames method
getAnnotation method
getArray method
getAsText method
getAsText/setAsText methods
getAttribute method 2nd
getAttributes method 2nd
getAvailableLocales method 2nd
getBeanInfo method
getBlob method
GetBooleanArrayElements method
getBundle method
getByName method
getCanonicalPath method
getCategory method
getCellEditorValue method 2nd 3rd
getCellRenderer method
getChannel method
getCharacterStream method
getChild method 2nd
getChildNodes method
getClob method
getCollationKey method
getColorModel method
getColumn method
getColumnClass method
getColumnCount method 2nd
getColumnName method
getConcurrency method
getConnection method 2nd
getContent method
getCurrencyInstance method 2nd
getData method
getDataElements method

getDateInstance method
getDefault method
getDisplayName method
getDocumentElement method 2nd
getDrive method
getDropLocation method
getElementAt method
getEngineFactories method
getErrorCode

getErrorStream method
getEventType method
getFieldDescription method
GetFieldID function
getFields method
getFilePointer method 2nd
getFirstChild method

getFontRenderContext method
getHeaderField method
getHeaderFieldKey method 2nd
getHeaderFields method
getHeight method
getIcon method
getImageableHeight method
getImageableWidth method
getImageableX method
getImageableY method
getImageReadersByMIMEType method
getImageReadersBySuffix method
getIndexOfChild method
getInputStream method 2nd 3rd
getInstance factory method
getInstance method
  of AlphaComposite
  of Cipher
  of Currency
getIntegerInstance method
getJavaFileForOutput method
getJavaInitializationString method
getLastChild method
getLastPathComponent method
getLastSelectedPathComponent method
getLength method
getLocalHost method
getMaxStatements method
getMethodCallSyntax method
GetMethodID function 2nd
getModel method
getMoreResults method
getName method
getNewValue method
getNextEntry method
getNextException method
getNextSibling method
getNextValue method 2nd
getNodeName method
getNodeValue method
getNumberInstance method
getNumImages method
getNumThumbnails method

getObject method
GetObjectArrayElement method
GetObjectClass function 2nd
getOrientation method
getOutline method
getOutputStream method
getPageCount method
getParameter method
getPathToRoot method
getPercentInstance method
getPixel method
getPixels method
getPointCount method
getPreviousValue method 2nd
getPrintService method
getProperty method
getPropertyDescriptors method 2nd
getRaster method
getReaderFileSuffixes method
getResource method
getReturnAuthorization method
getRGB method
getRoot method
getRowCount method 2nd
getSecurityManager method
getSelectedColumns method
getSelectedIndex method
getSelectedRows method
getSelectedValue convenience method
getSelectedValues method
getSelectionModel
getSelectionPath method 2nd
getSelectionPaths method
get/set naming pattern, exception to
getSourceActions method
getSQLState method
getSQLStateType method
GetStaticFieldID function
GetStaticMethodID function
GetStringRegion method
GetStringUTFChars function 2nd
GetStringUTFLength method
GetStringUTFRegion method
GetSuperclass method
getSystemClipboard method
getTableCellEditorComponent method
getTableCellRendererComponent method
getTables method
getTagName method
getTags method
getTask method
getTime method
getTransferable method 2nd
getTreeCellRendererComponent method
getType method
getUpdateCount method
getURL method
getValue method 2nd
    defining for a spinner

of JSpinner
returning the integer value of an attribute
getValueAt method 2nd
getWidth method
getWriteFormatNames method
getWriterFormats helper method
Get*Xxx*ArrayElements function
Get*Xxx*ArrayRegion method
GIF files, writing

GIF image
Global scope
Gnu C compiler
Gödel's theorem
GradientPaint class 2nd
GradientPaint object
grant clause 2nd

grant entries, in a policy file
Graph character class
Graphic Java 2: Mastering the JFC, Volume II: Swing (Geary) 2nd
Graphics, printing
Graphics class 2nd 3rd 4th
Graphics classes, using float coordinates
Graphics object, clipped
Graphics2D class 2nd 3rd 4th 5th
Grid bag
Grid bag pane
Grid width
gridbag.dtd 2nd
GridBagLayout
GridBagPane class
GridBagPane.java
GridBagTest.java
gridbag.xsd
Groovy engine 2nd 3rd
groupCount method
Grouping, in regular expressions
Groups
defining subexpressions
nested
GSS-API
GUI design tools
GUI events
GUI-based property editors

Index

Half-close
The Handbook of Applied Cryptography
Handles, for subtrees
`hashCode` method
Header(s)
    table rendering
    of an XML document
Header information, querying the server for
Header types, querying values
`HelloNative.java`
`HelloNativeTest.java`
Hex editor, modifying byte codes
Hidden commands, in comments
Hiding, table columns
Hierarchical databases 2nd
Hierarchical URIs
Hierarchy
    array types
    attribute sets
    attributes for printing
    for bundles
    class loader
    of countries, states, and cities
    for input and output streams 2nd
    permission classes
    property files
    reader and writer
    of text components and documents
`HIGH` constant
Hints [See Rendering, hints.]
Horizontal line style, tree with
`HORIZONTAL_SPLIT`, for a split pane
`HORIZONTAL_WRAP`, for a list box
Host names 2nd
Host variable, in a prepared query
Hot deployment
`HrefMatch.java`
HTML
    compared to XML 2nd
    displaying program help in
    displaying with `JEditorPane`
    form
    help system
    making XML compliant
    opening with snippets of Java code
    page 2nd 3rd
    rule for attribute usage
    table 2nd
    transforming XML files into
`HTMLDocument` class
HTTP
HTTP request, response header fields from
/`https:` URLs, accessing
`HttpURLConnection` class

Human-readable name, of a data flavor
Hyperlink(s) 2nd
`HyperlinkEvent` class
`HyperlinkListener` class
`HyperlinkListener` interface
`hyperlinkUpdate` method
Hypertext references, locating all

UNREGISTERED VERSION OF CHM TO PDF CONVERTER PRO BY THETA-SOFTWARE

Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Z]

IANA Character Set Registry
IBM Tivoli Directory Server
ICC profiles 2nd
Icon(s) 2nd
Icon images, loading
Icon objects, list filled with
Icon state, of a frame
ID construct
Identical character differences
Identity transformation
IDL (Interface Definition Language)
IDREF attribute value
IDREFS attribute value
ifModifiedSince property
IIOImage class
IIOImage object
IIOP (Inter-ORB Protocol)
IIOServiceProvider class
Illegal input, provided by users
IllegalAccessException
IllegalArgumentException 2nd 3rd 4th
IllegalStateException
Image(s)
    blurring
    building
    creating
    filtering
    readers and writers for
    rotating about the center
    storing
    superimposing on existing
    transferring into the clipboard
Image class
Image control, in Visual Basic
Image file types
Image format
Image icon
Image manipulation
Image processing operations
Image size, getting
Image types, menu of all supported
Imageable area
ImageInputStream
ImageIO class 2nd
ImageIOTest.java
ImageList drag-and-drop application
ImageListDragDrop.java
ImageProcessingTest.java
ImageReader class
ImageReaderWriterSpi class
ImageTransferTest.java
ImageViewer bean 2nd
ImageViewerBean component

ImageViewerBean.java
ImageWriter class 2nd
IMAP (Internet Message Access Protocol)
implies method 2nd
importData method 2nd
In*Block* character class
In*Category* character class
include method
Incremental rendering, of images

Indented output
Indeterminate progress bar
Indeterminate property
Indexed properties
IndexedPropertyChangeEvent class
IndexedPropertyDescriptor class
IndexOutOfBoundsException

Inequality testing, in SQL
InetAddress class 2nd
InetAddress object 2nd
InetAddressTest.java
InetSocketAddress class
Infinite tree
Information
    locating in an XML document
    using URLConnection to retrieve
Inheritance trees 2nd
@Inherited meta-annotation
InitialContext class
InitialDirContext class
Initialization code, for shared libraries
initialize method
Input, splitting into an array
Input fields, formatted
Input reader, reading keystrokes
Input stream(s)
    as an input source
    keeping open
    monitoring the progress of
Input stream filter
Input validation mask
INPUT_STREAM data source
InputSource class
InputStream class 2nd
InputStream object
InputStreamReader class
InputVerifier class
Insert row
INSERT statement, in SQL
Insert string command
insertNodeInto method
insertRow method 2nd
insertString method
insertTab method
insertUpdate method
Inside Java 2 Platform Security: Architecture, API Design, and Implementation (Gong/Ellison/Dageforde)
Instance fields 2nd 3rd
Instance methods, calling from native code
instanceof operator
Instrumentation API, installing a bytecode transformer
Integer(s), methods of storing
INTEGER (INT) data type, in SQL 2nd

`Integer` constructor
Integer formatter
`Integer` identifier type
Integer input, text field for
Interactive scripting tool
`@interface` declaration
Interface Definition Language (IDL)
Interface description
Internal frames
    cascading on the desktop
    dialogs in
    displaying multiple
    setting the size of
    tiled
`internalFrameClosing` method
`InternalFrameListener`
`InternalFrameTest.java`
International Color Consortium (ICC) 2nd
International currency character, Euro symbol replacing
International Organization for Standardization [See ISO ; *specific standards*.]
Internationalization
Internet, delivery over the public
Internet addresses
Internet hosts, services provided by
Internet Message Access Protocol (IMAP)
Internet Printing Protocol 1.1 (RFC 2911)
Inter-ORB Protocol [See IIOP (Inter-ORB Protocol).]
Interpolation strategies
Interruptible sockets
`intersect` operation 2nd
intranet, delivery in
`Introspector` class
`InverseEditor.java`
`InverseEditorPanel.java`
Investment, growth of
`InvestmentTable.java`
`Invocable` interface 2nd
Invocation API
`InvocationTest.c`
`invokeFunction` method
`IOException` 2nd
IP addresses, customizing 4-byte
IPv6 Internet addresses, supporting
`isAdjusting` method
`IsAssignableFrom` method
`isCanceled` method
`isCellEditable` method 2nd
`isDesktopSupported` method
`isDirectory` method
`isEditValid` method 2nd
`isFile` method
`isIcon` method
`isIndeterminate` method
`isLeaf` method 2nd
ISO 216 paper sizes
ISO 639-1
ISO 3166-1
ISO 4217
ISO 8859-1
ISO-8859-1 2nd
ISO-8859-15
iSQL-Viewer

is/set naming pattern
isShared method
isStringPainted method
isSupported method 2nd
item method
Item.java
Items, selecting in a list box
ItemSearch operation
ItemSearchRequest parameter type
Iterable objects
Iterator interface
iterator method

UNREGISTERED VERSION OF CHM TO PDF CONVERTER PRO BY THETA-SOFTWARE

Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Z]

JAAS
JAAS login modules
`JAASTest.java`
JAR file(s)
  for the database driver
  packaging beans in
  registering the driver class
  signing
  signing and verifying
  as ZIP file with a manifest
JAR file resources
`jarsigner` tool 2nd
Java 2D API 2nd
Java API, for SQL access
Java applications [See also Application(s).]
  data copying between two instances of
  splash screens difficult for
  with three internal frames 2nd
  writing internationalized
Java code [See also Code.]
  dynamic generation
  iterating through multiple result sets
Java compiler, tools invoking
Java data types
Java Database Connectivity
Java deployment directory
Java exception, native C++ method in
Java method name, for a C function
Java methods, calling from native code
Java Native Interface [See JNI (Java Native Interface).]
Java objects, transferring via the system clipboard
Java platform security
Java Plug-in tool
Java program
  copying a native program to
  copying to a native program
Java RMI technology [See RMI (Remote Method Invocation).]
Java servlets
Java `String` objects, converting
Java types, compared to C types
Java virtual machine [See Virtual machine(s).]
The Java Virtual Machine Specification (Lindholm/Yellin)
`java.awt.datatransfer` package
`java.awt.Desktop` class
`java.awt.dnd` package
`java.awt.geom` package
JavaBeans 2nd [See also Beans.]
JavaBeans persistence
  for arbitrary data
  complete example
`java.beans.Beans` class
`JavaCompiler` class
JavaDB [See Derby database.]
Javadoc comments

JavaFileManager
JavaFileObject interface
JavaFileObject subclass
javah utility
JavaHelp
javaLowerCase character class
JavaMail API
javaMirrored character class
java.nio package

  making memory mapping simple
  new I/O in
  unifying characterset conversion
java.policy files
JavaScript—The Definitive Guide (Flanagan)
java.security configuration file
JavaServer Faces (JSF) 2nd
UNREGISTERED VERSION OF CHM TO PDF CONVERTER PRO BY THETA-SOFTWARE
JavaServer Pages (JSP)
javaUpperCase character class
javaWhitespace character class
javax.imageio package
javax.sql.rowset package
JAX-WS technology 2nd
jclass type, in C
JComponent, attaching a verifier to
JComponent class 2nd 3rd 4th 5th
JDBC
  application deploying
  configuration
  design of
  driver types
  drivers currently available
  requests
  syntax describing data sources
  tracing, enabling
  typical uses of
  ultimate goal of
  version numbers
JDBC 4
JDBC API
JDBC driver 2nd 3rd
JDBC Driver API
JDBC/ODBC bridge
JDBC-related problems, debugging
JdbcRowSet interface
JDesktopPane 2nd 3rd
JDialog class
JEditorPane class
  API notes
  displaying HTML with
  in edit mode by default
  extending JTextComponent
  showing and editing styled text
JFormattedTextField class 2nd
JFrame class
JFrame object
JInternalFrame class 2nd 3rd 4th
JInternalFrame windows, constructing
JList class
  API notes 2nd 3rd 4th
  calling get methods of
  configuring for writing custom renderers
  responsible for visual appearance of data

JList component
JList constructors
JList object
JNDI service
JNI (Java Native Interface) 2nd
JNI API, finding
JNI debugging mode
JNI functions 2nd 3rd
JNI_CreateJavaVM
JNI_OnLoad method
JobAttributes class, as obsolete
Join style, for thick strokes
Joining, tables 2nd
JoinRowSet interface
Joint styles
JPEG files
JProgressBar 2nd
JSP engine
JSpinner class
JSpinner component 2nd
JSplitPane class 2nd
jstring type 2nd
JTabbedPane class
JTabbedPane object
JTable class
    API notes 2nd 3rd
    picking a renderer
JTable component
JTextPane subclass
JTree, constructing 2nd
JTree class
    API notes 2nd 3rd
    calling methods to find tree nodes
JTree constructor
JUnit 4 testing tool
jvm pointer
JXplorer

Index

Kerberos protocol

Kernel, of a convolution operation
Kernel object 2nd
Keyboard, reading information from
KeyGenerator class
Keys
    distributing
    generating 2nd

    native methods enumerating 2nd
    retrieving autogenerated
Keystore(s) 2nd
Keystore password
Keystrokes
    monitoring
    reading from the console
    trying to filter
keytool

Index

Label
Language design features, of Java
Language locales
Large objects (LOBs) 2nd
A Layman's Guide to a Subset of ASN.1, BER, and DER (Kaliski)
Layout algorithm
Layout orientation, for a list box
`layoutPages` method
LCD values
`LD_LIBRARY.PATH`
LDAP (Lightweight Directory Access Protocol) 2nd
LDAP Browser
LDAP directory
    accessing
    keeping all data in a tree structure
    modifying
LDAP server
LDAP user, configuring 2nd
`LDAPTest.java`
LDIF data
LDIF file
Least common denominator approach
Leaves, of a tree 2nd 3rd 4th
Legacy classes
Legacy code, containing an enumerated type
Legacy data, converting into XML
Legion of Bouncy Castle provider
`length` method 2nd
Lenient date format
`lenient` flag
Levels of security
Lightweight Directory Access Protocol [See LDAP (Lightweight Directory Access Protocol).]
Lightweight Directory Interchange Format data [See LDIF data.]
`LIKE` operator, in SQL
Limit, of a buffer
Line segments, testing the miter limit
Lines
    counting in a file
    terminating in e-mail
`lineTo` method
Link action
Link to the file, placing
Linux 2nd
List(s)
    very long
List box(es)
    adding or removing items in
    filled with strings program
    populating with planets
    with rendered cells
    scrolling
    of strings
List cell renderers 2nd
List components, reacting to double clicks
List display

list method 2nd
List models
List selection listener
List values
List<String> interface
ListCellRenderer 2nd
ListDataListener
Listener interface, for events
Listener management methods
Listeners 2nd
Listening
    to hyperlinks
    to tree events
listFiles method
ListModel class
ListModel interface
ListRenderingTest.java
ListResourceBundle class
ListSelectionEvent method
ListSelectionListener class
ListSelectionModel class
ListTest.java
LITTLE_ENDIAN constant
Little-endian method
Load time
loadClass method
loadImage convenience method
loadLibrary method
LOBs (large objects) 2nd
Local clipboard
Local encoding schemes
Local host
Local language
    ISO codes for
    translating to
Local name, in the DOM parser
Local parameter and result objects
Local variables, annotations for
Locale(s)
    defined
    described
    formatting numbers for
    getting a list of currently supported
    no connection with character encodings
    program for selecting
    for the retirement calculator
Locale class 2nd
Locale objects 2nd
Locale-dependent utility classes
Location (L) component
lock method 2nd
Logging, RMI activity
Logging instructions
LoggingPermission permission
Login(s)
    management of
    separating from action code
Login code
    basic outline of
    separating from business logic
Login information, storing
Login modules 2nd 3rd

Login policy
LoginContext class
LoginModule class
LONG NVARCHAR data type, in SQL
LONG VARCHAR data type, in SQL
LongListTest.java
Long-term storage, JavaBeans persistence suitable for
Lookup table
LookupOp operation 2nd
lookupPrintServices method
LookupTable class
lostOwnership method
Lower character class
Lower limit, in a choice format

Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Z]

Macintosh
clipboard implementation of
executable program
Magic number, beginning every file
Mail header, sending
Mail messages [See also E-mail.]
sending
using sockets to send plain text

MailTest.java
main method, executing
makehtml.xsl
makeprop.xsl
makeShape methods
makeVisible method
Mandelbrot set, drawing
Mangled signatures
Mangling, rules for
Manifest entry, in JAR files
Manifest file
Map interface
map method
MapClassLoader.java
MappedByteBuffer
Mapping modes
Mark, of a buffer
mark method, of InputStream
Marker annotation
MarshalledObject class 2nd
MaskFormatter 2nd
Mastering Regular Expressions (Friedl)
match attribute, in XSLT
Matcher class
Matcher object 2nd
matches method
Matching, in SQL
Matrices 2nd
Matrix transformations
Maximum state, of a frame
Maximum value, for a progress bar
maxoccurs attribute, in XML Schema
MD5 algorithm
MDI (multiple document interface)
Memory mapping
Message digests
Message formatting
Message signing
Message strings, defining in an external location
MessageDigest class 2nd
MessageDigestTest.java
MessageFormat class 2nd
Messages, varying
Meta-annotations 2nd 3rd
Metadata
Metal look and feel

    frame icon displayed
    grabber areas of internal frames
    selected frame in
    selecting multiple items
    for a tree
Method(s)
    of an annotation interface
    executing Java
    of graphics classes
Method IDs
    compared to `Method` objects
    needed to call a method
    obtaining
Method names
    for beans
    for a C function
    capitalization pattern for
Method signatures 2nd
Method verification error
Metric system, adoption of
Microsoft Active Directory
Microsoft Active Server Pages (ASP)
Microsoft compiler
Microsoft Windows, clipboard implementation of
MIME (Multipurpose Internet Mail Extension) standard
MIME type name, of a data flavor
MIME types
    for print services
    reader or writer matching
    transferring an arbitrary Java object reference
    transferring local, serialized, and remote Java objects
`MimeUtility` class
Minimum value, for a progress bar
`minoccurs` attribute, in XML Schema
`MissingResourceException`
Miter join 2nd
Miter limit
Mixed contents
    parsing
    in the XML specification
`mkdir` method
Mnemonics, for tab labels
Model, obtaining a reference to
`model` object
Modernist painting 2nd 3rd
Modifier, annotation used like
`modifyAttributes` method
Mouse events, trapping
Move action, changing to a copy action 2nd
`moveColumn` method
`moveToCurrentRow`
`moveToInsertRow` method
Moving, a column in a table
Multicast lookup, of remote objects
`MULTILINE` flag
Multipage printout
Multiple document interface (MDI)
Multiple images
    program displaying
    reading and writing files with
    writing a file with
Multiple-page printing
multithreaded server

`MULTITHREADED` value, for scripts
`MutableTreeNode` class
`MutableTreeNode` interface

UNREGISTERED VERSION OF CHM TO PDF CONVERTER PRO BY THETA-SOFTWARE

Index

NameCallback class
NameClassPair helper class 2nd
NamedNodeMap class
NamedNodeMap object
Namespace(s)
    turning on support for
    using
    using class loaders as
Namespace mechanism, in XML
Namespace processing 2nd
Namespace URI, in the DOM parser
Namespace URL
Name/value pairs, in a property file
Naming class
Naming convention, for resource bundles
Naming pattern, for properties
NamingEnumeration class
NamingEnumeration<T> class
NanoHTTPD web server 2nd 3rd
    starting
National character string (NCHAR)
Native C code, compiling
Native character encoding, changing
Native code 2nd
native keyword
Native methods
    calling Java methods
    enumerating keys 2nd
    example
    handling error conditions
    implementing registry access functions as
    implementing with C++ 2nd
    overloading
    throwing exceptions 2nd
Native print dialog box
Native program
    copying a Java program to
    copying to a Java program
Native storage, for XML data
native2ascii utility
NCHAR data type, in SQL
NCLOB data type, in SQL
Negative byte values
Nested groups
Nesting filters
NetBeans integrated development environment
NetBeans version 6, importing beans into
NetPermission permission
Network address, for a remote object
Network connections, to remote locations
Network password dialog box
Network programming, debugging tool
Network sniffer
New I/O

New Project dialog box, in NetBeans 6
NewByteArray
newDocument method
NewGlobalRef
Newline character, displaying
NewObject function
newOutputStream method
NewStringUTF function 2nd
    calling to create a new string

    constructing a new jstring
New*Xxx*Array function
next method
nextElement method 2nd
nextPage method
NIOTest.java
NMTOKEN attribute value

NMTOKENS attribute value
NO_DECOMPOSITION collator value
Node(s)
    changing the appearance of
    displaying as leaves
    generating on demand
    identifying in a tree
    rendering
    in a tree 2nd
Node class 2nd 3rd
Node enumeration
Node interface, with subinterfaces
Node label, formatting
Node renderer
Node set, converting to a string
nodeChanged method
NodeList class
NodeList collection type
Non-ASCII characters, changing to Unicode
Non-deterministic parsing
Nonremote objects 2nd
Non-XML legacy data, converting into XML
NORMAL constant
Normalization forms
Normalization process
Normalized attribute value
Normalized color values
Normalizer class 2nd
NoSuchAlgorithmException
NoSuchElementException
NOT NULL constraint, in SQL
NotBoundException
notFilter method
Novell eDirectory
-noverify option
n-tier models
NULL, in SQL
Null references, storing
NullPointerException
Number filter
Number formats
Number formatters
Number models, for spinners
Number superclass
NumberFormat class 2nd

NumberFormat type
NumberFormatException
NumberFormatTest.java
Numbers
formatting
printf formatting
reading from a file
writing to a buffer
NUMERIC data type, in SQL 2nd
NVARCHAR data type, in SQL

Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Z]

Object(s)
allowing arbitrary inside cells
reading back in
saving a network of 2nd
saving in object serialization format
saving in text format
serial numbers for
shared by several objects
as the solution to all problems
storing in object serialization format
transferring via the clipboard
transmitting between client and server
writing and reading
writing to a stream and reading back
Object array, accessing elements in
Object classes, in LDAP
Object data fields, accessing
Object data, saving
Object files, evolution of classes
Object inspection tree
Object references, transferring 2nd
Object serialization
associating serial numbers
compared to JavaBeans persistence
file format
modifying the default mechanism
Object stream 2nd
Object values
ObjectInputStream 2nd
ObjectInspectorTest.java
ObjectOutputStream 2nd
ObjectRefTest program
ObjectStreamConstants
ObjectStreamTest.java
ODBC 2nd
One-touch expand icons
Opaque absolute URI
OPEN arc type
openConnection method
Opened nonleaf icon
OpenLDAP 2nd
OpenSSL software package
openStream method
Operating systems, character encoding
optional module
Ordering, of permissions
orFilter method
Organization (O) component
Organizational Unit (OU) component
Orientation, for a progress bar
Original PC encoding, for file names
Outer join
Outline dragging
Outline shape

Output stream 2nd 3rd 4th
OutputStream class 2nd
OutputStreamWriter class
OverlappingFileLockException
Overloading, native methods
@Override annotation
Overtype mode, mask formatter in
Overwrite mode, DefaultFormatter in

Index

Packages
    annotations for
    using to avoid name clashes
Packets, sending
Padding scheme
Page, multiple calls for
Page format measurements
Page orientation
Page setup dialog box 2nd 3rd 4th
Page size
Pageable interface
PageAttributes class, as obsolete
pageDialog method
PageFormat class
PageFormat parameter
Paint 2nd
Paint interface
paint method
paintComponent method 2nd 3rd
paintValue method
Paper margins
Paper sizes 2nd
Parameter marshalling
Parameters
    attaching the end of a URL
    parsing by serializing
Parent, of every node 2nd
Parent nodes
Parent/child relationships
    of class loaders
    establishing between tree nodes
parse method 2nd 3rd
Parse tree
ParseException 2nd 3rd 4th
Parsers 2nd 3rd
Parsing
    experimenting with
    by URIs
    XML documents
PasswordCallback class
Password-protected file by FTP
Password-protected web page
Path(s)
    finding from an ancestor to a given node
    of objects
    program creating sample
Path names, resolving
path parameter
Path2D class
Path2D.Float class
pathFromAncestorEnumeration method
Pattern class
Pattern object
Patterns 2nd

UNREGISTERED VERSION OF CHM TO PDF CONVERTER PRO BY THETA-SOFTWARE

#PCDATA 2nd
PCDATA abbreviation
PEM (Privacy Enhanced Mail) format
Periods, replacing with underscores
Permission classes 2nd
Permission files
Permissions
    attaching a set of
    custom
    defined
    describing in the policy file
    implying other permissions
    listing of
    restricting to certain users
    structure of
PermissionText.java
Permutations, algorithm determining
Persist behavior, with setFocusLostBehavior
Persistence delegate
PersistenceDelegate class
PersistenceDelegatTest.java
PersistentFrameTest.java
Phase, of the dash pattern
PIE arc type
Pixels [See also Device coordinates.]
    composing
    interpolating
    reading
    setting individual
    setting to a particular color
Placeholder character 2nd
Placeholder index
Placeholders
Plain text, turning an XML file into 2nd
PlainDocument class
Planet data, table with
PlanetTable.java
Platform integration
Platform-specific code, installing onto the client
Plugins [See also Java Plug-in tool.]
    packaged as JAR files
Point2D class
Point2D.Double class
Points, paper size measured in
Policy class 2nd
Policy files
    adding role-based permissions into
    building to grant specific permissions
    creating 2nd
    locations for
    sample 2nd
    security
    supplying
Policy URLs, in the policy file
policytool
Polygon
Polygon2D class [See GeneralPath class.]
Pooling, connections
POP before SMTP rule
Populating, a database
Pop-up menu, for a tray icon
PopupMenu class
Port

Port ranges
Porter-Duff composition rules
Position, of a buffer 2nd
position function
POST data 2nd
POST response command 2nd
@PostConstruct annotation 2nd
PostgreSQL
    database
    drivers
Postorder traversal
postOrderTraversal method
PostScript files
PostTest.java
Predefined filters
@PreDestroy annotation
preOrderTraversal method
Prepared statements
PreparedStatement class
PreparedStatement object
Primary character differences
Primitive type values
Primitive types, arrays of
Principal class
Principal objects
Principals
Print character class
Print dialog box 2nd
Print job 2nd 3rd
print methods
    of the Printable object
    of the Printable sections
    of PrinterJob
    of PrintWriter 2nd
    for a table
Print preview
Print request attributes
Print service attributes
Print services
    compared to stream print services
    document flavors for
    finding
    printing an image file
Print writer
Printable interface 2nd 3rd
Printable.NO_SUCH_PAGE value
Printable.PAGE_EXISTS value
printDialog method
Printer graphics context
Printer settings
PrinterException
PrinterJob class 2nd 3rd
printf, formatting numbers
Printf1 class
Printf1.java
Printf1Test.java
Printf2.java
Printf2Test.java
Printf3Test.java
Printf4.java
println method

Printing
    attribute hierarchy
    attribute set hierarchy
    multiple-page
Printing attributes
    listing of
PrintJobAttribute interface 2nd
Printouts, generating
PrintPreviewDialog class
PrintQuality attribute
PrintRequestAttribute interface 2nd
PrintRequestAttributeSet interface
PrintService class 2nd
PrintService objects
PrintServiceAttribute interface
PrintServiceLookup class
PrintServiceTest.java
PrintStream class
PrintTest.java
PrintWriter class 2nd 3rd
Privacy Enhanced Mail (PEM) format
Private keys 2nd
PRIVATE mapping mode
PrivilegedAction interface 2nd
PrivilegedExceptionAction interface 2nd
processAnnotations method 2nd
Processing instructions, in XML documents
Processing tools, for annotations
Processor interface
Product class
Product.java
Program code, controlling the source of
Programs [See also Java program.]
    launching from the command line
    signing executable
    supporting cut and paste of data types
    switching the default locale of
Progress bars 2nd
Progress indicators
Progress monitor dialog box
Progress monitors 2nd
Progress value, setting
ProgressBarTest.java
ProgressMonitor 2nd 3rd
ProgressMonitorInputStream 2nd 3rd
ProgressMonitorInputStreamTest.java
ProgressMonitorTest.java
Properties
    array of descriptors for
    Boolean valued
    bound
    changing the setting of in the NetBeans environment
    constrained
    constructing objects from
    exposing in beans
    at a higher level than instance fields
    indexed
    in the NetBeans environment
    simple
    transient
Properties class
Properties window, in Visual Basic

`@Property` annotation
Property editors
    in builder tools
    GUI-based
    string-based
    supplying customizers
    writing
Property files
    describing program configuration

    flat hierarchy of
    specifying string resources
    for strings
    unique key requirement
Property inspectors
    displaying current property values in
    listing bean property names 2nd

    in Visual Basic
Property permission targets
Property setter statements
Property settings, vetoing
Property values, editing
`PropertyChange` event
`PropertyChangeEvent` class 2nd 3rd
`PropertyChangeEvent` object 2nd
`PropertyChangeListener` interface 2nd
`PropertyChangeSupport` class 2nd
`PropertyDescriptor` 2nd 3rd
`PropertyEditor` class
`PropertyEditor` interface
`PropertyEditorSupport` class 2nd
`Property.java`
`PropertyPermission` permission
`PropertyVetoException`
    API notes 2nd
    catching
    throwing 2nd 3rd 4th 5th
Protection domain
`ProtectionDomain` class
Prototype cell value
Proxies, communicating
Proxy classes, for annotation interfaces
Proxy objects 2nd
Public certificates, keystore for
Public class, permission class as
`PUBLIC` identifier 2nd
Public key
Public key algorithms
Public key ciphers
Public key cryptography
Public Key Cryptography Standard (PKCS) #5
Pull parser
`Punct` character class
Pure rule
Pushback input stream
`PushbackInputStream`
`put` methods 2nd
`putNextEntry` method 2nd

Index

QuadCurve2D.Double class
Quadratic curves
quadTo method
Qualified name, in the DOM parser
Quantifiers 2nd
Queries
    building manually
    constraining
    executing
    using SQL
Query by example (QBE) tools
Query results
Query statements
QueryDB application
QueryDB.java
Question-mark characters, in date output
Quotation marks, optional in HTML

Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Z]

"r"

for read access
read-only mode
raiseSalary method 2nd 3rd
Random access 2nd
Random input, from a hardware device
Random numbers
Random-access files

RandomAccessFile class 2nd 3rd
RandomFileTest.java
Randomness
Ranges of cells
Raster class
Raster images, constructing
Raster point
RasterImageTest.java
read method
of DataInput interface
of ImageIO
of InputStream 2nd
of the progress monitor stream
of Reader
of ZipInputStream
Read permission
READ_ONLY mapping mode
READ_WRITE mapping mode
Readable interface 2nd 3rd
ReadableByteChannel interface
Reader class
READER data source
readExternal method
readFixedString method
Reading, text input
readLine method
readObject method
of the Date class
of ObjectInputStream 2nd
as private
of a serializable class
readResolve method
Read/write property
REAL data type, in SQL 2nd
Records
computing size of fixed
reading
Rectangle2D class
Rectangle2D.Double class
RectangularShape superclass
Redundancy elimination
Reflection 2nd 3rd
ReflectPermission permission
regedit command, in the DOS shell
regexFilter method 2nd

RegexTest.java
register method
Registered objects, displaying names of
Registration mechanism
Registry
    accessing
    Java platform interface for accessing
    overview of
Registry access functions, implementing as native methods
Registry editor
Registry functions, program testing 2nd
Registry keys 2nd
Registry object references
Regular expressions
    in an element specification
    replacing all occurrences of
    rows having a string value matching
    syntax of 2nd
    uses for
    vertical bar character in
Relational database
Relational model, distributing data
Relative identifiers, handling
Relative URI
Relative URLs 2nd
Relativization, of a URI
Relax NG
ReleaseStringUTFChars function 2nd
Release*Xxx*ArrayElements function
Reliability, of remote method calls
reload method
remaining method
Remote interface
Remote method call(s) 2nd
Remote method invocation [See RMI (Remote Method Invocation).]
Remote methods
Remote objects
    activation of
    clone method
    comparing
    equals method
    garbage-collecting
    hashCode method
    interfaces for
    passing
    registering 2nd
    transferring
Remote references
    invoking methods on
    with multiple interfaces
    passing
    transferring objects as
Remote resource, connecting to
Remote Warehouse interface
RemoteException 2nd 3rd
removeColumn method
removeElement method
removeMode property
removeNodeFromParent method
removePropertyChangeListener method 2nd
removeTabAt method
removeUpdate method

removeVetoableChangeListener method
Rendered cells, in a list box
RenderHints class 2nd
Rendering
  actions
  hints 2nd
  list values
  nodes
  pipeline
  shapes

RenderingHints class
RenderQualityTest.java
Rental car, damage report for
replace method
replaceAll method
replaceFirst method
Representation class

Request headers
required module
requisite module
Rescale operator
RescaleOp operation 2nd
Rescaling operation
reset method
reshape method
Resizable state, of a frame
Resizing
  columns
  columns in a table
  rows in JTable
resolveEntity method
Resolving
  a class
  a relative URL
Resource(s)
  alternate mechanisms for storing
  annotations for managing
  bundle classes
  bundles
  data
  files
  hierarchy, for bundles
  injection
  kinds of
Resource annotation
@Resource annotation
Response header fields
Response page
Result interface
Result sets
  analyzing
  concurrency values
  enhancements to
  managing
  retrieving multiple
  scrollable and updatable
  type value
  updatable 2nd
Results, query returning multiple
ResultSet class 2nd 3rd 4th 5th
ResultSet type
ResultSetMetaData class 2nd

`@Retention` meta-annotation
Retention policies
`Retire.java`
Retirement calculator applet
`RetireResources_de.java`
`RetireResources_zh.java`
`RetireResources.java`
Return character, displaying
Reverting, an input string
RFC 2279
RFC 2368
RFC 2396
RFC 2781
RGB color model
Rhino engine 2nd 3rd 4th
Rhino interpreter
Rich text format (RTF)
`RIGHT OUTER JOIN`
Rivest, Ronald
RMI (Remote Method Invocation)
    activation daemon
    activity, logging
    applications, deploying
    communication between client and middle tier
    deploying applications using
    loggers, listing of
    method calls between distributed objects
    programming model
    protocol
    registry
    registry, starting
    URLs
`rmid` program 2nd
`rmiregistry` service
Role-based authentication
Roles, login module supporting
`rollback` method
Rolled back transactions 2nd
Root
    certificate
    element, of an XML document
    handle, tree with
    hiding altogether
    node 2nd 3rd 4th
`rotate` method 2nd
Rotation transformation
Round cap
Round join 2nd
Rounded rectangle
`RoundRectangle2D` class 2nd
`RoundRectangle2D.Double` class
Row(s)
    adding to the database
    in a database
    determining selected
    filtering
    inspecting individual
    resizing
    selecting 2nd
    selection and filtering of
    sorting 2nd
Row height, setting
Row position, of a node

Row sets
RowFilter class 2nd 3rd
ROWID data type, in SQL
ROWID values
RowSet class
RowSet interface
RSA algorithm 2nd
RSATest.java
RTF (rich text format)

Rules, in a DTD
run method
RUNTIME retention policy, for annotations
RuntimePermission permission
"rw"
  read/write access
  read/write mode

"rwd", read/write mode
"rws", read/write mode

Index

Sample values 2nd
Sandbox
SASL (Simple Authentication and Security Layer)
Save points
Savepoint class
SAX parser
SAX XML reader
SAXParseException class
SAXParser class
SAXParserFactory class
SAXSource 2nd
SAXTest.java
Scalable Vector Graphics (SVG) format
Scalar functions
scale method 2nd
Scaling operation
Scaling transformation 2nd
Scanner, constructing
Scanner class 2nd
Schema
Schema file
schemeSpecificPart, of a URI
Scopes, collection of
Script(s)
    compiling
    executing in multiple threads
    invoking
    redirecting
    for server-side programs
Script class, accessing
Script engines, invoking functions
ScriptContext class
ScriptContext interface
ScriptEngine class 2nd
ScriptEngineFactory class
ScriptEngineManager 2nd 3rd
Scripting
    API
    GUI events
    engine 2nd 3rd
    engine factories
    for the Java platform
    languages
    statements, variables bound by
ScriptTest.java
Scroll pane, scrolling
Scrollable result
    sets 2nd
Scrolling
    mode
scrollPathToVisible method
Secondary character differences
Secret key, generating
SecretKeyFactory 2nd

SecretKeySpec class
Secure Hash Algorithm [See SHA (Secure Hash Algorithm).]
Secure random generator
Secure web pages
SecureRandom class
Securing Java: Getting Down to Business with Mobile Code (McGraw/Felten)
Security
  levels of
  mechanisms
Security manager class

Security managers
  configuring standard
  reading policy files
  in RMI applications
Security policy 2nd
Security policy files [See Policy files.]
SecurityException 2nd

SecurityManager class 2nd
SecurityPermission permission
Seek forward only mode
seek method 2nd
SELECT queries
SELECT statement
  adding to a batch
  executing to read a LOB
  in SQL 2nd
Selected frame
Selection(s)
  choosing from a very long list of
  moving from current frame to the next
Selection model, for rows
Selection state, setting for tree nodes
Semicolon (;), annotation placed without
separator field
Serial number, saving objects with
Serial version unique ID
SerialCloneable class
SerialCloneTest.java
@Serializable annotation
Serializable class
Serializable interface 2nd
SerializablePermission permission
Serialization
  copying objects using
  mechanism 2nd
  performance of
  unsuitable for long-term storage
  using for cloning
Serialized Java objects
SerialTransferTest.java
serialver program
serialVersionUID constant
Server(s)
  connecting to
  harvesting information from
  implementing
  role in distributed programming
  starting on a given URL
Server calls
Server program
Server-side script
ServerSocket class 2nd
Service provider interface, of a reader

SERVICE_FORMATTED data source
set methods 2nd 3rd
Set of nodes, XPath describing
Set operations, in regular expressions
setAllowsChildren method
setAllowUserInteraction method
setAsksAllowsChildren method
setAsText method
setAttribute method
setAutoCreateRowSorter method 2nd
setAutoResizeMode method
setBackground method
setCellRenderer method
setCellSelectionEnabled method
setClip operation
setClosed method
setColor method
setColumns method
setColumnSelectionAllowed method
setComparator method
setComposite method 2nd
setContextClassLoader method
setContinuousLayout method
setCurrency method
setDataElements method
setDefaultRenderer method
setDoInput method
setDoOutput method 2nd
setDragEnabled method 2nd
setDragMode method
setDropMode method
setEditable method
setEntityResolver method
setErrorHandler method
setFillsViewportHeight method
setFocusLostBehavior method
setHeaderRenderer method
setHeaderValue method
setIfModifiedSince method
setIndeterminate method
setLenient method
setMaximum method 2nd
setMaxWidth method
setMillisToDecideToPopup method
setMinimum method
setMinWidth method
setMnemonicAt method
setNamespaceAware method
setObject method
setObjectArrayElement method
setOneTouchExpandable method
setOverwriteMode method
setPage method
setPageable method
setPageSize method
setPaint method 2nd
setPixel methods 2nd
setPlaceholderCharacter method
setPreferredWidth method

setProgress method
setPropertyEditorClass method
setReader method
setRenderingHint method
setRenderingHints method 2nd
setRequestProperty method
setResizable method
setRootVisible method
setRowFilter method 2nd

setRowHeight method
setRowMargin method
setRowSelectionAllowed method
setSecurityManager method
setSeed method
setSelected method
setSelectedIndex method

setSelectionMode method 2nd
setSoTimeout method
setStringPainted method
setStroke method 2nd
setTabComponentAt method
setTabLayoutPolicy method
setTable method
SetTest program 2nd
SetTest.java
setText method
setTitle method
setTransform operation
setUseCaches method
setValue method 2nd 3rd
setValueAt method
setVisible method 2nd
setVisibleRowCount method
setWidth method
setWriter method
Set*Xxx*ArrayRegion method
SGML (Standard Generalized Markup Language)
SHA (Secure Hash Algorithm)
SHA1 (secure hash algorithm #1)
Shape classes
    relationships between
    using
Shape interface 2nd
Shape maker classes
Shape makers
ShapeMaker abstract superclass
ShapePanel class
Shapes
    composing from areas
    creating
    drawing
    rendering
    superimposing
ShapeTest.java
shared locks
shear method
Shear transformation 2nd
short values
ShortLookupTable subclass 2nd
shouldSelectCell method

showInternal*Xxx*Dialog methods
showWindowWithoutWarningBanner target
Side files
Signatures
    encoding
    of a field
    mangling
Signed applet 2nd
Simple Authentication and Security Layer (SASL)
Simple Mail Transport Protocol [See SMTP (Simple Mail Transport Protocol).]
Simple Object Access Protocol [See SOAP (Simple Object Access Protocol).]
Simple properties
Simple type
SimpleBeanInfo convenience class 2nd
SimpleCallbackHandler.java
SimpleDateFormat class
SimpleDoc class 2nd
SimpleJavaFileObject class
SimpleLoginModule.java
SimplePrincipal.java
SimpleTree.java
SimulatedActivity class
Single quotes, in SQL
Single value annotation
SINGLE_TREE_SELECTION
Singleton object, splash screen as
Singletons, serializing
SISC Scheme engine 2nd
size element
Skewed angle, for an elliptical arc
skip method
slapd.conf file
Slow activity, progress of
SMALLINT data type, in SQL 2nd
SMTP (Simple Mail Transport Protocol)
    specification
SOAP (Simple Object Access Protocol) 2nd
    message
    traffic
Social Security numbers
Socket(s) 2nd
Socket class 2nd 3rd
Socket constructor
Socket object
Socket operation, interrupting
Socket permission targets
Socket timeouts
SocketChannel class
SocketChannel feature, of java.nio
SocketPermission permission
SocketTest.java
SocketTimeoutException
Software developer certificates
Solaris, compiling InvocationTest.c
Sorting, rows
Source file annotations, tools harvesting
Source files 2nd
Source interface
Source level, processing annotations at
Source pixel
SOURCE retention policy
Source-level annotation process

Space character class
Spelling rule sets, in Norway
Spinner(s) 2nd
Spinner model
SpinnerDateModel class
SpinnerListModel 2nd
SpinnerNumberModel 2nd
SpinnerTest.java
Splash screens
drawing directly on
indicating the loading process on
replacing with a follow-up window
SplashScreen class
SplashScreenTest.java
split method
of Pattern
of String
Split panes
SplitPaneTest.java
Splitter bar
sprintf C function
SQL (Structured Query Language) 2nd
changing data inside a database
data types 2nd
exceptions
types
writing keywords in capital letters
SQL ARRAY
SQL statement file, program reading 2nd
SQL statements
executing
executing arbitrary
SQLException class 2nd
SQLPermission permission
SQLWarning class
SQLXML data type, in SQL
SQLXML interface
Square cap
SQuirrel
SRC rule
SRC_ATOP rule
SRC_IN rule 2nd
SRC_OUT rule 2nd
SRC_OVER rule 2nd 3rd
sRGB standard
SSL
Standard annotations
Standard extensions, loading
Standard Generalized Markup Language (SGML)
StandardJavaFileManager class
Start angle, of an arc 2nd 3rd
startElement method
startNameEnumeration function
State (ST) component
stateChanged method
STATELESS value, for scripts
Statement class 2nd 3rd 4th
Statement object 2nd
Statements, managing
Static fields
Static initialization block 2nd
Static methods, calling from native methods

StAX parser 2nd
StAXTest.java
stopCellEditing method 2nd
Stored procedures
Stream(s)
  assembling bytes into data types
  classes 2nd
  closing
  filters 2nd
  in the Java API
  keeping track of intermediate
  print services
  retrieving bytes from files
  sending print data to
  types
Streaming parsers 2nd
StreamPrintService class
StreamPrintServiceFactory class
StreamResult class 2nd
StreamSource 2nd
Strength, of a collator
String(s)
  converting into normalized forms
  filter looking for matching
  internationalizing
  objects, saving [See also Java String objects, converting.]
  painted property
  parameters
  patterns, specifying with regular expressions
  transferring to and from native methods
  writing and reading fixed-size
STRING data source
String parameter, of getPrice
StringBuffer class
StringBuilder class
StringBuilderJavaSource.java
StringSelection class 2nd
stringToValue method
Stroke interface
Strokes
  control over
  controlling placement of
  selecting
StrokeTest.java
Structure of a database
Structured Query Language [See SQL (Structured Query Language).]
Stub classes
Stubs
Style, in a placeholder index
style attribute
Style sheet 2nd
StyledDocument interface
Subcontext
Subject, login authenticating
Subject class
subtract operation 2nd
Subtrees
SUCCESS_NO_INFO value
sufficient module
Sun compiler
Sun DOM parser
Sun Java System Directory Server for Solaris

supportCustomEditor
@SupportedAnnotationTypes annotation
SupportedValuesAttribute interface
supportsBatchUpdates method
supportsResultSetConcurrency method
supportsResultSetType method
@SuppressWarnings annotation 2nd
SVG (Scalable Vector Graphics) format
Swing, data transfer support in

Swing code, generating dynamic
Swing components
    drag-and-drop behavior of
    layout manager for
Swing table, as asymmetric
Swing user interface toolkit
SwingDnDTest.java

SwingWorker class
Symbols [See also *specific* symbols.]
    in choice formats
    in a mask formatter
Symmetric ciphers 2nd
SyncProviderException 2nd
SysPropAction.java
System class
System class loader 2nd 3rd
System clipboard 2nd
SYSTEM declaration, in a DTD
SYSTEM identifier
System properties, in policy files
System tray
System.err
System.in
System.out
SystemTray class 2nd
SystemTrayTest.java

Index

Tab
    labels
    layout
    layout policy
    titles
Tabbed pane(s) 2nd
    user interface
TabbedPaneTest.java
Table(s)
    constructing from arrays
    inserting values into
    inspecting and linking
    joining 2nd
    manipulating rows and columns in
    with planet data
    printing
    producing
    selecting data from multiple
    simple
    types array for
Table cell renderers
Table classes 2nd
Table columns 2nd
Table index values
Table models 2nd
Table names 2nd
Table view, removing a column from
TableCellEditor class
TableCellEditor interface
TableCellRenderer class
TableCellRenderer interface
TableCellRenderTest.java
TableColumn class 2nd
TableColumn object 2nd
TableColumn type
TableColumnModel class
TableColumnModel object
TableModel class 2nd
TableRowSorter <M> object
TableRowSorter class
TableSelectionTest.java
TableStringConverter class
Tabs 2nd
Tag name, of an element
@Target meta-annotation
Target names, for permissions
TCP (Transmission Control Protocol)
telnet
    accessing an HTTP port
    activating in Windows Vista
    connecting to java.sun.com
Telnet windows
Tertiary character differences
@Test annotation

TestDB.java
Text
components, in the Swing library
input and output
transferring to and from the clipboard
transmitting through sockets
Text field(s)
editor
for integer input
losing focus
program showing various formatted
tracking changes in
user supplying input to
Text file, inside a ZIP file
Text format
for saving data
saving objects in
Text fragments
Text input, reading
Text nodes
constructing
as only children
Text output, writing
Text strings
converting back to a property value
property editors working with
saving
TextFileTest.java
TextLayout class
TextLayout object
TextTransferTest.java
TexturePaint class 2nd 3rd
TexturePaint object
this argument object
Thread(s)
executing scripts in multiple
forcing loading in a separate
making connections using
referencing class loaders
Thread class
ThreadedEchoHandler class
ThreadedEchoServer.java
THREAD-ISOLATED value
Three-tier applications
Three-tier model
Throw function
ThrowNew function
Thumbnails
Tiled internal frames
Tiling
frames
windows
Time
computing in different time zones
formatting
TIME data type, in SQL 2nd
Time of day service
Time picker
Timeout value, selecting
Timer, updating progress measurement
TIMESTAMP data type, in SQL 2nd
TimeZone class 2nd
TitlePositionEditor.java

Tödter, Kai
Tool class
Toolkit class
Tools, processing annotations
tools.jar, as no longer necessary
Tooltip, for a tray icon
Top-left corner, shifting
toString method
    calling to get a string
    displaying table objects
    returning a class name
    of the Variable class
Tracing
Tracking, in text components
Transactions
Transfer handler
    adding 2nd
    constructing
    installing
Transfer wrapper 2nd
Transferable interface 2nd 3rd
Transferable object
Transferable wrapper
TransferHandler class 2nd 3rd 4th
TransferSupport class
transform method 2nd 3rd 4th
Transformations
    composing 2nd
    supplying
    types of
    from user space to device space
    using 2nd
Transformer class
TransformerFactory class 2nd
TransformTest.java
Transient fields
transient keyword
Transient properties
Transitional events
translate method 2nd
Translation transformation
Transmission Control Protocol (TCP)
Transparency
Traversal order
Traversals
Tray icons 2nd
TrayIcon class
TrayIcon instance
Tree(s)
    cell renderer 2nd 3rd
    classes
    composed of nodes 2nd
    describing an infinite
    editing
    events
    leaves of 2nd
    parsers
    paths 2nd
    program displaying with a few nodes 2nd 3rd
    selection listener
    simple
    structures 2nd
    with/without connecting lines

Tree model(s)
    constructing 2nd
    custom
    linking nodes together
    obtaining
Tree nodes
    accessing with XPath
    changing font for individual
    determining currently selected
    editing
    iterating through
TreeCellRenderer class
TreeCellRenderer interface 2nd
TreeEditTest.java
TreeModel class 2nd
TreeModel interface 2nd 3rd
TreeModelEvent class
TreeModelEvent object
TreeModelListener class
TreeModelListener interface
TreeNode array
TreeNode class 2nd
TreeNode interface 2nd
treeNodesChanged method
treeNodesInserted method
treeNodesRemoved method
TreePath class 2nd
TreePath constructor
TreePath objects
TreeSelectionEvent class 2nd
TreeSelectionListener class
TreeSelectionListener interface
TreeSelectionModel
treeStructureChanged method
trim method
True Odds: How Risks Affect Your Everyday Life (Walsh)
Trust, giving to an applet
Trust models, assuming a chain of trust
try/catch block
try/finally block
tryLock method 2nd
Type(s)
    defined by a schema
    of images
    nesting definitions for
    in a placeholder index
Type drivers
TYPE_INT_ARGB
TYPE_INT_ARGB type
Typesafe enumerations

Index

UDP (User Datagram Protocol)
UI-intensive Windows programs, Visual Basic optimized for
Unambiguous DTD
Unicast
UnicastRemoteObject class 2nd
Unicode
    characters 2nd
    "replacement character" ('\uFFFD')
    strings
    using for all strings
UNICODE_CASE flag
Uniform Resource Identifier [See URI (Uniform Resource Identifier).]
Uniform resource name (URN)
Unique identifier, for a remote object
UNIX user, checking the name of
UNIX_LINES flag
UnknownHostException
UnsatisfiedLinkError
Unwrap mode
Updatable result sets 2nd
update methods 2nd
UPDATE statement 2nd
updateRow method 2nd
Upper case, turning characters of a string to
Upper character class
URI (Uniform Resource Identifier) 2nd
URI class
URL(s)
    compared to URIs
    connections
    forms of
    specifying a Derby database
    specifying for a DTD
    types of
URL class 2nd 3rd
URL data source
URL object
URLClassLoader class
URLConnection
URLConnection class
    API notes
    compared to Socket
    methods
    using to retrieve information
URLConnection object 2nd
URLConnectionTest.java
URLDecoder class
URLEncoder class
URN (uniform resource name)
US-ASCII character encoding
User(s)
    authentication
    coordinates, in transformations 2nd

drop action
interface components
names
objects 2nd
providing illegal input
restricting permissions to certain
User Datagram Protocol (UDP)
UTF-8 character encoding 2nd 3rd
UTF-16 character encoding 2nd 3rd

Index

Validating, XML documents
Validation
    of input
    languages
    turning on
VALUE_RENDER_QUALITY
VALUE_STROKE_NORMALIZE
valueChanged method 2nd
valueToString method
VARCHAR data type, in SQL 2nd
Variable class
Variable-byte encodings
Variants, in locales
Vendor name, of a reader
Verification
Verifiers 2nd
VerifierTest.java
verify method
VeriSign, Inc. 2nd
VeriSign certificate
Version number
    of the object serialization format
    of a reader
Versioning
VERTICAL, for a list box
VERTICAL_SPLIT, for a split pane
VERTICAL_WRAP, for a list box
Very long lists
Vetoable change listeners
vetoableChange method
VetoableChangeListener 2nd 3rd 4th
VetoableChangeSupport class 2nd
Vetoing 2nd
ViewDB application
ViewDB.java
Virtual machine(s)
    embedding into C or C++ programs
    function terminating
    launching
    loading class files
    setting up and calling the main method of Welcome
    terminating
    transferring values between
    writing strings intended for
Visual Basic 2nd
Visual feedback 2nd
Visual presentation

Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Z]

WarehouseActivator.java

WarehouseClient program 2nd
WarehouseImpl.java 2nd 3rd
Warehouse.java
WarehouseServer server program
WarehouseServer.java 2nd
WarehouseService class
Warning class

Warnings, retrieving
Weak certificates
WeakReference objects
Web applications
Web browser 2nd
Web crawler program
    code for
    implemented with the StAX parser
    implementing
Web or enterprise environment, JDBC applications in
Web pages, accessing secure
Web servers, invoking programs
Web service client
Web services
    architecture
    components of
    concrete example of
    in Java
Web Services Description Language [See WSDL (Web Services Description Language).]
Web Start applications
@WebParam annotation
WebRowSet interface
@WebService
WHERE clause, in SQL
Whitespace 2nd 3rd
Wild card characters, in SQL
Win32RegKey class
Win32RegKey.java
Win32RegKeyn class
Win32RegKeyNameEnumeration class
Win32RegKeyTest.java
Window listener
Windows [See also Microsoft Windows.]
    cascading all
    compiling InvocationTest.c
Windows executable program
Windows look and feel
    standard commands for cascading and tiling
    tree with 2nd
Windows Vista, activating telnet
Word check permissions
WordCheckPermission class
WordCheckPermission.java
Worker thread, blocking indefinitely
Working directory, finding

wrap method
Wrap mode
Wrapper class
WritableByteChannel interface
WritableRaster class
WritableRaster type
Write, then read cycle
write method
  of ImageIO
  of OutputStream 2nd
  of Writer
  writing out the first image
writeAttribute
writeCharacters
writeData method
writeDouble method
writeEmptyElement
writeEndDocument
writeEndElement
writeExternal method
writeFixedString method
writeInt method
writeObject method
  of the Date class
  of ObjectOutputStream 2nd
  as private
  of a serializable class
Write-only property
Writer class
writeStartDocument
writeStartElement
writeUTF method
Writing, text output
WS-* [See Web services.]
WSDL (Web Services Description Language) 2nd
  for the Amazon E-Commerce Service
  file
wsgen class
wsimport utility

Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Z]

x- prefix, indicating an experimental name
X Window System
X.500 distinguished names
X.509 certificate format
XDigit character class
XHTML 2nd
XML
    approaches for writing
    compared to HTML
    describing a grid bag layout
    format, expressing hierarchical structures
    header
    introducing
    layout, defining a font dialog
    output 2nd 3rd
    parsers
    protocol, advantage of
    reader, generating SAX events
    standard
    use of in a realistic setting
XML documents
    generating
    parsing
    reading
    structure of
    transforming into other formats
    validating
    writing with StAX
XML files
    describing a gridbag layout
    describing a program configuration
    format of
    parsing with a schema
    transforming into HTML
XML Schema 2nd
XMLDecoder 2nd
XMLEncoder 2nd 3rd
XMLInputFactory class
XMLOutputFactory class
XMLReader interface 2nd
XMLStreamReader class
XMLStreamWriter 2nd
XMLWriteTest.java
XOR rule
XPath
    expressions
    functions
    language
XPath class
XPath object
XPathFactory class
XPathTest.java
xsd prefix
xsd:choice construct

`xsd:schema` element
`xsd:sequence` construct
`xsl:output` element
XSLT (XSL Transformations) 2nd
XSLT processor 2nd 3rd
XSLT style sheet 2nd
`xsl:value-of` statement
Xxx`2D` classes
Xxx`2D.Double` class
Xxx`2D.Float` class

Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Z]

ZIP archives
ZIP file
    opening
    reading numbers from 2nd
    reading through
    writing
ZIP input stream
ZIP streams
`ZipEntry` class
`ZipEntry` constructor
`ZipEntry` object
`ZipException`
`ZipFile`
`ZipInputStream` 2nd 3rd
`ZipOutputStream` 2nd 3rd
`ZipTest.java`