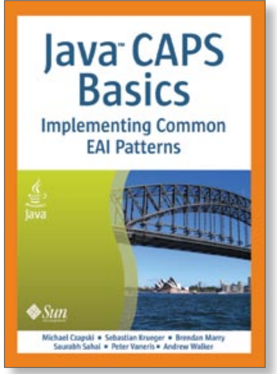
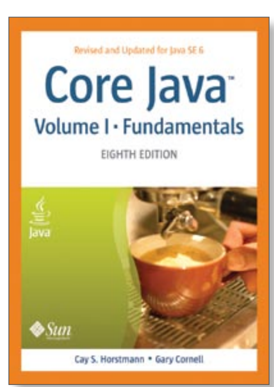
THE JAVA DEVELOPER'S DIGEST 2008 EDITION





A COLLECTION OF EXCERPTS FROM JAVA EXPERTS

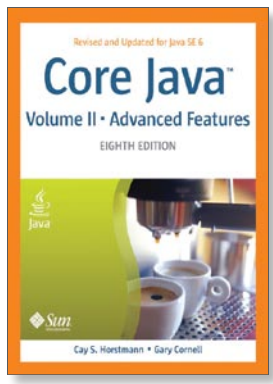
Michael Czapski, et al

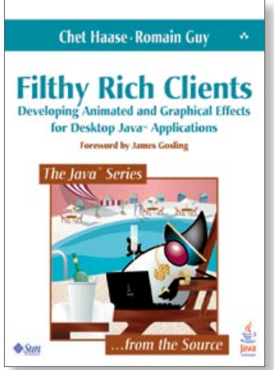
Chet Haase | Romain Guy

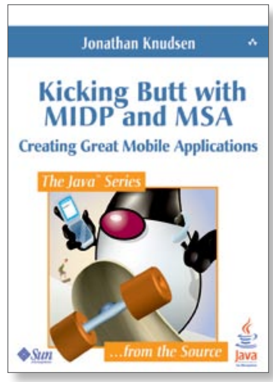
Marty Hall | Larry Brown
Yaakov Chaikin

Cay Horstmann | Gary Cornell

Jonathan Knudsen







Compliments of Addison-Wesley Professional, Prentice Hall Professional, and Sun Microsystems Press

THE JAVA DEVELOPER'S DIGEST

2008 EDITION

Michael Czapski, et al

Chet Haase | Romain Guy

Marty Hall | Larry Brown
Yaakov Chaikin

Cay Horstmann | Gary Cornell

Jonathan Knudsen

UPPER SADDLE RIVER, NJ | BOSTON | INDIANAPOLIS | SAN FRANCISCO | NEW YORK | TORONTO | MONTREAL
LONDON | MUNICH | PARIS | MADRID | CAPETOWN | SYDNEY | TOKYO | SINGAPORE | MEXICO CITY

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

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

The publisher offers discounts on this book when ordered in quantity for bulk purchases and special sales. For more information, please contact:

U.S. CORPORATE AND GOVERNMENT SALES
(800) 382-3419
corpsales@pearsontechgroup.com

FOR SALES OUTSIDE OF THE U.S., PLEASE CONTACT:
International Sales
(317) 581-3793
international@pearsontechgroup.com

VISIT ADDISON-WESLEY PROFESSIONAL AND
PRENTICE HALL PROFESSIONAL ON THE WEB:
www.informit.com

Copyright © 2008 by Pearson Education, Inc.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher. Printed in the United States of America.

For information on obtaining permission for use of material from this work, please submit a written request to:

Pearson Education, Inc.
Rights and Contracts Department
75 Arlington Street, Suite 300
Boston, MA 02116
Fax: (617) 848-7047

Text printed on recycled paper April, 2008

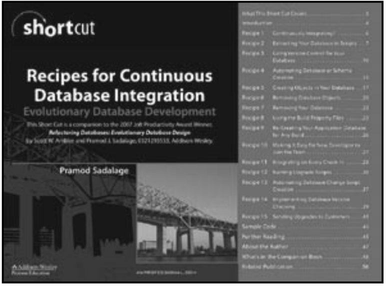
NOT FOR RESALE

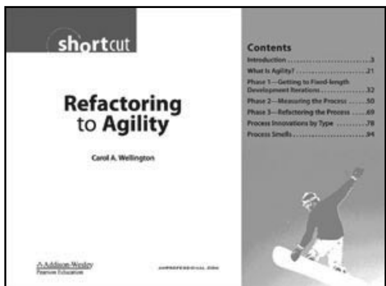
Short on time and need information fast... take a shortcut

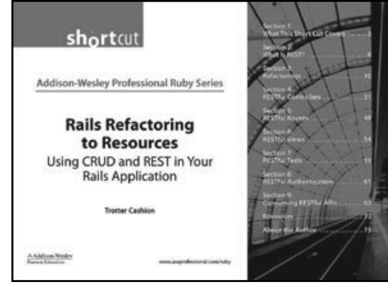
Short Cuts are short, concise PDF documents on cutting-edge new technology that shows great promise, or an existing technology that has reached the "tipping point" and is about to take off.

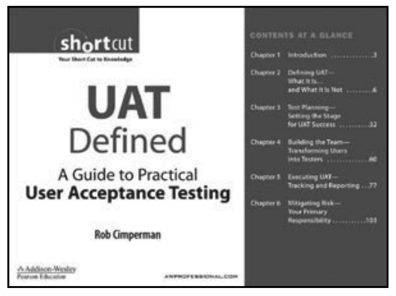
Written by industry experts and bestselling authors, Short Cuts are published with you in mind—getting you the technical information you need now.

There are more than 50 Short Cuts published—here are just a few samples









View Titles Available and Purchase Short Cuts at

informit.com/shortcut

Cisco Press

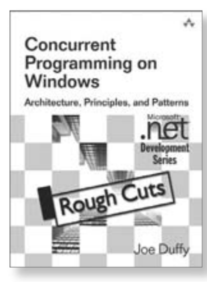
Get behind the Scenes to Stay Ahead of the Curve with

When you need to gain early access to information on cutting edge technologies, turn to the Rough Cuts service from Safari Books Online. With Rough Cuts you'll access the book's content as it is being written.

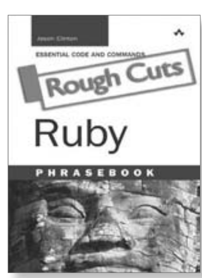
You can choose to purchase online access to the book with unlimited viewing and PDF downloads of each revision, pre-purchase the print book, or get the best of both worlds—online access immediately and the print book later. Any way you cut it, you will receive the finished product as soon as it is published, whether as a pdf or as a printed book.

Here are some samples of titles that were once in the Rough Cuts program.

0136135161

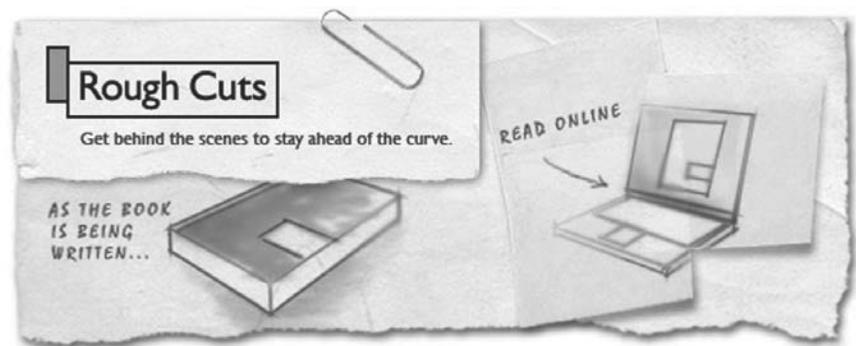


032143482X



0768666759

0137144482



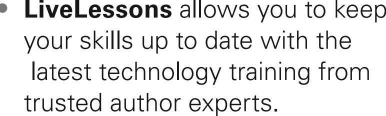
View Titles Available and Purchase **Rough Cuts** at

http://safari.informit.com/roughcuts

Cisco Press

livelessons

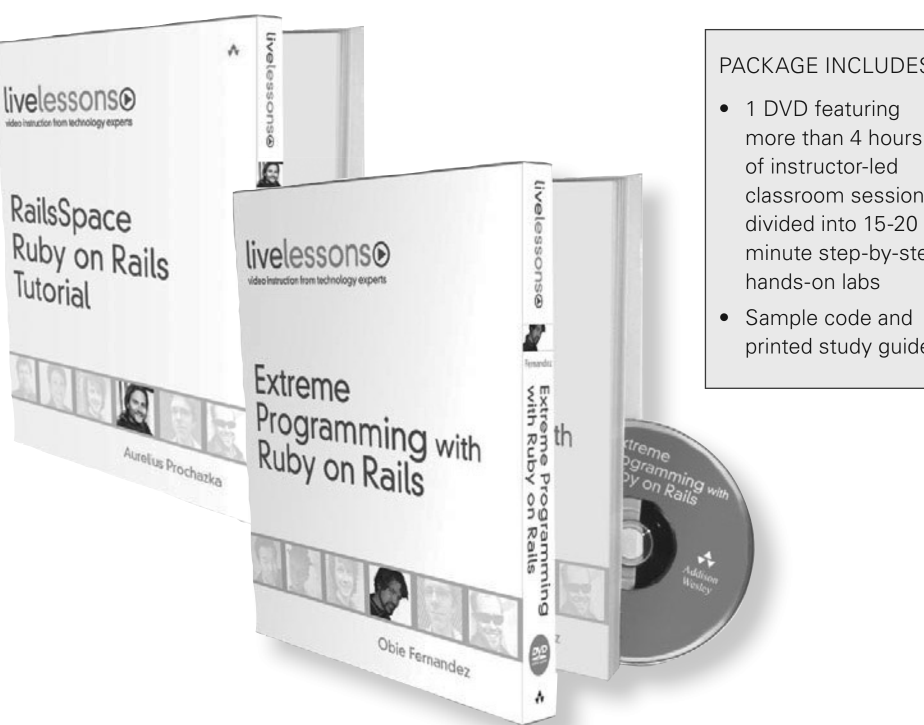
video instruction from technology experts



- **LiveLessons** allows you to keep your skills up to date with the latest technology training from trusted author experts.
- **LiveLessons** is a cost effective alternative to expensive off-site training programs.
- **LiveLessons** provides the ability to learn at your own pace and avoid hours in a classroom.

LiveLessons: self-paced, personal video instruction from the world's leading technology experts

- INSTRUCTORS YOU TRUST
- CUTTING EDGE TOPICS
- CUSTOMIZED, SELF-PACED LEARNING
- LEARN BY DOING



PACKAGE INCLUDES:

- 1 DVD featuring more than 4 hours of instructor-led classroom sessions divided into 15-20 minute step-by-step hands-on labs
- Sample code and printed study guide

The power of the world's leading experts at your fingertips!

To learn more about **LiveLessons** visit
www.mylivelessons.com

TABLE OF CONTENTS

MESSAGE ROUTING

From JAVA CAPS BASICS - IMPLEMENTING COMMON EAI PATTERNS
by Michael Czapski, Sebastian Krueger, Brendan Marry
Saurabh Sahai, Peter Vaneris, and Andrew Walker 1

- Introduction 3
- Overview 3
- Fixed Router 5
- Content-Based Router 7
- Message Filter 10
- Recipient List 11
- Splitter 13
- Aggregator 14
- Resequencer 15
- Composed Message Processor 17
- Scatter-Gather 17
- Routing Slip 18
- Process Manager 19
- Message Broker 19
- Chapter Summary 20

IMAGING PROCESSING

From FILTHY RICH CLIENTS: DEVELOPING ANIMATED AND GRAPHICAL
EFFECTS FOR DESKTOP JAVA™ APPLICATIONS
by Chet Haase and Romain Guy 21

- Image Filters 24
- Processing an Image with BufferedImageOp 25
- AffineTransformOp 27
- ColorConvertOp 28
- ConvolveOp 30
- LookupOp 35
- RescaleOp 37
- Custom BufferedImageOp 38
- A Note about Filters Performance 46
- Summary 46

TAG LIBRARIES: THE BASICS

From CORE SERVLETS AND JAVASERVER PAGES, VOLUME 2:
ADVANCED TECHNOLOGIES
by Marty Hall, Larry Brown, and Yaakov Chaikinr 47

- Tag Library Components 50
- Example: Simple Prime Tag 55
- Assigning Attributes to Tags 59
- Example: Prime Tag with Variable Length 61
- Including Tag Body in the Tag Output 64

TABLE OF CONTENTS

Example: Heading Tag 66
Example: Debug Tag 70
Creating Tag Files 73
Example: Simple Prime Tag Using Tag Files 74
Example: Prime Tag with Variable Length Using Tag Files 76
Example: Heading Tag Using Tag Files 78

INHERITANCE

From CORE JAVA VOLUME I, EIGHTH EDITION: FUNDAMENTALS
by Cay Horstmann and Gary Cornell 81

Classes, Superclasses, and Subclasses 84
Object: The Cosmic Superclass 109
Generic Array Lists 116
Object Wrappers and Autoboxing 123
Methods with a Variable Number of Parameters 126
Enumeration Classes 127
Reflection....129
Design Hints for Inheritance 150

SECURITY

From CORE JAVA VOLUME II: ADVANCED FEATURES, EIGHTH EDITION
by Cay Horstmann and Gary Cornell 153

Class Loaders 156
Bytecode Verification 167
Security Managers and Permissions 171
User Authentication 190
Digital Signatures 205
Code Signing 222
Encryption 228

BASIC USER INTERPACE

From KICKING BUTT WITH MIDP AND MSA:
CREATING GREAT MOBILE APPLICATIONS
by Jonathan Knudsen 241

How to Show Screens 246
TextBox, the Runt of the Litter 247
Input Modes 248
Using Alerts for Notifications 249
A Very Quick Introduction to Images 250
Putting It Together 250
Good for the Old Ticker 252
The Whole Story on Commands 252
Command Placement 255
Summery 256

Use Java CAPS to Streamline IT Services and
Leverage Legacy Applications without Extensive Coding

MESSAGE ROUTING

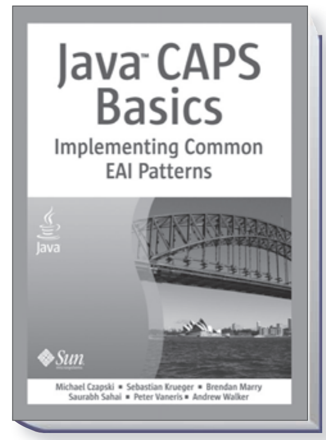
FROM

JAVA CAPS BASICS

Implementing Common EAI Patterns

by Michael Czapski, Sebastian Krueger, Brendan Marry,
Saurabh Sahai, Peter Vaneris, and Andrew Walker

©2008 | 512 PAGES | ISBN: 0-13-713071-6



ALSO AVAILABLE

- SAFARI BOOKS ONLINE
- E-BOOK: 0137130716
- MOBI POCKET: 0137146140
- SONY READER: 0137146167

TABLE OF CONTENTS

PART I: PRELIMINARIES

Chapter One:
Enterprise Integration Styles

Chapter Two:
Java CAPS Architecture

Chapter Three:
Project Structure and Deployment

PART II: PATTERNS REVIEW AND APPLICATION

Chapter Four:
Message Exchange Patterns

Chapter Five:
Messaging Infrastructure

Chapter Six:
Message Routing

Chapter Seven:
Message Construction

Chapter Eight:
Message Transformation

Chapter Nine:
Messaging Endpoints

Chapter Ten:
System Management

PART III: SPECIALIZED JAVA CAPS TOPICS

Chapter Eleven:
Message Correlation

Chapter Twelve:
Reusability

Chapter Thirteen:
Scalability and Resilience

Chapter Fourteen:
Security Features

FOR MORE INFORMATION
informit.com/title/9780137130719

PRENTICE HALL

CHAPTER SIX

Message Routing

6.1 Introduction

This chapter discusses message routing patterns. It includes discussion and application of patterns from [EIP] Messaging Systems and Message Routing. The chapter briefly discusses where a Java CAPS solution developer can make routing decisions and discusses each of the routing patterns in turn, specifically Splitter, Aggregator, Resequencer, Scatter-Gather, Routing Slip, Process Manager, and Message Broker.

6.2 Overview

A messaging-based integration solution, whether or not and however it transforms messages as they pass through, inevitably routes messages from one or more sources to one or more destinations. A Java CAPS solution can make message routing decisions in four areas: the JMS Message Server, the connectivity map, the Java Collaboration definition, and the eInsight Business Process. Typical solutions that use just the eGate infrastructure would perform routing through the JMS Message Server, the connectivity map, and possibly the Java Collaborations. Typical solutions that use eInsight Business Process Management (BPM) would perform routing predominantly within eInsight Business Processes but may also route in the connectivity map. In all but the simplest solutions, routing will likely be performed by multiple components.

Routing in the JMS Message Server is performed as a consequence of configuring nondefault redelivery handling, which can divert messages to Dead Letter Queues. This issue was discussed in Chapter 5, “Messaging Infrastructure,” section 5.13.

The connectivity map, the graphical representation of how Java CAPS components are connected, is the means to both collect all integration solution components that will be deployed as part of a single enterprise application and to configure certain aspects of the message endpoints that are logical in nature, such as JMS Destination names and properties, or names and name patterns for file system objects. The simplest functional Java CAPS solution must have a minimum of two components: a message source and a service that operates on messages from that source. Unlikely as it may seem, in special circumstance, such an apparently useless solution might be valid and reasonable. What [EIP] calls the Channel Purger would be an example of a solution that receives messages from an endpoint and routes them to nowhere. Figure 6-1 shows a connectivity map for a basic Channel Purger.

This is the simplest example of message routing: Fixed Routing [EIP].

Note
A Java CAPS implementer would typically look at the connectivity map for routing information—which components publish and subscribe to which JMS Destinations and how many, and which JMS Destinations are subscribed to/published to by an eInsight Business Process. For that reason, a solution that makes explicit routing decisions in Java Collaboration Definitions (JCDs) or Business Processes will be more difficult to analyze by an implementer new to it. It will also make it harder for the original developers to recall where and how routing decisions are made. If no other considerations dictate specific choices, given a choice of explicit routing in a JCD and explicit routing in an eInsight Business Process, choose the latter, as its graphical depiction of processing logic makes it more obvious that explicit routing takes place. Multiple subscriptions and/or publications by a service on a connectivity map are a strong hint that explicit routing is taking place inside a service component.

Message Router [EIP], a specialized Filter [EIP], represents a component in an integration solution that causes messages to be passed from a source to a destination depending on a possibly empty set of criteria. Unlike connectivity map–

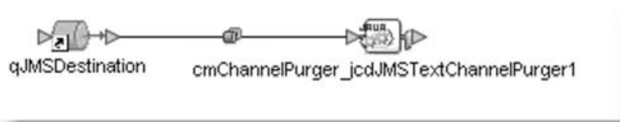


Figure 6-1: Channel Purger

based fixed routing, Message Router variants that make explicit routing decisions programmatically can all be implemented in a Java CAPS solution using either JCDs or eInsight Business Processes or both.

The following sections discuss implementation of most of the router patterns using Java CAPS as the infrastructure.

6.3 Fixed Router

A fixed router, one where a single channel is a source of messages and a single channel is a destination, is the most trivial form of a Message Router. You would typically configure the connectivity map source and destination to configure a fixed router. If necessary, however, a Java Collaboration or a Business Process can be constructed to explicitly choose a destination if that destination is a JMS Destination.

Given the connectivity map shown in Figure 6-2, we would expect that the Java Collaboration publishes messages to the JMS Destination (queue) qDummy-Destination.

Inspection of the collaboration source, shown in Figure 6-3, reveals that it is the JMS Destination (queue) qNewQueue that is the actual destination of messages. This destination is hardcoded in the fixed router.

The same effect could be achieved by explicit assignment of the destination queue name prior to sending the message, as shown in Figure 6-4.

Given the connectivity map shown in Figure 6-5, we would again expect the queue qDummyDestination to be the destination of messages.

Inspecting the business rules embedded in the eInsight Business Process, shown in Figure 6-6, reveals this to not be the case.

In this example, an explicit assignment of a JMS Destination name to the destination node of the JMS OTD results in messages being explicitly routed to a JMS Destination (queue) qNewJMSDestination.

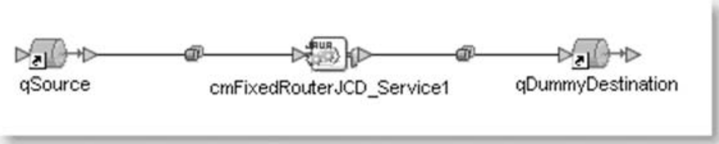


Figure 6-2: Connectivity map of an implicit fixed router

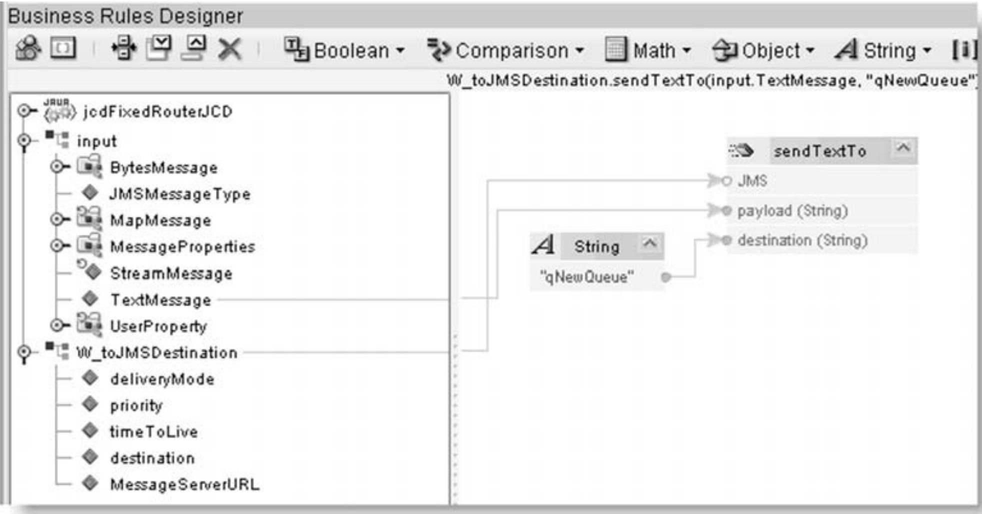


FIGURE 6-3: Hardcoded JMS queue name in a fixed router, which uses a sendTo() OTD method

FIGURE 6-4: Hardcoded JMS queue name in a fixed router using a "destination" OTD node

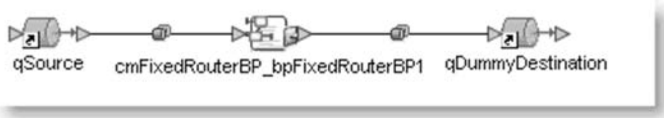


FIGURE 6-5: Implicit fixed router connectivity map

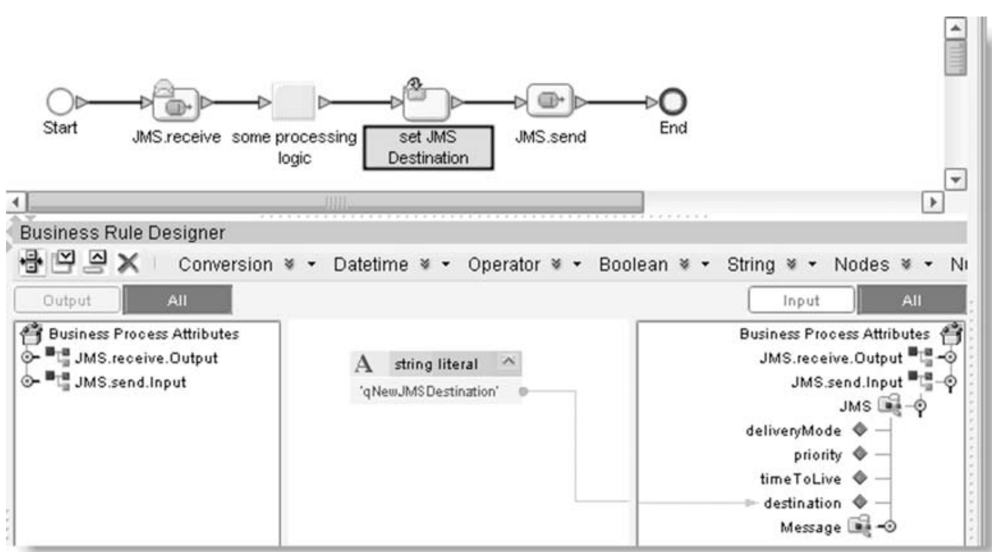


FIGURE 6-6: Explicit JMS queue assignment in a Business Process

Note

In all of the previous examples, the JMS Destination name contained the literal Dummy. This is a hint to the developer who inspects the connectivity map that the actual destination is likely different and is configured within the component that publishes to the "dummy" destination. This is a good practice suggestion, since no part of Java CAPS enforces naming conventions.

6.4 Content-based Router

Content of the message may dictate the destination to which the message must be delivered. Content-based Router [EIP] inspects the message it receives and sends it to a destination depending on the content.

In a simple case, a Java Collaboration or a Business Process would have a set of destinations hardcoded within a switch or an if-then-else construct that operates on all or part of the message. An example in Figure 6-7 is a simple Java Collaboration that illustrates dynamic JMS Destination selection.

```
public void receive( com.stc.connectors.jms.Message input, com.stc.connectors.jms.JMS W_toJMSDestination )
    throws Throwable
{
    if (input.getTextMessage().toUpperCase().endsWith( "PRIMARY" )) {
        W_toJMSDestination.setDestination( "qToPrimary" );
    } else if (input.getTextMessage().toUpperCase().endsWith( "SECONDAR" )) {
        W_toJMSDestination.setDestination( "qToSecondary" );
    } else {
        W_toJMSDestination.setDestination( "qToCatchOther" );
    }
    W_toJMSDestination.sendText( input.getTextMessage() );
}
```

FIGURE 6-7: Hardcode dynamic router

A Business Process that implements a dynamic router can be constructed similarly, as the example in Figure 6-8 shows.

Here a decision gate inspects a message to determine which branch to follow. A Business Rules activity assigns a string literal to the JMS Destination's destination attribute, and the JMS.send activity gets the message delivered to the destination so set.

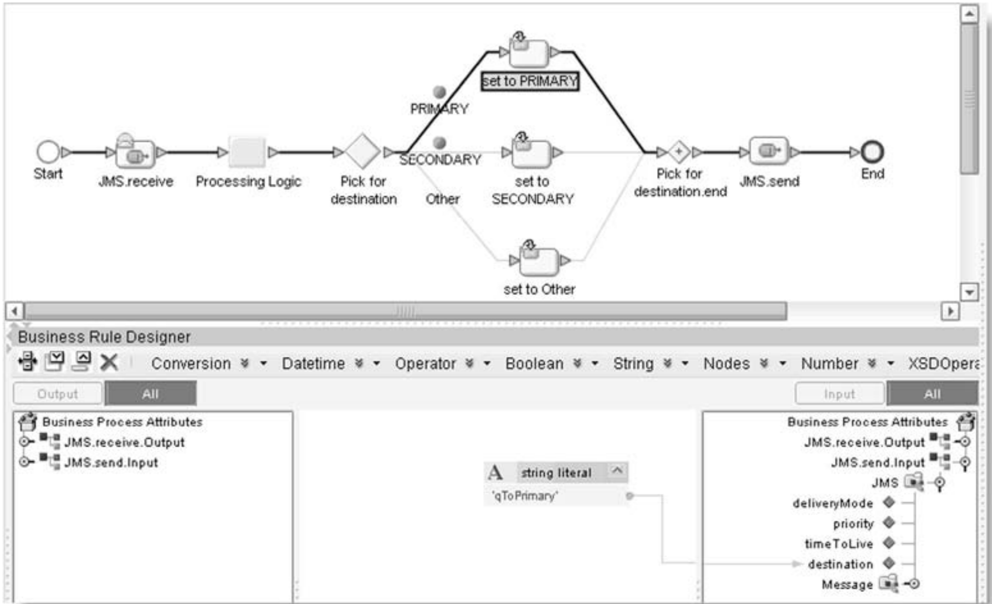


FIGURE 6-8: eInsight Business Process–based dynamic router

Whether a JCD or a Business Process is used, the connectivity map for the example will be identical to that used for the fixed router in the example in the previous section (i.e., the name of the JMS Destination will be unrelated to the actual JMS Destination to which messages will be delivered).

In the two examples shown in Figures 6-7 and 6-8, a conditional was evaluated to determine the destination of the message, which was hardcoded. If additional destinations were required, the collaboration or the process would have to be modified, and the application containing it would have to be redeployed to propagate changes to the runtime environment. This implementation of a Content-based Router is potentially a high-maintenance implementation if destinations change frequently.

In a special case, you could use the message, or the message component, as the complete name or a part of the name of the destination. You would not need to use a conditional or hardcode destination names.

The JCD in Figure 6-9 appends the first 10 characters of the input message to a literal qDest to form the name of the destination. The message is then written to the destination with the resulting name.

If the first 10 characters of messages were PRIMARY??? and SECONDARY?, where ? represents a space character, the resulting destination names would be qDestPRIMARY and qDestSECONDARY respectively. If using the Sun SeeBeyond JMS implementation, which does not require you to preconfigure JMS

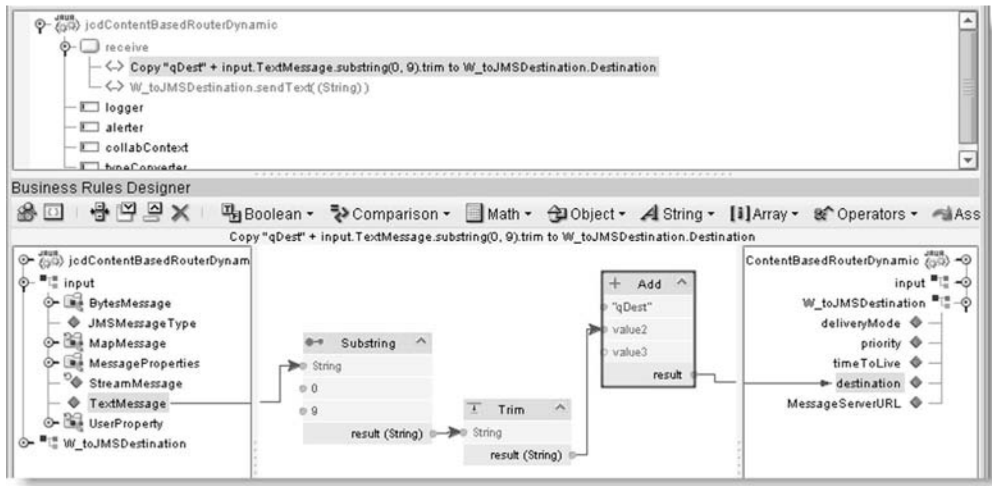


FIGURE 6-9: Dynamically generating JMS Destination names

Destinations ahead of time but rather creates JMS Destinations on first reference if they don't already exist, this could result in a completely dynamic Content-based Router. You could introduce a message whose initial characters were something other than PRIMARY??? and SECONDARY?, and the Sun SeeBeyond JMS implementation would create a new JMS Destination with that appropriate name and deliver the message there. There may not be a receiver for the message delivered to the new destination, but the Content-based Router itself would be dynamic and would not require modification. Addition of a destination would not require redeployment of the application containing such a router if no other changes were needed to take advantage of the new route. This solution does not require maintenance of the router if the number of destinations changes but makes it difficult to determine to how many destinations messages are routed, as it removes the setting of the content, upon which routing decisions are made, from the router to some other component upstream from the router or even outside the integration solution altogether.

In the previous examples, a very simple text message was used and some leading or trailing characters were extracted from the text for use in the conditional or as a part of a destination name. Messaging systems will rarely deal with such unstructured text messages. Much more likely, messages will be structured. The contents of one or more fields in the message will then be used for routing decisions or destination name derivation. For simplicity, we will continue using simple text messages wherever message structure has no bearing on the discussion.

These trivial examples demonstrate how explicit routing can be performed programmatically within a JCD or a Business Process. This method will be used to set destinations for more complex Message Routers.

6.5 Message Filter

Message Filter [EIP] is a component in an integration solution that selectively processes messages. A Java CAPS solution offers two ways in which a Message Filter can determine whether or not to process a message.

If the message source is a JMS Destination, such as a queue or a topic, the Message Filter can be configured, through the connectivity map, to only accept messages whose attributes match an SQL-like selection expression. This method leverages the JMS selector mechanism. Rather than receiving a message and, if not of interest, discarding it, the JMS selector–based Message Filter prevents

delivery of messages that do not match the selector expression to the filtering receiver. This mechanism is static in that the selection expression is configured through the connectivity map and cannot be changed without redeploying the enterprise application.

Java CAPS provides the means to implement a dynamic selection solution using a Java Collaboration. This technique, discussed at length in Chapter 5, section 5.6.7, and Chapter 11, "Message Correlation," section 11.11, allows selection expression to change at runtime, thus providing the means to implement dynamic routing solutions.

6.6 Recipient List

By Saurabh Sahai

Often, it is required that a message be selectively sent to more than one recipient. The recipients that are to receive each message are determined either dynamically, based on the message content, or statically, based on external business rules. For example, an expense approval request message, pertaining to expenses below a certain amount, may get sent to the immediate manager for approval, whereas a message above the defined limit must also be sent to the business unit head for special approval.

A recipient list processor is similar to a Content-based Router; however, unlike a Content-based Router that routes the message to a specific destination based on message content, a recipient list processor sends the message to one or more designated recipients. The list of recipients can be static, hardcoded within the implementation, or dynamic, provided to the implementation from an externally maintained source. The latter approach provides greater flexibility as it allows the recipient list to be dynamically configured.

In Java CAPS, a recipient list can be implemented using either a JCD or a Business Process. In either case, once the message is received, the list of intended recipients is computed from the available recipients, and the message is forwarded as required.

The Java Collaboration shown in Figure 6-10 receives an expense report message for an amount greater than $300. Based on business rules, the collaboration looks up additional approvers that are required to approve this expense report and sends a copy of the message to these approvers in addition to the default

approver. This is an example of a dynamic recipient list, where the collaboration uses information stored in an external store such as the organization's Lightweight Directory Access Protocol (LDAP) server to create the required recipient list. Exception processing has been omitted in Figure 6-10 to focus on the essentials of the example.

In the example in Figure 6-10, the additional approval threshold has been hardcoded within the Java Collaboration. In a more realistic example, externally configurable delegation of authority rules would be loaded into the collaboration using one of the techniques for dealing with dynamic runtime reconfiguration of components, discussed elsewhere in the book.

The example uses a hypothetical sendMail() method to send the expense report to a recipient for approval. The collaboration could have equally validly used multiple JMS Destinations, a single JMS Destination with target recipient indicated using JMS user properties, a Batch eWay, WebSphere MQ eWay, or any number of other endpoints, as dictated by the environment or business requirements.

```
private static final float DEF_APPROVER_EXPENSE_LIMIT = 300.0f;

public void receive( com.stc.connectors.jms.Message input, xsd.ExpenseReport516785849.ExpenseReport_ otdExpenseReport,
                     com.stc.connectors.jms.JMS otdJMS ) throws Throwable
{
    // unmarshal the incoming expense report into the ExpenseReport OTD
    String expenseReport = input.getTextMessage();
    otdExpenseReport.unmarshalFromString( expenseReport );
    // compute the expense total
    float expenseTotal = 0f;
    int numExpenseItems = otdExpenseReport.getExpenseDetails().countExpenseItem();
    for (int i = 0; i < numExpenseItems; i++) {
        expenseTotal += otdExpenseReport.getExpenseDetails().getExpenseItem( i ).getAmount();
    }
    // Determine the recipients for this expense report
    String employeeID = otdExpenseReport.getEmployeeID();
    if (expenseTotal <= DEF_APPROVER_EXPENSE_LIMIT) {
        // send the expense report to the default approver.
        // lookup the email id for the employee's default approver
        String defaultApproverEmailId = lookupApprovingManagerEmailId( employeeID );
        sendMail( defaultApproverEmailId, expenseReport );
    } else {
        // lookup email IDs for senior approvers required to approve expenses above the predefined limit
        String[] snrApproverEmailIDs = lookupSeniorApproveEmailIds( employeeID );
        for (int i = 0; i < snrApproverEmailIDs.length; i++) {
            // send expense report to each designated approver
            sendMail( snrApproverEmailIDs[i], expenseReport );
        }
    }
}
```

FIGURE 6-10: Recipient list example

6.7 Splitter

By Saurabh Sahai

An incoming message may encapsulate one or more submessages. It is often desired to process submessages independently as separate messages. For example, an order message may consist of multiple order line items, each of which corresponds to a unique item type and may be fulfilled by a separate inventory store.

A splitter solves the problem of processing a composite message comprising multiple submessages, each of which may be processed differently by breaking up the message into individual messages and sending each separate message for further processing by a downstream component.

A splitter can be implemented in Java CAPS in multiple ways. Java Collaborations can be used to receive the composite message, iterate over the individual submessages, and, on the basis of the message content, send each of them to a unique destination that is responsible for processing a specific type of message.

The collaboration shown in Figure 6-11 is an example of processing an incoming message consisting of multiple order items. The collaboration iterates over each order item and creates a new message, enriched with the original order item information, and sends it for processing by a specific system. The item number contained in each order item is used to determine the destination address where the enriched order item message is to be sent. Exception processing has been omitted in the example to focus on the essentials of the example.

Splitter is a component that, as the name suggests, breaks messages into component parts. How easy or difficult it is to split original messages and create component messages largely depends on the size and complexity of the message structures involved. As a general rule, it is easier to handle a composite message with more than one level of components using a Java Collaboration than to do so using an eInsight Business Process. Implementing nested loops in a Java Collaboration is easier and more compact than doing the same in Business Process Execution Language (BPEL) using the graphical environment. While loops, whether single-level or nested, are clearly visible in an eInsight Business Process graphic, making it obvious that splitting is taking place, it is necessary to reset the target OTD structure prior to its being populated in each iteration, which is neither obvious nor easily discovered by the casual observer.

```
/**
 * Receives an Order consisting of multiple order items each of which is to be processed by an
 * individual inventory system. The collaboration iterates over each of these individual
 * order items and sends an enriched message containing the order item and the original order id
 * to the appropriate inventory item for the fulfillment of the order item. For this example,
 * the queue for each inventory that will be used to fulfill the order item is determined using
 * the item number embedded in the order item message.
 */
public void receive( com.stc.connectors.jms.Message input, xsd.Order595920194.CompositeOrder otdCompositeOrder,
                    xsd.Order595920194.SubOrder otdSubOrder, com.stc.connectors.jms.JMS otdJMS )
    throws Throwable
{
    String inputOrder = input.getTextMessage();
    /*
      unmarshal the incoming composite message into the COmpositeMessage OTD
     */
    otdCompositeOrder.unmarshalFromString( inputOrder );
    int numOrderItems = otdCompositeOrder.countOrderItem();
    String orderId = otdCompositeOrder.getOrderId();
    otdSubOrder.setOrderId( orderId );
    for (int i = 0; i < numOrderItems; i++) {
        String itemNumber = otdCompositeOrder.getOrderItem( i ).getItemNumber();
        int itemQty = otdCompositeOrder.getOrderItem( i ).getQuantity();
        /*
          Update the outgoing message with the itemNumber and quantity.
         */
        otdSubOrder.getOrderItem().setItemNumber( itemNumber );
        otdSubOrder.getOrderItem().setQuantity( itemQty );
        /*
          Send the individual order item to the inventory system.
          The inventory queue is identified by concatnating "q" with the item number.
         */
        String qInventorySystem = "q" + itemNumber;
        otdJMS.sendTextTo( otdSubOrder.marshalToString(), qInventorySystem );
    }
}
```

FIGURE 6-11: Dynamic content-based routing

6.8 AGGREGATOR

Aggregator [EIP] is a special Filter that collects related messages until some completeness condition has been reached, at which point it processes the related messages to obtain a single aggregated message that is then passed on to the next component.

An Aggregator must be able to correlate related messages, store related messages until ready to process, determine when the completeness condition is met, and implement the aggregation logic.

Java CAPS eInsight engine is a convenient tool to use for building Aggregators, as it inherently supports correlations and transparently stores related messages. The specific eInsight Business Process must only implement the completeness condition and the aggregation logic in order to become a specific Aggregator.

At the heart of every Aggregator is correlation logic, logic that determines which messages are related and therefore are subject to aggregation. Chapter 11 discusses at length the topic of message correlation with references to a number of specific examples presented in Part II (located on the accompanying CD-ROM). Chapter 11, section 11.10, discusses in detail a number of correlation implementations that incorporate an Aggregator.

Implementing an Aggregator without the benefit of eInsight is much harder. In addition to having to implement a completeness condition and aggregation logic, the solution designer must also do all the work related to storing and correlating messages. Chapter 11, section 11.11, discusses this topic and presents an example of how to accomplish the task of storing and correlating messages using just eGate and the Sun SeeBeyond JMS Message Server.

[EIP] discusses a number of completeness conditions an Aggregator might implement: Wait for All, Timeout, First Best, Timeout with Override, and External Event. Variants of these completeness conditions are discussed in Chapter 11, section 11.10. Implementing these conditions using eInsight with correlations is rather trivial. Implementing most of them using just eGate is much more difficult.

[EIP] also discusses a number of aggregation algorithms, including Select Best, Condense, and Collect for Later. Variants of these are also discussed in Chapter 11, section 11.10. Since aggregation will not start until all related messages are collected—that is, until the completeness condition has been satisfied—implementing aggregation logic is equally simple whether eInsight or eGate is used.

6.9 Resequencer

By Sebastian Krueger

Messages can arrive out of order for many reasons. If these messages are required to be delivered in sequence to a downstream component, the easiest solution would be to make sure that they never get out of order in the first place. Such approaches were discussed in Chapter 4, "Message Exchange Patterns," section 4.8. However, there may be times when we don't have a choice of how the upstream components are implemented; for example, we may control only the receiving side. Thus, the need to implement a component that will reorder messages may arise.

A number of implementations of a resequencer are possible. Chapter 4, section 4.2, "Resequencer," in Part II, discusses and illustrates two implementations

using examples: a simple buffered resequencer and a persisted resequencer, both of which are discussed in the remainder of this section.

The simple buffered resequencer operates as follows. When the resequencer receives a message, it adds that message to an internal buffer. It then sends all consecutive messages from the buffer.

In order to send all consecutive messages, the resequencer component needs to know the current sequence number index and whether a message with this index is in the buffer. If the index is not found in the buffer, then the message has not arrived yet. The resequencer will not send out buffered messages until at least the next message arrives.

A resequencer implementation requires all messages to have a unique sequence number. Not only do these sequence numbers have to be unique, they also have to be consecutive. That is, no gaps are allowed to exist in the sequence.

If a message gets lost and never arrives, messages would be queued up and would never be sent out because the resequencer component would be waiting for a message that will never arrive. To get around this issue, the designer could implement a solution whereby the resequencer only waits a set time for a message to arrive and then moves on to the next messages, effectively ignoring the message that never arrived. However, what if the message arrives late? What if a duplicate message arrives? Strategies for dealing with these conditions would have to be considered in designing a robust resequencer. The designer could, for example, discard the message, or send it to a Dead Letter Channel for alerting and auditing purposes.

When a simple resequencer starts, it expects the first message to have a sequence number of 0. However, what if this is not the case? For example, the resequencer might restart. Unless the sequence is persisted, the resequencer would expect the message sequence to start at 0 again. There are two ways to get around this problem. An initialization message could be sent that informs the resequencer which number is the start of the sequence. Alternatively, the sequence could be persisted so that it can be recovered in appropriate circumstances.

Another point that a simple resequencer does not handle is buffer overrun. If too many messages get queued up, the HashMap, used to store messages while assembling message sequences, may get too large to fit into the JVM allocated memory.

Implementation of a robust and scalable resequencer would require a significant amount of code and would likely be domain specific. An improved resequencer is discussed next. Implementation of a perfect resequencer is beyond the scope of this book.

An improvement to the previous resequencer would be to implement the buffer as a database table. By moving the message buffer to a persistent store, we solve two problems exhibited by the previous implementation. First, we effectively have an unlimited buffer, so no buffer overflow will occur. Second, in case of a server failure, buffered messages are not lost.

There are still unresolved issues with this persisted resequencer. The initialization of sequence numbers expects the first message to always start at 0. Also, we have not accounted for messages that never arrive. We briefly touched on the some of these issues in the previous section. They are out of the scope of this chapter.

While the two resequencers discussed in this chapter are by no means perfect, they do give examples of simple resequencers and give an indication of what is required to implement a robust resequencer.

6.10 Composed Message Processor

Composed Message Processor [EIP] is a higher-order component of a messaging system that accepts a message, breaks it up into submessages that are dispatched and processed by multiple lower-order components, then reassembles submessages into a final message.

In Java CAPS, as in any messaging system, implementation of a Composed Message Processor requires the use of correlations. Superficially, Composed Message Processor pattern is no different from the Scatter-Gather pattern [EIP]. Both involve breaking a message and reassembling the pieces once they are processed by independent intermediate components.

Chapter 11 discusses correlation implementation options provided by Java CAPS and presents a number of correlation examples. Section 11.10.5 provides a Java CAPS example of a Scatter-Gather pattern and Composed Message Processor pattern implementation.

6.11 Scatter-Gather

The Scatter-Gather [EIP] pattern involves breaking up a message, or replicating a message, delivering multiple messages to multiple components, then collecting related messages back together. Implementation of a Scatter-Gather pattern requires the use of correlations. It is discussed in various sections of Chapter 11.

6.12 Routing Slip

Routing Slip [EIP] is a mechanism that can be used to dynamically route a message through a series of components such that individual components do not embed routing logic. The route the message is to take can be computed by a router that embeds the necessary logic. This route is then attached to the message as a Routing Slip, and each component through which the message passes, once it performs its processing, forwards the message onto the next component specified in the Routing Slip. [EIP] discusses at length the rationale behind a desire to implement the Routing Slip pattern. Java CAPS, and its underlying JMS Message Server implementation, lends itself to building Routing Slip–based solutions; however, eInsight Business Processes may be better, in many circumstances, as an approach to conditional component invocation and dynamic route determination.

In a fixed Routing Slip solution, the message, once it leaves the router, is passed from component to component. There is no opportunity to change the message route once it is computed. An alternative to this approach is to have the router compute the next component to which to send the message, send the message to it, and have the component return the message back to the router once it is done. Routing decisions are still centralized, making routing logic simple to maintain, and the route the message takes can be changed by the router at any time based on the outcomes of message processing.

In Java CAPS, a Routing Slip, or return destination, can be attached to the message in one of two ways. It can be passed via JMS user-defined properties if the message is passed from component to component over JMS, or the message can be packaged into an Envelope Wrapper and the Routing Slip can be incorporated into the Envelope metadata.

Envelope Wrapper is discussed at length in Chapter 8, "Message Transformation," section 8.2. The route, computed by the router, could be represented in the Envelope node as a series of labels delimited by some delimiter, or an ordered, repeating collection of labels.

Chapter 4, section 4.3, "Routing Slip," in Part II, illustrates this discussion with an example implementation of a Routing Slip pattern using Java Collaborations and JMS.

We could have used an eInsight Business Process to implement the kind of functionality that the Routing Slip facilitates. Each processing component could have been implemented as a New Web Service Java Collaboration or an eInsight subprocess. The Business Process would orchestrate execution of these components according to routing logic rules it implements.

6.13 Process Manager

One of the Routing Slip solutions involves a single routing component that determines the next component to which a message must be sent and receives the message back once the component is finished with it. This central routing component directs flow of messages using routing logic it embeds. Since it always receives messages that are processed by the processing components, it can modify the route a message is to take based on the outcome of processing by a particular component. Thus, the route the message finally takes may be different from the route a fixed router, which does not use intermediate processing results for routing decisions, would have determined.

Process Manager is a component that implements conditional routing logic and orchestrates execution of other processing components. Java CAPS supports implementation of the Process Manager, with functionality as described in the opening paragraph, as a Java Collaboration using JMS, possibly in Request/Reply mode, to dispatch and receive messages to and from processing components. The disadvantage of this approach is that the routing logic is hidden away in the Java code and, depending on the size and complexity of logic involved, may be difficult to understand.

Java CAPS eInsight Business Process Manager provides a graphical Business Process modeling environment. It overcomes the understandability limitations of a Java-only implementation and offers a number of features for Business Process modeling, component orchestration, and runtime monitoring that are not available with Java-only implementations.

Using eInsight Business Process Manager, you can implement any desired routing and component orchestration solutions. eInsight examples appear in most sections of this book and illustrate all manner of solutions of varying complexity. When a dynamic routing or component orchestration is required, eInsight Business Process can be developed to satisfy the requirement.

6.14 Message Broker

Message Broker [EIP] is an EAI architectural style wherein a component of a messaging system implements centralized routing for all messages flowing through the system. [EIP] also uses the term hub-and-spoke when referring to this architectural style. SeeBeyond's DataGate 3.6 product, predecessor to eGate 4.x, ICAN 5.0, and Java CAPS 5.1, is a Message Broker–based EAI package.

Message Broker architecture allows decoupling of senders from receivers. The senders need not know where the messages are going, and receivers need not know from where the messages are coming. The Message Broker embeds all routing logic necessary to get messages from senders to receivers. This centralizes routing logic maintenance.

In Java CAPS, and ICAN before it, each connectivity map could be considered to represent a Message Broker–based solution. In effect, all collaborations and Business Processes present in a connectivity map route messages from sources to destinations. Each eInsight Business Process could also be considered a Message Broker implementation, as it, too, makes routing decisions when orchestrating a series of activities. Collections of connectivity maps sharing common channels could be considered hierarchies of Message Brokers [EIP].

While Java CAPS can certainly be used to implement centralized routing solutions in the spirit of Message Broker, doing so does not appear particularly necessary or particularly advantageous.

6.15 Chapter Summary

This chapter discussed [EIP] message routing and message routing–related patterns. It included discussion and application of patterns from [EIP] Messaging Systems and Message Routing.

The chapter briefly discussed where a Java CAPS solution developer can make routing decisions and discussed each of the routing patterns, specifically Splitter, Aggregator, Resequencer, Scatter-Gather, Routing Slip, Process Manager, and Message Broker.

Build better, more effective, cooler desktop applications
that intensify the user experience

IMAGE PROCESSING

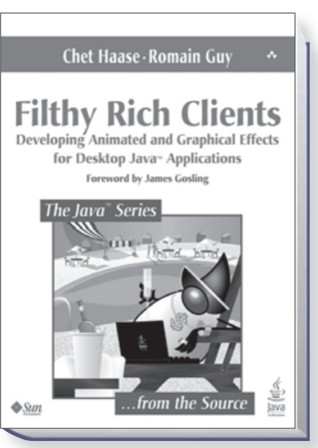
FROM

FILTHY RICH CLIENTS

Developing Animated and Graphical Effects for Desktop Java Applications

by Chet Haase and Romain Guy

©2008 | 608 PAGES | ISBN: 0-132-41393-0



ALSO AVAILABLE

- SAFARI BOOKS ONLINE
- E-BOOK: 0132345366

TABLE OF CONTENTS

PART I: GRAPHICS AND GUI FUNDAMENTALS

Chapter 1: Desktop Java Graphics APIs: Swing, AWT, and Java 2D

Chapter 2: Swing Rendering Fundamentals

Chapter 3: Graphics Fundamentals

Chapter 4: Images

Chapter 5: Performance

PART II: ADVANCED GRAPHICS RENDERING

Chapter 6: Composites

Chapter 7: Gradients

Chapter 8: Image Processing

Chapter 9: Glass Pane

Chapter 10: Layered Panes

Chapter 11: Repaint Manager

PART III: ANIMATION

Chapter 12: Animation Fundamentals

Chapter 13: Smooth Moves

Chapter 14: Timing Framework: Fundamentals

Chapter 15: Timing Framework: Advanced Features

PART IV: EFFECTS

Chapter 16: Static Effects

Chapter 17: Dynamic Effects

Chapter 18: Animated Transitions

Chapter 19: Birth of a Filthy Rich Client

FOR MORE INFORMATION

informit.com/title/9780132413930

Addison
Wesley

8

Image Processing

IMAGE-PROCESSING tools such as Adobe Photoshop and The GIMP offer a wide variety of filters you can apply on your pictures to create various special effects (see Figure 8-1). When you are designing a user interface, it is very tempting to use those effects. For instance, you could use a filter to blur an out-of-focus element in the UI. You could also increase the brightness of an image as the user moves the mouse over a component.

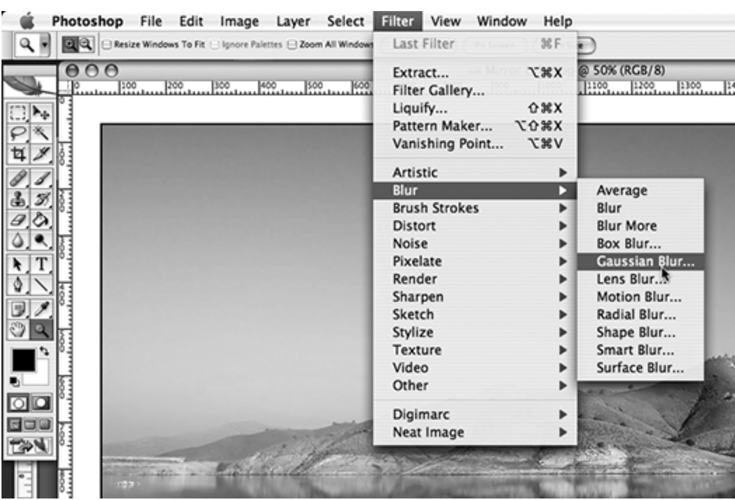


Figure 8-1 Applications like Adobe Photoshop have advanced image-processing capabilities.

Image Filters

Despite the impressive-looking results, image processing is not a difficult task to implement. Processing an image, or applying a filter, is just a matter of calculating a new color for each pixel of a source image. The information required to compute the new pixels varies greatly from one filter to another. Some filters, a grayscale filter for instance, need only the current color of a pixel; other filters, such as a sharpening filter, may also need the color of the surrounding pixels; still other filters, such as a rotation filter, may need additional parameters.

Since the introduction of Java 2D in J2SE 1.2, Java programmers have access to a straightforward image-processing model. You might have learned or read about the old producer-consumer model of Java 1.1. If you did, forget everything you know about it because the new model is much easier and more versatile. Java 2D's image-processing model revolves around the `java.awt.image.BufferedImage` class and the `java.awt.image.BufferedImageOp` interface.

A `BufferedImageOp` implementation takes a `BufferedImage` as input, called the source, and outputs another `BufferedImage`, called the destination, which is altered according to specific rules. Figure 8-2 shows how a blur filter produces the final image.

While the JDK does not offer concrete image filters, it does provide the foundations for you to create your own. If you need a sharpening or blurring filter, for example, you must know how to provide parameters to a `ConvolveOp` filter. We teach you such techniques in this chapter. Before we delve further into image-processing theory, let's see how we can use a `BufferedImageOp` to process an image.

Figure 8-2 Filtering an image with Java 2D.

Processing an Image with BufferedImageOp

Filtering a `BufferedImage` can be done onscreen at painting time or offscreen. In both cases, you need a source image and an operation, an instance of `BufferedImageOp`. Processing the image at painting time is the easiest approach; here is how you might do it:

```
// createImageOp returns a useful image filter
BufferedImageOp op = createImageOp();
// loadSourceImage returns a valid image
BufferedImage sourceImage = loadSourceImage();

@Override
protected void paintComponent(Graphics g) {
  Graphics2D g2 = (Graphics2D) g;
  // Filter the image with a BufferedImageOp, then draw it
  g2.drawImage(sourceImage, op, 0, 0);
}
```

You can filter an image at painting time by invoking the `drawImage(BufferedImage, BufferedImageOp, int, int)` method in `Graphics2D` that filters the source image and draws it at the specified location.

Warning: Use Image Filters with Care. The `drawImage(BufferedImage, BufferedImageOp, int, int)` method is very convenient but often has poor runtime performance. An image filter is likely to perform at least a few operations for every pixel in the source image, which easily results in hundreds of thousands, or even millions, of operations on medium or large images. Besides, this method might have to create a temporary image, which takes time and memory. For every filter you want to use, you will have to see whether the runtime performance is acceptable or not.

Here is an example of how to preprocess an image by doing all the operations offscreen:

```
BufferedImageOp op = createImageOp();
BufferedImage sourceImage = loadSourceImage();
BufferedImage destination;

destination = op.filter(sourceImage, null);
```

Calling the `filter()` method on a `BufferedImageOp` triggers the processing of the source image and the generation of the destination image. The second parameter, set to null here, is actually the destination image, which, when set to null, tells the `filter()` method to create a new image of the appropriate size. You can, instead, pass a non-null `BufferedImage` object as this parameter to avoid creating a new one on each invocation. Doing so can save performance by reducing costly image creations.

The following code example shows how you can optimize a routine applying the same filter on several images of the same size:

```
BufferedImageOp op = createImageOp();
BufferedImage[] sourceImagesArray = loadImages();
BufferedImage destination = null;

for (BufferedImage sourceImage : sourceImagesArray) {
  // on the first pass, destination is null
  // so we need to retrieve the reference to
  // the newly created BufferedImage
  destination = op.filter(sourceImage, destination);
  saveImage(destination);
}
```

After the first pass in the loop, the destination will be non-null and `filter()` will not create a new `BufferedImage` when invoked. By doing so, we also make sure that the destination is in a format optimized for the filter, as it is created by the filter itself.

Processing an image with Java 2D is an easy task. No matter which method you choose, you will need to write only one line of code. But we haven't seen any concrete `BufferedImageOp` yet and have just used an imaginary `createImageOp()` method that was supposedly returning a useful filter. As of Java SE 6, the JDK contains five implementations of `BufferedImageOp` we can rely on to write our own filters: `AffineTransformOp`, `ColorConvertOp`, `ConvolveOp`, `LookupOp`, and `RescaleOp`.

You can also write your own implementation of `BufferedImageOp` from scratch if the JDK does not fulfill your needs. Before learning how to write your own, let's take a closer look at what the JDK has to offer. Each filter we investigate will be applied to the sample picture shown in Figure 8-3 to give you a better idea of the result.

The complete source code for all the examples can be found on this book's Web site in the project named `ImageOps`.

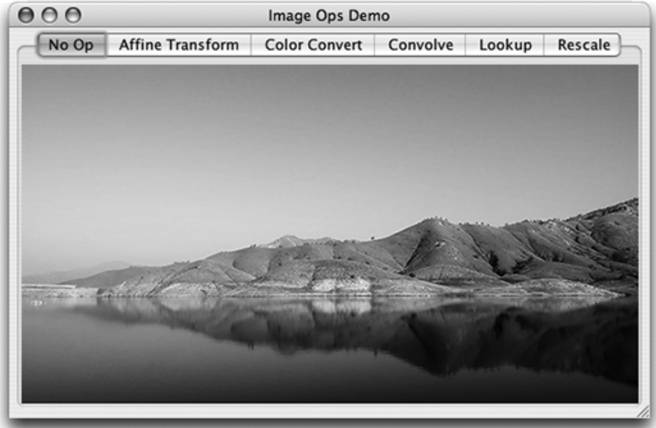


Figure 8-3 The image used in our filter examples.

AffineTransformOp

An `AffineTransformOp` is a geometry filter. It does not work on the actual color of the pixels but on the shape of the picture. As its name suggests, however, it is not meant to perform any kind of geometry transformation. Instead, it is limited to linear mapping from 2D coordinates in the source image to 2D coordinates in the destination image.

This kind of filter is created with an `AffineTransform` instance, which you should be familiar with if you have worked with the `Graphics2D` class (this class is also discussed in Chapter 3, “Graphics Fundamentals”). An `AffineTransform` can be used to rotate, scale, translate, and shear objects in a 2D space.

The following code illustrates how to divide the size of an image by two using an `AffineTransformOp`:

```
BufferedImage dstImage = null;
AffineTransform transform =
    AffineTransform.getScaleInstance(0.5, 0.5);
AffineTransformOp op = new AffineTransformOp(transform,
    AffineTransformOp.TYPE_BILINEAR);
dstImage = op.filter(sourceImage, null);
```

The `AffineTransformOp` constructor used in this example takes two parameters: an `AffineTransform`, in this case a scale operation of 50 percent on both axes,

Figure 8-4 The size of the original image is reduced by 50 percent with an `AffineTransformOp`.

and an interpolation type, which is equivalent to the interpolation rendering hint you can find in the `RenderingHints` class (see Chapter 3). You can also pass a `RenderingHints` instance instead of the interpolation type, in which case the interpolation rendering hint will be used.

Figure 8-4 shows the result of our scaling operation.

ColorConvertOp

This `BufferedImageOp` implementation performs a pixel-by-pixel color conversion of the source image into the destination image. This particular image-processing operation has an interesting feature: It transforms a given pixel from one color model to another. To do this, the filter needs the color value of only this single pixel, which means that it is possible to use the same image as both the source and the destination.

Converting an image from one color model to another has little practical use if you are not building an advanced imaging tool. And it is definitely useless if terms like “CMYK,” “sRGB,” and “Adobe RGB 1998 color profile” mean nothing to you. Color spaces are very useful, but describing them and their applications goes way beyond the scope of this book. Even so, we can use a `ColorConvertOp` to create something more basic and potentially useful to us: a grayscale version of a source image.

You first need to create a `ColorSpace` instance that represents the color model to which you want to convert your image. A `ColorSpace` can be instantiated by invoking `ColorSpace.getInstance(int)` and passing one of the five following constants:

```
ColorSpace.CS_CIEXYZ
ColorSpace.CS_GRAY
ColorSpace.CS_LINEAR_RGB
ColorSpace.CS_PYCC
ColorSpace.CS_sRGB
```

You might have already guessed which one is best suited to our purpose of performing a grayscale conversion:

```
BufferedImage dstImage = null;
ColorSpace colorSpace = ColorSpace.getInstance(
    ColorSpace.CS_GRAY);
ColorConvertOp op = new ColorConvertOp(colorSpace, null);
dstImage = op.filter(sourceImage, null);
```

Similar to `AffineTransformOp`, `ColorConvertOp` can use a set of `RenderingHints` to control the quality of the color conversion and the dithering. Figure 8-5 shows the result of the our color conversion.

Last but not least, it is important to know that performing such a conversion on an image may make it incompatible with your graphics display hardware, thus hurting the performance if you need to paint the filtered image to the Swing window.

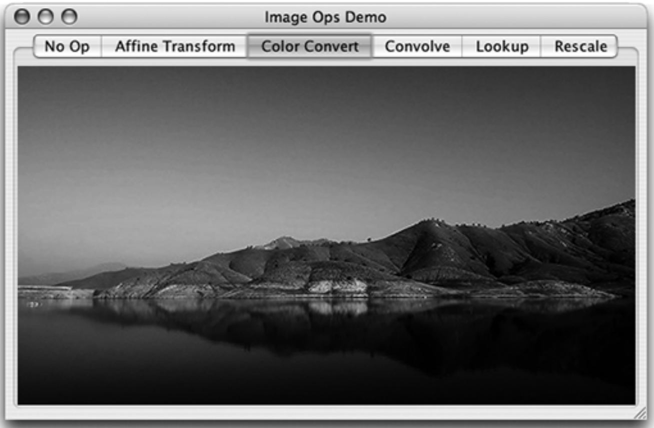


Figure 8-5 `ColorConvertOp` can be used to create a grayscale version of an image.

You may want to convert the image to be a compatible image instead for general usage. See Chapter 3 for more information.

ConvolveOp

The `ConvolveOp` is the most complicated `BufferedImageOp` but also the most versatile. It is the only `BufferedImageOp` you should master and know by heart. A `ConvolveOp` is used to perform a convolution from the source image to the destination. If you never took any math courses, or more likely, if you forgot everything about what you learned during those classes, a convolution is a spatial operation that computes the destination pixel by multiplying the source pixel and its neighbors by a convolution kernel.[1] Don't be frightened: You will soon understand what this gobbledygook means.

Any convolution operation relies on a convolution kernel, which is just a matrix of numbers. Here is an example:

$$kernel = \begin{bmatrix} 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \end{bmatrix}$$

The kernel defined here represents a 3×3 matrix of floating-point numbers. When you perform a convolution operation, this matrix is used as a sliding mask over the pixels of the source image. For instance, to compute the result of the convolution for a pixel located at the coordinates (x, y) in the source image, the center of the kernel is positioned at these coordinates. In the case of a 3×3 kernel, here are the coordinates, in the source image, of the pixels that each corresponding kernel value is applied to:

$$kernel\ coordinates = \begin{bmatrix} x-1, y-1 & x, y-1 & x+1, y-1 \\ x-1, y & x, y & x+1, y \\ x-1, y+1 & x, y+1 & x+1, y+1 \end{bmatrix}$$

1. And if this does not make sense to you, consider the mathematical definition from the Wikipedia: "Convolution is a mathematical operator which takes two functions, f and g, and produces a third function that in a sense represents the amount of overlap between f and a reversed and translated version of g." At least my version talks about pixels.

To compute the value of the destination pixel at (x, y), Java 2D multiplies the kernel values with their corresponding color values in the source image. Imagine a 3 × 3 white image with a single black pixel in its center, as suggested in Figure 8-6.

To convolve the black pixel with our 3 × 3 kernel, we must start by placing the matrix over the pixels, as shown in Figure 8-7.

255, 255, 255	255, 255, 255	255, 255, 255
255, 255, 255	0, 0, 0	255, 255, 255
255, 255, 255	255, 255, 255	255, 255, 255

Figure 8-6 A black pixel surrounded by white pixels. The numbers show the RGB value of each pixel.

255, 255, 255 x 1/9	255, 255, 255 x 1/9	255, 255, 255 x 1/9
255, 255, 255 x 1/9	0, 0, 0 x 1/9	255, 255, 255 x 1/9
255, 255, 255 x 1/9	255, 255, 255 x 1/9	255, 255, 255 x 1/9

Figure 8-7 Each color value is multiplied by the corresponding value of the kernel.

Now we can compute all the multiplications, add up the results, and get the color value of the destination pixel:

$$R = 255 \cdot \frac{1}{9} + 255 \cdot \frac{1}{9} + 255 \cdot \frac{1}{9} + 255 \cdot \frac{1}{9} + 0 \cdot \frac{1}{9} + 255 \cdot \frac{1}{9} + 255 \cdot \frac{1}{9} + 255 \cdot \frac{1}{9} + 255 \cdot \frac{1}{9} + 255 \cdot \frac{8}{9}$$

$$B = 255 \cdot \frac{1}{9} + 255 \cdot \frac{1}{9} + 255 \cdot \frac{1}{9} + 255 \cdot \frac{1}{9} + 0 \cdot \frac{1}{9} + 255 \cdot \frac{1}{9} + 255 \cdot \frac{1}{9} + 255 \cdot \frac{1}{9} + 255 \cdot \frac{1}{9} + 255 \cdot \frac{8}{9}$$

$$G = 255 \cdot \frac{1}{9} + 255 \cdot \frac{1}{9} + 255 \cdot \frac{1}{9} + 255 \cdot \frac{1}{9} + 0 \cdot \frac{1}{9} + 255 \cdot \frac{1}{9} + 255 \cdot \frac{1}{9} + 255 \cdot \frac{1}{9} + 255 \cdot \frac{1}{9} + 255 \cdot \frac{8}{9}$$

The destination pixel is therefore a light gray; its RGB value is (227, 227, 227), or #E3E3E3. By now, you might have guessed what this kernel does: It replaces each pixel by the average color of its surroundings. Such a convolution operation is commonly known as a blur. We discuss blurring filters in more detail in Chapter 16, “Static Effects.”

Constructing a Kernel

There are no particular restrictions about the size and contents of the kernels you can use with Java 2D. However, you should be aware of several important characteristics of kernels.

First, the values of a kernel should add up to 1.0 in the typical case, as in the previous example where all nine entries have the value 1/9. If these values do not add up to 1.0, the luminosity of the picture will not be preserved. This can, however, be turned to your advantage. For instance, you can increase the luminosity of a picture by 10 percent with a 1 × 1 kernel containing the value 1.1. Similarly, you can darken a picture by 10 percent with a 1 × 1 kernel containing the value 0.9. When dealing with larger kernels, the sum of the values defines the new luminosity. For instance, if the sum equals 0.5, then the luminosity will be cut in half. Keep that in mind when creating a kernel.

The size of a kernel defines the strength of a filter. For instance, a 3 × 3 blurring kernel produces a slightly blurry picture, whereas a 40 × 40 blurring kernel produces an indistinguishable blob from the original image.

The dimensions of the kernel are equally important. Kernels are usually odd-sided. While it is perfectly safe to use a 4 × 4 or a 12 × 12 kernel, it is not recommended. An even-sided kernel will not be centered over the source pixel and might give unbalanced visual results, which you should avoid. Also, it is easier for code readers to understand how an odd-sided kernel will behave. The Java 2D documentation defines the value of the matrix used as the center of the kernel

as being the one at the coordinates (w – 1)/2, (h – 1)/2. This definition makes it harder to know which value is used as the center.

Your kernels do not have to be square shaped. Vertical kernels, for example with a 1 × 5 matrix, and horizontal kernels, for example with a 5 × 1 matrix, can be used to apply effects that work in only one direction. Chapter 16 presents examples of such kernels.

Last but not least, avoid using large kernels. When convolving a picture with a 3 × 3 kernel, Java 2D performs at least 17 operations (9 multiplications and 8 additions) per color component per pixel. Convolving a 640 × 480 picture requires at least 640 × 480 × 3 × 17 = 15,667,200 operations! That's quite a lot.[2] And this number does not even include the operations of reading and writing the actual pixel values from and to the source and destination pictures. We therefore strongly advise you not to perform convolve operations at painting time. Instead perform the operations once prior to painting and cache the results instead.

No matter what kernel you create, writing the code to perform the convolution is simple:

```
BufferedImage dstImage = null;
float[] sharpen = new float[] {
     0.0f, -1.0f,  0.0f,
    -1.0f,  5.0f, -1.0f,
     0.0f, -1.0f,  0.0f
};
Kernel kernel = new Kernel(3, 3, sharpen);
ConvolveOp op = new ConvolveOp(kernel);
dstImage = op.filter(sourceImage, null);
```

In Java 2D, a kernel is an array of floats and two dimensions. In this case, we use a 3 × 3 sharpening kernel to create an array of nine floats and tell the `Kernel` class that we want this array to be treated as a 3 × 3 matrix.

Figure 8-8 shows the result of the convolution with the 3 × 3 sharpening kernel shown in the previous code example.

Working on the Edge

Everything is not perfect yet. Take a close look at the generated result: You should see a black border surrounding the picture. During the convolve operation, Java

2. Even with today's CPU, it's still a lot. Really. And we are talking about convolving a small picture with a small kernel.

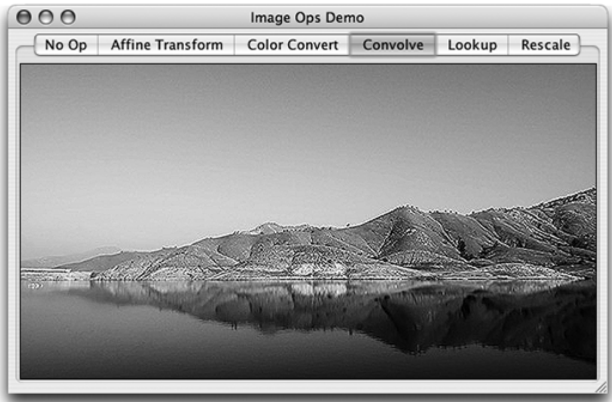


Figure 8-8 The sharpened picture shows enhanced details.

2D always matches the center of the kernel with one pixel of the source image. This works well for every pixel except the ones on the edges of the picture. Try to line up a 3×3 kernel with any pixel on the edge of an image and you will see that some parts of the kernel lie outside of the image. To work around this problem, Java 2D replaces the pixels it cannot compute with black pixels, which results in darkened edges because of the extra black introduced into the convolve operations for these edge pixels. To avoid this result, you can instruct Java 2D to do nothing and to keep the original color:

```
// the default is ConvolveOp.EDGE_ZERO_FILL
// the last parameter is the RenderingHints set
ConvolveOp op = new ConvolveOp(kernel,
    ConvolveOp.EDGE_NO_OP, null);
```

Unfortunately, neither of these solutions generates good-looking results. To get rid of any problem on the edges, you can simply increase the size of the original picture, as follows:

```
int kernelWidth = 3;
int kernelHeight = 3;

int xOffset = (kernelWidth - 1) / 2;
int yOffset = (kernelHeight - 1) / 2;

BufferedImage newSource = new BufferedImage(
  sourceImage.getWidth() + kernelWidth - 1,
  sourceImage.getHeight() + kernelHeight - 1,
  BufferedImage.TYPE_INT_ARGB);
Graphics2D g2 = newSource.createGraphics();
```

```
g2.drawImage(sourceImage, xOffset, yOffset, null);
g2.dispose();

ConvolveOp op = new ConvolveOp(kernel,
    ConvolveOp.EDGE_NO_OP, null);
dstImage = op.filter(newSource, null);
```

The original image is drawn centered into a new, larger, transparent image. Because we added enough transparent pixels on each side of the original image, the convolution operation will not affect the pixels of the original image. It is important to use the `ConvolveOp.EDGE_NO_OP` edge condition so you will keep the pixels transparent around the image. This technique of adding transparent pixels on the sides provides better-looking results, but you have to take the extraneous pixels into account.

LookupOp

A `LookupOp` maps the color values of the source to new color values in the destination. This operation is achieved with a lookup table that contains the destination values for each possible source value.

Lookup operations can be used to generate several common filters, such as negative filters, posterizing filters, and thresholding filters. Negative filters are interesting because they help illustrate how lookup tables work. Pixel colors are usually represented using three components (red, green, and blue) stored in 8 bits each. As a result, the color values of a negative image are the 8 bits' complements of the source image color values:

```
dstR = 255 - srcR;
dstG = 255 - srcG;
dstB = 255 - srcB;
```

To apply such a conversion to the source image, you must create a lookup table that associates all the values in the 8 bits range (from 0 to 255) to their complements:

```
short[] data = new short[256];
for (short i = 0; i < 256; i++) {
    data[i] = 255 - i;
}

BufferedImage dstImage = null;
LookupTable lookupTable = new ShortLookupTable(0, data);
LookupOp op = new LookupOp(lookupTable, null);
dstImage = op.filter(sourceImage, null);
```

Figure 8-9 shows the result of this negative filter.

The `LookupTable` from this example contains only one lookup array, used for all of the color components of the source image, resulting in the same conversion of all of the color components.

To perform a different conversion for each component, you simply need to create one lookup array per color component in the source image. Since the example relies on an RGB picture, we can create a filter that inverts only the red component by defining three lookup arrays:

```
short[] red = new short[256];
short[] green = new short[256];
short[] blue = new short[256];

for (short i = 0; i < 256; i++) {
    red[i] = 255 - i;
    green[i] = blue[i] = i;
}

short[][] data = new short[][] {
    red, green, blue
};

BufferedImage dstImage;
LookupTable lookupTable = new ShortLookupTable(0, data);
dstImage = op.filter(sourceImage, null);
LookupOp op = new LookupOp(lookupTable, null);
```

Figure 8-9 A simple lookup operation can be used to produce a negative image.

You do not need to provide a lookup array for the alpha channel of your picture, if present. In this case, Java 2D will simply preserve the original alpha values. Whenever you create a new `LookupOp`, ensure that the number and size of your lookup arrays match the source image structure.

RescaleOp

`RescaleOp` does not scale the size of an image as you would expect it to. Instead, `RescaleOp` performs a rescaling operation by multiplying the color value of each pixel in the source image by a scale factor and then adding an offset. Here is the formula applied to each color component of the source pixels:

```
dstR = (srcR * scaleFactor) + offset
dstG = (srcG * scaleFactor) + offset
dstB = (srcB * scaleFactor) + offset
```

Rescaling operations can be used to brighten, darken, or tint images. The following code example increases the overall brightness of the picture by 10 percent:

```
BufferedImage dstImage = null;
RescaleOp op = new RescaleOp(1.1f, 0.0f, null);
dstImage = op.filter(sourceImage, null);
```

The first two parameters of the `RescaleOp` constructor are respectively the scale factor and the offset. Note that a `RescaleOp` with an offset of 0 is no different from a `ConvolveOp` with a 1×1 kernel. You can also adjust each color component independently:

```
BufferedImage dstImage = null;
float[] factors = new float[] {
    1.4f, 1.4f, 1.4f
};
float[] offsets = new float[] {
    0.0f, 0.0f, 30.0f
};
RescaleOp op = new RescaleOp(factors, offsets, null);
dstImage = op.filter(sourceImage, null);
```

In this case, the overall brightness is increased by 40 percent, and all of the pixel colors are shifted toward the blue color. The offset of 30 increases the blue component of each pixel by 12 percent (30/256). Remember, the offset is added to the color value and must therefore be a value between 0 and 255, as opposed to the scale factor, which acts as a percentage.

Figure 8-10 The image is brighter and the blues are bluer after processing.

Figure 8-10 shows the result produced by a `RescaleOp` with a scale factor of 1.4 for each component and an offset of 30 for the blue component.

Just as in `LookupOp`, the number of values used in the scale factors and offset arrays depend on the number of components in the source image. Working on `TYPE_INT_RGB` or `TYPE_INT_ARGB` pictures is therefore easier than working on other types of `BufferedImage`. When the source image contains an alpha channel, you do not need to specify a factor and an offset for the alpha component. Java 2D automatically preserves the original values.

These five `BufferedImageOps` will probably be all you need for most situations. Nevertheless, you might want to create your own specialized `BufferedImageOp` to create advanced graphical effects.

Custom BufferedImageOp

Creating a new filter from scratch is not a very complicated task. To prove it, we show you how to implement a color tint filter. This kind of filter can be used to mimic the effect of the colored filters photographers screw in front of their lenses. For instance, an orange color tint filter gives a sunset mood to a scene, while a blue filter cools down the tones in the picture.

You first need to create a new class that implements the `BufferedImageOp` interface and its five methods. To make the creation of several filters easier, we first

define a new abstract class entitled `AbstractFilter`. As you will soon discover, all filters based on this class are nonspatial, linear color filters. That means that they will not affect the geometry of the source image and that they assume the destination image has the same size as the source image.

The complete source code of our custom `BufferedImage` is available on this book's Web site in the project entitled `CustomImageOp`.

Base Filter Class

`AbstractFilter` implements all the methods from `BufferedImageOp` except for `filter()`, which actually processes the source image into the destination and hence belongs in the subclasses:

```
public abstract class AbstractFilter
    implements BufferedImageOp {
  public abstract BufferedImage filter(
    BufferedImage src, BufferedImage dest);

  public Rectangle2D getBounds2D(BufferedImage src) {
    return new Rectangle(0, 0, src.getWidth(),
                         src.getHeight());
  }

  public BufferedImage createCompatibleDestImage(
    BufferedImage src, ColorModel destCM) {
    if (destCM == null) {
        destCM = src.getColorModel();
    }

    return new BufferedImage(destCM,
        destCM.createCompatibleWritableRaster(
          src.getWidth(), src.getHeight()),
        destCM.isAlphaPremultiplied(), null);
  }

  public Point2D getPoint2D(Point2D srcPt,
                            Point2D dstPt) {
    return (Point2D) srcPt.clone();
  }

  public RenderingHints getRenderingHints() {
    return null;
  }
}
```

The `getRenderingHints()` method must return a set of `RenderingHints` when the image filter relies on rendering hints. Since this will probably not be the case for our custom filters, the abstract class simply returns null.

The two methods `getBounds2D()` and `getPoint2D()` are very important for spatial filters, such as `AffineTransformOp`. The first method, `getBounds2D()`, returns the bounding box of the filtered image. If your custom filter modifies the dimension of the source image, you must implement this method accordingly. The implementation proposed here makes the assumption that the filtered image will have the same size as the source image.

The other method, `getPoint2D()`, returns the corresponding destination point given a location in the source image. As for `getBounds2D()`, `AbstractFilter` makes the assumption that no geometry transformation will be applied to the image, and the returned location is therefore the source location.

`AbstractFilter` also assumes that the only data needed to compute the pixel for (x, y) in the destination is the pixel for (x, y) in the source.

The last implemented method is `createCompatibleDestImage()`. Its role is to produce an image with the correct size and number of color components to contain the filtered image. The implementation shown in the previous source code creates an empty clone of the source image; it has the same size and the same color model regardless of the source image type.

Color Tint Filter

The color tint filter, cleverly named `ColorTintFilter`, extends `AbstractFilter` and implements `filter()`, the only method left from the `BufferedImageOp` interface. Before we delve into the source code, we must first define the operation that the filter will perform on the source image. A color tint filter mixes every pixel from the source image with a given color. The strength of the mix is defined by a mix value. A mix value of 0 means that all of the pixels remain the same, whereas a mix value of 1 means that all of the source pixels are replaced by the tinting color. Given those two parameters, a color and a mix percentage, we can compute the color value of the destination pixels:

```
dstR = srcR * (1 – mixValue) + mixR * mixValue
dstG = srcG * (1 – mixValue) + mixG * mixValue
dstB = srcB * (1 – mixValue) + mixB * mixValue
```

If you tint a picture with 40 percent white, the filter will retain 60 percent (1 or 1 – `mixValue`) of the source pixel color values to preserve the overall luminosity of the picture.

The following source code shows the skeleton of `ColorTintFilter`, an immutable class.

Note: Immutability. It is very important to ensure that your filters are immutable to avoid any problem during the processing of the source images. Imagine what havoc a thread could cause by modifying one of the parameters of the filter while another thread is filtering an image. Rather than synchronizing code blocks or spending hours in a debugger, go the easy route and make your `BufferedImageOp` implementations immutable.

```
public class ColorTintFilter extends AbstractFilter {
  private final Color mixColor;
  private final float mixValue;

  public ColorTintFilter(Color mixColor, float mixValue) {
    if (mixColor == null) {
      throw new IllegalArgumentException(
          "mixColor cannot be null");
    }

    this.mixColor = mixColor;
    if (mixValue < 0.0f) {
      mixValue = 0.0f;
    } else if (mixValue > 1.0f) {
      mixValue = 1.0f;
    }
    this.mixValue = mixValue;
  }

  public float getMixValue() {
    return mixValue;
  }

  public Color getMixColor() {
    return mixColor;
  }

  @Override
  public BufferedImage filter(BufferedImage src,
                              BufferedImage dst) {
    // filters src into dst
  }
}
```

The most interesting part of this class is the implementation of the `filter()` method:

```
@Override
public BufferedImage filter(BufferedImage src,
                            BufferedImage dst) {
  if (dst == null) {
      dst = createCompatibleDestImage(src, null);
  }

  int width = src.getWidth();
  int height = src.getHeight();

  int[] pixels = new int[width * height];
  GraphicsUtilities.getPixels(src, 0, 0, width,
                              height, pixels);
  mixColor(pixels);
  GraphicsUtilities.setPixels(dst, 0, 0, width,
                              height, pixels);

  return dst;
}
```

The first few lines of this method create an acceptable destination image when the caller provides none. The `javadoc` of the `BufferedImageOp` interface dictates this behavior: "If the destination image is null, a `BufferedImage` with an appropriate `ColorModel` is created."

Instead of working directly on the source and destination images, the color tint filter reads all the pixels of the source image into an array of integers. The implications are threefold. First, all of the color values are stored on four ARGB 8-bit components packed as an integer. Then, the source and the destination can be the same, since all work will be performed on the array of integers. Finally, despite the increased memory usage, it is faster to perform one read and one write operation on the images rather than reading and writing pixel by pixel. Before we take a closer look at `mixColor()`, where the bulk of the work is done, here is the code used to read all the pixels at once into a single array of integers:

```
public static int[] getPixels(BufferedImage img,
                              int x, int y,
                              int w, int h,
                              int[] pixels) {
  if (w == 0 || h == 0) {
    return new int[0];
  }
```

```
    if (pixels == null) {
      pixels = new int[w * h];
    } else if (pixels.length < w * h) {
      throw new IllegalArgumentException(
          "pixels array must have a length >= w*h");
    }

    int imageType = img.getType();
    if (imageType == BufferedImage.TYPE_INT_ARGB ||
        imageType == BufferedImage.TYPE_INT_RGB) {
        Raster raster = img.getRaster();
        return (int[]) raster.getDataElements(x, y, w, h, pixels);
    }

    return img.getRGB(x, y, w, h, pixels, 0, w);
  }
```

There are two different code paths, depending on the nature of the image from which the pixels are read. When the image is of type `INT_ARGB` or `INT_RGB`, we know for sure that the data elements composing the image are integers. We can therefore call `Raster.getDataElements()` and cast the result to an array of integers. This solution is not only fast but preserves all the optimizations of managed images performed by Java 2D.

When the image is of another type, for instance `TYPE_3BYTE_BGR`, as is often the case with JPEG pictures loaded from disk, the pixels are read by calling the `BufferedImage.getRGB(int, int, int, int, int[], int, int)` method. This invocation has two major problems. First, it needs to convert all the data elements into integers, which can take quite some time for large images. Second, it throws away all the optimizations made by Java 2D, resulting in slower painting operations, for instance. The picture is then said to be unmanaged. To learn more details about managed images, please refer to the Chapter 5, "Performance."

Note: Performance and `getRGB()`. The class `BufferedImage` offers two variants of the `getRGB()` method. The one discussed previously has the following signature:

```
int[] getRGB(int startX, int startY, int w, int h,
             int[] rgbArray, int offset, int scansize)
```

This method is used to retrieve an array of pixels at once, and invoking it will punt the optimizations made by Java 2D. Consider the second variant of `getRGB()`:

```
int getRGB(int x, int y)
```

This method is used to retrieve a single pixel and does not throw away the optimizations made by Java 2D. Be very careful about which one of these methods you decide to use.

The `setPixels()` method is very similar to `getPixels()`:

```
public static void setPixels(BufferedImage img,
                             int x, int y,
                             int w, int h,
                             int[] pixels) {
  if (pixels == null || w == 0 || h == 0) {
      return;
  } else if (pixels.length < w * h) {
    throw new IllegalArgumentException(
        "pixels array must have a length >= w*h");
  }

  int imageType = img.getType();
  if (imageType == BufferedImage.TYPE_INT_ARGB ||
      imageType == BufferedImage.TYPE_INT_RGB) {
      WritableRaster raster = img.getRaster();
      raster.setDataElements(x, y, w, h, pixels);
  } else {
      img.setRGB(x, y, w, h, pixels, 0, w);
  }
}
```

Performance Tip: Working on a `TYPE_INT_RGB` or `TYPE_INT_ARGB` results in better performance, since no type conversion is required to store the processed pixels into the destination image.

Reading and writing pixels from and to images would be completely useless if we did not process them in between operations. The implementation of the color tint equations is straightforward:

```
private void mixColor(int[] inPixels) {
  int mix_a = mixColor.getAlpha();
  int mix_r = mixColor.getRed();
  int mix_b = mixColor.getBlue();
  int mix_g = mixColor.getGreen();

  for (int i = 0; i < inPixels.length; i++) {
      int argb = inPixels[i];

      int a = argb & 0xFF000000;
      int r = (argb >> 16) & 0xFF;
      int g = (argb >>  8) & 0xFF;
      int b = (argb      ) & 0xFF;
```

```
            r = (int) (r * (1.0f - mixValue) + mix_r * mixValue);
            g = (int) (g * (1.0f - mixValue) + mix_g * mixValue);
            b = (int) (b * (1.0f - mixValue) + mix_b * mixValue);

            inPixels[i] = a << 24 | r << 16 | g << 8 | b;
        }
    }
```

Before applying the equations, we must split the pixels into their four color components. Some bit shifting and masking is all you need in this situation. Once each color component has been filtered, the destination pixel is computed by packing the four modified color components into a single integer. Figure 8-11 shows a picture tinted with 50 percent red.

The above implementation works well but can be vastly improved performance-wise. The `ColorTintFilter` class in the `CustomImageOp` project on this book's Web site offers a better implementation that uses a few tricks to avoid doing all of the computations in the loop.

Note: As an exercise, you can try to improve this implementation on your own before looking at the final version. (Hint: You can use lookup arrays.)

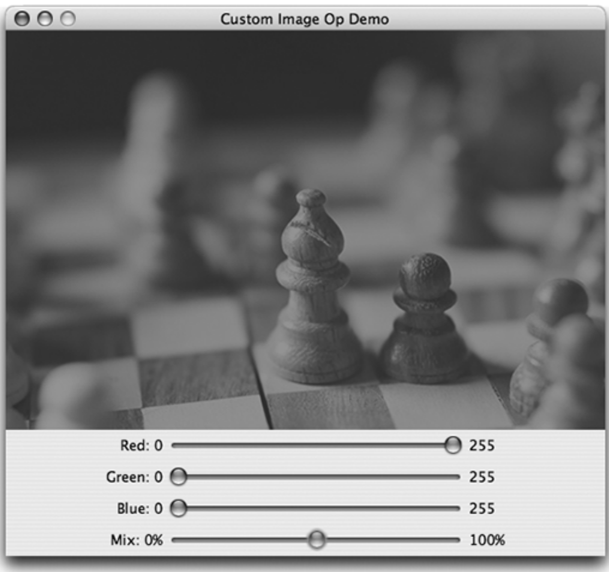


Figure 8-11 A red-tinted picture.

A Note about Filters Performance

Image filters perform a lot of operations on images, and performance can easily degrade if you do not pay attention to a few details. Whenever you write a filter assuming the source image will be of type `INT_RGB` or `INT_ARGB`, make sure the source image is actually of that type.

Usually, compatible images (images created with `GraphicsConfiguration.createCompatibleImage()`), which are designed to be in the same format as the screen, are stored as integers. It is often the case that the user's display is in 32-bit format and not the older 8-, 16-, and 24-bit formats. Therefore, it is a good idea to always load your images as compatible images.

The `CustomImageOp` demo loads a JPEG picture, which would normally be of type `3BYTE_BGR`, and turns it into a compatible image of type `INT_RGB`. You can look for the call to `GraphicsUtilities.loadCompatibleImage()` in the source code of the demo and replace it with `ImageIO.read()` to see the difference when moving the sliders of the user interface. As a rule of thumb, do not hesitate to use the various methods from the `GraphicsUtilities` class to always use compatible images.

Summary

Java 2D offers several powerful facilities to perform image processing on your pictures. The built-in `BufferedImageOp` implementations let you write your own custom filters very quickly. And if you need more flexibility, you can even create a new `BufferedImageOp` implementation from scratch.

The definitive guide to the advanced features and capabilities provided by servlets and JSP

TAG LIBRARIES: THE BASICS

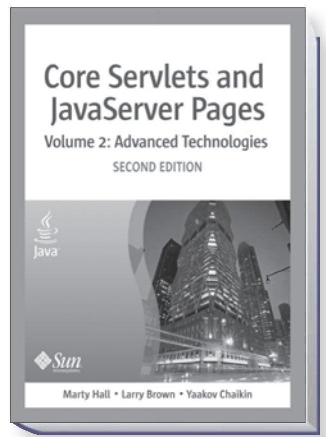
FROM

CORE SERVLETS AND JAVASERVER PAGES, Volume 2

Advanced Technologies, Second Edition

by Marty Hall, Larry Brown, and Yaakov Chaikin

©2008 | 736 PAGES | ISBN: 0-13-148260-2



ALSO AVAILABLE

- SAFARI BOOKS ONLINE
- E-BOOK: 013813765X
- MOBI POCKET: 0136041906
- SONY READER: 013500148X

TABLE OF CONTENTS

Chapter 1:
Using And Deploying Web Applications

Chapter 2:
Controlling Web Application Behavior With Web.xml

Chapter 3:
Declarative Security

Chapter 4:
Programmatic Security

Chapter 5:
Servlet And Jsp Filters

Chapter 6:
The Application Events Framework

Chapter 7:
Tag Libraries: The Basics

Chapter 8:
Tag Libraries: Advanced Features

Chapter 9:
Jsp Standard Tag Library (Jstl)

Chapter 10:
The Struts Framework: Basics

Chapter 11:
The Struts Framework: Doing More

Chapter 12:
The Struts Framework: Validating User Input

Appendix:
Developing Applications With Apache A

FOR MORE INFORMATION

informit.com/title/9780131482609

PRENTICE HALL

Chapter 7

As discussed in Volume 1 (Section 11.2) of *Core Servlets and JavaServer Pages*, you have many options when it comes to generating dynamic content inside the JSP page. These options are as follows:

- Scripting elements calling servlet code directly
- Scripting elements calling servlet code indirectly (by means of utility classes)
- Beans
- Servlet/JSP combo (MVC)
- MVC with JSP expression language
- Custom tags

The options at the top of the list are much simpler to use and are just as legitimate as the options at the bottom of the list. However, industry has adopted a best practice to avoid placing Java code inside the JSP page. This best practice stems from it being much harder to debug and maintain Java code inside the JSP page. In addition, JSP pages should concentrate only on the presentation logic. Introducing Java code into the JSP page tends to divert its purpose and, inevitably, business logic starts to creep in. To enforce this best practice, version 2.4 of the servlet specification went so far as to provide a way to disable any type of JSP scripting for a group of JSP pages. We discuss how to disable scripting in Section 2.14 (Configuring JSP Pages).

That said, there are cases where the presentation logic itself is quite complex and using the non-Java code options in the JSP page to express that logic becomes either too clunky and unreadable or, sometimes, just impossible to achieve. This is where

logic through the familiar HTML-like structures.

This chapter discusses how to create and use custom tags utilizing the new `SimpleTag` API, which was introduced in version 2.4 of the servlet specification. As its name suggests, `SimpleTag` API is very easy to use in comparison to its predecessor, now known as the classic tag API.

Although the `SimpleTag` API completely replaces the classic tag API, you should keep in mind that it works only in containers compliant with servlet specification 2.4 and above. Because there are still a lot of applications running on servlet 2.3-compliant containers, you should consider avoiding the `SimpleTag` API if you are not sure what type of container your code will end up on.

7.1 Tag Library Components

To use custom JSP tags, you need to define three separate components:

- The tag handler class that defines the tag's behavior
- The TLD file that maps the XML element names to the tag implementations
- The JSP file that uses the tag library

The rest of this section gives an overview of each of these components, and the following sections give details on how to build these components for various styles of tags. Most people find that the first tag they write is the hardest—the difficulty being in knowing where each component should go, not in writing the components. So, we suggest that you start by just downloading the simplest of the examples of this chapter from http://volume2.coreservlets.com/ and getting those examples to work on your machine. After that, you can move on and try creating some of your own tags.

The Tag Handler Class

When defining a new tag, your first task is to define a Java class that tells the system what to do when it sees the tag. This class must implement the`SimpleTag` interface. In practice, you extend `SimpleTagSupport`, which implements the `SimpleTag` interface and supplies standard implementations for some of its methods. Both the `SimpleTag` interface and the `SimpleTagSupport` class reside in the `javax.servlet.jsp.tagext` package.

The very first action the container takes after loading the tag handler class is instantiating it with its no-arg constructor. This means that every tag handler must have a no-arg constructor or its instantiation will fail. Remember that the Java compiler provides one for you automatically unless you define a constructor with arguments. In that case, be sure to define a no-arg constructor yourself.

The code that does the actual work of the tag goes inside the `doTag` method. Usually, this code outputs content to the JSP page by invoking the `print` method of the `JspWriter` class. To obtain an instance of the `JstWriter` class you call `getJspContext().getOut()` inside the `doTag` method. The `doTag` method is called at request time. It's important to note that, unlike the classic tag model, the `SimpleTag` model never reuses tag handler instances. In fact, a new instance of the tag handler class is created for every tag occurrence on the page. This alleviates worries about race conditions and cached values even if you use instance variables in the tag handler class.

You place the compiled tag handler in the same location you would place a regular servlet, inside the WEB-INF/classes directory, keeping the package structure intact. For example, if your tag handler class belongs to the `mytags` package and its class name is `MyTag`, you would place theMyTag.class file inside the WEB-INF/classes/mytags/ directory.

Listing 7.1 shows an example of a tag handler class.

Listing 7.1 Example Tag Handler Class

```
package somepackage;

import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.*;

public class ExampleTag extends SimpleTagSupport {
  public void doTag() throws JspException, IOException {
    JspWriter out = getJspContext().getOut();
    out.print("<b>Hello World!</b>");
  }
}
```

The Tag Library Descriptor File

Once you have defined a tag handler, your next task is to identify this class to the server and to associate it with a particular XML tag name. This task is accomplished by means of a TLD file in XML format. This file contains some fixed information (e.g., XML Schema instance declaration), an arbitrary short name for your library, a short description, and a series of tag descriptions. Listing 7.2 shows an example TLD file.

Listing 7.2 Example Tag Library Descriptor File

```
<?xml version="1.0" encoding="UTF-8" ?>
<taglib xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
  http://java.sun.com/xml/ns/j2ee/web-jsptaglibrary_2_0.xsd"
  version="2.0">
  <tlib-version>1.0</tlib-version>
  <short-name>csajsp-taglib</short-name>
  <tag>
    <description>Example tag</description>
    <name>example</name>
    <tag-class>package.TagHandlerClass</tag-class>
    <body-content>empty</body-content>
  </tag>
</taglib>
```

We describe the details of the contents of the TLD file in later sections. For now, just note that the `tag` element through the following subelements in their required order defines the custom tag.

- **`description`.** This optional element allows the tag developer to document the purpose of the custom tag.
- **`name`.** This required element defines the name of the tag as it will be referred to by the JSP page (really tag suffix, as will be seen shortly).
- **`tag-class`.** This required element identifies the fully qualified name of the implementing tag handler class.
- **`body-content`.** This required element tells the container how to treat the content between the beginning and ending occurrence of the tag, if any. The value that appears here can be either `empty`, `scriptless`, `tagdependent`, or `JSP`.

  The value of `empty` means that no content is allowed to appear in the body of the tag. This would mean that the declared tag can only appear in the form:

  ```
  <prefix:tag/>
  ```

  or

  ```
  <prefix:tag></prefix:tag>
  ```

  (without any spaces between the opening and closing tags). Placing any content inside the tag body would generate a page translation error.

The value of `scriptless` means that the tag body is allowed to have JSP content as long as it doesn't contain any scripting elements like `<% ... %>` or `<%= ... %>`. If present, the body of the tag would be processed just like any other JSP content.

The value of `tagdependent` means that the tag is allowed to have any type of content as its body. However, this content is not processed at all and completely ignored. It is up to the developer of the tag handler to get access to that content and do something with it. For example, if you wanted to develop a tag that would allow the JSP page developer to execute an SQL statement, providing the SQL in the body of the tag, you would use `tagdependent` as the value of the `body-content` element.

Finally, the value of `JSP` is provided for backward compatibility with the classic custom tag model. It is not a legal value when used with the `SimpleTag` API.

Note that there is no legal way of allowing any scripting elements to appear as the tag body under the new `SimpleTag` API model.

Core Warning

When using the `SimpleTag` API, it is illegal to include scripting elements in the body of the tag.

The TLD file must be placed inside the WEB-INF directory or any subdirectory thereof.

Core Note

The TLD file must be placed inside the WEB-INF directory or a subdirectory thereof.

We suggest that you don't try to retype the TLD every time you start a new tag library, but start with a template. You can download such a template from http://volume2.coreservlets.com/.

The JSP File

Once you have a tag handler implementation and a TLD, you are ready to write a JSP file that makes use of the tag. Listing 7.3 gives an example. Somewhere in the JSP page you need to place the `taglib` directive. This directive has the following form:

```
<%@ taglib uri="..." prefix="..." %>
```

The required `uri` attribute can be either an absolute or relative URL referring to a TLD file like the one shown in Listing 7.2. For now, we will use a simple URL relative to the Web application's root directory. This makes it easy to refer to the same TLD file from multiple JSP pages in different directories. Remember that the TLD file must be placed somewhere inside the WEB-INF directory. Because this URL will be resolved on the server and not the client, it is allowed to refer to the WEB-INF directory, which is always protected from direct client access.

The required `prefix` attribute specifies a prefix to use in front of any tag name defined in the TLD of this `taglib` declaration. For example, if the TLD file defines a tag named `tag1` and the `prefix` attribute has a value of `test`, the JSP page would need to refer to the tag as `test:tag1`. This tag could be used in either of the following two ways, depending on whether it is defined to be a container that makes use of the tag body:

```
<test:tag1>Arbitrary JSP</test:tag1>
```

or just

```
<test:tag1 />
```

Listing 7.3 Example JSP File

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Example JSP page</TITLE>
<LINK REL=STYLESHEET
      HREF="JSP-Styles.css"
      TYPE="text/css">
</HEAD>
<BODY>
<%@ taglib uri="/WEB-INF/tlds/example.tld"
           prefix="test" %>
<test:example/>
<test:example></test:example>
</BODY></HTML>
```

7.2 Example: Simple Prime Tag

In this example we create a simple custom tag that would output a random 50-digit prime number to the JSP page (a real treat!). We accomplish this task with the help of the `Primes` class shown in Listing 7.4.

We define a tag handler class `SimplePrimeTag` that extends the `SimpleTagSupport` class. In its `doTag` method, we obtain a reference to the `JspWriter` by calling `getJspContext().getOut()`. Then, by using the static method `Primes.nextPrime` we generate our random 50-digit prime number. We output this number to the JSP page by invoking the `print` method on the `JspWriter` object reference. The code for SimplePrimeTag.java is shown in Listing 7.5.

Listing 7.4 Primes.java

```
package coreservlets;
import java.math.BigInteger;

/** A few utilities to generate a large random BigInteger,
 *  and find the next prime number above a given BigInteger.
 */
public class Primes {
  private static final BigInteger ZERO = BigInteger.ZERO;
  private static final BigInteger ONE = BigInteger.ONE;
  private static final BigInteger TWO = new BigInteger("2");

  // Likelihood of false prime is less than 1/2^ERR_VAL
  // Presumably BigInteger uses the Miller-Rabin test or
  // equivalent, and thus is NOT fooled by Carmichael numbers.
  // See section 33.8 of Cormen et al.'s Introduction to
  // Algorithms for details.
  private static final int ERR_VAL = 100;

  public static BigInteger nextPrime(BigInteger start) {
    if (isEven(start))
      start = start.add(ONE);
    else
      start = start.add(TWO);
    if (start.isProbablePrime(ERR_VAL))
      return(start);
    else
      return(nextPrime(start));
  }
```

Listing 7.4 Primes.java *(continued)*

```
private static boolean isEven(BigInteger n) {
  return(n.mod(TWO).equals(ZERO));
}

private static StringBuffer[] digits =
  { new StringBuffer("0"), new StringBuffer("1"),
    new StringBuffer("2"), new StringBuffer("3"),
    new StringBuffer("4"), new StringBuffer("5"),
    new StringBuffer("6"), new StringBuffer("7"),
    new StringBuffer("8"), new StringBuffer("9") };
private static StringBuffer randomDigit(boolean isZeroOK) {
  int index;
  if (isZeroOK) {
    index = (int)Math.floor(Math.random() * 10);
  } else {
    index = 1 + (int)Math.floor(Math.random() * 9);
  }
  return(digits[index]);
}

/** Create a random big integer where every digit is
 *  selected randomly (except that the first digit
 *  cannot be a zero).
 */
public static BigInteger random(int numDigits) {
  StringBuffer s = new StringBuffer("");
  for(int i=0; i<numDigits; i++) {
    if (i == 0) {
      // First digit must be non-zero.
      s.append(randomDigit(false));
    } else {
      s.append(randomDigit(true));
    }
  }
  return(new BigInteger(s.toString()));
}

/** Simple command-line program to test. Enter number
 *  of digits, and it picks a random number of that
 *  length and then prints the first 50 prime numbers
 *  above that.
 */

public static void main(String[] args) {
  int numDigits;
```

Listing 7.4 Primes.java *(continued)*

```
    try {
      numDigits = Integer.parseInt(args[0]);
    } catch (Exception e) { // No args or illegal arg.
      numDigits = 150;
    }
    BigInteger start = random(numDigits);
    for(int i=0; i<50; i++) {
      start = nextPrime(start);
      System.out.println("Prime " + i + " = " + start);
    }
  }
}
```

Listing 7.5 SimplePrimeTag.java

```
package coreservlets.tags;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.*;
import java.math.*;
import coreservlets.Primes;

/**
 *  SimplePrimeTag output a random 50-digit prime number
 *  to the JSP page.
 */
public class SimplePrimeTag extends SimpleTagSupport {
  protected int length = 50;

  public void doTag() throws JspException, IOException {
    JspWriter out = getJspContext().getOut();
    BigInteger prime = Primes.nextPrime(Primes.random(length));
    out.print(prime);
  }
}
```

Now that we have our tag handler class, we need to describe our tag to the container. We do this using the TLD csajsp-taglib.tld shown in Listing 7.6. Because all our tag does is output a prime number, we don't need to allow the tag to include a body, and so we specify `empty` as the value of the `body-content` element. We place the csajsp-taglib.tld file in the WEB-INF/tlds folder.

Listing 7.6 Excerpt from csajsp-taglib.tld

```
<?xml version="1.0" encoding="UTF-8" ?>
<taglib xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
  http://java.sun.com/xml/ns/j2ee/web-jsptaglibrary_2_0.xsd"
  version="2.0">
  <tlib-version>1.0</tlib-version>
  <short-name>csajsp-taglib</short-name>

  <tag>
    <description>Outputs 50-digit primes</description>
    <name>simplePrime</name>
    <tag-class>coreservlets.tags.SimplePrimeTag</tag-class>
    <body-content>empty</body-content>
  </tag>
  ...
</taglib>
```

Listing 7.7 shows the simple-primes-1.jsp page, which uses the simple prime tag. We assign csajsp as the prefix for all tags (so far just simplePrime) in the /WEB-INF/tlds/csajsp-taglib.tld library. Also note that it is perfectly legal to use a closing tag with the body-content of empty as long as there is nothing, not even a space, between the opening tag and the closing tag, as shown by the last occurrence of the tag in the simple-primes-1.jsp page; that is, <csajsp:simplePrime></csajsp:simplePrime>. The resulting output is shown in Figure 7–1.

Listing 7.7 simple-primes-1.jsp

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Some 50-Digit Primes</TITLE>
<LINK REL=STYLESHEET
      HREF="JSP-Styles.css"
      TYPE="text/css">
</HEAD>
<BODY>
<H1>Some 50-Digit Primes</H1>
<%@ taglib uri="/WEB-INF/tlds/csajsp-taglib.tld"
           prefix="csajsp" %>
```

Listing 7.7 simple-primes-1.jsp *(continued)*

```
<UL>
  <LI><csajsp:simplePrime />
  <LI><csajsp:simplePrime />
  <LI><csajsp:simplePrime />
  <LI><csajsp:simplePrime></csajsp:simplePrime>
</UL>
</BODY></HTML>
```

Figure 7–1 Result of simple-primes-1.jsp.

7.3 Assigning Attributes to Tags

Allowing tags like

```
<prefix:name attribute1="value1" attribute2="value2"... />
```

adds significant flexibility to your tag library because the attributes allow us to pass information to the tag. This section explains how to add attribute support to your tags.

Tag Attributes: Tag Handler Class

Providing support for attributes is straightforward. Use of an attribute called `attribute1` simply results in a call to a method called `setAttribute1` in your class that extends `SimpleTagSupport` (or that otherwise implements the `Simple-Tag` interface). Consequently, adding support for an attribute named `attribute1` is merely a matter of implementing the following method in your tag handler class:

```
public void setAttribute1(String value1) {
  doSomethingWith(value1);
}
```

Note that an attribute with the name of `attributeName` (lowercase `a`) corresponds to a method called `setAttributeName` (uppercase `A`).

One of the most common things to do in the attribute handler is to simply store the attribute in a field for later use by the `doTag` method. For example, the following is a code snippet of a tag implementation that adds support for the `message` attribute:

```
private String message = "Default Message";
public void setMessage(String message) {
  this.message = message;
}
```

If the tag handler is accessed from other classes, it is a good idea to provide a `getAttributeName` method in addition to the `setAttributeName` method. Only `setAttributeName` is required, however.

Tag Attributes: Tag Library Descriptor

Tag attributes must be declared inside the `tag` element by means of an `attribute` element. The `attribute` element has three nested elements that can appear between `<attribute>` and `</attribute>`.

- **`name.`** This is a required element that defines the case-sensitive attribute name.
- **`required.`** This is an optional element that stipulates whether the attribute must always be supplied, `true`, or is optional, `false` (default). If `required` is `false` and the JSP page omits the attribute, no call is made to the `setAttributeName` method, so be sure to give default values to the fields that the method sets if the attribute is not declared as required. Omitting a tag attribute, which is declared with the `required` element equal to `true`, results in an error at page translation time.
- **`rtexprvalue.`** This is an optional element that indicates whether the attribute value can be either a JSP scripting expression like `<%= expression %>` or JSP EL like `${bean.value}` (`true`), or whether it must be a fixed string (`false`). The default value is `false`, so this element is usually omitted except when you want to allow attributes to have values determined at request time. Note that even though it is never legal for the body of the tag to contain JSP scripting expressions like `<%= expression %>`, they are nevertheless legal as attribute values.

Tag Attributes: JSP File

As before, the JSP page has to declare the tag library using the `taglib` directive. This is done in the following form:

```
<%@ taglib uri="..." prefix="..." %>
```

The usage of the tag is very similar, except now we are able to specify a custom attribute as well. Remember that just like tag names, the attribute names are case-sensitive and have to appear in the JSP page exactly as they were declared inside the TLD file. Because custom tags are based on XML syntax, the value of an attribute has to be enclosed by either single or double quotes. For example:

```
<some-prefix:tag1 attribute1="value" />
```

7.4 Example: Prime Tag with Variable Length

In this example, we modify the previous prime number example, shown in Section 7.2 (Example: Simple Prime Tag), to provide an attribute for specifying the length of the prime number. Listing 7.8 shows the `PrimeTag` class, a subclass of `SimplePrimeTag` that adds support for the `length` attribute. This change is achieved by supplying an additional method, `setLength`. When this method is called, it attempts to convert its `String` argument into an `int` and store it in an instance variable `length`. If it fails, the originally initialized value for the instance variable `length` is used.

The TLD, shown in Listing 7.9, declares the optional attribute `length`. It is this declaration that tells the container to call the `setLength` method if the attribute `length` appears in the tag when it's used in the JSP page.

The JSP page, shown in Listing 7.10, declares the tag library with the `taglib` directive as before. However, now we are able to specify how long our prime number should be. If we omit the `length` attribute, the prime tag defaults to `50`. Figure 7–2 shows the result of this page.

Listing 7.8 PrimeTag.java

```
package coreservlets.tags;

/** PrimeTag outputs a random prime number
 *  to the JSP page. The length of the prime number is
 *  specified by the length attribute supplied by the JSP
 *  page. If not supplied, it defaults to 50.
 */
public class PrimeTag extends SimplePrimeTag {
  public void setLength(String length) {
    try {
      this.length = Integer.parseInt(length);
    } catch(NumberFormatException nfe) {
      // Do nothing as length is already set to 50
    }
  }
}
```

Listing 7.9 Excerpt from csajsp-taglib.tld

```
<?xml version="1.0" encoding="UTF-8" ?>
<taglib xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
  http://java.sun.com/xml/ns/j2ee/web-jsptaglibrary_2_0.xsd"
  version="2.0">
  <tlib-version>1.0</tlib-version>
  <short-name>csajsp-taglib</short-name>

  <tag>
    <description>Outputs an N-digit prime</description>
    <name>prime</name>
    <tag-class>coreservlets.tags.PrimeTag</tag-class>
    <body-content>empty</body-content>
    <attribute>
      <description>N (prime number length)</description>
      <name>length</name>
      <required>false</required>
    </attribute>
  </tag>
</taglib>
```

Listing 7.10 primes-1.jsp

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Some N-Digit Primes</TITLE>
<LINK REL=STYLESHEET
      HREF="JSP-Styles.css"
      TYPE="text/css">
</HEAD>
<BODY>
<H1>Some N-Digit Primes</H1>
<%@ taglib uri="/WEB-INF/tlds/csajsp-taglib.tld"
           prefix="csajsp" %>
<UL>
  <LI>20-digit: <csajsp:prime length="20" />
  <LI>40-digit: <csajsp:prime length="40" />
  <LI>80-digit: <csajsp:prime length="80" />
  <LI>Default (50-digit): <csajsp:prime />
</UL>
</BODY></HTML>
```

Figure 7–2 Result of primes-1.jsp.

7.5 Including Tag Body in the Tag Output

Up to this point, all of the custom tags you have seen did not allow a body and thus were always used as standalone tags of the following form:

```
<prefix:tagname/>
<prefix:tagname></prefix:tagname>
```

Note that the second tag shown does not have any space between the opening and closing tags. The fact that these tags were not allowed to include a body was a direct result of supplying the element `body-content` with the value of `empty`.

In this section, we see how to define tags that use their body content and are thus written in the following matter:

```
<prefix:tagname>scriptless JSP content</prefix:tagname>
```

Tag Bodies: Tag Handler Class

Supporting tag bodies does not introduce any structural changes to the tag handler class. You still need to include setter methods for any attributes you are planning to declare and use. You still need to override the `doTag` method. To output the body content of the tag, inside the `doTag` method you need to acquire the `JspFragment` instance representing the body of the tag by calling the `getJspBody` method, then using its `invoke` method passing it `null` as its argument. Usually, this is done in a single step as follows:

```
getJspBody().invoke(null);
```

The container processes the JSP content found in the body of the tag just like any other JSP page content. If the `invoke` method is passed `null` as its argument, the resulting output of that JSP content is passed verbatim to the client. Therefore, the `doTag` method has no way of accessing the tag body output. All it can do is pass it along. We show how to access and modify the output of the tag body content before it's sent to the client in Section 8.1 (Manipulating Tag Body). It's important to stress, however, that it is the output resulting from the execution of the JSP code in the tag body, not the JSP code itself, that is passed to the client.

Core Note

When `getJspBody().invoke(null)` is called, it is the output resulting from the execution of the tag body's JSP content that gets passed to the client, not the JSP code itself.

In practice, you almost always output something before or after outputting the tag body as follows:

```
JspWriter out = getJspContext().getOut();
out.print("...");
getJspBody().invoke(null);
out.print("...");
```

Note that because sending the JSP content of the tag body boils down to a simple method invocation, it is very easy to create a tag that conditionally sends the JSP content to the client by surrounding the method call with an `if` statement. We show an example of this in Section 7.7 (Example: Debug Tag). It is also trivial to output the tag body content several times, as the method call can be placed inside a `for` loop and invoked many times. We show an example of this in Section 8.4 (Example: Simple Looping Tag).

Tag Bodies: Tag Library Descriptor

The change to the TLD is trivial. Instead of the value of `empty` for the required `body-content` element, we need to provide the value of `scriptless`.

Tag Bodies: JSP File

There are no changes to the JSP file. You still need to declare and assign a `prefix` to the TLD through the `taglib` directive. However, now we can use our tags with nonempty bodies.

Remember, however, that the `body-content` was declared as `scriptless`, and that `scriptless` means we are allowed to place JSP content into the body of the tag, but are not allowed to place JSP scriptlets there. So, the following is a legal usage of the tag:

```
<prefix:tagname>
  some content with ${bean.property}
</prefix:tagname>
```

The following would be illegal:

```
<prefix:tagname>
  some content with <%= bean.property %>
</prefix:tagname>
```

7.6 Example: Heading Tag

Listing 7.11 shows HeadingTag.java, which defines a tag for a heading element that is more flexible than the standard HTML H1 through H6 elements. (Yes, we know that the entire problem could be solved more elegantly with Cascading Style Sheets [CSS] and without the use of a custom tag, but this is for demonstration purposes only, so work with us.) This new element allows a precise font size, a list of preferred font names (the first entry that is available on the client system will be used), a foreground color, a background color, a border, and an alignment (LEFT, CENTER, RIGHT). Only the alignment capability is available with the H1 through H6 elements. The heading is implemented through use of a one-cell table enclosing a SPAN element that has embedded stylesheet attributes.

The doTag method first generates the <TABLE> and <SPAN> start tags, then invokes getJspBody().invoke(null) to instruct the system to include the tag body, and then generates the </SPAN> and </TABLE> tags. We use various setAttributeName methods to handle the attributes like bgColor and fontSize.

Listing 7.12 shows the excerpt from the csajsp-taglib.tld file that defines the heading tag. Listing 7.13 shows heading-1.jsp, which uses the heading tag. Figure 7–3 shows the resulting JSP page.

Listing 7.11 HeadingTag.java

```
package coreservlets.tags;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.*;

/** Heading tag allows the JSP developer to create
 *  a heading and specify alignment, background color,
 *  foreground color, font, etc. for that heading.
 */
public class HeadingTag extends SimpleTagSupport {
  private String align;
  private String bgColor;
  private String border;
  private String fgColor;
  private String font;
  private String size;

  public void setAlign(String align) {
    this.align = align;
  }
```

Listing 7.11 HeadingTag.java *(continued)*

```
  public void setBgColor(String bgColor) {
    this.bgColor = bgColor;
  }
  public void setBorder(String border) {
    this.border = border;
  }
  public void setFgColor(String fgColor) {
    this.fgColor = fgColor;
  }
  public void setFont(String font) {
    this.font = font;
  }
  public void setSize(String size) {
    this.size = size;
  }

 public void doTag() throws JspException, IOException {
    JspWriter out = getJspContext().getOut();
    out.print("<TABLE ALIGN=\"" + align + "\"\n" +
              "       BGCOLOR=\"" + bgColor + "\"\n" +
              "       BORDER=" + border + "\">\n");
    out.print("<TR><TH>");
    out.print("<SPAN STYLE=\"color: " + fgColor + ";\n" +
              "              font-family: " + font + ";\n" +
              "              font-size: " + size + "px; " +
              "\">\n");
    // Output content of the body
    getJspBody().invoke(null);
    out.println("</SPAN></TH></TR></TABLE>" +
                "<BR CLEAR=\"ALL\"><BR>");
  }
}
```

Listing 7.12 Excerpt from csajsp-taglib.tld

```
<?xml version="1.0" encoding="UTF-8" ?>
<taglib xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
  http://java.sun.com/xml/ns/j2ee/web-jsptaglibrary_2_0.xsd"
  version="2.0">
  <tlib-version>1.0</tlib-version>
  <short-name>csajsp-taglib</short-name>
```

Listing 7.12 Excerpt from csajsp-taglib.tld *(continued)*

```
<tag>
    <description>Formats enclosed heading</description>
    <name>heading</name>
    <tag-class>coreservlets.tags.HeadingTag</tag-class>
    <body-content>scriptless</body-content>
    <attribute>
      <name>align</name>
      <required>true</required>
    </attribute>
    <attribute>
      <name>bgColor</name>
      <required>true</required>
    </attribute>
    <attribute>
      <name>border</name>
      <required>true</required>
    </attribute>
    <attribute>
      <name>fgColor</name>
      <required>true</required>
    </attribute>
    <attribute>
      <name>font</name>
      <required>true</required>
    </attribute>
    <attribute>
      <name>size</name>
      <required>true</required>
    </attribute>
  </tag>
</taglib>
```

Listing 7.13 heading-1.jsp

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD><TITLE>Headings</TITLE>
<LINK REL=STYLESHEET
      HREF="JSP-Styles.css"
      TYPE="text/css">
</HEAD>
<BODY>
<%@ taglib uri="/WEB-INF/tlds/csajsp-taglib.tld"
           prefix="csajsp" %>
```

Listing 7.13 heading-1.jsp *(continued)*

```
<csajsp:heading align="LEFT" bgColor="CYAN"
                border="10" fgColor="BLACK"
                font="Arial Black" size="78">
  First Heading
</csajsp:heading>

<csajsp:heading align="RIGHT" bgColor="RED"
                border="1" fgColor="YELLOW"
                font="Times New Roman" size="50">
  Second Heading
</csajsp:heading>

<csajsp:heading align="CENTER" bgColor="#C0C0C0"
                border="20" fgColor="BLUE"
                font="Arial Narrow" size="100">
  Third Heading
</csajsp:heading>
</BODY></HTML>
```

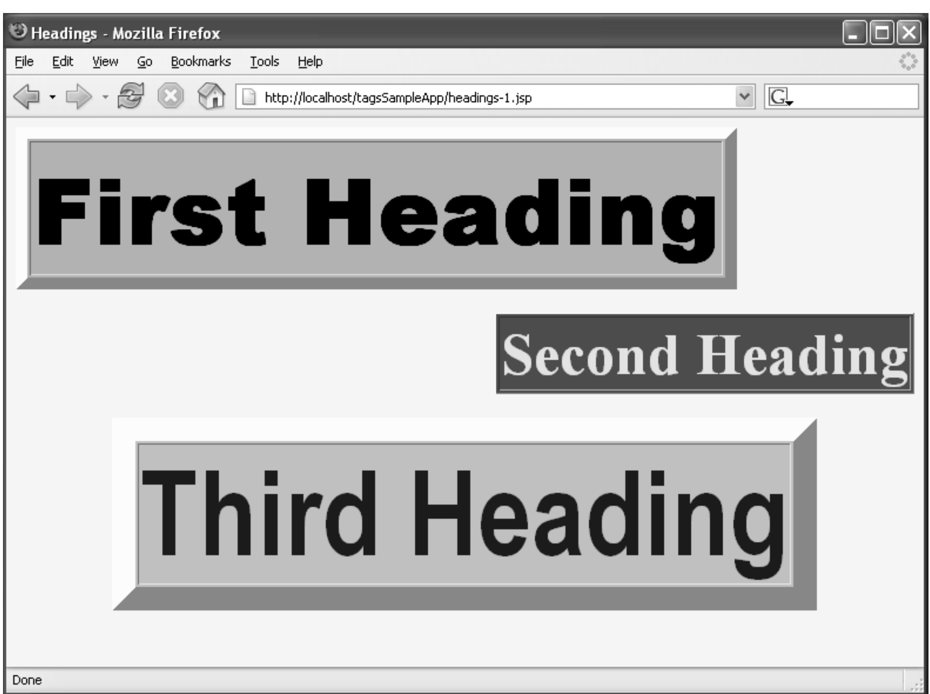


Figure 7–3 Result of heading-1.jsp.

7.7 Example: Debug Tag

In Section 7.5 (Including Tag Body in the Tag Output), we explained that to send the JSP content of the tag body to the client, one need only call the `getJspBody().invoke(null)` method inside the `doTag` method of the tag handler class. This simplicity allows us to easily create tags that output their bodies conditionally. This functionality can be achieved by simply surrounding the `getJspBody().invoke(null)` invocation within an `if` statement.

In this section, we present an example of a custom tag that conditionally outputs its tag body. It's quite often the case when the output of the JSP page is something other than what you expected. In such a case, it's useful to have the option of seeing some debugging information right on the page without having to resort to embedding `System.out.print` statements throughout the page. However, we do not want the user to see the debugging information in the production system. To solve this problem, we create a custom tag that conditionally outputs its body based on the presence of the `debug` request parameter. If the `debug` request parameter is present, it would signal to the JSP page to output the debugging information.

Listing 7.14 shows the DebugTag.java file. In its `doTag` method, we output the tag body if the `debug` request parameter is present and skip the body of the tag if it's not. Inside the JSP page, shown in Listing 7.16, we surround the debugging information with our `debug` tag. Listing 7.15 shows the excerpt from the csajsp-taglib.tld file declaring the `debug` tag to the container. Listing 7.16 shows the debug.jsp page that uses the `debug` tag. Figure 7–4 shows the result of the debug.jsp page when the `debug` request parameter is not present. Figure 7–5 shows the result of the debug.jsp page when the `debug` request parameter is supplied.

Listing 7.14 DebugTag.java

```
package coreservlets.tags;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.*;
import javax.servlet.http.*;

/**
 *  DebugTag outputs its body if the request parameter
 *  'debug' is present and skips it if it's not.
 */
```

Listing 7.14 DebugTag.java *(continued)*

```
public class DebugTag extends SimpleTagSupport {
  public void doTag() throws JspException, IOException {
    PageContext context = (PageContext) getJspContext();
    HttpServletRequest request =
      (HttpServletRequest) context.getRequest();
    // Output body of tag only if debug param is present.
    if (request.getParameter("debug") != null) {
      getJspBody().invoke(null);
    }
  }
}
```

Listing 7.15 Excerpt from csajsp-taglib.tld

```
<?xml version="1.0" encoding="UTF-8" ?>
<taglib xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
  http://java.sun.com/xml/ns/j2ee/web-jsptaglibrary_2_0.xsd"
  version="2.0">
  <tlib-version>1.0</tlib-version>
  <short-name>csajsp-taglib</short-name>

  <tag>
    <description>Conditionally outputs enclosed body</description>
    <name>debug</name>
    <tag-class>coreservlets.tags.DebugTag</tag-class>
    <body-content>scriptless</body-content>
  </tag>
</taglib>
```

Listing 7.16 debug.jsp

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Some Hard-to-Debug Page</TITLE>
<LINK REL=STYLESHEET
      HREF="JSP-Styles.css"
      TYPE="text/css">
</HEAD>
```

Listing 7.16 debug.jsp *(continued)*

```
<BODY>
<H1>Some Hard-to-Debug Page</H1>
<%@ taglib uri="/WEB-INF/tlds/csajsp-taglib.tld"
           prefix="csajsp" %>
Top of regular page. Blah, blah, blah.
Yadda, yadda, yadda.
<csajsp:debug>
<H2>Debug Info:</H2>
********************<BR>
-Remote Host: ${pageContext.request.remoteHost}<BR>
-Session ID: ${pageContext.session.id}<BR>
-The foo parameter: ${param.foo}<BR>
********************<BR>
</csajsp:debug>
<P>
Bottom of regular page. Blah, blah, blah.
Yadda, yadda, yadda.
</BODY></HTML>
```

Figure 7–4 Result of debug.jsp page without supplying the `debug` request parameter.

Figure 7–5 Result of debug.jsp page when the `debug` request parameter is supplied.

7.8 Creating Tag Files

JSP specification version 2.0 introduced a JSP-based way to create custom tags using tag files. One of the key differences between what we talk about in the beginning of this chapter, Java-based custom tags, and tag files (or JSP-based custom tags) is that with Java-based tags the tag handler is a Java class, whereas with JSP-based tags the tag handler is a JSP page. Tag files are also a bit simpler to write because they don't require you to provide a TLD.

The guidelines for when to develop a JSP-based custom tag versus a Java-based custom tag are analogous to the guidelines for when to use a JSP page versus a servlet. When there is a lot of logic, use Java to create output. When there is a lot of HTML formatting, use tag files to create output. To review the general benefits of JSPs versus servlets, please see Section 10.2 of Volume 1.

There is one caveat that might force your choice between tag files and Java-based custom tags. Tag files run only in JSP 2.0, whereas Java-based custom tags have a "classic" version that does not rely on the new `SimpleTag` API. So, if the container you are targeting is only compliant with earlier versions of the specification, you have to use classic Java-based custom tag development. The bad news is that classic

Java-based custom tag development is quite more complicated than the `SimpleTag` API and we do not cover classic tags in this book. The good news is that almost all mainstream containers have been updated to be compliant with servlet specification 2.4 and JSP specification 2.0, so chances are you won't need to develop the classic Java-based custom tags.

In general, there are two steps to creating a JSP-based custom tag.

- **Create a JSP-based tag file.** This file is a fragment of a JSP page with some special directives and a .tag extension. It must be placed inside the WEB-INF/tags directory or a subdirectory thereof.
- **Create a JSP page that uses the tag file.** The JSP page points to the directory where the tag file resides. The name of the tag file (minus the .tag extension) becomes the name of the custom tag and therefore no TLD connecting the implementation of the tag with its name is needed.

In the next few sections, we reproduce the same custom tags we developed earlier in this chapter, but we use tag files to accomplish it.

7.9 Example: Simple Prime Tag Using Tag Files

Let's rewrite the simple prime custom tag example using tag files. Listing 7.17 shows simplePrime2.tag. It consists of just one line invoking the static method `nextPrime` of the `Primes` class. The Primes.java file is shown in Listing 7.4. We place the simplePrime2.tag file into the WEB-INF/tags directory. Listing 7.18 shows simple-primes-2.jsp, which uses our JSP-based custom tag. Note that the `taglib` directive no longer has a `uri` attribute, but uses a `tagdir` attribute instead. This attribute tells the container which directory contains the tag files. Figure 7–6 shows the result of simple-primes-2.jsp.

Listing 7.17 simplePrime2.tag

```
<%= coreservlets.Primes.nextPrime
      (coreservlets.Primes.random(50)) %>
```

Listing 7.18 simple-primes-2.jsp

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Some 50-Digit Primes</TITLE>
<LINK REL=STYLESHEET
      HREF="JSP-Styles.css"
      TYPE="text/css">
</HEAD>
<BODY>
<H1>Some 50-Digit Primes</H1>
<%@ taglib tagdir="/WEB-INF/tags" prefix="csajsp" %>
<UL>
  <LI><csajsp:simplePrime2 />
  <LI><csajsp:simplePrime2 />
  <LI><csajsp:simplePrime2 />
  <LI><csajsp:simplePrime2 />
</UL>
</BODY></HTML>
```

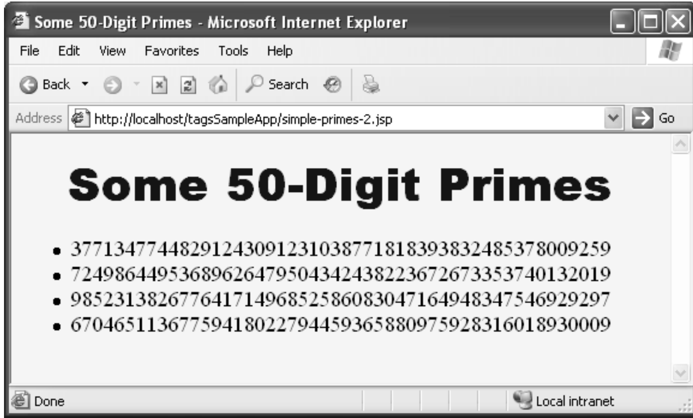


Figure 7–6 Result of simple-primes-2.jsp.

7.10 Example: Prime Tag with Variable Length Using Tag Files

In this section, we rewrite the example of Section 7.4 (Example: Prime Tag with Variable Length) with a JSP-based custom tag. To use attributes with a JSP-based custom tag, each attribute must be declared inside the tag file. This declaration is accomplished by the `attribute` directive. The `attribute` directive itself has attributes that provide the same information that the `attribute` subelements inside the TLD would provide. For example, you can specify whether an attribute is required or not by supplying a `required` attribute with a value of either `true` or `false`. When the value is passed through an attribute to the tag file, it is automatically stored into a scoped variable for access from the JSP EL and into a local variable for access from Java code (scriptlets and scripting expressions). Note once again that because the tag file has the ability to describe itself to the container, no TLD is required.

Listing 7.19 shows prime2.tag declaring an optional attribute called `length`. Note that we are able to refer to that attribute just like to any other local variable inside the Java code. The JSP page, primes-2.jsp, shown in Listing 7.20, uses our tag file to output prime numbers of different lengths. Figure 7–7 shows the result of primes-2.jsp.

Listing 7.19 prime2.tag

```
<%@ attribute name="length" required="false" %>
<%
int len = 50;
try {
  len = Integer.parseInt(length);
} catch(NumberFormatException nfe) {}
%>
<%= coreservlets.Primes.nextPrime
      (coreservlets.Primes.random(len)) %>
```

Listing 7.20 primes-2.jsp

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Some N-Digit Primes</TITLE>
<LINK REL=STYLESHEET
      HREF="JSP-Styles.css"
      TYPE="text/css">
</HEAD>
<BODY>
<H1>Some N-Digit Primes</H1>
<%@ taglib tagdir="/WEB-INF/tags" prefix="csajsp" %>
<UL>
  <LI>20-digit: <csajsp:prime2 length="20" />
  <LI>40-digit: <csajsp:prime2 length="40" />
  <LI>80-digit: <csajsp:prime2 length="80" />
  <LI>Default (50-digit): <csajsp:prime2 />
</UL>
</BODY></HTML>
```

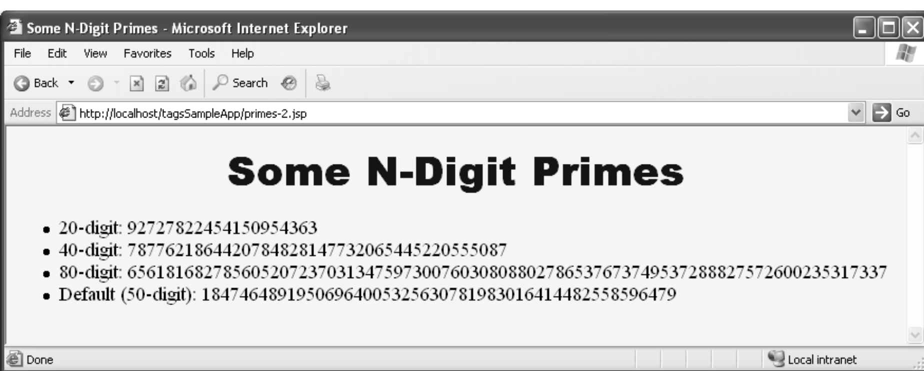


Figure 7–7 Result of primes-2.jsp.

7.11 Example: Heading Tag Using Tag Files

In this section, we rewrite the heading example of Section 7.6 (Example: Heading Tag) with a JSP-based custom tag. Outputting the tag body inside a tag file is as simple as providing a `<jsp:doBody/>` tag. That's it! No additional configurations, no TLD file, and the access to attributes is still the same simple process described in Section 7.10 (Example: Prime Tag with Variable Length Using Tag Files). Just place `<jsp:doBody/>` where you want the tag body to appear in the final output and you are done.

Listing 7.21 shows the `heading2.tag` file. It declares quite a number of required attributes and then proceeds to use them as regular scoped variables. We use `<jsp:doBody/>` to output the body of the tag to the client. Listing 7.22 shows the `headings-2.jsp` file, which uses the `heading2.tag` custom tag. Figure 7–8 shows the result of `headings-2.jsp`.

Listing 7.21 heading2.tag

```
<%@ attribute name="align" required="true" %>
<%@ attribute name="bgColor" required="true" %>
<%@ attribute name="border" required="true" %>
<%@ attribute name="fgColor" required="true" %>
<%@ attribute name="font" required="true" %>
<%@ attribute name="size" required="true" %>
<TABLE ALIGN="${align}"
       BGCOLOR="${bgColor}"
       BORDER="${border}">
  <TR><TH>
      <SPAN STYLE="color: ${fgColor};
                   font-family: ${font};
                   font-size: ${size}px;">
      <jsp:doBody/></SPAN>
</TABLE><BR CLEAR="ALL"><BR>
```

Listing 7.22 headings-2.jsp

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD><TITLE>Headings</TITLE>
<LINK REL=STYLESHEET
      HREF="JSP-Styles.css"
      TYPE="text/css">
</HEAD>
```

Listing 7.22 headings-2.jsp *(continued)*

```
<BODY>
<%@ taglib tagdir="/WEB-INF/tags" prefix="csajsp" %>
<csajsp:heading2 align="LEFT" bgColor="CYAN"
                 border="10" fgColor="BLACK"
                 font="Arial Black" size="78">
  First Heading
</csajsp:heading2>
<csajsp:heading2 align="RIGHT" bgColor="RED"
                 border="1" fgColor="YELLOW"
                 font="Times New Roman" size="50">
  Second Heading
</csajsp:heading2>
<csajsp:heading2 align="CENTER" bgColor="#C0C0C0"
                 border="20" fgColor="BLUE"
                 font="Arial Narrow" size="100">
  Third Heading
</csajsp:heading2>
</BODY></HTML>
```

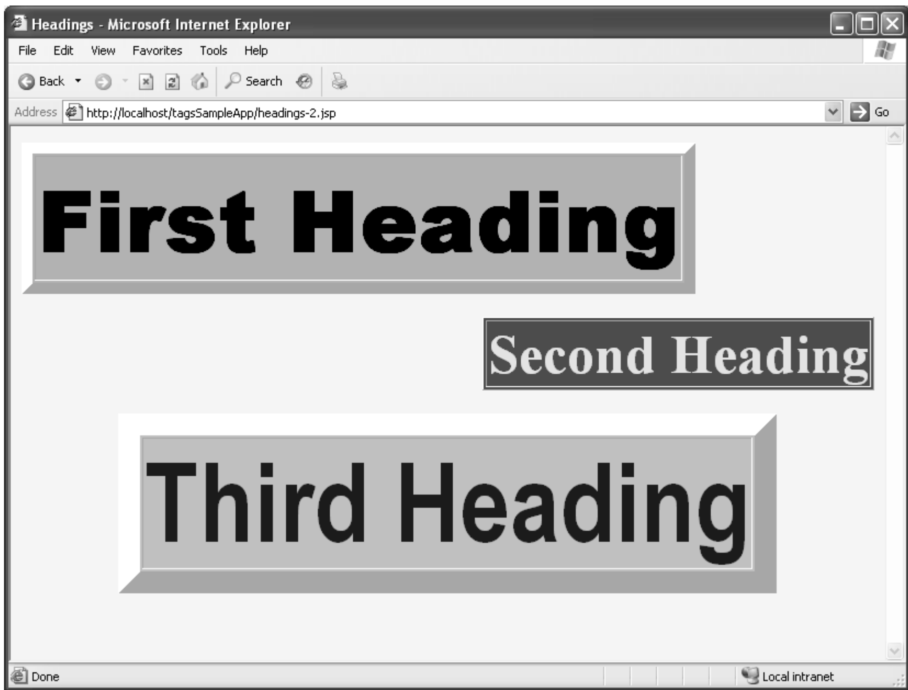


Figure 7–8 Result of headings-2.jsp.

The definitive guide to Java for serious programmers who want to put Java to work on real projects

INHERITANCE

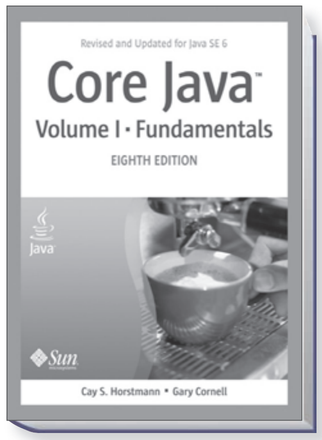
FROM

CORE JAVA, VOLUME I—

Fundamentals, Eighth Edition

by Cay Horstmann and Gary Cornell

©2008 | 864 PAGES | 0-13-235476-4



ALSO AVAILABLE

- SAFARI BOOKS ONLINE
- E-BOOK: 0132357267
- MOBI POCKET: 0131353004

FOR MORE INFORMATION

informit.com/title/9780132354769

PRENTICE HALL

TABLE OF CONTENTS

Chapter 1:
An Introduction to Java

Chapter 2:
The Java Programming Environment

Chapter 3:
Fundamental Programming Structures in Java

Chapter 4:
Objects and Classes

Chapter 5:
Inheritance

Chapter 6:
Interfaces and Inner Classes

Chapter 7:
Graphics Programming

Chapter 8:
Event Handling

Chapter 9:
User Interface Components with Swing

Chapter 10:
Deploying Applications and Applets

Chapter 11:
Exceptions, Logging, Assertions, and Debugging

Chapter 12:
Generic Programming

Chapter 13:
Collections

Chapter 14:
Multithreading

Chapter 5

INHERITANCE

- CLASSES, SUPERCLASSES, AND SUBCLASSES
- `Object`: THE COSMIC SUPERCLASS
- GENERIC ARRAY LISTS
- OBJECT WRAPPERS AND AUTOBOXING
- METHODS WITH A VARIABLE NUMBER OF PARAMETERS
- ENUMERATION CLASSES
- REFLECTION
- DESIGN HINTS FOR INHERITANCE

Chapter 4 introduced you to classes and objects. In this chapter, you learn about *inheritance,* another fundamental concept of object-oriented programming. The idea behind inheritance is that you can create new classes that are built on existing classes. When you inherit from an existing class, you reuse (or inherit) its methods and fields and you add new methods and fields to adapt your new class to new situations. This technique is essential in Java programming.

As with the previous chapter, if you are coming from a procedure-oriented language like C, Visual Basic, or COBOL, you will want to read this chapter carefully. For experienced C++ programmers or those coming from another object-oriented language like Smalltalk, this chapter will seem largely familiar, but there are many differences between how inheritance is implemented in Java and how it is done in C++ or in other object-oriented languages.

This chapter also covers *reflection,* the ability to find out more about classes and their properties in a running program. Reflection is a powerful feature, but it is undeniably complex. Because reflection is of greater interest to tool builders than to application programmers, you can probably glance over that part of the chapter upon first reading and come back to it later.

Classes, Superclasses, and Subclasses

Let's return to the `Employee` class that we discussed in the previous chapter. Suppose (alas) you work for a company at which managers are treated differently from other employees. Managers are, of course, just like employees in many respects. Both employees and managers are paid a salary. However, while employees are expected to complete their assigned tasks in return for receiving their salary, managers get *bonuses* if they actually achieve what they are supposed to do. This is the kind of situation that cries out for inheritance. Why? Well, you need to define a new class, `Manager`, and add functionality. But you can retain some of what you have already programmed in the `Employee` class, and *all* the fields of the original class can be preserved. More abstractly, there is an obvious "is–a" relationship between `Manager` and `Employee`. Every manager *is an* employee: This "is–a" relationship is the hallmark of inheritance.

Here is how you define a `Manager` class that inherits from the `Employee` class. You use the Java keyword `extends` to denote inheritance.

```
class Manager extends Employee
{
   added methods and fields
}
```

C++ NOTE: Inheritance is similar in Java and C++. Java uses the `extends` keyword instead of the `:` token. All inheritance in Java is public inheritance; there is no analog to the C++ features of private and protected inheritance.

The keyword `extends` indicates that you are making a new class that derives from an existing class. The existing class is called the *superclass, base class,* or *parent class.* The new class is called the *subclass, derived class,* or *child class.* The terms superclass and subclass are those most commonly used by Java programmers, although some programmers prefer the parent/child analogy, which also ties in nicely with the "inheritance" theme.

The `Employee` class is a superclass, but not because it is superior to its subclass or contains more functionality. *In fact, the opposite is true:* subclasses have *more* functionality than their superclasses. For example, as you will see when we go over the rest of the `Manager` class code, the `Manager` class encapsulates more data and has more functionality than its superclass `Employee`.

NOTE: The prefixes *super* and *sub* come from the language of sets used in theoretical computer science and mathematics. The set of all employees contains the set of all managers, and this is described by saying it is a *superset* of the set of managers. Or, put it another way, the set of all managers is a *subset* of the set of all employees.

Our `Manager` class has a new field to store the bonus, and a new method to set it:

```
class Manager extends Employee
{
   . . .

   public void setBonus(double b)
   {
      bonus = b;
   }

   private double bonus;
}
```

There is nothing special about these methods and fields. If you have a `Manager` object, you can simply apply the `setBonus` method.

```
Manager boss = . . .;
boss.setBonus(5000);
```

Of course, if you have an `Employee` object, you cannot apply the `setBonus` method—it is not among the methods that are defined in the `Employee` class.

However, you *can* use methods such as `getName` and `getHireDay` with `Manager` objects. Even though these methods are not explicitly defined in the `Manager` class, they are automatically inherited from the `Employee` superclass.

Similarly, the fields `name`, `salary`, and `hireDay` are inherited from the superclass. Every `Manager` object has four fields: `name`, `salary`, `hireDay`, and `bonus`.

When defining a subclass by extending its superclass, you only need to indicate the *differences* between the subclass and the superclass. When designing classes, you place the most general methods into the superclass and more specialized methods in the subclass. Factoring out common functionality by moving it to a superclass is common in object-oriented programming.

However, some of the superclass methods are not appropriate for the `Manager` subclass. In particular, the `getSalary` method should return the sum of the base salary and the bonus. You need to supply a new method to *override* the superclass method:

```
class Manager extends Employee
{
   . . .
```

```
    public double getSalary()
    {
        . . .
    }
    . . .
}
```

How can you implement this method? At first glance, it appears to be simple—just return the sum of the `salary` and `bonus` fields:

```
public double getSalary()
{
    return salary + bonus; // won't work
}
```

However, that won't work. The `getSalary` method of the `Manager` class *has no direct access to the private fields of the superclass.* This means that the `getSalary` method of the `Manager` class cannot directly access the `salary` field, even though every `Manager` object has a field called `salary`. Only the methods of the `Employee` class have access to the private fields. If the `Manager` methods want to access those private fields, they have to do what every other method does—use the public interface, in this case, the public `getSalary` method of the `Employee` class.

So, let's try this again. You need to call `getSalary` instead of simply accessing the `salary` field.

```
public double getSalary()
{
    double baseSalary = getSalary(); // still won't work
    return baseSalary + bonus;
}
```

The problem is that the call to `getSalary` simply calls *itself*, because the `Manager` class has a `getSalary` method (namely, the method we are trying to implement). The consequence is an infinite set of calls to the same method, leading to a program crash.

We need to indicate that we want to call the `getSalary` method of the `Employee` superclass, not the current class. You use the special keyword `super` for this purpose. The call

```
super.getSalary()
```

calls the `getSalary` method of the `Employee` class. Here is the correct version of the `getSalary` method for the `Manager` class:

```
public double getSalary()
{
    double baseSalary = super.getSalary();
    return baseSalary + bonus;
}
```

NOTE: Some people think of `super` as being analogous to the `this` reference. However, that analogy is not quite accurate—`super` is not a reference to an object. For example, you cannot assign the value `super` to another object variable. Instead, `super` is a special keyword that directs the compiler to invoke the superclass method.

As you saw, a subclass can *add* fields, and it can *add* or *override* methods of the superclass. However, inheritance can never take away any fields or methods.

C++ NOTE: Java uses the keyword super to call a superclass method. In C++, you would use the name of the superclass with the :: operator instead. For example, the getSalary method of the Manager class would call Employee::getSalary instead of super.getSalary.

Finally, let us supply a constructor.

```
public Manager(String n, double s, int year, int month, int day)
{
   super(n, s, year, month, day);
   bonus = 0;
}
```

Here, the keyword super has a different meaning. The instruction

```
super(n, s, year, month, day);
```

is shorthand for "call the constructor of the Employee superclass with n, s, year, month, and day as parameters."

Because the Manager constructor cannot access the private fields of the Employee class, it must initialize them through a constructor. The constructor is invoked with the special super syntax. The call using super must be the first statement in the constructor for the subclass.

If the subclass constructor does not call a superclass constructor explicitly, then the default (no-parameter) constructor of the superclass is invoked. If the superclass has no default constructor and the subclass constructor does not call another superclass constructor explicitly, then the Java compiler reports an error.

NOTE: Recall that the this keyword has two meanings: to denote a reference to the implicit parameter and to call another constructor of the same class. Likewise, the super keyword has two meanings: to invoke a superclass method and to invoke a superclass constructor. When used to invoke constructors, the this and super keywords are closely related. The constructor calls can only occur as the first statement in another constructor. The construction parameters are either passed to another constructor of the same class (this) or a constructor of the superclass (super).

C++ NOTE: In a C++ constructor, you do not call super, but you use the initializer list syntax to construct the superclass. The Manager constructor looks like this in C++:

```
Manager::Manager(String n, double s, int year, int month, int day) // C++
: Employee(n, s, year, month, day)
{
   bonus = 0;
}
```

Having redefined the getSalary method for Manager objects, managers will *automatically* have the bonus added to their salaries.

Here's an example of this at work: we make a new manager and set the manager's bonus:

```
Manager boss = new Manager("Carl Cracker", 80000, 1987, 12, 15);
boss.setBonus(5000);
```

We make an array of three employees:

```
Employee[] staff = new Employee[3];
```

We populate the array with a mix of managers and employees:

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

We print out everyone's salary:

```
for (Employee e : staff)
   System.out.println(e.getName() + " " + e.getSalary());
```

This loop prints the following data:

```
Carl Cracker 85000.0
Harry Hacker 50000.0
Tommy Tester 40000.0
```

Now `staff[1]` and `staff[2]` each print their base salary because they are `Employee` objects. However, `staff[0]` is a `Manager` object and its `getSalary` method adds the bonus to the base salary.

What is remarkable is that the call

```
e.getSalary()
```

picks out the *correct* `getSalary` method. Note that the *declared* type of `e` is `Employee`, but the *actual* type of the object to which `e` refers can be either `Employee` or `Manager`.

When `e` refers to an `Employee` object, then the call `e.getSalary()` calls the `getSalary` method of the `Employee` class. However, when `e` refers to a `Manager` object, then the `getSalary` method of the `Manager` class is called instead. The virtual machine knows about the actual type of the object to which `e` refers, and therefore can invoke the correct method.

The fact that an object variable (such as the variable `e`) can refer to multiple actual types is called *polymorphism.* Automatically selecting the appropriate method at runtime is called *dynamic binding.* We discuss both topics in more detail in this chapter.

C++ NOTE: In Java, you do not need to declare a method as virtual. Dynamic binding is the default behavior. If you do *not* want a method to be virtual, you tag it as `final`. (We discuss the `final` keyword later in this chapter.)

Listing 5–1 contains a program that shows how the salary computation differs for `Employee` and `Manager` objects.

Listing 5–1 ManagerTest.java

```
1. import java.util.*;
2.
3. /**
4.  * This program demonstrates inheritance.
5.  * @version 1.21 2004-02-21
6.  * @author Cay Horstmann
7.  */
8. public class ManagerTest
9. {
10.    public static void main(String[] args)
11.    {
12.       // construct a Manager object
13.       Manager boss = new Manager("Carl Cracker", 80000, 1987, 12, 15);
14.       boss.setBonus(5000);
15.
16.       Employee[] staff = new Employee[3];
17.
18.       // fill the staff array with Manager and Employee objects
19.
20.       staff[0] = boss;
21.       staff[1] = new Employee("Harry Hacker", 50000, 1989, 10, 1);
22.       staff[2] = new Employee("Tommy Tester", 40000, 1990, 3, 15);
23.
24.       // print out information about all Employee objects
25.       for (Employee e : staff)
26.          System.out.println("name=" + e.getName() + ",salary=" + e.getSalary());
27.    }
28. }
29.
30. class Employee
31. {
32.    public Employee(String n, double s, int year, int month, int day)
33.    {
34.       name = n;
35.       salary = s;
36.       GregorianCalendar calendar = new GregorianCalendar(year, month - 1, day);
37.       hireDay = calendar.getTime();
38.    }
39.
40.    public String getName()
41.    {
42.       return name;
43.    }
44.
45.    public double getSalary()
46.    {
47.       return salary;
48.    }
49.
```

Listing 5–1 ManagerTest.java (continued)

```
50.    public Date getHireDay()
51.    {
52.       return hireDay;
53.    }
54.
55.    public void raiseSalary(double byPercent)
56.    {
57.       double raise = salary * byPercent / 100;
58.       salary += raise;
59.    }
60.
61.    private String name;
62.    private double salary;
63.    private Date hireDay;
64. }
65.
66. class Manager extends Employee
67. {
68.    /**
69.     * @param n the employee's name
70.     * @param s the salary
71.     * @param year the hire year
72.     * @param month the hire month
73.     * @param day the hire day
74.     */
75.    public Manager(String n, double s, int year, int month, int day)
76.    {
77.       super(n, s, year, month, day);
78.       bonus = 0;
79.    }
80.
81.    public double getSalary()
82.    {
83.       double baseSalary = super.getSalary();
84.       return baseSalary + bonus;
85.    }
86.
87.    public void setBonus(double b)
88.    {
89.       bonus = b;
90.    }
91.
92.    private double bonus;
93. }
```

Inheritance Hierarchies

Inheritance need not stop at deriving one layer of classes. We could have an `Executive` class that extends `Manager`, for example. The collection of all classes extending from a common superclass is called an *inheritance hierarchy,* as shown in Figure 5–1. The path from a particular class to its ancestors in the inheritance hierarchy is its *inheritance chain.*

There is usually more than one chain of descent from a distant ancestor class. You could form a subclass `Programmer` or `Secretary` that extends `Employee`, and they would have nothing to do with the `Manager` class (or with each other). This process can continue as long as is necessary.

C++ NOTE: Java does not support multiple inheritance. (For ways to recover much of the functionality of multiple inheritance, see the section on Interfaces in the next chapter.)

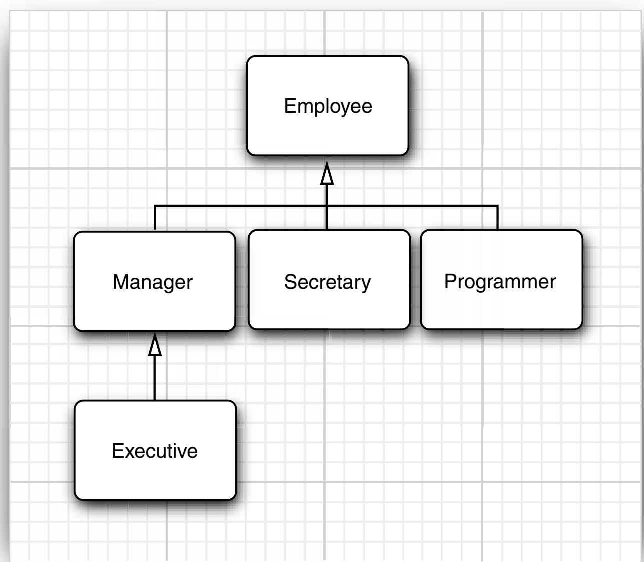


Figure 5–1 `Employee` inheritance hierarchy

Polymorphism

A simple rule enables you to know whether or not inheritance is the right design for your data. The "is–a" rule states that every object of the subclass is an object of the superclass. For example, every manager is an employee. Thus, it makes sense for the `Manager` class to be a subclass of the `Employee` class. Naturally, the opposite is not true—not every employee is a manager.

Another way of formulating the "is–a" rule is the *substitution principle.* That principle states that you can use a subclass object whenever the program expects a superclass object.

For example, you can assign a subclass object to a superclass variable.

```
Employee e;
e = new Employee(. . .);  // Employee object expected
e = new Manager(. . .); // OK, Manager can be used as well
```

In the Java programming language, object variables are *polymorphic.* A variable of type `Employee` can refer to an object of type `Employee` or to an object of any subclass of the `Employee` class (such as `Manager`, `Executive`, `Secretary`, and so on).

We took advantage of this principle in Listing 5–1:

```
Manager boss = new Manager(. . .);
Employee[] staff = new Employee[3];
staff[0] = boss;
```

In this case, the variables `staff[0]` and `boss` refer to the same object. However, `staff[0]` is considered to be only an `Employee` object by the compiler.

That means, you can call

```
boss.setBonus(5000); // OK
```

but you can't call

```
staff[0].setBonus(5000); // ERROR
```

The declared type of `staff[0]` is `Employee`, and the `setBonus` method is not a method of the `Employee` class.

However, you cannot assign a superclass reference to a subclass variable. For example, it is not legal to make the assignment

```
Manager m = staff[i]; // ERROR
```

The reason is clear: Not all employees are managers. If this assignment were to succeed and `m` were to refer to an `Employee` object that is not a manager, then it would later be possible to call `m.setBonus(...)` and a runtime error would occur.

CAUTION: In Java, arrays of subclass references can be converted to arrays of superclass references without a cast. For example, consider this array of managers:

```
Manager[] managers = new Manager[10];
```

It is legal to convert this array to an `Employee[]` array:

```
Employee[] staff = managers; // OK
```

Sure, why not, you may think. After all, if `manager[i]` is a `Manager`, it is also an `Employee`. But actually, something surprising is going on. Keep in mind that `managers` and `staff` are references to the same array. Now consider the statement

```
staff[0] = new Employee("Harry Hacker", ...);
```

The compiler will cheerfully allow this assignment. But `staff[0]` and `manager[0]` are the same reference, so it looks as if we managed to smuggle a mere employee into the management ranks. That would be very bad—calling `managers[0].setBonus(1000)` would try to access a nonexistent instance field and would corrupt neighboring memory.

To make sure no such corruption can occur, all arrays remember the element type with which they were created, and they monitor that only compatible references are stored into them. For example, the array created as `new Manager[10]` remembers that it is an array of managers. Attempting to store an `Employee` reference causes an `ArrayStoreException`.

Dynamic Binding

It is important to understand what happens when a method call is applied to an object. Here are the details:

1. The compiler looks at the declared type of the object and the method name. Let's say we call `x.f(param)`, and the implicit parameter `x` is declared to be an object of class `C`. Note that there may be multiple methods, all with the same name, `f`, but with different parameter types. For example, there may be a method `f(int)` and a method `f(String)`. The compiler enumerates all methods called `f` in the class `C` and all `public` methods called `f` in the superclasses of `C`.

   Now the compiler knows all possible candidates for the method to be called.
2. Next, the compiler determines the types of the parameters that are supplied in the method call. If among all the methods called `f` there is a unique method whose parameter types are a best match for the supplied parameters, then that method is chosen to be called. This process is called *overloading resolution.* For example, in a call `x.f("Hello")`, the compiler picks `f(String)` and not `f(int)`. The situation can get complex because of type conversions (`int` to `double`, `Manager` to `Employee`, and so on). If the compiler cannot find any method with matching parameter types or if multiple methods all match after applying conversions, then the compiler reports an error.

   Now the compiler knows the name and parameter types of the method that needs to be called.

NOTE: Recall that the name and parameter type list for a method is called the method's *signature*. For example, `f(int)` and `f(String)` are two methods with the same name but different signatures. If you define a method in a subclass that has the same signature as a superclass method, then you override that method.

The return type is not part of the signature. However, when you override a method, you need to keep the return type compatible. Prior to Java SE 5.0, the return types had to be identical. However, it is now legal for the subclass to change the return type of an overridden method to a subtype of the original type. For example, suppose that the `Employee` class has a

```
public Employee getBuddy() { ... }
```

Then the `Manager` subclass can override this method as

```
public Manager getBuddy() { ... } // OK in Java SE 5.0
```

We say that the two `getBuddy` methods have *covariant* return types.

---

3. If the method is `private`, `static`, `final`, or a constructor, then the compiler knows exactly which method to call. (The `final` modifier is explained in the next section.) This is called *static binding.* Otherwise, the method to be called depends on the actual type of the implicit parameter, and dynamic binding must be used at runtimeruntime. In our example, the compiler would generate an instruction to call `f(String)` with dynamic binding.
4. When the program runs and uses dynamic binding to call a method, then the virtual machine must call the version of the method that is appropriate for the *actual* type of the object to which `x` refers. Let's say the actual type is `D`, a subclass of `C`. If the class `D`

defines a method `f(String)`, that method is called. If not, `D`'s superclass is searched for a method `f(String)`, and so on.

It would be time consuming to carry out this search every time a method is called. Therefore, the virtual machine precomputes for each class a *method table* that lists all method signatures and the actual methods to be called. When a method is actually called, the virtual machine simply makes a table lookup. In our example, the virtual machine consults the method table for the class `D` and looks up the method to call for `f(String)`. That method may be `D.f(String)` or `X.f(String)`, where `X` is some superclass of `D`.

There is one twist to this scenario. If the call is `super.f(param)`, then the compiler consults the method table of the superclass of the implicit parameter.

Let's look at this process in detail in the call `e.getSalary()` in Listing 5–1. The declared type of `e` is `Employee`. The `Employee` class has a single method, called `getSalary`, with no method parameters. Therefore, in this case, we don't worry about overloading resolution.

Because the `getSalary` method is not `private`, `static`, or `final`, it is dynamically bound. The virtual machine produces method tables for the `Employee` and `Manager` classes. The `Employee` table shows that all methods are defined in the `Employee` class itself:

```
Employee:
   getName() -> Employee.getName()
   getSalary() -> Employee.getSalary()
   getHireDay() -> Employee.getHireDay()
   raiseSalary(double) -> Employee.raiseSalary(double)
```

Actually, that isn't the whole story—as you will see later in this chapter, the `Employee` class has a superclass `Object` from which it inherits a number of methods. We ignore the `Object` methods for now.

The `Manager` method table is slightly different. Three methods are inherited, one method is redefined, and one method is added.

```
Manager:
   getName() -> Employee.getName()
   getSalary() -> Manager.getSalary()
   getHireDay() -> Employee.getHireDay()
   raiseSalary(double) -> Employee.raiseSalary(double)
   setBonus(double) -> Manager.setBonus(double)
```

At runtime, the call `e.getSalary()` is resolved as follows:

1. First, the virtual machine fetches the method table for the actual type of `e`. That may be the table for `Employee`, `Manager`, or another subclass of `Employee`.
2. Then, the virtual machine looks up the defining class for the `getSalary()` signature. Now it knows which method to call.
3. Finally, the virtual machine calls the method.

Dynamic binding has a very important property: it makes programs *extensible* without the need for modifying existing code. Suppose a new class `Executive` is added and there is the possibility that the variable `e` refers to an object of that class. The code containing the call `e.getSalary()` need not be recompiled. The `Executive.getSalary()` method is called automatically if `e` happens to refer to an object of type `Executive`.

CAUTION: When you override a method, the subclass method must be *at least as visible* as the superclass method. In particular, if the superclass method is `public`, then the subclass method must also be declared as `public`. It is a common error to accidentally omit the `public` specifier for the subclass method. The compiler then complains that you try to supply a weaker access privilege.

Preventing Inheritance: Final Classes and Methods

Occasionally, you want to prevent someone from forming a subclass from one of your classes. Classes that cannot be extended are called *final* classes, and you use the `final` modifier in the definition of the class to indicate this. For example, let us suppose we want to prevent others from subclassing the `Executive` class. Then, we simply declare the class by using the `final` modifier as follows:

```
final class Executive extends Manager
{
   . . .
}
```

You can also make a specific method in a class `final`. If you do this, then no subclass can override that method. (All methods in a `final` class are automatically `final`.) For example:

```
class Employee
{
   . . .
   public final String getName()
   {
      return name;
   }
   . . .
}
```

NOTE: Recall that fields can also be declared as `final`. A final field cannot be changed after the object has been constructed. However, if a class is declared as `final`, only the methods, not the fields, are automatically `final`.

There is only one good reason to make a method or class `final`: to make sure that the semantics cannot be changed in a subclass. For example, the `getTime` and `setTime` methods of the `Calendar` class are `final`. This indicates that the designers of the `Calendar` class have taken over responsibility for the conversion between the `Date` class and the calendar state. No subclass should be allowed to mess up this arrangement. Similarly, the `String` class is a `final` class. That means nobody can define a subclass of `String`. In other words, if you have a `String` reference, then you know it refers to a `String` and nothing but a `String`.

Some programmers believe that you should declare all methods as `final` unless you have a good reason that you want polymorphism. In fact, in C++ and C#, methods do not use polymorphism unless you specifically request it. That may be a bit extreme, but we agree that it is a good idea to think carefully about final methods and classes when you design a class hierarchy.

In the early days of Java, some programmers used the `final` keyword in the hope of avoiding the overhead of dynamic binding. If a method is not overridden, and it is short, then a compiler can optimize the method call away—a process called *inlining*. For example, inlining the call `e.getName()` replaces it with the field access `e.name`. This is a worthwhile improvement—CPUs hate branching because it interferes with their strategy of prefetching instructions while processing the current one. However, if `getName` can be overridden in another class, then the compiler cannot inline it because it has no way of knowing what the overriding code may do.

Fortunately, the just-in-time compiler in the virtual machine can do a better job than a traditional compiler. It knows exactly which classes extend a given class, and it can check whether any class actually overrides a given method. If a method is short, frequently called, and not actually overridden, the just-in-time compiler can inline the method. What happens if the virtual machine loads another subclass that overrides an inlined method? Then the optimizer must undo the inlining. That's slow, but it happens rarely.

C++ NOTE: In C++, a method is not dynamically bound by default, and you can tag it as `inline` to have method calls replaced with the method source code. However, there is no mechanism that would prevent a subclass from overriding a superclass method. In C++, you can write classes from which no other class can derive, but doing so requires an obscure trick, and there are few reasons to write such a class. (The obscure trick is left as an exercise to the reader. Hint: Use a virtual base class.)

Casting

Recall from Chapter 3 that the process of forcing a conversion from one type to another is called casting. The Java programming language has a special notation for casts. For example,

```
double x = 3.405;
int nx = (int) x;
```

converts the value of the expression `x` into an integer, discarding the fractional part.

Just as you occasionally need to convert a floating-point number to an integer, you also need to convert an object reference from one class to another. To actually make a cast of an object reference, you use a syntax similar to what you use for casting a numeric expression. Surround the target class name with parentheses and place it before the object reference you want to cast. For example:

```
Manager boss = (Manager) staff[0];
```

There is only one reason why you would want to make a cast—to use an object in its full capacity after its actual type has been temporarily forgotten. For example, in the `ManagerTest` class, the `staff` array had to be an array of `Employee` objects because *some* of its entries were regular employees. We would need to cast the managerial elements of the array back to `Manager` to access any of its new variables. (Note that in the sample code for the first section, we made a special effort to avoid the cast. We initialized the `boss` variable with a `Manager` object before storing it in the array. We needed the correct type to set the bonus of the manager.)

As you know, in Java every object variable has a type. The type describes the kind of object the variable refers to and what it can do. For example, `staff[i]` refers to an `Employee` object (so it can also refer to a `Manager` object).

The compiler checks that you do not promise too much when you store a value in a variable. If you assign a subclass reference to a superclass variable, you are promising less, and the compiler will simply let you do it. If you assign a superclass reference to a subclass variable, you are promising more. Then you must use a cast so that your promise can be checked at runtimeruntime.

What happens if you try to cast down an inheritance chain and you are "lying" about what an object contains?

```
Manager boss = (Manager) staff[1]; // ERROR
```

When the program runs, the Java runtime system notices the broken promise and generates a `ClassCastException`. If you do not catch the exception, your program terminates. Thus, it is good programming practice to find out whether a cast will succeed before attempting it. Simply use the `instanceof` operator. For example:

```
if (staff[1] instanceof Manager)
{
   boss = (Manager) staff[1];
   . . .
}
```

Finally, the compiler will not let you make a cast if there is no chance for the cast to succeed. For example, the cast

```
Date c = (Date) staff[1];
```

is a compile-time error because `Date` is not a subclass of `Employee`.

To sum up:

- You can cast only within an inheritance hierarchy.
- Use `instanceof` to check before casting from a superclass to a subclass.

NOTE: The test

```
x instanceof C
```

does not generate an exception if `x` is `null`. It simply returns `false`. That makes sense. Because `null` refers to no object, it certainly doesn't refer to an object of type `C`.

Actually, converting the type of an object by performing a cast is not usually a good idea. In our example, you do not need to cast an `Employee` object to a `Manager` object for most purposes. The `getSalary` method will work correctly on both objects of both classes. The dynamic binding that makes polymorphism work locates the correct method automatically.

The only reason to make the cast is to use a method that is unique to managers, such as `setBonus`. If for some reason you find yourself wanting to call `setBonus` on `Employee` objects, ask yourself whether this is an indication of a design flaw in the superclass. It may make sense to redesign the superclass and add a `setBonus` method. Remember, it takes only one uncaught `ClassCastException` to terminate your program. In general, it is best to minimize the use of casts and the `instanceof` operator.

C++ NOTE: Java uses the cast syntax from the "bad old days" of C, but it works like the safe `dynamic_cast` operation of C++. For example,

```
Manager boss = (Manager) staff[1]; // Java
```

is the same as

```
Manager* boss = dynamic_cast<Manager*>(staff[1]); // C++
```

with one important difference. If the cast fails, it does not yield a null object but throws an exception. In this sense, it is like a C++ cast of *references*. This is a pain in the neck. In C++, you can take care of the type test and type conversion in one operation.

```
Manager* boss = dynamic_cast<Manager*>(staff[1]); // C++
if (boss != NULL) . . .
```

In Java, you use a combination of the `instanceof` operator and a cast.

```
if (staff[1] instanceof Manager)
{
   Manager boss = (Manager) staff[1];
   . . .
}
```

Abstract Classes

As you move up the inheritance hierarchy, classes become more general and probably more abstract. At some point, the ancestor class becomes *so* general that you think of it more as a basis for other classes than as a class with specific instances you want to use. Consider, for example, an extension of our `Employee` class hierarchy. An employee is a person, and so is a student. Let us extend our class hierarchy to include classes `Person` and `Student`. Figure 5–2 shows the inheritance relationships between these classes.

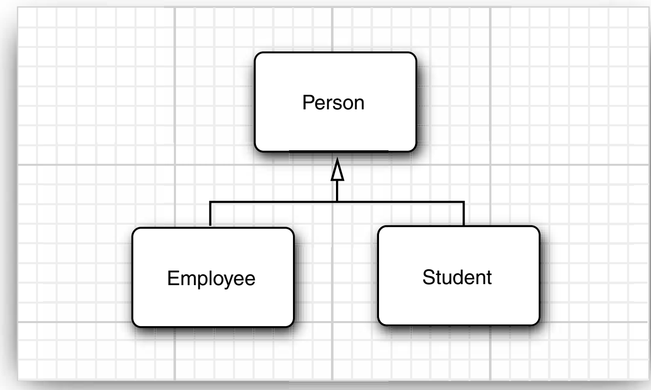


Figure 5–2 Inheritance diagram for `Person` and its subclasses

Why bother with so high a level of abstraction? There are some attributes that make sense for every person, such as the name. Both students and employees have names,

and introducing a common superclass lets us factor out the `getName` method to a higher level in the inheritance hierarchy.

Now let's add another method, `getDescription`, whose purpose is to return a brief description of the person, such as

```
an employee with a salary of $50,000.00
a student majoring in computer science
```

It is easy to implement this method for the `Employee` and `Student` classes. But what information can you provide in the `Person` class? The `Person` class knows nothing about the person except the name. Of course, you could implement `Person.getDescription()` to return an empty string. But there is a better way. If you use the `abstract` keyword, you do not need to implement the method at all.

```
public abstract String getDescription();
   // no implementation required
```

For added clarity, a class with one or more abstract methods must itself be declared abstract.

```
abstract class Person
{  . . .
   public abstract String getDescription();
}
```

In addition to abstract methods, abstract classes can have fields and concrete methods. For example, the `Person` class stores the name of the person and has a concrete method that returns it.

```
abstract class Person
{
   public Person(String n)
   {
      name = n;
   }

   public abstract String getDescription();

   public String getName()
   {
       return name;
   }

   private String name;
}
```

TIP: Some programmers don't realize that abstract classes can have concrete methods. You should always move common fields and methods (whether abstract or not) to the superclass (whether abstract or not).

Abstract methods act as placeholders for methods that are implemented in the subclasses. When you extend an abstract class, you have two choices. You can leave some or all of the abstract methods undefined. Then you must tag the subclass as abstract as well. Or you can define all methods. Then the subclass is no longer abstract.

For example, we will define a `Student` class that extends the abstract `Person` class and implements the `getDescription` method. Because none of the methods of the `Student` class are abstract, it does not need to be declared as an abstract class.

A class can even be declared as `abstract` even though it has no abstract methods.

Abstract classes cannot be instantiated. That is, if a class is declared as `abstract`, no objects of that class can be created. For example, the expression

```
new Person("Vince Vu")
```

is an error. However, you can create objects of concrete subclasses.

Note that you can still create *object variables* of an abstract class, but such a variable must refer to an object of a nonabstract subclass. For example:

```
Person p = new Student("Vince Vu", "Economics");
```

Here `p` is a variable of the abstract type `Person` that refers to an instance of the nonabstract subclass `Student`.

C++ NOTE: In C++, an abstract method is called a *pure virtual function* and is tagged with a trailing = 0, such as in

```
class Person // C++
{
public:
   virtual string getDescription() = 0;
   . . .
};
```

A C++ class is abstract if it has at least one pure virtual function. In C++, there is no special keyword to denote abstract classes.

Let us define a concrete subclass `Student` that extends the abstract `Person` class:

```
class Student extends Person
{
   public Student(String n, String m)
   {
      super(n);
      major = m;
   }

   public String getDescription()
   {
      return "a student majoring in " + major;
   }

   private String major;
}
```

The `Student` class defines the `getDescription` method. Therefore, all methods in the `Student` class are concrete, and the class is no longer an abstract class.

The program shown in Listing 5–2 defines the abstract superclass `Person` and two concrete subclasses, `Employee` and `Student`. We fill an array of `Person` references with employee and student objects:

```
Person[] people = new Person[2];
people[0] = new Employee(. . .);
people[1] = new Student(. . .);
```

We then print the names and descriptions of these objects:

```
for (Person p : people)
   System.out.println(p.getName() + ", " + p.getDescription());
```

Some people are baffled by the call

```
p.getDescription()
```

Isn't this call an undefined method? Keep in mind that the variable `p` never refers to a `Person` object because it is impossible to construct an object of the abstract `Person` class. The variable `p` always refers to an object of a concrete subclass such as `Employee` or `Student`. For these objects, the `getDescription` method is defined.

Could you have omitted the abstract method altogether from the `Person` superclass and simply defined the `getDescription` methods in the `Employee` and `Student` subclasses? If you did that, then you wouldn't have been able to invoke the `getDescription` method on the variable `p`. The compiler ensures that you invoke only methods that are declared in the class.

Abstract methods are an important concept in the Java programming language. You will encounter them most commonly inside *interfaces*. For more information about interfaces, turn to Chapter 6.

Listing 5–2 PersonTest.java

```
1. import java.util.*;
2.
3. /**
4.  * This program demonstrates abstract classes.
5.  * @version 1.01 2004-02-21
6.  * @author Cay Horstmann
7.  */
8. public class PersonTest
9. {
10.    public static void main(String[] args)
11.    {
12.       Person[] people = new Person[2];
13.
14.       // fill the people array with Student and Employee objects
15.       people[0] = new Employee("Harry Hacker", 50000, 1989, 10, 1);
16.       people[1] = new Student("Maria Morris", "computer science");
17.
18.       // print out names and descriptions of all Person objects
19.       for (Person p : people)
20.          System.out.println(p.getName() + ", " + p.getDescription());
21.    }
```

Listing 5–2 PersonTest.java (continued)

```
22. }
23.
24. abstract class Person
25. {
26.    public Person(String n)
27.    {
28.       name = n;
29.    }
30.
31.    public abstract String getDescription();
32.
33.    public String getName()
34.    {
35.       return name;
36.    }
37.
38.    private String name;
39. }
40.
41. class Employee extends Person
42. {
43.    public Employee(String n, double s, int year, int month, int day)
44.    {
45.       super(n);
46.       salary = s;
47.       GregorianCalendar calendar = new GregorianCalendar(year, month - 1, day);
48.       hireDay = calendar.getTime();
49.    }
50.
51.    public double getSalary()
52.    {
53.       return salary;
54.    }
55.
56.    public Date getHireDay()
57.    {
58.       return hireDay;
59.    }
60.
61.    public String getDescription()
62.    {
63.       return String.format("an employee with a salary of $%.2f", salary);
64.    }
65.
66.    public void raiseSalary(double byPercent)
67.    {
68.       double raise = salary * byPercent / 100;
69.       salary += raise;
70.    }
```

Listing 5–2 PersonTest.java (continued)

```
71.
72.    private double salary;
73.    private Date hireDay;
74. }
75.
76. class Student extends Person
77. {
78.    /**
79.     * @param n the student's name
80.     * @param m the student's major
81.     */
82.    public Student(String n, String m)
83.    {
84.       // pass n to superclass constructor
85.       super(n);
86.       major = m;
87.    }
88.
89.    public String getDescription()
90.    {
91.       return "a student majoring in " + major;
92.    }
93.
94.    private String major;
95. }
```

Protected Access

As you know, fields in a class are best tagged as `private`, and methods are usually tagged as `public`. Any features declared `private` won't be visible to other classes. As we said at the beginning of this chapter, this is also true for subclasses: a subclass cannot access the private fields of its superclass.

There are times, however, when you want to restrict a method to subclasses only or, less commonly, to allow subclass methods to access a superclass field. In that case, you declare a class feature as `protected`. For example, if the superclass `Employee` declares the `hireDay` field as `protected` instead of private, then the `Manager` methods can access it directly.

However, the `Manager` class methods can peek inside the `hireDay` field of `Manager` objects only, not of other `Employee` objects. This restriction is made so that you can't abuse the protected mechanism and form subclasses just to gain access to the protected fields.

In practice, use `protected` fields with caution. Suppose your class is used by other programmers and you designed it with protected fields. Unknown to you, other programmers may inherit classes from your class and then start accessing your protected fields. In this case, you can no longer change the implementation of your class without upsetting the other programmers. That is against the spirit of OOP, which encourages data encapsulation.

Protected methods make more sense. A class may declare a method as `protected` if it is tricky to use. This indicates that the subclasses (which, presumably, know their ancestors well) can be trusted to use the method correctly, but other classes cannot.

A good example of this kind of method is the `clone` method of the `Object` class—see Chapter 6 for more details.

C++ NOTE: As it happens, protected features in Java are visible to all subclasses as well as to all other classes in the same package. This is slightly different from the C++ meaning of protected, and it makes the notion of `protected` in Java even less safe than in C++.

Here is a summary of the four access modifiers in Java that control visibility:

1. Visible to the class only (`private`).
2. Visible to the world (`public`).
3. Visible to the package and all subclasses (`protected`).
4. Visible to the package—the (unfortunate) default. No modifiers are needed.

`Object`: The Cosmic Superclass

The `Object` class is the ultimate ancestor—every class in Java extends `Object`. However, you never have to write

```
class Employee extends Object
```

The ultimate superclass `Object` is taken for granted if no superclass is explicitly mentioned. Because *every* class in Java extends `Object`, it is important to be familiar with the services provided by the `Object` class. We go over the basic ones in this chapter and refer you to later chapters or to the on-line documentation for what is not covered here. (Several methods of `Object` come up only when dealing with threads—see Volume II for more on threads.)

You can use a variable of type `Object` to refer to objects of any type:

```
Object obj = new Employee("Harry Hacker", 35000);
```

Of course, a variable of type `Object` is only useful as a generic holder for arbitrary values. To do anything specific with the value, you need to have some knowledge about the original type and then apply a cast:

```
Employee e = (Employee) obj;
```

In Java, only the *primitive types* (numbers, characters, and `boolean` values) are not objects.

All array types, no matter whether they are arrays of objects or arrays of primitive types, are class types that extend the `Object` class.

```
Employee[] staff = new Employee[10];
obj = staff; // OK
obj = new int[10]; // OK
```

C++ NOTE: In C++, there is no cosmic root class. However, every pointer can be converted to a `void*` pointer.

The `equals` *Method*

The `equals` method in the `Object` class tests whether one object is considered equal to another. The `equals` method, as implemented in the `Object` class, determines whether two object references are identical. This is a pretty reasonable default—if two objects are identical, they should certainly be equal. For quite a few classes, nothing else is required. For example, it makes little sense to compare two `PrintStream` objects for equality. However, you will often want to implement state-based equality testing, in which two objects are considered equal when they have the same state.

For example, let us consider two employees equal if they have the same name, salary, and hire date. (In an actual employee database, it would be more sensible to compare IDs instead. We use this example to demonstrate the mechanics of implementing the `equals` method.)

```
class Employee
{
   . . .
   public boolean equals(Object otherObject)
   {
      // a quick test to see if the objects are identical
      if (this == otherObject) return true;

      // must return false if the explicit parameter is null
      if (otherObject == null) return false;

      // if the classes don't match, they can't be equal
      if (getClass() != otherObject.getClass())
         return false;

      // now we know otherObject is a non-null Employee
      Employee other = (Employee) otherObject;

      // test whether the fields have identical values
      return name.equals(other.name)
         && salary == other.salary
         && hireDay.equals(other.hireDay);
   }
}
```

The `getClass` method returns the class of an object—we discuss this method in detail later in this chapter. In our test, two objects can only be equal when they belong to the same class.

When you define the `equals` method for a subclass, first call `equals` on the superclass. If that test doesn't pass, then the objects can't be equal. If the superclass fields are equal, then you are ready to compare the instance fields of the subclass.

```
class Manager extends Employee
{
   . . .
   public boolean equals(Object otherObject)
   {
      if (!super.equals(otherObject)) return false;
```

```
        // super.equals checked that this and otherObject belong to the same class
        Manager other = (Manager) otherObject;
        return bonus == other.bonus;
    }
}
```

Equality Testing and Inheritance

How should the `equals` method behave if the implicit and explicit parameters don't belong to the same class? This has been an area of some controversy. In the preceding example, the `equals` method returns `false` if the classes don't match exactly. But many programmers use an `instanceof` test instead:

```
if (!(otherObject instanceof Employee)) return false;
```

This leaves open the possibility that `otherObject` can belong to a subclass. However, this approach can get you into trouble. Here is why. The Java Language Specification requires that the `equals` method has the following properties:

1. It is *reflexive*: For any non-null reference `x`, `x.equals(x)` should return `true`.
2. It is *symmetric*: For any references `x` and `y`, `x.equals(y)` should return `true` if and only if `y.equals(x)` returns `true`.
3. It is *transitive*: For any references `x`, `y`, and `z`, if `x.equals(y)` returns `true` and `y.equals(z)` returns `true`, then `x.equals(z)` should return `true`.
4. It is *consistent*: If the objects to which `x` and `y` refer haven't changed, then repeated calls to `x.equals(y)` return the same value.
5. For any non-null reference `x`, `x.equals(null)` should return `false`.

These rules are certainly reasonable. You wouldn't want a library implementor to ponder whether to call `x.equals(y)` or `y.equals(x)` when locating an element in a data structure.

However, the symmetry rule has subtle consequences when the parameters belong to different classes. Consider a call

```
e.equals(m)
```

where `e` is an `Employee` object and `m` is a `Manager` object, both of which happen to have the same name, salary, and hire date. If `Employee.equals` uses an `instanceof` test, the call returns `true`. But that means that the reverse call

```
m.equals(e)
```

also needs to return `true`—the symmetry rule does not allow it to return `false` or to throw an exception.

That leaves the `Manager` class in a bind. Its `equals` method must be willing to compare itself to any `Employee`, without taking manager-specific information into account! All of a sudden, the `instanceof` test looks less attractive!

Some authors have gone on record that the `getClass` test is wrong because it violates the substitution principle. A commonly cited example is the `equals` method in the `AbstractSet` class that tests whether two sets have the same elements. The `AbstractSet` class has two concrete subclasses, `TreeSet` and `HashSet`, that use different algorithms for locating set elements. You really want to be able to compare any two sets, no matter how they are implemented.

However, the set example is rather specialized. It would make sense to declare `AbstractSet.equals` as `final`, because nobody should redefine the semantics of set equality. (The method is not actually `final`. This allows a subclass to implement a more efficient algorithm for the equality test.)

The way we see it, there are two distinct scenarios:

- If subclasses can have their own notion of equality, then the symmetry requirement forces you to use the `getClass` test.
- If the notion of equality is fixed in the superclass, then you can use the `instanceof` test and allow objects of different subclasses to be equal to another.

In the example of the employees and managers, we consider two objects to be equal when they have matching fields. If we have two `Manager` objects with the same name, salary, and hire date, but with different bonuses, we want them to be different. Therefore, we used the `getClass` test.

But suppose we used an employee ID for equality testing. This notion of equality makes sense for all subclasses. Then we could use the `instanceof` test, and we should declare `Employee.equals` as `final`.

NOTE: The standard Java library contains over 150 implementations of `equals` methods, with a mishmash of using `instanceof`, calling `getClass`, catching a `ClassCastException`, or doing nothing at all.

Here is a recipe for writing the perfect `equals` method:

1. Name the explicit parameter `otherObject`—later, you need to cast it to another variable that you should call `other`.
2. Test whether `this` happens to be identical to `otherObject`:
   ```
   if (this == otherObject) return true;
   ```
   This statement is just an optimization. In practice, this is a common case. It is much cheaper to check for identity than to compare the fields.
3. Test whether `otherObject` is `null` and return `false` if it is. This test is required.
   ```
   if (otherObject == null) return false;
   ```
4. Compare the classes of `this` and `otherObject`. If the semantics of `equals` can change in subclasses, use the `getClass` test:
   ```
   if (getClass() != otherObject.getClass()) return false;
   ```
   If the same semantics holds for *all* subclasses, you can use an `instanceof` test:
   ```
   if (!(otherObject instanceof ClassName)) return false;
   ```
5. Cast `otherObject` to a variable of your class type:
   ```
   ClassName other = (ClassName) otherObject
   ```
6. Now compare the fields, as required by your notion of equality. Use `==` for primitive type fields, `equals` for object fields. Return `true` if all fields match, `false` otherwise.
   ```
   return field1 == other.field1
      && field2.equals(other.field2)
      && . . .;
   ```
   If you redefine `equals` in a subclass, include a call to `super.equals(other)`.

TIP: If you have fields of array type, you can use the static `Arrays.equals` method to check that corresponding array elements are equal.

CAUTION: Here is a common mistake when implementing the `equals` method. Can you spot the problem?

```
public class Employee
{
   public boolean equals(Employee other)
   {
      return name.equals(other.name)
         && salary == other.salary
         && hireDay.equals(other.hireDay);
   }
   ...
}
```

This method declares the explicit parameter type as `Employee`. As a result, it does not override the `equals` method of the `Object` class but defines a completely unrelated method.

Starting with Java SE 5.0, you can protect yourself against this type of error by tagging methods that are intended to override superclass methods with `@Override`:

```
@Override public boolean equals(Object other)
```

If you made a mistake and you are defining a new method, the compiler reports an error. For example, suppose you add the following declaration to the `Employee` class:

```
@Override public boolean equals(Employee other)
```

An error is reported because this method doesn't override any method from the `Object` superclass.

API `java.util.Arrays` 1.2

- `static boolean equals(`*type*`[] a, `*type*`[] b)` 5.0
  returns `true` if the arrays have equal lengths and equal elements in corresponding positions. The arrays can have component types `Object`, `int`, `long`, `short`, `char`, `byte`, `boolean`, `float`, or `double`.

The `hashCode` *Method*

A hash code is an integer that is derived from an object. Hash codes should be scrambled—if `x` and `y` are two distinct objects, there should be a high probability that `x.hashCode()` and `y.hashCode()` are different. Table 5–1 lists a few examples of hash codes that result from the `hashCode` method of the `String` class.

The `String` class uses the following algorithm to compute the hash code:

```
int hash = 0;
for (int i = 0; i < length(); i++)
   hash = 31 * hash + charAt(i);
```

Table 5–1 Hash Codes Resulting from the hashCode Function

String	Hash Code
Hello	69609650
Harry	69496448
Hacker	–2141031506

The hashCode method is defined in the Object class. Therefore, every object has a default hash code. That hash code is derived from the object's memory address. Consider this example:

```
String s = "Ok";
StringBuilder sb = new StringBuilder(s);
System.out.println(s.hashCode() + " " + sb.hashCode());
String t = new String("Ok");
StringBuilder tb = new StringBuilder(t);
System.out.println(t.hashCode() + " " + tb.hashCode());
```

Table 5–2 shows the result.

Table 5–2 Hash Codes of Strings and String Builders

Object	Hash Code
s	2556
sb	20526976
t	2556
tb	20527144

Note that the strings s and t have the same hash code because, for strings, the hash codes are derived from their *contents.* The string builders sb and tb have different hash codes because no hashCode method has been defined for the StringBuilder class, and the default hashCode method in the Object class derives the hash code from the object's memory address.

If you redefine the equals method, you will also need to redefine the hashCode method for objects that users might insert into a hash table. (We discuss hash tables in Chapter 2 of Volume II.)

The hashCode method should return an integer (which can be negative). Just combine the hash codes of the instance fields so that the hash codes for different objects are likely to be widely scattered.

For example, here is a hashCode method for the Employee class:

```
class Employee
{
   public int hashCode()
   {
```

```
        return 7 * name.hashCode()
           + 11 * new Double(salary).hashCode()
           + 13 * hireDay.hashCode();
     }
     . . .
  }
```

Your definitions of `equals` and `hashCode` must be compatible: if `x.equals(y)` is true, then `x.hashCode()` must be the same value as `y.hashCode()`. For example, if you define `Employee.equals` to compare employee IDs, then the `hashCode` method needs to hash the IDs, not employee names or memory addresses.

> TIP: If you have fields of array type, you can use the static `Arrays.hashCode` method to compute a hash code that is composed of the hash codes of the array elements.

API `java.lang.Object` 1.0

- `int hashCode()`
  returns a hash code for this object. A hash code can be any integer, positive or negative. Equal objects need to return identical hash codes.

API `java.util.Arrays` 1.2

- `static int hashCode(`*type*`[] a)` 5.0
  computes the hash code of the array `a`, which can have component type `Object`, `int`, `long`, `short`, `char`, `byte`, `boolean`, `float`, or `double`.

The `toString` *Method*

Another important method in `Object` is the `toString` method that returns a string representing the value of this object. Here is a typical example. The `toString` method of the `Point` class returns a string like this:

```
java.awt.Point[x=10,y=20]
```

Most (but not all) `toString` methods follow this format: the name of the class, followed by the field values enclosed in square brackets. Here is an implementation of the `toString` method for the `Employee` class:

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

Actually, you can do a little better. Rather than hardwiring the class name into the `toString` method, call `getClass().getName()` to obtain a string with the class name.

```
public String toString()
{
   return getClass().getName()
```

```
         + "[name=" + name
         + ",salary=" + salary
         + ",hireDay=" + hireDay
         + "]";
}
```

The toString method then also works for subclasses.

Of course, the subclass programmer should define its own toString method and add the subclass fields. If the superclass uses getClass().getName(), then the subclass can simply call super.toString(). For example, here is a toString method for the Manager class:

```
class Manager extends Employee
{
   . . .
   public String toString()
   {
      return super.toString()
        + "[bonus=" + bonus
        + "]";
   }
}
```

Now a Manager object is printed as

```
Manager[name=...,salary=...,hireDay=...][bonus=...]
```

The toString method is ubiquitous for an important reason: whenever an object is concatenated with a string by the "+" operator, the Java compiler automatically invokes the toString method to obtain a string representation of the object. For example:

```
Point p = new Point(10, 20);
String message = "The current position is " + p;
   // automatically invokes p.toString()
```

TIP: Instead of writing x.toString(), you can write "" + x. This statement concatenates the empty string with the string representation of x that is exactly x.toString(). Unlike toString, this statement even works if x is of primitive type.

If x is any object and you call

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

then the println method simply calls x.toString() and prints the resulting string.

The Object class defines the toString method to print the class name and the hash code of the object. For example, the call

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

produces an output that looks like this:

```
java.io.PrintStream@2f6684
```

The reason is that the implementor of the PrintStream class didn't bother to override the toString method.

CAUTION: Annoyingly, arrays inherit the `toString` method from `Object`, with the added twist that the array type is printed in an archaic format. For example,

```
int[] luckyNumbers = { 2, 3, 5, 7, 11, 13 };
String s = "" + luckyNumbers;
```

yields the string `"[I@1a46e30"`. (The prefix `[I` denotes an array of integers.) The remedy is to call the static `Arrays.toString` method instead. The code

```
String s = Arrays.toString(luckyNumbers);
```

yields the string `"[2, 3, 5, 7, 11, 13]"`.

To correctly print multidimensional arrays (that is, arrays of arrays), use `Arrays.deepToString`.

The `toString` method is a great tool for logging. Many classes in the standard class library define the `toString` method so that you can get useful information about the state of an object. This is particularly useful in logging messages like this:

```
System.out.println("Current position = " + position);
```

As we explain in Chapter 11, an even better solution is

```
Logger.global.info("Current position = " + position);
```

TIP: We strongly recommend that you add a `toString` method to each class that you write. You, as well as other programmers who use your classes, will be grateful for the logging support.

The program in Listing 5–3 implements the `equals`, `hashCode`, and `toString` methods for the `Employee` and `Manager` classes.

Listing 5–3 EqualsTest.java

```
1. import java.util.*;
2.
3. /**
4.  * This program demonstrates the equals method.
5.  * @version 1.11 2004-02-21
6.  * @author Cay Horstmann
7.  */
8. public class EqualsTest
9. {
10.    public static void main(String[] args)
11.    {
12.       Employee alice1 = new Employee("Alice Adams", 75000, 1987, 12, 15);
13.       Employee alice2 = alice1;
14.       Employee alice3 = new Employee("Alice Adams", 75000, 1987, 12, 15);
15.       Employee bob = new Employee("Bob Brandson", 50000, 1989, 10, 1);
16.
17.       System.out.println("alice1 == alice2: " + (alice1 == alice2));
18.
```

Listing 5–3 EqualsTest.java (continued)

```
19.       System.out.println("alice1 == alice3: " + (alice1 == alice3));
20.
21.       System.out.println("alice1.equals(alice3): " + alice1.equals(alice3));
22.
23.       System.out.println("alice1.equals(bob): " + alice1.equals(bob));
24.
25.       System.out.println("bob.toString(): " + bob);
26.
27.       Manager carl = new Manager("Carl Cracker", 80000, 1987, 12, 15);
28.       Manager boss = new Manager("Carl Cracker", 80000, 1987, 12, 15);
29.       boss.setBonus(5000);
30.       System.out.println("boss.toString(): " + boss);
31.       System.out.println("carl.equals(boss): " + carl.equals(boss));
32.       System.out.println("alice1.hashCode(): " + alice1.hashCode());
33.       System.out.println("alice3.hashCode(): " + alice3.hashCode());
34.       System.out.println("bob.hashCode(): " + bob.hashCode());
35.       System.out.println("carl.hashCode(): " + carl.hashCode());
36.    }
37. }
38.
39. class Employee
40. {
41.    public Employee(String n, double s, int year, int month, int day)
42.    {
43.       name = n;
44.       salary = s;
45.       GregorianCalendar calendar = new GregorianCalendar(year, month - 1, day);
46.       hireDay = calendar.getTime();
47.    }
48.
49.    public String getName()
50.    {
51.       return name;
52.    }
53.
54.    public double getSalary()
55.    {
56.       return salary;
57.    }
58.
59.    public Date getHireDay()
60.    {
61.       return hireDay;
62.    }
63.
64.    public void raiseSalary(double byPercent)
65.    {
66.       double raise = salary * byPercent / 100;
67.       salary += raise;
68.    }
```

Listing 5–3 EqualsTest.java (continued)

```
69.
70.    public boolean equals(Object otherObject)
71.    {
72.       // a quick test to see if the objects are identical
73.       if (this == otherObject) return true;
74.
75.       // must return false if the explicit parameter is null
76.       if (otherObject == null) return false;
77.
78.       // if the classes don't match, they can't be equal
79.       if (getClass() != otherObject.getClass()) return false;
80.
81.       // now we know otherObject is a non-null Employee
82.       Employee other = (Employee) otherObject;
83.
84.       // test whether the fields have identical values
85.       return name.equals(other.name) && salary == other.salary && hireDay.equals(other.hireDay);
86.    }
87.
88.    public int hashCode()
89.    {
90.       return 7 * name.hashCode() + 11 * new Double(salary).hashCode() + 13 * hireDay.hashCode();
91.    }
92.
93.    public String toString()
94.    {
95.       return getClass().getName() + "[name=" + name + ",salary=" + salary + ",hireDay=" + hireDay
96.             + "]";
97.    }
98.
99.    private String name;
100.   private double salary;
101.   private Date hireDay;
102. }
103.
104. class Manager extends Employee
105. {
106.   public Manager(String n, double s, int year, int month, int day)
107.   {
108.      super(n, s, year, month, day);
109.      bonus = 0;
110.   }
111.
112.   public double getSalary()
113.   {
114.      double baseSalary = super.getSalary();
115.      return baseSalary + bonus;
116.   }
117.
```

Listing 5–3 EqualsTest.java (continued)

```
118.
119.    public void setBonus(double b)
120.    {
121.       bonus = b;
122.    }
123.
124.    public boolean equals(Object otherObject)
125.    {
126.       if (!super.equals(otherObject)) return false;
127.       Manager other = (Manager) otherObject;
128.       // super.equals checked that this and other belong to the same class
129.       return bonus == other.bonus;
130.    }
131.
132.    public int hashCode()
133.    {
134.       return super.hashCode() + 17 * new Double(bonus).hashCode();
135.    }
136.
137.    public String toString()
138.    {
139.       return super.toString() + "[bonus=" + bonus + "]";
140.    }
141.
142.    private double bonus;
143. }
```

API java.lang.Object 1.0

- `Class getClass()`
  returns a class object that contains information about the object. As you see later in this chapter, Java has a runtime representation for classes that is encapsulated in the `Class` class.
- `boolean equals(Object otherObject)`
  compares two objects for equality; returns `true` if the objects point to the same area of memory, and `false` otherwise. You should override this method in your own classes.
- `String toString()`
  returns a string that represents the value of this object. You should override this method in your own classes.
- `Object clone()`
  creates a clone of the object. The Java runtime system allocates memory for the new instance and copies the memory allocated for the current object.

NOTE: Cloning an object is important, but it also turns out to be a fairly subtle process filled with potential pitfalls for the unwary. We will have a lot more to say about the `clone` method in Chapter 6.

API `java.lang.Class` 1.0

- `String getName()`
  returns the name of this class.
- `Class getSuperclass()`
  returns the superclass of this class as a `Class` object.

Generic Array Lists

In many programming languages—in particular, in C—you have to fix the sizes of all arrays at compile time. Programmers hate this because it forces them into uncomfortable trade-offs. How many employees will be in a department? Surely no more than 100. What if there is a humongous department with 150 employees? Do we want to waste 90 entries for every department with just 10 employees?

In Java, the situation is much better. You can set the size of an array at runtime.

```
int actualSize = . . .;
Employee[] staff = new Employee[actualSize];
```

Of course, this code does not completely solve the problem of dynamically modifying arrays at runtime. Once you set the array size, you cannot change it easily. Instead, the easiest way in Java to deal with this common situation is to use another Java class, called `ArrayList`. The `ArrayList` class is similar to an array, but it automatically adjusts its capacity as you add and remove elements, without your needing to write any code.

As of Java SE 5.0, `ArrayList` is a *generic class* with a *type parameter*. To specify the type of the element objects that the array list holds, you append a class name enclosed in angle brackets, such as ArrayList<Employee>. You will see in Chapter 13 how to define your own generic class, but you don't need to know any of those technicalities to use the `ArrayList` type.

Here we declare and construct an array list that holds `Employee` objects:

```
ArrayList<Employee> staff = new ArrayList<Employee>();
```

NOTE: Before Java SE 5.0, there were no generic classes. Instead, there was a single `ArrayList` class, a "one size fits all" collection that holds elements of type `Object`. If you must use an older version of Java, simply drop all <. . .> suffixes. You can still use `ArrayList` without a <. . .> suffix in Java SE 5.0 and beyond. It is considered a "raw" type, with the type parameter erased.

NOTE: In even older versions of the Java programming language, programmers used the `Vector` class for dynamic arrays. However, the `ArrayList` class is more efficient, and there is no longer any good reason to use the `Vector` class.

You use the add method to add new elements to an array list. For example, here is how you populate an array list with employee objects:

```
staff.add(new Employee("Harry Hacker", . . .));
staff.add(new Employee("Tony Tester", . . .));
```

The array list manages an internal array of object references. Eventually, that array will run out of space. This is where array lists work their magic: If you call add and the internal array is full, the array list automatically creates a bigger array and copies all the objects from the smaller to the bigger array.

If you already know, or have a good guess, how many elements you want to store, then call the ensureCapacity method before filling the array list:

```
staff.ensureCapacity(100);
```

That call allocates an internal array of 100 objects. Then, the first 100 calls to add do not involve any costly reallocation.

You can also pass an initial capacity to the ArrayList constructor:

```
ArrayList<Employee> staff = new ArrayList<Employee>(100);
```

CAUTION: Allocating an array list as

```
new ArrayList<Employee>(100) // capacity is 100
```

is *not* the same as allocating a new array as

```
new Employee[100] // size is 100
```

There is an important distinction between the capacity of an array list and the size of an array. If you allocate an array with 100 entries, then the array has 100 slots, ready for use. An array list with a capacity of 100 elements has the *potential* of holding 100 elements (and, in fact, more than 100, at the cost of additional reallocations); but at the beginning, even after its initial construction, an array list holds no elements at all.

The size method returns the actual number of elements in the array list. For example,

```
staff.size()
```

returns the current number of elements in the staff array list. This is the equivalent of

```
a.length
```

for an array a.

Once you are reasonably sure that the array list is at its permanent size, you can call the trimToSize method. This method adjusts the size of the memory block to use exactly as much storage space as is required to hold the current number of elements. The garbage collector will reclaim any excess memory.

Once you trim the size of an array list, adding new elements will move the block again, which takes time. You should only use trimToSize when you are sure you won't add any more elements to the array list.

C++ NOTE: The ArrayList class is similar to the C++ vector template. Both ArrayList and vector are generic types. But the C++ vector template overloads the [] operator for convenient element access. Because Java does not have operator overloading, it must use explicit method calls instead. Moreover, C++ vectors are copied by value. If a and b are two vectors, then the assignment a = b makes a into a new vector with the same length as b, and all elements are copied from b to a. The same assignment in Java makes both a and b refer to the same array list.

API **java.util.ArrayList<T> 1.2**

- `ArrayList<T>()`
  constructs an empty array list.
- `ArrayList<T>(int initialCapacity)`
  constructs an empty array list with the specified capacity.

  Parameters: `initialCapacity` the initial storage capacity of the array list
- `boolean add(T obj)`
  appends an element at the end of the array list. Always returns true.

  Parameters: `obj` the element to be added
- `int size()`
  returns the number of elements currently stored in the array list. (Of course, this is never larger than the array list's capacity.)
- `void ensureCapacity(int capacity)`
  ensures that the array list has the capacity to store the given number of elements without reallocating its internal storage array.

  Parameters: `capacity` the desired storage capacity
- `void trimToSize()`
  reduces the storage capacity of the array list to its current size.

Accessing Array List Elements

Unfortunately, nothing comes for free. The automatic growth convenience that array lists give requires a more complicated syntax for accessing the elements. The reason is that the ArrayList class is not a part of the Java programming language; it is just a utility class programmed by someone and supplied in the standard library.

Instead of using the pleasant [] syntax to access or change the element of an array, you use the get and set methods.

For example, to set the ith element, you use

```
staff.set(i, harry);
```

This is equivalent to

```
a[i] = harry;
```

for an array a. (As with arrays, the index values are zero-based.)

CAUTION: Do not call `list.set(i, x)` until the *size* of the array list is larger than `i`. For example, the following code is wrong:

```
ArrayList<Employee> list = new ArrayList<Employee>(100); // capacity 100, size 0
list.set(0, x); // no element 0 yet
```

Use the `add` method instead of `set` to fill up an array, and use `set` only to replace a previously added element.

To get an array list element, use

```
Employee e = staff.get(i);
```

This is equivalent to

```
Employee e = a[i];
```

NOTE: Before Java SE 5.0, there were no generic classes, and the `get` method of the raw `ArrayList` class had no choice but to return an `Object`. Consequently, callers of `get` had to cast the returned value to the desired type:

```
Employee e = (Employee) staff.get(i);
```

The raw `ArrayList` is also a bit dangerous. Its `add` and `set` methods accept objects of any type. A call

```
staff.set(i, new Date());
```

compiles without so much as a warning, and you run into grief only when you retrieve the object and try to cast it. If you use an `ArrayList<Employee>` instead, the compiler will detect this error.

You can sometimes get the best of both worlds—flexible growth and convenient element access—with the following trick. First, make an array list and add all the elements:

```
ArrayList<X> list = new ArrayList<X>();
while (. . .)
{
   x = . . .;
   list.add(x);
}
```

When you are done, use the `toArray` method to copy the elements into an array:

```
X[] a = new X[list.size()];
list.toArray(a);
```

Sometimes, you need to add elements in the middle of an array list. Use the `add` method with an index parameter:

```
int n = staff.size() / 2;
staff.add(n, e);
```

The elements at locations `n` and above are shifted up to make room for the new entry. If the new size of the array list after the insertion exceeds the capacity, then the array list reallocates its storage array.

Similarly, you can remove an element from the middle of an array list:

```
Employee e = staff.remove(n);
```

The elements located above it are copied down, and the size of the array is reduced by one.

Inserting and removing elements is not terribly efficient. It is probably not worth worrying about for small array lists. But if you store many elements and frequently insert and remove in the middle of a collection, consider using a linked list instead. We explain how to program with linked lists in Chapter 13.

As of Java SE 5.0, you can use the "for each" loop to traverse the contents of an array list:

```
for (Employee e : staff)
   do something with e
```

This loop has the same effect as

```
for (int i = 0; i < staff.size(); i++)
{
   Employee e = staff.get(i);
   do something with e
}
```

Listing 5–4 is a modification of the `EmployeeTest` program of Chapter 4. The `Employee[]` array is replaced by an `ArrayList<Employee>`. Note the following changes:

- You don't have to specify the array size.
- You use `add` to add as many elements as you like.
- You use `size()` instead of `length` to count the number of elements.
- You use `a.get(i)` instead of `a[i]` to access an element.

Listing 5–4 ArrayListTest.java

```
1. import java.util.*;
2.
3. /**
4.  * This program demonstrates the ArrayList class.
5.  * @version 1.1 2004-02-21
6.  * @author Cay Horstmann
7.  */
8. public class ArrayListTest
9. {
10.    public static void main(String[] args)
11.    {
12.       // fill the staff array list with three Employee objects
13.       ArrayList<Employee> staff = new ArrayList<Employee>();
14.
15.       staff.add(new Employee("Carl Cracker", 75000, 1987, 12, 15));
16.       staff.add(new Employee("Harry Hacker", 50000, 1989, 10, 1));
17.       staff.add(new Employee("Tony Tester", 40000, 1990, 3, 15));
18.
```

Listing 5–4 ArrayListTest.java (continued)

```
19.       // raise everyone's salary by 5%
20.       for (Employee e : staff)
21.          e.raiseSalary(5);
22.
23.       // print out information about all Employee objects
24.       for (Employee e : staff)
25.          System.out.println("name=" + e.getName() + ",salary=" + e.getSalary() + ",hireDay="
26.                + e.getHireDay());
27.    }
28. }
29.
30. class Employee
31. {
32.    public Employee(String n, double s, int year, int month, int day)
33.    {
34.       name = n;
35.       salary = s;
36.       GregorianCalendar calendar = new GregorianCalendar(year, month - 1, day);
37.       hireDay = calendar.getTime();
38.    }
39.
40.    public String getName()
41.    {
42.       return name;
43.    }
44.
45.    public double getSalary()
46.    {
47.       return salary;
48.    }
49.
50.    public Date getHireDay()
51.    {
52.       return hireDay;
53.    }
54.
55.    public void raiseSalary(double byPercent)
56.    {
57.       double raise = salary * byPercent / 100;
58.       salary += raise;
59.    }
60.
61.    private String name;
62.    private double salary;
63.    private Date hireDay;
64. }
```

API `java.util.ArrayList<T>` 1.2

- `void set(int index, T obj)`
  puts a value in the array list at the specified index, overwriting the previous contents.

  Parameters: `index` the position (must be between `0` and `size() - 1`)
  `obj` the new value

- `T get(int index)`
  gets the value stored at a specified index.

  Parameters: `index` the index of the element to get (must be between `0` and `size() - 1`)

- `void add(int index, T obj)`
  shifts up elements to insert an element.

  Parameters: `index` the insertion position (must be between `0` and `size()`)
  `obj` the new element

- `T remove(int index)`
  removes an element and shifts down all elements above it. The removed element is returned.

  Parameters: `index` the position of the element to be removed (must be between `0` and `size() - 1`)

Compatibility between Typed and Raw Array Lists

When you write new code with Java SE 5.0 and beyond, you should use type parameters, such as `ArrayList<Employee>`, for array lists. However, you may need to interoperate with existing code that uses the raw `ArrayList` type.

Suppose that you have the following legacy class:

```
public class EmployeeDB
{
   public void update(ArrayList list) { ... }
   public ArrayList find(String query) { ... }
}
```

You can pass a typed array list to the `update` method without any casts.

```
ArrayList<Employee> staff = ...;
employeeDB.update(staff);
```

The `staff` object is simply passed to the `update` method.

CAUTION: Even though you get no error or warning from the compiler, this call is not completely safe. The `update` method might add elements into the array list that are not of type `Employee`. When these elements are retrieved, an exception occurs. This sounds scary, but if you think about it, the behavior is simply as it was before Java SE 5.0. The integrity of the virtual machine is never jeopardized. In this situation, you do not lose security, but you also do not benefit from the compile-time checks.

Conversely, when you assign a raw `ArrayList` to a typed one, you get a warning.

```
ArrayList<Employee> result = employeeDB.find(query); // yields warning
```

NOTE: To see the text of the warning, compile with the option `-Xlint:unchecked`.

Using a cast does not make the warning go away.

```
ArrayList<Employee> result = (ArrayList<Employee>)
   employeeDB.find(query); // yields another warning
```

Instead, you get a different warning, telling you that the cast is misleading.

This is the consequence of a somewhat unfortunate limitation of generic types in Java. For compatibility, the compiler translates all typed array lists into raw `ArrayList` objects after checking that the type rules were not violated. In a running program, all array lists are the same—there are no type parameters in the virtual machine. Thus, the casts `(ArrayList)` and `(ArrayList<Employee>)` carry out identical runtime checks.

There isn't much you can do about that situation. When you interact with legacy code, study the compiler warnings and satisfy yourself that the warnings are not serious.

Object Wrappers and Autoboxing

Occasionally, you need to convert a primitive type like `int` to an object. All primitive types have class counterparts. For example, a class `Integer` corresponds to the primitive type `int`. These kinds of classes are usually called *wrappers*. The wrapper classes have obvious names: `Integer`, `Long`, `Float`, `Double`, `Short`, `Byte`, `Character`, `Void`, and `Boolean`. (The first six inherit from the common superclass `Number`.) The wrapper classes are immutable—you cannot change a wrapped value after the wrapper has been constructed. They are also `final`, so you cannot subclass them.

Suppose we want an array list of integers. Unfortunately, the type parameter inside the angle brackets cannot be a primitive type. It is not possible to form an `ArrayList<int>`. Here, the `Integer` wrapper class comes in. It is ok to declare an array list of `Integer` objects.

```
ArrayList<Integer> list = new ArrayList<Integer>();
```

CAUTION: An `ArrayList<Integer>` is far less efficient than an `int[]` array because each value is separately wrapped inside an object. You would only want to use this construct for small collections when programmer convenience is more important than efficiency.

Another Java SE 5.0 innovation makes it easy to add and get array elements. The call

```
list.add(3);
```

is automatically translated to

```
list.add(new Integer(3));
```

This conversion is called *autoboxing*.

NOTE: You might think that *autowrapping* would be more consistent, but the “boxing” metaphor was taken from C#.

Conversely, when you assign an `Integer` object to an `int` value, it is automatically unboxed. That is, the compiler translates

```
int n = list.get(i);
```

into

```
int n = list.get(i).intValue();
```

Automatic boxing and unboxing even works with arithmetic expressions. For example, you can apply the increment operator to a wrapper reference:

```
Integer n = 3;
n++;
```

The compiler automatically inserts instructions to unbox the object, increment the resulting value, and box it back.

In most cases, you get the illusion that the primitive types and their wrappers are one and the same. There is just one point in which they differ considerably: identity. As you know, the `==` operator, applied to wrapper objects, only tests whether the objects have identical memory locations. The following comparison would therefore probably fail:

```
Integer a = 1000;
Integer b = 1000;
if (a == b) ...
```

However, a Java implementation *may*, if it chooses, wrap commonly occurring values into identical objects, and thus the comparison might succeed. This ambiguity is not what you want. The remedy is to call the `equals` method when comparing wrapper objects.

NOTE: The autoboxing specification requires that `boolean`, `byte`, `char` ≤ 127, and `short` and `int` between –128 and 127 are wrapped into fixed objects. For example, if `a` and `b` had been initialized with 100 in the preceding example, then the comparison would have had to succeed.

Finally, let us emphasize that boxing and unboxing is a courtesy of the *compiler*, not the virtual machine. The compiler inserts the necessary calls when it generates the bytecodes of a class. The virtual machine simply executes those bytecodes.

You will often see the number wrappers for another reason. The designers of Java found the wrappers a convenient place to put certain basic methods, like the ones for converting strings of digits to numbers.

To convert a string to an integer, you use the following statement:

```
int x = Integer.parseInt(s);
```

This has nothing to do with `Integer` objects—`parseInt` is a static method. But the `Integer` class was a good place to put it.

The API notes show some of the more important methods of the `Integer` class. The other number classes implement corresponding methods.

CAUTION: Some people think that the wrapper classes can be used to implement methods that can modify numeric parameters. However, that is not correct. Recall from Chapter 4 that it is impossible to write a Java method that increments an integer parameter because parameters to Java methods are always passed by value.

```
public static void triple(int x) // won't work
{
   x = 3 * x; // modifies local variable
}
```

Could we overcome this by using an `Integer` instead of an int?

```
public static void triple(Integer x) // won't work
{
   ...
}
```

The problem is that `Integer` objects are *immutable*: the information contained inside the wrapper can't change. You cannot use these wrapper classes to create a method that modifies numeric parameters.

If you do want to write a method to change numeric parameters, you can use one of the *holder* types defined in the `org.omg.CORBA` package. There are types `IntHolder`, `BooleanHolder`, and so on. Each holder type has a public (!) field `value` through which you can access the stored value.

```
public static void triple(IntHolder x)
{
   x.value = 3 * x.value;
}
```

API **`java.lang.Integer`** 1.0

- `int intValue()`
  returns the value of this `Integer` object as an `int` (overrides the `intValue` method in the `Number` class).
- `static String toString(int i)`
  returns a new `String` object representing the number `i` in base 10.
- `static String toString(int i, int radix)`
  lets you return a representation of the number `i` in the base specified by the `radix` parameter.
- `static int parseInt(String s)`
- `static int parseInt(String s, int radix)`
  returns the integer whose digits are contained in the string `s`. The string must represent an integer in base 10 (for the first method) or in the base given by the `radix` parameter (for the second method).
- `static Integer valueOf(String s)`
- `static Integer valueOf(String s, int radix)`
  returns a new `Integer` object initialized to the integer whose digits are contained in the string `s`. The string must represent an integer in base 10 (for the first method) or in the base given by the `radix` parameter (for the second method).

API `java.text.NumberFormat` 1.1

- `Number parse(String s)`
  returns the numeric value, assuming the specified `String` represents a number.

Methods with a Variable Number of Parameters

Before Java SE 5.0, every Java method had a fixed number of parameters. However, it is now possible to provide methods that can be called with a variable number of parameters. (These are sometimes called "varargs" methods.)

You have already seen such a method: `printf`. For example, the calls

```
System.out.printf("%d", n);
```

and

```
System.out.printf("%d %s", n, "widgets");
```

both call the same method, even though one call has two parameters and the other has three.

The `printf` method is defined like this:

```
public class PrintStream
{
   public PrintStream printf(String fmt, Object... args) { return format(fmt, args); }
}
```

Here, the ellipsis ... is a part of the Java code. It denotes that the method can receive an arbitrary number of objects (in addition to the `fmt` parameter).

The `printf` method actually receives two parameters, the format string, and an `Object[]` array that holds all other parameters. (If the caller supplies integers or other primitive type values, autoboxing turns them into objects.) It now has the unenviable task of scanning the `fmt` string and matching up the ith format specifier with the value `args[i]`.

In other words, for the implementor of `printf`, the `Object...` parameter type is exactly the same as `Object[]`.

The compiler needs to transform each call to `printf`, bundling the parameters into an array and autoboxing as necessary:

```
System.out.printf("%d %s", new Object[] { new Integer(n), "widgets" } );
```

You can define your own methods with variable parameters, and you can specify any type for the parameters, even a primitive type. Here is a simple example: a function that computes the maximum of a variable number of values.

```
public static double max(double... values)
{
   double largest = Double.MIN_VALUE;
   for (double v : values) if (v > largest) largest = v;
   return largest;
}
```

Simply call the function like this:

```
double m = max(3.1, 40.4, -5);
```

The compiler passes a `new double[] { 3.1, 40.4, -5 }` to the `max` function.

NOTE: It is legal to pass an array as the last parameter of a method with variable parameters. For example:

```
System.out.printf("%d %s", new Object[] { new Integer(1), "widgets" } );
```

Therefore, you can redefine an existing function whose last parameter is an array to a method with variable parameters, without breaking any existing code. For example, `MessageFormat.format` was enhanced in this way in Java SE 5.0. If you like, you can even declare the `main` method as

```
public static void main(String... args)
```

Enumeration Classes

You saw in Chapter 3 how to define enumerated types in Java SE 5.0 and beyond. Here is a typical example:

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

The type defined by this declaration is actually a class. The class has exactly four instances—it is not possible to construct new objects.

Therefore, you never need to use `equals` for values of enumerated types. Simply use `==` to compare them.

You can, if you like, add constructors, methods, and fields to an enumerated type. Of course, the constructors are only invoked when the enumerated constants are constructed. Here is an example.

```
enum Size
{
   SMALL("S"), MEDIUM("M"), LARGE("L"), EXTRA_LARGE("XL");

   private Size(String abbreviation) { this.abbreviation = abbreviation; }
   public String getAbbreviation() { return abbreviation; }

   private String abbreviation;
}
```

All enumerated types are subclasses of the class `Enum`. They inherit a number of methods from that class. The most useful one is `toString`, which returns the name of the enumerated constant. For example, `Size.SMALL.toString()` returns the string `"SMALL"`.

The converse of `toString` is the static `valueOf` method. For example, the statement

```
Size s = (Size) Enum.valueOf(Size.class, "SMALL");
```

sets `s` to `Size.SMALL`.

Each enumerated type has a static `values` method that returns an array of all values of the enumeration. For example, the call

```
Size[] values = Size.values();
```

returns the array with elements `Size.SMALL`, `Size.MEDIUM`, `Size.LARGE`, and `Size.EXTRA_LARGE`.

The `ordinal` method yields the position of an enumerated constant in the `enum` declaration, counting from zero. For example, `Size.MEDIUM.ordinal()` returns 1.

The short program in Listing 5–5 demonstrates how to work with enumerated types.

NOTE: The `Enum` class has a type parameter that we have ignored for simplicity. For example, the enumerated type `Size` actually extends `Enum<Size>`. The type parameter is used in the `compareTo` method. (We discuss the `compareTo` method in Chapter 6 and type parameters in Chapter 12.)

Listing 5–5 EnumTest.java

```
1. import java.util.*;
2.
3. /**
4.  * This program demonstrates enumerated types.
5.  * @version 1.0 2004-05-24
6.  * @author Cay Horstmann
7.  */
8. public class EnumTest
9. {
10.    public static void main(String[] args)
11.    {
12.       Scanner in = new Scanner(System.in);
13.       System.out.print("Enter a size: (SMALL, MEDIUM, LARGE, EXTRA_LARGE) ");
14.       String input = in.next().toUpperCase();
15.       Size size = Enum.valueOf(Size.class, input);
16.       System.out.println("size=" + size);
17.       System.out.println("abbreviation=" + size.getAbbreviation());
18.       if (size == Size.EXTRA_LARGE)
19.          System.out.println("Good job--you paid attention to the _.");
20.    }
21. }
22.
23. enum Size
24. {
25.    SMALL("S"), MEDIUM("M"), LARGE("L"), EXTRA_LARGE("XL");
26.
27.    private Size(String abbreviation) { this.abbreviation = abbreviation; }
28.    public String getAbbreviation() { return abbreviation; }
29.
30.    private String abbreviation;
31. }
```

API **java.lang.Enum<E> 5.0**

- `static Enum valueOf(Class enumClass, String name)`
  returns the enumerated constant of the given class with the given name.
- `String toString()`
  returns the name of this enumerated constant.
- `int ordinal()`
  returns the zero-based position of this enumerated constant in the `enum` declaration.

- `int compareTo(E other)`
  returns a negative integer if this enumerated constant comes before `other`, zero if `this == other`, and a positive integer otherwise. The ordering of the constants is given by the `enum` declaration.

Reflection

The *reflection library* gives you a very rich and elaborate toolset to write programs that manipulate Java code dynamically. This feature is heavily used in *JavaBeans*, the component architecture for Java (see Volume II for more on JavaBeans). Using reflection, Java can support tools like the ones to which users of Visual Basic have grown accustomed. In particular, when new classes are added at design or runtime, rapid application development tools can dynamically inquire about the capabilities of the classes that were added.

A program that can analyze the capabilities of classes is called *reflective*. The reflection mechanism is extremely powerful. As the next sections show, you can use it to

- Analyze the capabilities of classes at runtime;
- Inspect objects at runtime, for example, to write a single `toString` method that works for *all* classes;
- Implement generic array manipulation code; and
- Take advantage of `Method` objects that work just like function pointers in languages such as C++.

Reflection is a powerful and complex mechanism; however, it is of interest mainly to tool builders, not application programmers. If you are interested in programming applications rather than tools for other Java programmers, you can safely skip the remainder of this chapter and return to it later.

The `Class` *Class*

While your program is running, the Java runtime system always maintains what is called runtime type identification on all objects. This information keeps track of the class to which each object belongs. Runtime type information is used by the virtual machine to select the correct methods to execute.

However, you can also access this information by working with a special Java class. The class that holds this information is called, somewhat confusingly, `Class`. The `getClass()` method in the `Object` class returns an instance of `Class` type.

```
Employee e;
. . .
Class cl = e.getClass();
```

Just like an `Employee` object describes the properties of a particular employee, a `Class` object describes the properties of a particular class. Probably the most commonly used method of `Class` is `getName`. This returns the name of the class. For example, the statement

```
System.out.println(e.getClass().getName() + " " + e.getName());
```

prints

```
Employee Harry Hacker
```

if `e` is an employee, or

```
Manager Harry Hacker
```

if `e` is a manager.

If the class is in a package, the package name is part of the class name:

```
Date d = new Date();
Class cl = d.getClass();
String name = cl.getName(); // name is set to "java.util.Date"
```

You can obtain a `Class` object corresponding to a class name by using the static `forName` method.

```
String className = "java.util.Date";
Class cl = Class.forName(className);
```

You would use this method if the class name is stored in a string that varies at runtim-eruntime. This works if `className` is the name of a class or interface. Otherwise, the `forName` method throws a *checked exception*. See the section "A Primer on Catching Exceptions" on page 219 to see how to supply an *exception handler* whenever you use this method.

TIP: At startup, the class containing your `main` method is loaded. It loads all classes that it needs. Each of those loaded classes loads the classes that it needs, and so on. That can take a long time for a big application, frustrating the user. You can give users of your program the illusion of a faster start with the following trick. Make sure that the class containing the `main` method does not explicitly refer to other classes. First display a splash screen. Then manually force the loading of other classes by calling `Class.forName`.

A third method for obtaining an object of type `Class` is a convenient shorthand. If `T` is any Java type, then `T.class` is the matching class object. For example:

```
Class cl1 = Date.class; // if you import java.util.*;
Class cl2 = int.class;
Class cl3 = Double[].class;
```

Note that a `Class` object really describes a *type*, which may or may not be a class. For example, `int` is not a class, but `int.class` is nevertheless an object of type `Class`.

NOTE: As of Java SE 5.0, the `Class` class is parameterized. For example, `Employee.class` is of type `Class<Employee>`. We are not dwelling on this issue because it would further complicate an already abstract concept. For most practical purposes, you can ignore the type parameter and work with the raw `Class` type. See Chapter 13 for more information on this issue.

CAUTION: For historical reasons, the `getName` method returns somewhat strange names for array types:

- `Double[].class.getName()` returns `"[Ljava.lang.Double;"`
- `int[].class.getName()` returns `"[I"`

The virtual machine manages a unique `Class` object for each type. Therefore, you can use the `==` operator to compare class objects. For example:

```
if (e.getClass() == Employee.class) . . .
```

Another example of a useful method is one that lets you create an instance of a class on the fly. This method is called, naturally enough, `newInstance()`. For example,

```
e.getClass().newInstance();
```

creates a new instance of the same class type as `e`. The `newInstance` method calls the default constructor (the one that takes no parameters) to initialize the newly created object. An exception is thrown if the class has no default constructor.

Using a combination of `forName` and `newInstance` lets you create an object from a class name stored in a string.

```
String s = "java.util.Date";
Object m = Class.forName(s).newInstance();
```

NOTE: If you need to provide parameters for the constructor of a class you want to create by name in this manner, then you can't use statements like the preceding. Instead, you must use the `newInstance` method in the `Constructor` class.

C++ NOTE: The `newInstance` method corresponds to the idiom of a *virtual constructor* in C++. However, virtual constructors in C++ are not a language feature but just an idiom that needs to be supported by a specialized library. The `Class` class is similar to the `type_info` class in C++, and the `getClass` method is equivalent to the `typeid` operator. The Java `Class` is quite a bit more versatile than `type_info`, though. The C++ `type_info` can only reveal a string with the name of the type, not create new objects of that type.

A Primer on Catching Exceptions

We cover exception handling fully in Chapter 11, but in the meantime you will occasionally encounter methods that threaten to throw exceptions.

When an error occurs at runtime, a program can "throw an exception." Throwing an exception is more flexible than terminating the program because you can provide a *handler* that "catches" the exception and deals with it.

If you don't provide a handler, the program still terminates and prints a message to the console, giving the type of the exception. You may already have seen exception reports when you accidentally used a `null` reference or overstepped the bounds of an array.

There are two kinds of exceptions: *unchecked* exceptions and *checked* exceptions. With checked exceptions, the compiler checks that you provide a handler. However, many common exceptions, such as accessing a null reference, are unchecked. The compiler does not check whether you provide a handler for these errors—after all, you should spend your mental energy on avoiding these mistakes rather than coding handlers for them.

But not all errors are avoidable. If an exception can occur despite your best efforts, then the compiler insists that you provide a handler. The `Class.forName` method is an example of a method that throws a checked exception. In Chapter 11, you will see several exception handling strategies. For now, we just show you the simplest handler implementation.

Place one or more statements that might throw checked exceptions inside a `try` block. Then provide the handler code in the `catch` clause.

```
try
{
   statements that might throw exceptions
}
catch(Exception e)
{
   handler action
}
```

Here is an example:

```
try
{
   String name = . . .; // get class name
   Class cl = Class.forName(name); // might throw exception
   . . . // do something with cl
}
catch(Exception e)
{
   e.printStackTrace();
}
```

If the class name doesn't exist, the remainder of the code in the `try` block is skipped and the program enters the `catch` clause. (Here, we print a stack trace by using the `printStackTrace` method of the `Throwable` class. `Throwable` is the superclass of the `Exception` class.) If none of the methods in the `try` block throws an exception, the handler code in the `catch` clause is skipped.

You only need to supply an exception handler for checked exceptions. It is easy to find out which methods throw checked exceptions—the compiler will complain whenever you call a method that threatens to throw a checked exception and you don't supply a handler.

API **`java.lang.Class`** 1.0

- `static Class forName(String className)`
  returns the `Class` object representing the class with name `className`.
- `Object newInstance()`
  returns a new instance of this class.

API **`java.lang.reflect.Constructor`** 1.1

- `Object newInstance(Object[] args)`
  constructs a new instance of the constructor's declaring class.

  Parameters: `args` the parameters supplied to the constructor. See the section on reflection for more information on how to supply parameters.

API **`java.lang.Throwable` 1.0**

- `void printStackTrace()`
  prints the `Throwable` object and the stack trace to the standard error stream.

Using Reflection to Analyze the Capabilities of Classes

Here is a brief overview of the most important parts of the reflection mechanism for letting you examine the structure of a class.

The three classes `Field`, `Method`, and `Constructor` in the `java.lang.reflect` package describe the fields, methods, and constructors of a class, respectively. All three classes have a method called `getName` that returns the name of the item. The `Field` class has a method `getType` that returns an object, again of type `Class`, that describes the field type. The `Method` and `Constructor` classes have methods to report the types of the parameters, and the `Method` class also reports the return type. All three of these classes also have a method called `getModifiers` that returns an integer, with various bits turned on and off, that describes the modifiers used, such as `public` and `static`. You can then use the static methods in the `Modifier` class in the `java.lang.reflect` package to analyze the integer that `getModifiers` returns. Use methods like `isPublic`, `isPrivate`, or `isFinal` in the `Modifier` class to tell whether a method or constructor was `public`, `private`, or `final`. All you have to do is have the appropriate method in the `Modifier` class work on the integer that `getModifiers` returns. You can also use the `Modifier.toString` method to print the modifiers.

The `getFields`, `getMethods`, and `getConstructors` methods of the `Class` class return arrays of the *public* fields, methods, and constructors that the class supports. This includes public members of superclasses. The `getDeclaredFields`, `getDeclaredMethods`, and `getDeclaredConstructors` methods of the `Class` class return arrays consisting of all fields, methods, and constructors that are declared in the class. This includes private and protected members, but not members of superclasses.

Listing 5–6 shows you how to print out all information about a class. The program prompts you for the name of a class and then writes out the signatures of all methods and constructors as well as the names of all data fields of a class. For example, if you enter

```
java.lang.Double
```

the program prints

```
public class java.lang.Double extends java.lang.Number
{
   public java.lang.Double(java.lang.String);
   public java.lang.Double(double);

   public int hashCode();
   public int compareTo(java.lang.Object);
   public int compareTo(java.lang.Double);
   public boolean equals(java.lang.Object);
   public java.lang.String toString();
   public static java.lang.String toString(double);
   public static java.lang.Double valueOf(java.lang.String);
   public static boolean isNaN(double);
   public boolean isNaN();
   public static boolean isInfinite(double);
   public boolean isInfinite();
```

```
    public byte byteValue();
    public short shortValue();
    public int intValue();
    public long longValue();
    public float floatValue();
    public double doubleValue();
    public static double parseDouble(java.lang.String);
    public static native long doubleToLongBits(double);
    public static native long doubleToRawLongBits(double);
    public static native double longBitsToDouble(long);

    public static final double POSITIVE_INFINITY;
    public static final double NEGATIVE_INFINITY;
    public static final double NaN;
    public static final double MAX_VALUE;
    public static final double MIN_VALUE;
    public static final java.lang.Class TYPE;
    private double value;
    private static final long serialVersionUID;
 }
```

What is remarkable about this program is that it can analyze any class that the Java interpreter can load, not just the classes that were available when the program was compiled. We use this program in the next chapter to peek inside the inner classes that the Java compiler generates automatically.

Listing 5–6 ReflectionTest.java

```
1. import java.util.*;
2. import java.lang.reflect.*;
3.
4. /**
5.  * This program uses reflection to print all features of a class.
6.  * @version 1.1 2004-02-21
7.  * @author Cay Horstmann
8.  */
9. public class ReflectionTest
10. {
11.    public static void main(String[] args)
12.    {
13.       // read class name from command line args or user input
14.       String name;
15.       if (args.length > 0) name = args[0];
16.       else
17.       {
18.          Scanner in = new Scanner(System.in);
19.          System.out.println("Enter class name (e.g. java.util.Date): ");
20.          name = in.next();
21.       }
22.
23.       try
24.       {
```

Listing 5–6 ReflectionTest.java (continued)

```
25.          // print class name and superclass name (if != Object)
26.          Class cl = Class.forName(name);
27.          Class supercl = cl.getSuperclass();
28.          String modifiers = Modifier.toString(cl.getModifiers());
29.          if (modifiers.length() > 0) System.out.print(modifiers + " ");
30.          System.out.print("class " + name);
31.          if (supercl != null && supercl != Object.class) System.out.print(" extends "
32.                + supercl.getName());
33.
34.          System.out.print("\n{\n");
35.          printConstructors(cl);
36.          System.out.println();
37.          printMethods(cl);
38.          System.out.println();
39.          printFields(cl);
40.          System.out.println("}");
41.       }
42.       catch (ClassNotFoundException e)
43.       {
44.          e.printStackTrace();
45.       }
46.       System.exit(0);
47.    }
48.
49.    /**
50.     * Prints all constructors of a class
51.     * @param cl a class
52.     */
53.    public static void printConstructors(Class cl)
54.    {
55.       Constructor[] constructors = cl.getDeclaredConstructors();
56.
57.       for (Constructor c : constructors)
58.       {
59.          String name = c.getName();
60.          System.out.print("   ");
61.          String modifiers = Modifier.toString(c.getModifiers());
62.          if (modifiers.length() > 0) System.out.print(modifiers + " ");
63.          System.out.print(name + "(");
64.
65.          // print parameter types
66.          Class[] paramTypes = c.getParameterTypes();
67.          for (int j = 0; j < paramTypes.length; j++)
68.          {
69.             if (j > 0) System.out.print(", ");
70.             System.out.print(paramTypes[j].getName());
71.          }
72.          System.out.println(");");
73.       }
```

Listing 5–6 ReflectionTest.java (continued)

```
74.    }
75.
76.    /**
77.     * Prints all methods of a class
78.     * @param cl a class
79.     */
80.    public static void printMethods(Class cl)
81.    {
82.       Method[] methods = cl.getDeclaredMethods();
83.
84.       for (Method m : methods)
85.       {
86.          Class retType = m.getReturnType();
87.          String name = m.getName();
88.
89.          System.out.print("   ");
90.          // print modifiers, return type, and method name
91.          String modifiers = Modifier.toString(m.getModifiers());
92.          if (modifiers.length() > 0) System.out.print(modifiers + " ");
93.          System.out.print(retType.getName() + " " + name + "(");
94.
95.          // print parameter types
96.          Class[] paramTypes = m.getParameterTypes();
97.          for (int j = 0; j < paramTypes.length; j++)
98.          {
99.             if (j > 0) System.out.print(", ");
100.            System.out.print(paramTypes[j].getName());
101.         }
102.         System.out.println(");");
103.      }
104.   }
105.
106.   /**
107.    * Prints all fields of a class
108.    * @param cl a class
109.    */
110.   public static void printFields(Class cl)
111.   {
112.      Field[] fields = cl.getDeclaredFields();
113.
114.      for (Field f : fields)
115.      {
116.         Class type = f.getType();
117.         String name = f.getName();
118.         System.out.print("   ");
119.         String modifiers = Modifier.toString(f.getModifiers());
120.         if (modifiers.length() > 0) System.out.print(modifiers + " ");
121.         System.out.println(type.getName() + " " + name + ";");
122.      }
123.   }
124. }
```

API **java.lang.Class 1.0**

- `Field[] getFields()` 1.1
- `Field[] getDeclaredFields()` 1.1

  `getFields` returns an array containing `Field` objects for the public fields of this class or its superclasses; `getDeclaredField` returns an array of `Field` objects for all fields of this class. The methods return an array of length 0 if there are no such fields or if the `Class` object represents a primitive or array type.
- `Method[] getMethods()` 1.1
- `Method[] getDeclaredMethods()` 1.1

  returns an array containing `Method` objects: `getMethods` returns public methods and includes inherited methods; `getDeclaredMethods` returns all methods of this class or interface but does not include inherited methods.
- `Constructor[] getConstructors()` 1.1
- `Constructor[] getDeclaredConstructors()` 1.1

  returns an array containing `Constructor` objects that give you all the public constructors (for `getConstructors`) or all constructors (for `getDeclaredConstructors`) of the class represented by this `Class` object.

API **java.lang.reflect.Field 1.1**

API **java.lang.reflect.Method 1.1**

API **java.lang.reflect.Constructor 1.1**

- `Class getDeclaringClass()`

  returns the `Class` object for the class that defines this constructor, method, or field.
- `Class[] getExceptionTypes()` (in `Constructor` and `Method` classes)

  returns an array of `Class` objects that represent the types of the exceptions thrown by the method.
- `int getModifiers()`

  returns an integer that describes the modifiers of this constructor, method, or field. Use the methods in the `Modifier` class to analyze the return value.
- `String getName()`

  returns a string that is the name of the constructor, method, or field.
- `Class[] getParameterTypes()` (in `Constructor` and `Method` classes)

  returns an array of `Class` objects that represent the types of the parameters.
- `Class getReturnType()` (in `Method` classes)

  returns a `Class` object that represents the return type.

API **java.lang.reflect.Modifier 1.1**

- `static String toString(int modifiers)`

  returns a string with the modifiers that correspond to the bits set in `modifiers`.

- `static boolean isAbstract(int modifiers)`
- `static boolean isFinal(int modifiers)`
- `static boolean isInterface(int modifiers)`
- `static boolean isNative(int modifiers)`
- `static boolean isPrivate(int modifiers)`
- `static boolean isProtected(int modifiers)`
- `static boolean isPublic(int modifiers)`
- `static boolean isStatic(int modifiers)`
- `static boolean isStrict(int modifiers)`
- `static boolean isSynchronized(int modifiers)`
- `static boolean isVolatile(int modifiers)`
  tests the bit in the `modifiers` value that corresponds to the modifier in the method name.

Using Reflection to Analyze Objects at Runtime

In the preceding section, we saw how we can find out the *names* and *types* of the data fields of any object:

- Get the corresponding `Class` object.
- Call `getDeclaredFields` on the `Class` object.

In this section, we go one step further and actually look at the *contents* of the data fields. Of course, it is easy to look at the contents of a specific field of an object whose name and type are known when you write a program. But reflection lets you look at fields of objects that were not known at compile time.

The key method to achieve this examination is the `get` method in the `Field` class. If `f` is an object of type `Field` (for example, one obtained from `getDeclaredFields`) and `obj` is an object of the class of which `f` is a field, then `f.get(obj)` returns an object whose value is the current value of the field of `obj`. This is all a bit abstract, so let's run through an example.

```
Employee harry = new Employee("Harry Hacker", 35000, 10, 1, 1989);
Class cl = harry.getClass();
   // the class object representing Employee
Field f = cl.getDeclaredField("name");
   // the name field of the Employee class
Object v = f.get(harry);
   // the value of the name field of the harry object
   // i.e., the String object "Harry Hacker"
```

Actually, there is a problem with this code. Because the `name` field is a private field, the `get` method will throw an `IllegalAccessException`. You can only use the `get` method to get the values of accessible fields. The security mechanism of Java lets you find out what fields any object has, but it won't let you read the values of those fields unless you have access permission.

The default behavior of the reflection mechanism is to respect Java access control. However, if a Java program is not controlled by a security manager that disallows it, you can override access control. To do this, invoke the `setAccessible` method on a `Field`, `Method`, or `Constructor` object. For example:

```
f.setAccessible(true); // now OK to call f.get(harry);
```

The `setAccessible` method is a method of the `AccessibleObject` class, the common superclass of the `Field`, `Method`, and `Constructor` classes. This feature is provided for debuggers, persistent storage, and similar mechanisms. We use it for a generic `toString` method later in this section.

There is another issue with the `get` method that we need to deal with. The `name` field is a `String`, and so it is not a problem to return the value as an `Object`. But suppose we want to look at the `salary` field. That is a `double`, and in Java, number types are not objects. To handle this, you can either use the `getDouble` method of the `Field` class, or you can call `get`, whereby the reflection mechanism automatically wraps the field value into the appropriate wrapper class, in this case, `Double`.

Of course, you can also set the values that you can get. The call `f.set(obj, value)` sets the field represented by `f` of the object `obj` to the new value.

Listing 5–7 shows how to write a generic `toString` method that works for *any* class. It uses `getDeclaredFields` to obtain all data fields. It then uses the `setAccessible` convenience method to make all fields accessible. For each field, it obtains the name and the value. Listing 5–7 turns each value into a string by recursively invoking `toString`.

```
class ObjectAnalyzer
{
  public String toString(Object obj)
  {
      Class cl = obj.getClass();
      . . .
      String r = cl.getName();
      // inspect the fields of this class and all superclasses
      do
      {
         r += "[";
         Field[] fields = cl.getDeclaredFields();
         AccessibleObject.setAccessible(fields, true);
         // get the names and values of all fields
         for (Field f : fields)
         {
            if (!Modifier.isStatic(f.getModifiers()))
            {
               if (!r.endsWith("[")) r += ","
               r += f.getName() + "=";
               try
               {
                  Object val = f.get(obj);
                  r += toString(val);
               }
               catch (Exception e) { e.printStackTrace(); }
            }
         }
         r += "]";
         cl = cl.getSuperclass();
      }
      while (cl != null);
      return r;
   }
   . . .
}
```

The complete code in Listing 5–7 needs to address a couple of complexities. Cycles of references could cause an infinite recursion. Therefore, the `ObjectAnalyzer` keeps track of objects that were already visited. Also, to peek inside arrays, you need a different approach. You'll learn about the details in the next section.

You can use this `toString` method to peek inside any object. For example, the call

```
ArrayList<Integer> squares = new ArrayList<Integer>();
for (int i = 1; i <= 5; i++) squares.add(i * i);
System.out.println(new ObjectAnalyzer().toString(squares));
```

yields the printout

```
java.util.ArrayList[elementData=class java.lang.Object[]{java.lang.Integer[value=1][][],
java.lang.Integer[value=4][][],java.lang.Integer[value=9][][],java.lang.Integer[value=16][][],
java.lang.Integer[value=25][][],null,null,null,null,null},size=5][modCount=5][][]
```

You can use this generic `toString` method to implement the `toString` methods of your own classes, like this:

```
public String toString()
{
   return new ObjectAnalyzer().toString(this);
}
```

This is a hassle-free method for supplying a `toString` method that you may find useful in your own programs.

Listing 5–7 ObjectAnalyzerTest.java

```
1. import java.lang.reflect.*;
2. import java.util.*;
3.
4. /**
5.  * This program uses reflection to spy on objects.
6.  * @version 1.11 2004-02-21
7.  * @author Cay Horstmann
8.  */
9. public class ObjectAnalyzerTest
10. {
11.    public static void main(String[] args)
12.    {
13.       ArrayList<Integer> squares = new ArrayList<Integer>();
14.       for (int i = 1; i <= 5; i++)
15.          squares.add(i * i);
16.       System.out.println(new ObjectAnalyzer().toString(squares));
17.    }
18. }
19.
20. class ObjectAnalyzer
21. {
```

Listing 5–7 ObjectAnalyzerTest.java (continued)

```
22.    /**
23.     * Converts an object to a string representation that lists all fields.
24.     * @param obj an object
25.     * @return a string with the object's class name and all field names and
26.     * values
27.     */
28.    public String toString(Object obj)
29.    {
30.       if (obj == null) return "null";
31.       if (visited.contains(obj)) return "...";
32.       visited.add(obj);
33.       Class cl = obj.getClass();
34.       if (cl == String.class) return (String) obj;
35.       if (cl.isArray())
36.       {
37.          String r = cl.getComponentType() + "[]{";
38.          for (int i = 0; i < Array.getLength(obj); i++)
39.          {
40.             if (i > 0) r += ",";
41.             Object val = Array.get(obj, i);
42.             if (cl.getComponentType().isPrimitive()) r += val;
43.             else r += toString(val);
44.          }
45.          return r + "}";
46.       }
47.
48.       String r = cl.getName();
49.       // inspect the fields of this class and all superclasses
50.       do
51.       {
52.          r += "[";
53.          Field[] fields = cl.getDeclaredFields();
54.          AccessibleObject.setAccessible(fields, true);
55.          // get the names and values of all fields
56.          for (Field f : fields)
57.          {
58.             if (!Modifier.isStatic(f.getModifiers()))
59.             {
60.                if (!r.endsWith("[")) r += ",";
61.                r += f.getName() + "=";
62.                try
63.                {
64.                   Class t = f.getType();
65.                   Object val = f.get(obj);
66.                   if (t.isPrimitive()) r += val;
67.                   else r += toString(val);
68.                }
69.                catch (Exception e)
```

Listing 5–7 ObjectAnalyzerTest.java (continued)

```
70.                {
71.                   e.printStackTrace();
72.                }
73.             }
74.          }
75.          r += "]";
76.          cl = cl.getSuperclass();
77.       }
78.       while (cl != null);
79.
80.       return r;
81.    }
82.
83.    private ArrayList<Object> visited = new ArrayList<Object>();
84. }
```

API `java.lang.reflect.AccessibleObject` 1.2

- `void setAccessible(boolean flag)`
  sets the accessibility flag for this reflection object. A value of `true` indicates that Java language access checking is suppressed and that the private properties of the object can be queried and set.
- `boolean isAccessible()`
  gets the value of the accessibility flag for this reflection object.
- `static void setAccessible(AccessibleObject[] array, boolean flag)`
  is a convenience method to set the accessibility flag for an array of objects.

API `java.lang.Class` 1.1

- `Field getField(String name)`
- `Field[] getFields()`
  gets the public field with the given name, or an array of all fields.
- `Field getDeclaredField(String name)`
- `Field[] getDeclaredFields()`
  gets the field that is declared in this class with the given name, or an array of all fields.

API `java.lang.reflect.Field` 1.1

- `Object get(Object obj)`
  gets the value of the field described by this `Field` object in the object `obj`.
- `void set(Object obj, Object newValue)`
  sets the field described by this `Field` object in the object `obj` to a new value.

Using Reflection to Write Generic Array Code

The `Array` class in the `java.lang.reflect` package allows you to create arrays dynamically. For example, when you use this feature with the `arraycopy` method from Chapter 3, you can dynamically expand an existing array while preserving the current contents.

The problem we want to solve is pretty typical. Suppose you have an array of some type that is full and you want to grow it. And suppose you are sick of writing the grow-and-copy code by hand. You want to write a generic method to grow an array.

```
Employee[] a = new Employee[100];
. . .
// array is full
a = (Employee[]) arrayGrow(a);
```

How can we write such a generic method? It helps that an `Employee[]` array can be converted to an `Object[]` array. That sounds promising. Here is a first attempt to write a generic method. We simply grow the array by 10% + 10 elements (because the 10 percent growth is not substantial enough for small arrays).

```
static Object[] badArrayGrow(Object[] a) // not useful
{
   int newLength = a.length * 11 / 10 + 10;
   Object[] newArray = new Object[newLength];
   System.arraycopy(a, 0, newArray, 0, a.length);
   return newArray;
}
```

However, there is a problem with actually *using* the resulting array. The type of array that this code returns is an array of *objects* (`Object[]`) because we created the array using the line of code

```
new Object[newLength]
```

An array of objects *cannot* be cast to an array of employees (`Employee[]`). Java would generate a `ClassCastException` at runtime. The point is, as we mentioned earlier, that a Java array remembers the type of its entries, that is, the element type used in the `new` expression that created it. It is legal to cast an `Employee[]` temporarily to an `Object[]` array and then cast it back, but an array that started its life as an `Object[]` array can never be cast into an `Employee[]` array. To write this kind of generic array code, we need to be able to make a new array of the *same* type as the original array. For this, we need the methods of the `Array` class in the `java.lang.reflect` package. The key is the static `newInstance` method of the `Array` class that constructs a new array. You must supply the type for the entries and the desired length as parameters to this method.

```
Object newArray = Array.newInstance(componentType, newLength);
```

To actually carry this out, we need to get the length and component type of the new array.

We obtain the length by calling `Array.getLength(a)`. The static `getLength` method of the `Array` class returns the length of any array. To get the component type of the new array:

1. First, get the class object of `a`.
2. Confirm that it is indeed an array.
3. Use the `getComponentType` method of the `Class` class (which is defined only for class objects that represent arrays) to find the right type for the array.

Why is getLength a method of Array but getComponentType a method of Class? We don't know—the distribution of the reflection methods seems a bit ad hoc at times.

Here's the code:

```
static Object goodArrayGrow(Object a) // useful
{
   Class cl = a.getClass();
   if (!cl.isArray()) return null;
   Class componentType = cl.getComponentType();
   int length = Array.getLength(a);
   int newLength = length * 11 / 10 + 10;
   Object newArray = Array.newInstance(componentType, newLength);
   System.arraycopy(a, 0, newArray, 0, length);
   return newArray;
}
```

Note that this arrayGrow method can be used to grow arrays of any type, not just arrays of objects.

```
int[] a = { 1, 2, 3, 4 };
a = (int[]) goodArrayGrow(a);
```

To make this possible, the parameter of goodArrayGrow is declared to be of type Object, *not an array of objects* (Object[]). The integer array type int[] can be converted to an Object, but not to an array of objects!

Listing 5–8 shows both array grow methods in action. Note that the cast of the return value of badArrayGrow will throw an exception.

NOTE: We present this program to illustrate how to work with arrays through reflection. If you just want to grow an array, use the copyOf method in the Arrays class.

```
Employee[] a = new Employee[100];
. . .
// array is full
a = Arrays.copyOf(a, a.length * 11 / 10 + 10);
```

Listing 5–8 ArrayGrowTest.java

```
1. import java.lang.reflect.*;
2.
3. /**
4.  * This program demonstrates the use of reflection for manipulating arrays.
5.  * @version 1.01 2004-02-21
6.  * @author Cay Horstmann
7.  */
8. public class ArrayGrowTest
9. {
10.    public static void main(String[] args)
11.    {
12.       int[] a = { 1, 2, 3 };
13.       a = (int[]) goodArrayGrow(a);
```

Listing 5–8 ArrayGrowTest.java (continued)

```
14.       arrayPrint(a);
15.
16.       String[] b = { "Tom", "Dick", "Harry" };
17.       b = (String[]) goodArrayGrow(b);
18.       arrayPrint(b);
19.
20.       System.out.println("The following call will generate an exception.");
21.       b = (String[]) badArrayGrow(b);
22.    }
23.
24.    /**
25.     * This method attempts to grow an array by allocating a new array and copying all elements.
26.     * @param a the array to grow
27.     * @return a larger array that contains all elements of a. However, the returned array has
28.     * type Object[], not the same type as a
29.     */
30.    static Object[] badArrayGrow(Object[] a)
31.    {
32.       int newLength = a.length * 11 / 10 + 10;
33.       Object[] newArray = new Object[newLength];
34.       System.arraycopy(a, 0, newArray, 0, a.length);
35.       return newArray;
36.    }
37.
38.    /**
39.     * This method grows an array by allocating a new array of the same type and
40.     * copying all elements.
41.     * @param a the array to grow. This can be an object array or a primitive
42.     * type array
43.     * @return a larger array that contains all elements of a.
44.     */
45.    static Object goodArrayGrow(Object a)
46.    {
47.       Class cl = a.getClass();
48.       if (!cl.isArray()) return null;
49.       Class componentType = cl.getComponentType();
50.       int length = Array.getLength(a);
51.       int newLength = length * 11 / 10 + 10;
52.
53.       Object newArray = Array.newInstance(componentType, newLength);
54.       System.arraycopy(a, 0, newArray, 0, length);
55.       return newArray;
56.    }
57.
58.    /**
59.     * A convenience method to print all elements in an array
60.     * @param a the array to print. It can be an object array or a primitive type array
61.     */
62.    static void arrayPrint(Object a)
```

Listing 5–8 ArrayGrowTest.java (continued)

```
63.    {
64.       Class cl = a.getClass();
65.       if (!cl.isArray()) return;
66.       Class componentType = cl.getComponentType();
67.       int length = Array.getLength(a);
68.       System.out.print(componentType.getName() + "[" + length + "] = { ");
69.       for (int i = 0; i < Array.getLength(a); i++)
70.          System.out.print(Array.get(a, i) + " ");
71.       System.out.println("}");
72.    }
73. }
```

API `java.lang.reflect.Array` 1.1

- `static Object get(Object array, int index)`
- `static` *`xxx`* `get`*`Xxx`*`(Object array, int index)`
  (*xxx* is one of the primitive types `boolean`, `byte`, `char`, `double`, `float`, `int`, `long`, `short`.) These methods return the value of the given array that is stored at the given index.
- `static void set(Object array, int index, Object newValue)`
- `static set`*`Xxx`*`(Object array, int index,` *`xxx`* `newValue)`
  (*xxx* is one of the primitive types `boolean`, `byte`, `char`, `double`, `float`, `int`, `long`, `short`.) These methods store a new value into the given array at the given index.
- `static int getLength(Object array)`
  returns the length of the given array.
- `static Object newInstance(Class componentType, int length)`
- `static Object newInstance(Class componentType, int[] lengths)`
  returns a new array of the given component type with the given dimensions.

Method Pointers!

On the surface, Java does not have method pointers—ways of giving the location of a method to another method so that the second method can invoke it later. In fact, the designers of Java have said that method pointers are dangerous and error prone and that Java *interfaces* (discussed in the next chapter) are a superior solution. However, as of Java 1.1, it turns out that Java does have method pointers, as a (perhaps accidental) by-product of the reflection package.

NOTE: Among the nonstandard language extensions that Microsoft added to its Java derivative J++ (and its successor, C#) is another method pointer type, called a *delegate*, that is different from the `Method` class that we discuss in this section. However, inner classes (which we will introduce in the next chapter) are a more useful construct than delegates.

To see method pointers at work, recall that you can inspect a field of an object with the `get` method of the `Field` class. Similarly, the `Method` class has an `invoke` method that lets you call the method that is wrapped in the current `Method` object. The signature for the `invoke` method is

```
Object invoke(Object obj, Object... args)
```

The first parameter is the implicit parameter, and the remaining objects provide the explicit parameters. (Before Java SE 5.0, you had to pass an array of objects or `null` if the method had no explicit parameters.)

For a static method, the first parameter is ignored—you can set it to `null`.

For example, if `m1` represents the `getName` method of the `Employee` class, the following code shows how you can call it:

```
String n = (String) m1.invoke(harry);
```

As with the `get` and `set` methods of the `Field` type, there's a problem if the parameter or return type is not a class but a primitive type. You either rely on autoboxing or, before Java SE 5.0, wrap primitive types into their corresponding wrappers.

Conversely, if the return type is a primitive type, the `invoke` method will return the wrapper type instead. For example, suppose that `m2` represents the `getSalary` method of the `Employee` class. Then, the returned object is actually a `Double`, and you must cast it accordingly. As of Java SE 5.0, automatic unboxing takes care of the rest.

```
double s = (Double) m2.invoke(harry);
```

How do you obtain a `Method` object? You can, of course, call `getDeclaredMethods` and search through the returned array of `Method` objects until you find the method that you want. Or, you can call the `getMethod` method of the `Class` class. This is similar to the `getField` method that takes a string with the field name and returns a `Field` object. However, there may be several methods with the same name, so you need to be careful that you get the right one. For that reason, you must also supply the parameter types of the desired method. The signature of `getMethod` is

```
Method getMethod(String name, Class... parameterTypes)
```

For example, here is how you can get method pointers to the `getName` and `raiseSalary` methods of the `Employee` class:

```
Method m1 = Employee.class.getMethod("getName");
Method m2 = Employee.class.getMethod("raiseSalary", double.class);
```

(Before Java SE 5.0, you had to package the `Class` objects into an array or to supply `null` if there were no parameters.)

Now that you have seen the rules for using `Method` objects, let's put them to work. Listing 5–9 is a program that prints a table of values for a mathematical function such as `Math.sqrt` or `Math.sin`. The printout looks like this:

```
public static native double java.lang.Math.sqrt(double)
      1.0000 |      1.0000
      2.0000 |      1.4142
      3.0000 |      1.7321
      4.0000 |      2.0000
      5.0000 |      2.2361
      6.0000 |      2.4495
      7.0000 |      2.6458
      8.0000 |      2.8284
      9.0000 |      3.0000
      10.0000 |      3.1623
```

The code for printing a table is, of course, independent of the actual function that is being tabulated.

```
double dx = (to - from) / (n - 1);
for (double x = from; x <= to; x += dx)
{
   double y = (Double) f.invoke(null, x);
   System.out.printf("%10.4f | %10.4f%n", x, y);
}
```

Here, `f` is an object of type `Method`. The first parameter of `invoke` is `null` because we are calling a static method.

To tabulate the `Math.sqrt` function, we set `f` to

```
Math.class.getMethod("sqrt", double.class)
```

That is the method of the `Math` class that has the name `sqrt` and a single parameter of type `double`.

Listing 5–9 shows the complete code of the generic tabulator and a couple of test runs.

Listing 5–9 MethodPointerTest.java

```
1. import java.lang.reflect.*;
2.
3. /**
4.  * This program shows how to invoke methods through reflection.
5.  * @version 1.1 2004-02-21
6.  * @author Cay Horstmann
7.  */
8. public class MethodPointerTest
9. {
10.    public static void main(String[] args) throws Exception
11.    {
12.       // get method pointers to the square and sqrt methods
13.       Method square = MethodPointerTest.class.getMethod("square", double.class);
14.       Method sqrt = Math.class.getMethod("sqrt", double.class);
15.
16.       // print tables of x- and y-values
17.
18.       printTable(1, 10, 10, square);
19.       printTable(1, 10, 10, sqrt);
20.    }
21.
22.    /**
23.     * Returns the square of a number
24.     * @param x a number
25.     * @return x squared
26.     */
27.    public static double square(double x)
28.    {
29.       return x * x;
30.    }
```

Listing 5–9 MethodPointerTest.java (continued)

```
31.
32.    /**
33.     * Prints a table with x- and y-values for a method
34.     * @param from the lower bound for the x-values
35.     * @param to the upper bound for the x-values
36.     * @param n the number of rows in the table
37.     * @param f a method with a double parameter and double return value
38.     */
39.    public static void printTable(double from, double to, int n, Method f)
40.    {
41.       // print out the method as table header
42.       System.out.println(f);
43.
44.       double dx = (to - from) / (n - 1);
45.
46.       for (double x = from; x <= to; x += dx)
47.       {
48.          try
49.          {
50.             double y = (Double) f.invoke(null, x);
51.             System.out.printf("%10.4f | %10.4f%n", x, y);
52.          }
53.          catch (Exception e)
54.          {
55.             e.printStackTrace();
56.          }
57.       }
58.    }
59. }
```

As this example shows clearly, you can do anything with `Method` objects that you can do with function pointers in C (or delegates in C#). Just as in C, this style of programming is usually quite inconvenient and always error prone. What happens if you invoke a method with the wrong parameters? The `invoke` method throws an exception.

Also, the parameters and return values of `invoke` are necessarily of type `Object`. That means you must cast back and forth a lot. As a result, the compiler is deprived of the chance to check your code. Therefore, errors surface only during testing, when they are more tedious to find and fix. Moreover, code that uses reflection to get at method pointers is significantly slower than code that simply calls methods directly.

For that reason, we suggest that you use `Method` objects in your own programs only when absolutely necessary. Using interfaces and inner classes (the subject of the next chapter) is almost always a better idea. In particular, we echo the developers of Java and suggest not using `Method` objects for callback functions. Using interfaces for the callbacks (see the next chapter as well) leads to code that runs faster and is a lot more maintainable.

API **java.lang.reflect.Method 1.1**

- `public Object invoke(Object implicitParameter, Object[] explicitParameters)`
  invokes the method described by this object, passing the given parameters and returning the value that the method returns. For static methods, pass `null` as the implicit parameter. Pass primitive type values by using wrappers. Primitive type return values must be unwrapped.

Design Hints for Inheritance

We want to end this chapter with some hints that we have found useful when using inheritance.

1. *Place common operations and fields in the superclass.*

   This is why we put the name field into the `Person` class rather than replicating it in the `Employee` and `Student` classes.

2. *Don't use protected fields.*

   Some programmers think it is a good idea to define most instance fields as `protected`, "just in case," so that subclasses can access these fields if they need to. However, the `protected` mechanism doesn't give much protection, for two reasons. First, the set of subclasses is unbounded—anyone can form a subclass of your classes and then write code that directly accesses `protected` instance fields, thereby breaking encapsulation. And second, in the Java programming language, all classes in the same package have access to `protected` fields, whether or not they are subclasses.

   However, `protected` methods can be useful to indicate methods that are not ready for general use and should be redefined in subclasses. The `clone` method is a good example.

3. *Use inheritance to model the "is–a" relationship.*

   Inheritance is a handy code-saver, and sometimes people overuse it. For example, suppose we need a `Contractor` class. Contractors have names and hire dates, but they do not have salaries. Instead, they are paid by the hour, and they do not stay around long enough to get a raise. There is the temptation to form a subclass `Contractor` from `Employee` and add an `hourlyWage` field.

   ```
   class Contractor extends Employee
   {  . . .
      private double hourlyWage;
   }
   ```

   This is *not* a good idea, however, because now each contractor object has both a salary and hourly wage field. It will cause you no end of grief when you implement methods for printing paychecks or tax forms. You will end up writing more code than you would have by not inheriting in the first place.

   The contractor/employee relationship fails the "is–a" test. A contractor is not a special case of an employee.

4. *Don't use inheritance unless all inherited methods make sense.*

   Suppose we want to write a `Holiday` class. Surely every holiday is a day, and days can be expressed as instances of the `GregorianCalendar` class, so we can use inheritance.

   ```
   class Holiday extends GregorianCalendar { . . . }
   ```

Unfortunately, the set of holidays is not *closed* under the inherited operations. One of the public methods of `GregorianCalendar` is `add`. And `add` can turn holidays into nonholidays:

```
Holiday christmas;
christmas.add(Calendar.DAY_OF_MONTH, 12);
```

Therefore, inheritance is not appropriate in this example.

5. *Don't change the expected behavior when you override a method.*

   The substitution principle applies not just to syntax but, more important, to behavior. When you override a method, you should not unreasonably change its behavior. The compiler can't help you—it cannot check whether your redefinitions make sense. For example, you can "fix" the issue of the `add` method in the `Holiday` class by redefining `add`, perhaps to do nothing, or to throw an exception, or to move on to the next holiday.

   However, such a fix violates the substitution principle. The sequence of statements

   ```
   int d1 = x.get(Calendar.DAY_OF_MONTH);
   x.add(Calendar.DAY_OF_MONTH, 1);
   int d2 = x.get(Calendar.DAY_OF_MONTH);
   System.out.println(d2 - d1);
   ```

   should have the *expected behavior*, no matter whether `x` is of type `GregorianCalendar` or `Holiday`.

   Of course, therein lies the rub. Reasonable and unreasonable people can argue at length what the expected behavior is. For example, some authors argue that the substitution principle requires `Manager.equals` to ignore the `bonus` field because `Employee.equals` ignores it. These discussions are always pointless if they occur in a vacuum. Ultimately, what matters is that you do not circumvent the intent of the original design when you override methods in subclasses.

6. *Use polymorphism, not type information.*

   Whenever you find code of the form

   ```
   if (x is of type 1)
      action1(x);
   else if (x is of type 2)
      action2(x);
   ```

   think polymorphism.

   Do `action1` and `action2` represent a common concept? If so, make the concept a method of a common superclass or interface of both types. Then, you can simply call

   ```
   x.action();
   ```

   and have the dynamic dispatch mechanism inherent in polymorphism launch the correct action.

   Code using polymorphic methods or interface implementations is much easier to maintain and extend than code that uses multiple type tests.

7. *Don't overuse reflection.*

   The reflection mechanism lets you write programs with amazing generality, by detecting fields and methods at runtime. This capability can be extremely useful for systems programming, but it is usually not appropriate in applications. Reflection is fragile—the compiler cannot help you find programming errors. Any errors are found at runtime and result in exceptions.

You have now seen how Java supports the fundamentals of object-oriented programming: classes, inheritance, and polymorphism. In the next chapter, we will tackle two advanced topics that are very important for using Java effectively: interfaces and inner classes.

Revised to cover advanced user-interface programming and the enterprise features of the Java SE 6 platform

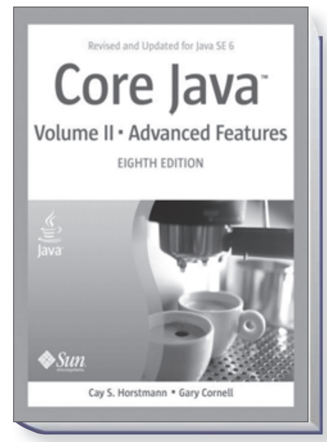
SECURITY

FROM

CORE JAVA, VOLUME II— Advanced Features, Eighth Edition

by Cay Horstmann and Gary Cornell

©2008 | 1056 PAGES | ISBN: 0-13-235479-9



ALSO AVAILABLE

- SAFARI BOOKS ONLINE Safari Books Online
- E-BOOK: 0137145136
- MOBI POCKET: 0137145128
- SONY READER: 013714511X

TABLE OF CONTENTS

Chapter 1:
Streams and Files

Chapter 2:
XML

Chapter 3:
Networking

Chapter 4:
Database Programming

Chapter 5:
Internationalization

Chapter 6:
Advanced Swing

Chapter 7:
Advanced AWT

Chapter 8:
Javabeans Components

Chapter 9:
Security

Chapter 10:
Distributed Objects

Chapter 11:
Scripting, Compiling,
and Annotation Processing

Chapter 12:
Native Methods

FOR MORE INFORMATION

informit.com/title/9780132354799

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.

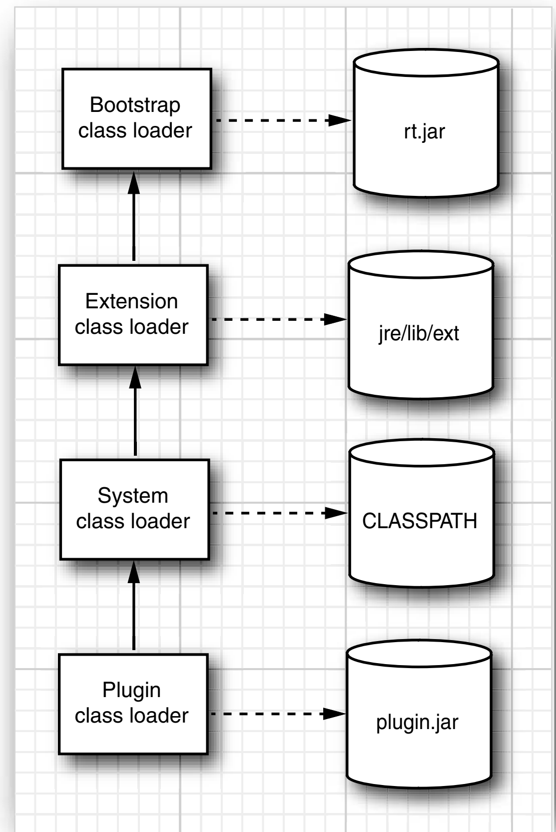


Figure 9–1 The class loader hierarchy

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.

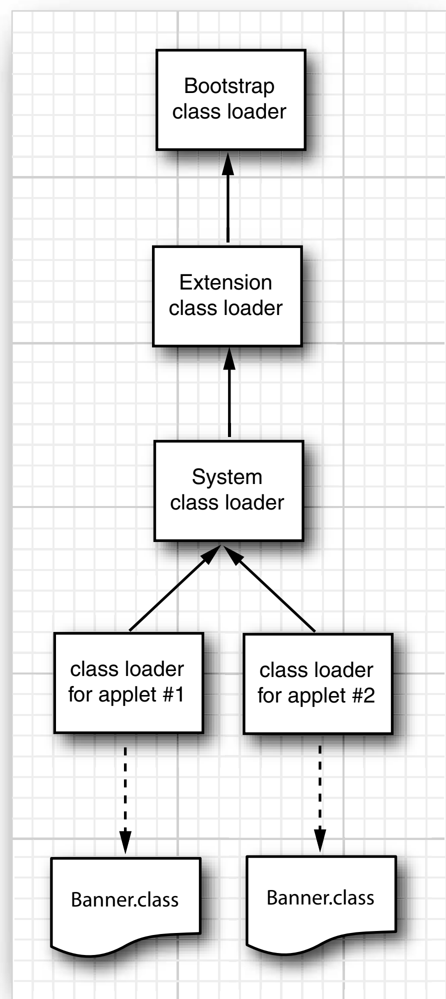


Figure 9–2 Two class loaders load different classes with the same name

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).

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.

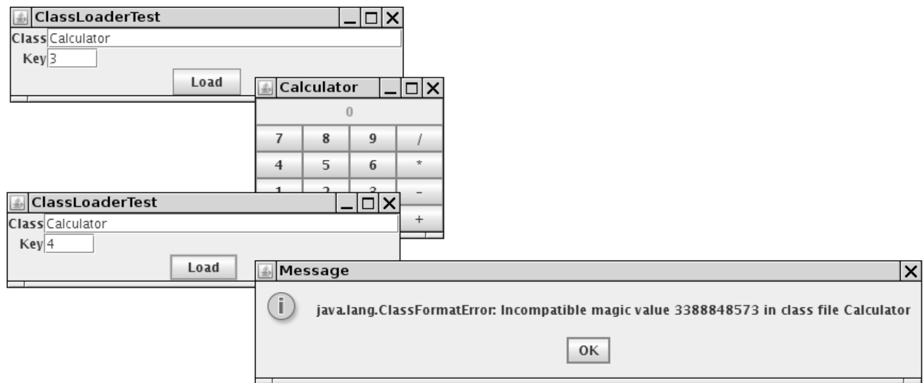


Figure 9–3 The ClassLoaderTest program

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`

```
1. import java.io.*;
2. import java.lang.reflect.*;
3. import java.awt.*;
4. import java.awt.event.*;
5. import javax.swing.*;
6.
7. /**
8.  * This program demonstrates a custom class loader that decrypts class files.
9.  * @version 1.22 2007-10-05
10.  * @author Cay Horstmann
11.  */
12. public class ClassLoaderTest
13. {
14.    public static void main(String[] args)
15.    {
16.       EventQueue.invokeLater(new Runnable()
17.          {
18.             public void run()
19.             {
20.
21.                JFrame frame = new ClassLoaderFrame();
22.                frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
23.                frame.setVisible(true);
24.             }
25.          });
26.    }
27. }
28.
```

Listing 9–1 ClassLoaderTest.java (continued)

```
29. /**
30.  * This frame contains two text fields for the name of the class to load and the decryption key.
31.  */
32. class ClassLoaderFrame extends JFrame
33. {
34.    public ClassLoaderFrame()
35.    {
36.       setTitle("ClassLoaderTest");
37.       setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
38.       setLayout(new GridBagLayout());
39.       add(new JLabel("Class"), new GBC(0, 0).setAnchor(GBC.EAST));
40.       add(nameField, new GBC(1, 0).setWeight(100, 0).setAnchor(GBC.WEST));
41.       add(new JLabel("Key"), new GBC(0, 1).setAnchor(GBC.EAST));
42.       add(keyField, new GBC(1, 1).setWeight(100, 0).setAnchor(GBC.WEST));
43.       JButton loadButton = new JButton("Load");
44.       add(loadButton, new GBC(0, 2, 2, 1));
45.       loadButton.addActionListener(new ActionListener()
46.          {
47.             public void actionPerformed(ActionEvent event)
48.             {
49.                runClass(nameField.getText(), keyField.getText());
50.             }
51.          });
52.       pack();
53.    }
54.
55.    /**
56.     * Runs the main method of a given class.
57.     * @param name the class name
58.     * @param key the decryption key for the class files
59.     */
60.    public void runClass(String name, String key)
61.    {
62.       try
63.       {
64.          ClassLoader loader = new CryptoClassLoader(Integer.parseInt(key));
65.          Class<?> c = loader.loadClass(name);
66.          Method m = c.getMethod("main", String[].class);
67.          m.invoke(null, (Object) new String[] {});
68.       }
69.       catch (Throwable e)
70.       {
71.          JOptionPane.showMessageDialog(this, e);
72.       }
73.    }
74.
```

Listing 9–1 ClassLoaderTest.java (continued)

```
75.    private JTextField keyField = new JTextField("3", 4);
76.    private JTextField nameField = new JTextField("Calculator", 30);
77.    private static final int DEFAULT_WIDTH = 300;
78.    private static final int DEFAULT_HEIGHT = 200;
79. }
80.
81. /**
82.  * This class loader loads encrypted class files.
83.  */
84. class CryptoClassLoader extends ClassLoader
85. {
86.    /**
87.     * Constructs a crypto class loader.
88.     * @param k the decryption key
89.     */
90.    public CryptoClassLoader(int k)
91.    {
92.       key = k;
93.    }
94.
95.    protected Class<?> findClass(String name) throws ClassNotFoundException
96.    {
97.       byte[] classBytes = null;
98.       try
99.       {
100.         classBytes = loadClassBytes(name);
101.      }
102.      catch (IOException e)
103.      {
104.         throw new ClassNotFoundException(name);
105.      }
106.
107.      Class<?> cl = defineClass(name, classBytes, 0, classBytes.length);
108.      if (cl == null) throw new ClassNotFoundException(name);
109.      return cl;
110.   }
111.
112.   /**
113.    * Loads and decrypt the class file bytes.
114.    * @param name the class name
115.    * @return an array with the class file bytes
116.    */
117.   private byte[] loadClassBytes(String name) throws IOException
118.   {
119.      String cname = name.replace('.', '/') + ".caesar";
120.      FileInputStream in = null;
121.      in = new FileInputStream(cname);
122.      try
123.      {
```

Listing 9–1 ClassLoaderTest.java (continued)

```
124.          ByteArrayOutputStream buffer = new ByteArrayOutputStream();
125.          int ch;
126.          while ((ch = in.read()) != -1)
127.          {
128.             byte b = (byte) (ch - key);
129.             buffer.write(b);
130.          }
131.          in.close();
132.          return buffer.toByteArray();
133.       }
134.       finally
135.       {
136.          in.close();
137.       }
138.    }
139.
140.    private int key;
141. }
```

Listing 9–2 Caesar.java

```
1. import java.io.*;
2.
3. /**
4.  * Encrypts a file using the Caesar cipher.
5.  * @version 1.00 1997-09-10
6.  * @author Cay Horstmann
7.  */
8. public class Caesar
9. {
10.    public static void main(String[] args)
11.    {
12.       if (args.length != 3)
13.       {
14.          System.out.println("USAGE: java Caesar in out key");
15.          return;
16.       }
17.
18.       try
19.       {
20.          FileInputStream in = new FileInputStream(args[0]);
21.          FileOutputStream out = new FileOutputStream(args[1]);
22.          int key = Integer.parseInt(args[2]);
23.          int ch;
24.          while ((ch = in.read()) != -1)
25.          {
```

Listing 9–2 Caesar.java (continued)

```
26.             byte c = (byte) (ch + key);
27.             out.write(c);
28.          }
29.          in.close();
30.          out.close();
31.       }
32.       catch (IOException exception)
33.       {
34.          exception.printStackTrace();
35.       }
36.    }
37. }
```

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.

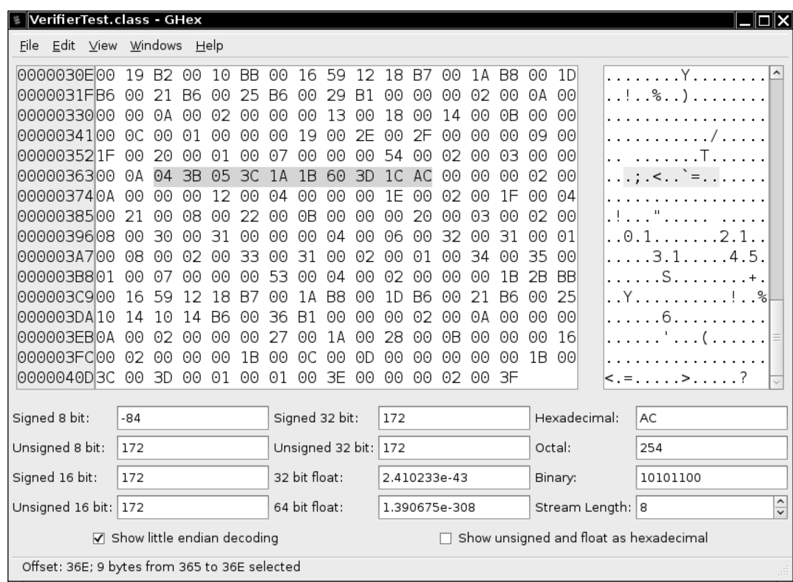


Figure 9–4 Modifying bytecodes with a hex editor

Change 3C to 3B and save the class file. Then try running the `VerifierTest` program. You get an error message:

```
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).

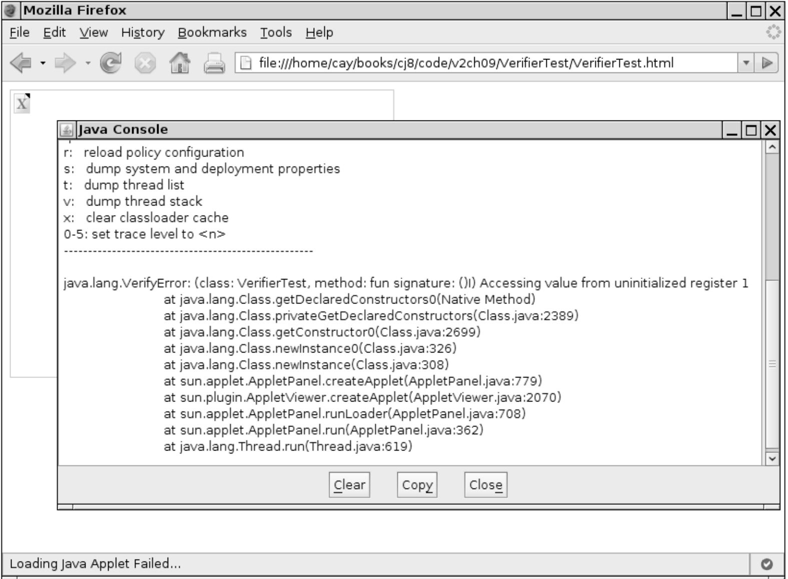


Figure 9–5 Loading a corrupted class file raises a method verification error

Listing 9–3 VerifierTest.java

```
1. import java.applet.*;
2. import java.awt.*;
3.
4. /**
5.  * This application demonstrates the bytecode verifier of the virtual machine. If you use a
6.  * hex editor to modify the class file, then the virtual machine should detect the tampering.
7.  * @version 1.00 1997-09-10
8.  * @author Cay Horstmann
9.  */
```

Listing 9–3 VerifierTest.java (continued)

```
10. public class VerifierTest extends Applet
11. {
12.    public static void main(String[] args)
13.    {
14.       System.out.println("1 + 2 == " + fun());
15.    }
16.
17.    /**
18.     * A function that computes 1 + 2
19.     * @return 3, if the code has not been corrupted
20.     */
21.    public static int fun()
22.    {
23.       int m;
24.       int n;
25.       m = 1;
26.       n = 2;
27.       // use hex editor to change to "m = 2" in class file
28.       int r = m + n;
29.       return r;
30.    }
31.
32.    public void paint(Graphics g)
33.    {
34.       g.drawString("1 + 2 == " + fun(), 20, 20);
35.    }
36. }
```

Security Managers and Permissions

Once a class has been loaded into the virtual machine and checked by the verifier, the second security mechanism of the Java platform springs into action: the *security manager.* The security manager is a class that controls whether a specific 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 operations are 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 security exception is thrown. Here is what happens in detail. The `exit` method of the `Runtime` class calls the `checkExit` method of 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 continues. However, if the security manager doesn't want to grant the request, the `checkExit` method throws a `SecurityException`.

The `exit` method continues only if no exception occurred. It then calls the *private native* `exitInternal` method that 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 virtual machine must go through the `exit` method and thus through the `checkExit` security check without triggering a security exception.

Clearly, the integrity of the security policy depends on careful coding. The providers of system services in the standard 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 control over 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 you can 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 the *sandbox.* Just like a child that can only play in a sandbox, remote code was only allowed to paint on the screen and interact with the user. The applet security manager denied all access to local resources. JDK 1.1 implemented a slight 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 had to play in the sandbox.

Starting with Java SE 1.2, the Java platform has a much more flexible mechanism. A *security policy* maps *code sources to permission sets* (see Figure 9–6).

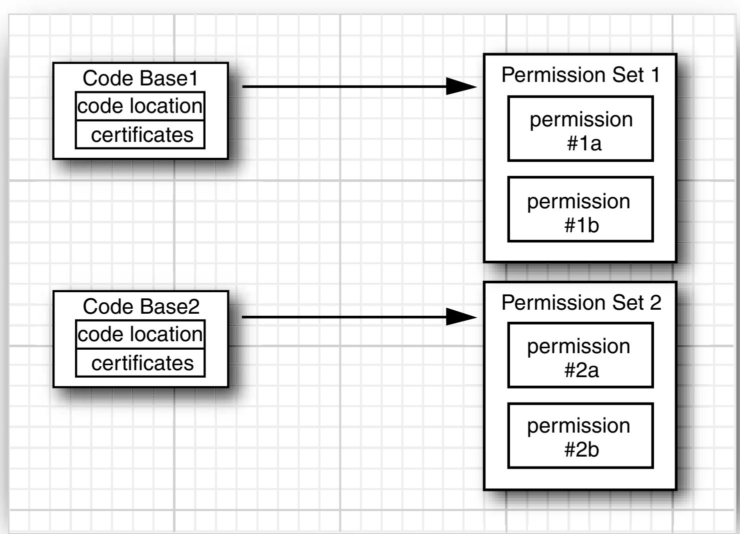


Figure 9–6 A security policy

A *code source* is specified by a *code base* and a set of *certificates*. The code base specifies the origin of the code. For example, the code base of remote applet code is the HTTP URL from which the applet is loaded. The code base of code in a JAR file is a file URL. A certificate, if present, is an assurance by some party that the code has not been tampered with. 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 permission classes, each of which encapsulates the details of a particular permission. For example, the following instance of the `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 permission classes have been added in subsequent Java releases.

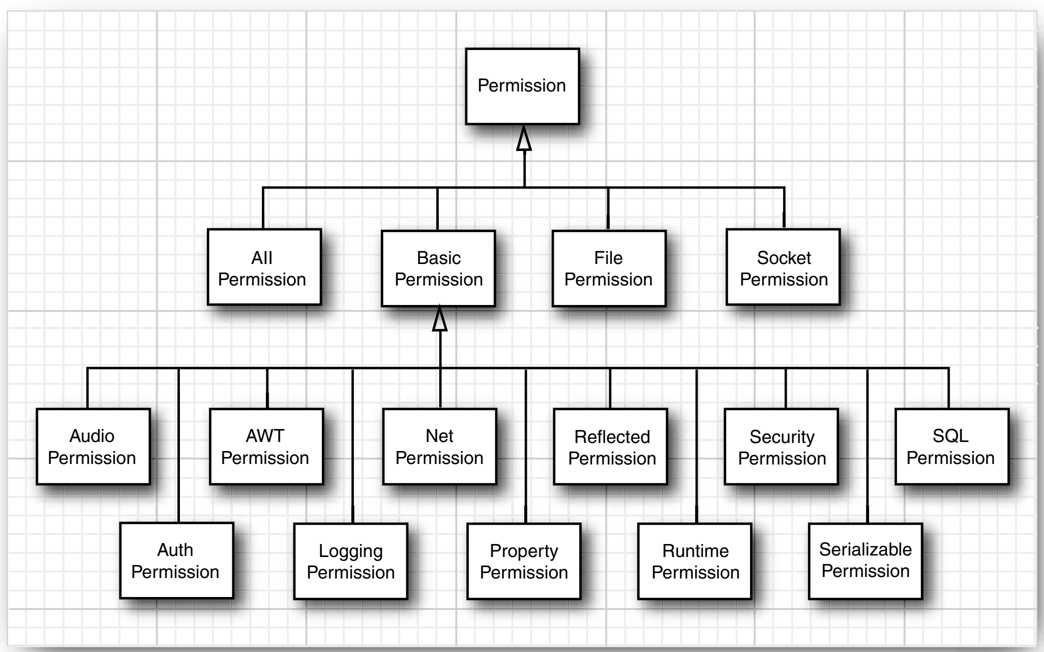


Figure 9–7 A part of the hierarchy of permission classes

In the preceding section, you saw that the `SecurityManager` class has security check methods such as `checkExit`. These methods exist only for the convenience of the programmer and for backward compatibility. They all map into standard 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 permissions of the class. When the `SecurityManager` needs to check a permission, it looks at the classes of all methods currently on the call stack. It then gets the protection domains of all classes and asks each protection domain if its permission collection 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. Suppose the `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 security manager, which finally calls `checkPermission` with a `FilePermission(name, "read"` object. Table 9–1 shows the call stack.

Table 9–1 Call Stack During Permission Checking

Class	Method	Code Source	Permissions
`SecurityManager`	`checkPermission`	`null`	`AllPermission`
`SecurityManager`	`checkRead`	`null`	`AllPermission`
`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 permissions consist of an instance of the `AllPermission` class, which allows all operations. Clearly, their permissions alone can't determine the outcome of the check. As you can see, the `checkPermission` method must take into account the restricted permissions of the applet class. By checking the entire call stack, the security mechanism ensures that one class can never 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.

API `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.

API `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 typical 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 the policy file that is required for each application. Place the permissions into a separate file, say, `MyApp.policy`. To apply the 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, such as

```
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 effect 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 virtual 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 the 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 names 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 example,

```
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. For 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 *target name* is a permission-specific value, for example, a file or directory name for the file permission, or a host and port for a socket permission. The *actionList* is also permission specific. It is a list of actions, such as `read` or `connect`, separated by commas. Some permission classes don't need target names and action lists. Table 9–2 lists the commonly used permission 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`, `delete`
`java.net.SocketPermission`	socket target (see text)	`accept`, `connect`, `listen`, `resolve`
`java.util.PropertyPermission`	property target (see text)	`read`, `write`

Table 9–2 Permissions and Their Associated Targets and Actions (continued)

Permission	Target	Action
java.lang.RuntimePermission	createClassLoader getClassLoader setContextClassLoader enableContextClassLoaderOverride createSecurityManager setSecurityManager 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	(none)
java.awt.AWTPermission	showWindowWithoutWarningBanner accessClipboard accessEventQueue createRobot fullScreenExclusive listenToAllAWTEvents readDisplayPixels replaceKeyboardFocusManager watchMousePointer setWindowAlwaysOnTop setAppletStub	(none)
java.net.NetPermission	setDefaultAuthenticator specifyStreamHandler requestPasswordAuthentication setProxySelector getProxySelector setCookieHandler getCookieHandler setResponseCache getResponseCache	(none)

Table 9–2 Permissions and Their Associated Targets and Actions (continued)

Permission	Target	Action
java.lang.reflect.ReflectPermission	suppressAccessChecks	(none)
java.io.SerializablePermission	enableSubclassImplementation enableSubstitution	(none)
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 operation as the target with an implied action `"permit"`. These permission classes all extend the `BasicPermission` class (see Figure 9–7 on page 774). However, the targets for the file, socket, and property permissions are more complex, and we need to 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 subdirectories.

```
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:

```
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 example, ${user.home} is replaced by the home directory of the user. Here is a typical use of this system property in a permission 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 explicit / or \\ 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 subdirectories.

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.

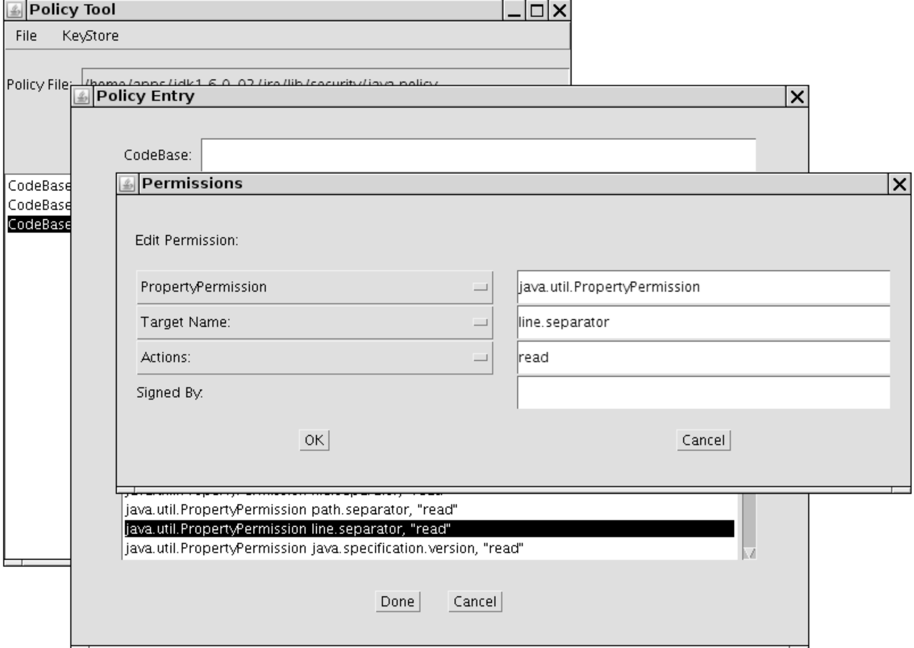


Figure 9–8 The policy tool

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* more 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 to open a file for reading, it checks whether it has permission to do so. For that check, a *specific* file permission object is passed to 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 implies 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 permission 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 implement the `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. The 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 `checkPermission` method throws an exception.

Word check permissions have two possible actions: `insert` (the permission to insert a specific text) and `avoid` (the permission to add any text that avoids certain bad words). You should run this program with the following policy file:

```
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 are the 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 example, the 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`. For 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 permission

  ```
  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 parse these strings. In particular, we use the following method to transform the comma-separated list of bad words of an `avoid` 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 `equals` method of a set class finds two sets to be equal if they contain the same elements in any order. For example, the sets 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 and click the Insert button. If the security check passes, the text is appended to the text area. If not, an error message is displayed (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 the standard permissions.

Listing 9–4 WordCheckPermission.java

```
1. import java.security.*;
2. import java.util.*;
3.
4. /**
5.  * A permission that checks for bad words.
6.  * @version 1.00 1999-10-23
7.  * @author Cay Horstmann
8.  */
9. public class WordCheckPermission extends Permission
10. {
11.    /**
12.     * Constructs a word check permission
13.     * @param target a comma separated word list
14.     * @param anAction "insert" or "avoid"
15.     */
16.    public WordCheckPermission(String target, String anAction)
17.    {
18.       super(target);
19.       action = anAction;
20.    }
21.
22.    public String getActions()
23.    {
24.       return action;
25.    }
26.
27.    public boolean equals(Object other)
28.    {
29.       if (other == null) return false;
30.       if (!getClass().equals(other.getClass())) return false;
31.       WordCheckPermission b = (WordCheckPermission) other;
32.       if (!action.equals(b.action)) return false;
33.       if (action.equals("insert")) return getName().equals(b.getName());
34.       else if (action.equals("avoid")) return badWordSet().equals(b.badWordSet());
35.       else return false;
36.    }
37.
38.    public int hashCode()
39.    {
40.       return getName().hashCode() + action.hashCode();
41.    }
42.
43.    public boolean implies(Permission other)
44.    {
```

Listing 9–4 WordCheckPermission.java (continued)

```
45.       if (!(other instanceof WordCheckPermission)) return false;
46.       WordCheckPermission b = (WordCheckPermission) other;
47.       if (action.equals("insert"))
48.       {
49.          return b.action.equals("insert") && getName().indexOf(b.getName()) >= 0;
50.       }
51.       else if (action.equals("avoid"))
52.       {
53.          if (b.action.equals("avoid")) return b.badWordSet().containsAll(badWordSet());
54.          else if (b.action.equals("insert"))
55.          {
56.             for (String badWord : badWordSet())
57.                if (b.getName().indexOf(badWord) >= 0) return false;
58.             return true;
59.          }
60.          else return false;
61.       }
62.       else return false;
63.    }
64.
65.    /**
66.     * Gets the bad words that this permission rule describes.
67.     * @return a set of the bad words
68.     */
69.    public Set<String> badWordSet()
70.    {
71.       Set<String> set = new HashSet<String>();
72.       set.addAll(Arrays.asList(getName().split(",")));
73.       return set;
74.    }
75.
76.    private String action;
77. }
```

Listing 9–5 PermissionTest.java

```
1. import java.awt.*;
2. import java.awt.event.*;
3. import javax.swing.*;
4.
5. /**
6.  * This class demonstrates the custom WordCheckPermission.
7.  * @version 1.03 2007-10-06
8.  * @author Cay Horstmann
9.  */
10. public class PermissionTest
11. {
```

Listing 9–5 PermissionTest.java (continued)

```
12.    public static void main(String[] args)
13.    {
14.       System.setProperty("java.security.policy", "PermissionTest.policy");
15.       System.setSecurityManager(new SecurityManager());
16.       EventQueue.invokeLater(new Runnable()
17.          {
18.             public void run()
19.             {
20.                JFrame frame = new PermissionTestFrame();
21.                frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
22.                frame.setVisible(true);
23.             }
24.          });
25.    }
26. }
27.
28. /**
29.  * This frame contains a text field for inserting words into a text area that is protected
30.  * from "bad words".
31.  */
32. class PermissionTestFrame extends JFrame
33. {
34.    public PermissionTestFrame()
35.    {
36.       setTitle("PermissionTest");
37.       setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
38.
39.       textField = new JTextField(20);
40.       JPanel panel = new JPanel();
41.       panel.add(textField);
42.       JButton openButton = new JButton("Insert");
43.       panel.add(openButton);
44.       openButton.addActionListener(new ActionListener()
45.          {
46.             public void actionPerformed(ActionEvent event)
47.             {
48.                insertWords(textField.getText());
49.             }
50.          });
51.
52.       add(panel, BorderLayout.NORTH);
53.
54.       textArea = new WordCheckTextArea();
55.       add(new JScrollPane(textArea), BorderLayout.CENTER);
56.    }
57.
```

Listing 9–5 PermissionTest.java (continued)

```
58.    /**
59.     * Tries to insert words into the text area. Displays a dialog if the attempt fails.
60.     * @param words the words to insert
61.     */
62.    public void insertWords(String words)
63.    {
64.       try
65.       {
66.          textArea.append(words + "\n");
67.       }
68.       catch (SecurityException e)
69.       {
70.          JOptionPane.showMessageDialog(this, "I am sorry, but I cannot do that.");
71.       }
72.    }
73.
74.    private JTextField textField;
75.    private WordCheckTextArea textArea;
76.    private static final int DEFAULT_WIDTH = 400;
77.    private static final int DEFAULT_HEIGHT = 300;
78. }
79.
80. /**
81.  * A text area whose append method makes a security check to see that no bad words are added.
82.  */
83. class WordCheckTextArea extends JTextArea
84. {
85.    public void append(String text)
86.    {
87.       WordCheckPermission p = new WordCheckPermission(text, "insert");
88.       SecurityManager manager = System.getSecurityManager();
89.       if (manager != null) manager.checkPermission(p);
90.       super.append(text);
91.    }
92. }
```

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:

```
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:

```
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`

```
1. import java.security.*;
2. import javax.security.auth.*;
3. import javax.security.auth.login.*;
4.
5. /**
6.  * This program authenticates a user via a custom login and then executes the SysPropAction
7.  * with the user's privileges.
8.  * @version 1.01 2007-10-06
9.  * @author Cay Horstmann
10.  */
```

Listing 9–6 AuthTest.java (continued)

```
11. public class AuthTest
12. {
13.    public static void main(final String[] args)
14.    {
15.       System.setSecurityManager(new SecurityManager());
16.       try
17.       {
18.          LoginContext context = new LoginContext("Login1");
19.          context.login();
20.          System.out.println("Authentication successful.");
21.          Subject subject = context.getSubject();
22.          System.out.println("subject=" + subject);
23.          PrivilegedAction<String> action = new SysPropAction("user.home");
24.          String result = Subject.doAsPrivileged(subject, action, null);
25.          System.out.println(result);
26.          context.logout();
27.       }
28.       catch (LoginException e)
29.       {
30.          e.printStackTrace();
31.       }
32.    }
33. }
```

Listing 9–7 SysPropAction.java

```
1. import java.security.*;
2.
3. /**
4.    This action looks up a system property.
5.  * @version 1.01 2007-10-06
6.  * @author Cay Horstmann
7. */
8. public class SysPropAction implements PrivilegedAction<String>
9. {
10.    /**
11.       Constructs an action for looking up a given property.
12.       @param propertyName the property name (such as "user.home")
13.    */
14.    public SysPropAction(String propertyName) { this.propertyName = propertyName; }
15.
16.    public String run()
17.    {
18.       return System.getProperty(propertyName);
19.    }
20.
21.    private String propertyName;
22. }
```

Listing 9–8 AuthTest.policy

```
1. grant codebase "file:login.jar"
2. {
3.    permission javax.security.auth.AuthPermission "createLoginContext.Login1";
4.    permission javax.security.auth.AuthPermission "doAsPrivileged";
5. };
6.
7. grant principal com.sun.security.auth.UnixPrincipal "harry"
8. {
9.    permission java.util.PropertyPermission "user.*", "read";
10. };
```

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).

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.

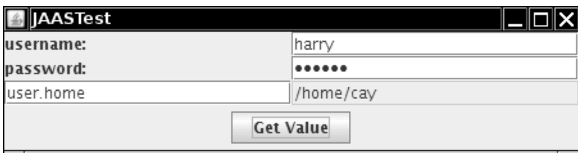


Figure 9–10 A custom login module

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

```
1. import java.security.*;
2.
3. /**
4.  * A principal with a named value (such as "role=HR" or "username=harry").
5.  * @version 1.0 2004-09-14
6.  * @author Cay Horstmann
7.  */
8. public class SimplePrincipal implements Principal
9. {
10.    /**
11.     * Constructs a SimplePrincipal to hold a description and a value.
12.     * @param roleName the role name
13.     */
14.    public SimplePrincipal(String descr, String value)
15.    {
16.       this.descr = descr;
17.       this.value = value;
18.    }
19.
20.    /**
21.     * Returns the role name of this principal
22.     * @return the role name
23.     */
```

Listing 9–9 SimplePrincipal.java (continued)

```
24.    public String getName()
25.    {
26.       return descr + "=" + value;
27.    }
28.
29.    public boolean equals(Object otherObject)
30.    {
31.       if (this == otherObject) return true;
32.       if (otherObject == null) return false;
33.       if (getClass() != otherObject.getClass()) return false;
34.       SimplePrincipal other = (SimplePrincipal) otherObject;
35.       return getName().equals(other.getName());
36.    }
37.
38.    public int hashCode()
39.    {
40.       return getName().hashCode();
41.    }
42.
43.    private String descr;
44.    private String value;
45. }
```

Listing 9–10 SimpleLoginModule.java

```
1. import java.io.*;
2. import java.security.*;
3. import java.util.*;
4. import javax.security.auth.*;
5. import javax.security.auth.callback.*;
6. import javax.security.auth.login.*;
7. import javax.security.auth.spi.*;
8.
9. /**
10.  * This login module authenticates users by reading usernames, passwords, and roles from a
11.  * text file.
12.  * @version 1.0 2004-09-14
13.  * @author Cay Horstmann
14.  */
15. public class SimpleLoginModule implements LoginModule
16. {
17.    public void initialize(Subject subject, CallbackHandler callbackHandler,
18.          Map<String, ?> sharedState, Map<String, ?> options)
19.    {
20.       this.subject = subject;
21.       this.callbackHandler = callbackHandler;
22.       this.options = options;
23.    }
```

Listing 9–10 SimpleLoginModule.java (continued)

```
24.
25.    public boolean login() throws LoginException
26.    {
27.       if (callbackHandler == null) throw new LoginException("no handler");
28.
29.       NameCallback nameCall = new NameCallback("username: ");
30.       PasswordCallback passCall = new PasswordCallback("password: ", false);
31.       try
32.       {
33.          callbackHandler.handle(new Callback[] { nameCall, passCall });
34.       }
35.       catch (UnsupportedCallbackException e)
36.       {
37.          LoginException e2 = new LoginException("Unsupported callback");
38.          e2.initCause(e);
39.          throw e2;
40.       }
41.       catch (IOException e)
42.       {
43.          LoginException e2 = new LoginException("I/O exception in callback");
44.          e2.initCause(e);
45.          throw e2;
46.       }
47.
48.       return checkLogin(nameCall.getName(), passCall.getPassword());
49.    }
50.
51.    /**
52.     * Checks whether the authentication information is valid. If it is, the subject acquires
53.     * principals for the user name and role.
54.     * @param username the user name
55.     * @param password a character array containing the password
56.     * @return true if the authentication information is valid
57.     */
58.    private boolean checkLogin(String username, char[] password) throws LoginException
59.    {
60.       try
61.       {
62.          Scanner in = new Scanner(new FileReader("" + options.get("pwfile")));
63.          while (in.hasNextLine())
64.          {
65.             String[] inputs = in.nextLine().split("\\|");
66.             if (inputs[0].equals(username) && Arrays.equals(inputs[1].toCharArray(), password))
67.             {
68.                String role = inputs[2];
69.                Set<Principal> principals = subject.getPrincipals();
70.                principals.add(new SimplePrincipal("username", username));
```

Listing 9–10 SimpleLoginModule.java (continued)

```
71.             principals.add(new SimplePrincipal("role", role));
72.             return true;
73.          }
74.       }
75.       in.close();
76.       return false;
77.    }
78.    catch (IOException e)
79.    {
80.       LoginException e2 = new LoginException("Can't open password file");
81.       e2.initCause(e);
82.       throw e2;
83.    }
84. }
85.
86. public boolean logout()
87. {
88.    return true;
89. }
90.
91. public boolean abort()
92. {
93.    return true;
94. }
95.
96. public boolean commit()
97. {
98.    return true;
99. }
100.
101. private Subject subject;
102. private CallbackHandler callbackHandler;
103. private Map<String, ?> options;
104. }
```

Listing 9–11 SimpleCallbackHandler.java

```
1. import javax.security.auth.callback.*;
2.
3. /**
4.  * This simple callback handler presents the given user name and password.
5.  * @version 1.0 2004-09-14
6.  * @author Cay Horstmann
7.  */
8. public class SimpleCallbackHandler implements CallbackHandler
9. {
```

Listing 9–11 SimpleCallbackHandler.java (continued)

```
10.    /**
11.     * Constructs the callback handler.
12.     * @param username the user name
13.     * @param password a character array containing the password
14.     */
15.    public SimpleCallbackHandler(String username, char[] password)
16.    {
17.       this.username = username;
18.       this.password = password;
19.    }
20.
21.    public void handle(Callback[] callbacks)
22.    {
23.       for (Callback callback : callbacks)
24.       {
25.          if (callback instanceof NameCallback)
26.          {
27.             ((NameCallback) callback).setName(username);
28.          }
29.          else if (callback instanceof PasswordCallback)
30.          {
31.             ((PasswordCallback) callback).setPassword(password);
32.          }
33.       }
34.    }
35.
36.    private String username;
37.    private char[] password;
38. }
```

Listing 9–12 JAASTest.java

```
1. import java.awt.*;
2. import java.awt.event.*;
3. import javax.security.auth.*;
4. import javax.security.auth.login.*;
5. import javax.swing.*;
6.
7. /**
8.  * This program authenticates a user via a custom login and then executes the SysPropAction
9.  * with the user's privileges.
10.  * @version 1.0 2004-09-14
11.  * @author Cay Horstmann
12.  */
13. public class JAASTest
14. {
```

Listing 9–12 JAASTest.java (continued)

```
15.    public static void main(final String[] args)
16.    {
17.       System.setSecurityManager(new SecurityManager());
18.       EventQueue.invokeLater(new Runnable()
19.          {
20.             public void run()
21.             {
22.                JFrame frame = new JAASFrame();
23.                frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
24.                frame.setVisible(true);
25.             }
26.          });
27.    }
28. }
29.
30. /**
31.  * This frame has text fields for user name and password, a field for the name of the requested
32.  * system property, and a field to show the property value.
33.  */
34. class JAASFrame extends JFrame
35. {
36.    public JAASFrame()
37.    {
38.       setTitle("JAASTest");
39.
40.       username = new JTextField(20);
41.       password = new JPasswordField(20);
42.       propertyName = new JTextField(20);
43.       propertyValue = new JTextField(20);
44.       propertyValue.setEditable(false);
45.
46.       JPanel panel = new JPanel();
47.       panel.setLayout(new GridLayout(0, 2));
48.       panel.add(new JLabel("username:"));
49.       panel.add(username);
50.       panel.add(new JLabel("password:"));
51.       panel.add(password);
52.       panel.add(propertyName);
53.       panel.add(propertyValue);
54.       add(panel, BorderLayout.CENTER);
55.
56.       JButton getValueButton = new JButton("Get Value");
57.       getValueButton.addActionListener(new ActionListener()
58.          {
59.             public void actionPerformed(ActionEvent event)
60.             {
61.                getValue();
62.             }
```

Listing 9–12 JAASTest.java (continued)

```
63.          });
64.       JPanel buttonPanel = new JPanel();
65.       buttonPanel.add(getValueButton);
66.       add(buttonPanel, BorderLayout.SOUTH);
67.       pack();
68.    }
69.
70.    public void getValue()
71.    {
72.       try
73.       {
74.          LoginContext context = new LoginContext("Login1", new SimpleCallbackHandler(username
75.             .getText(), password.getPassword()));
76.          context.login();
77.          Subject subject = context.getSubject();
78.          propertyValue.setText(""
79.             + Subject.doAsPrivileged(subject, new SysPropAction(propertyName.getText()), null));
80.          context.logout();
81.       }
82.       catch (LoginException e)
83.       {
84.          JOptionPane.showMessageDialog(this, e);
85.       }
86.    }
87.
88.    private JTextField username;
89.    private JPasswordField password;
90.    private JTextField propertyName;
91.    private JTextField propertyValue;
92. }
```

Listing 9–13 JAASTest.policy

```
1. grant codebase "file:login.jar"
2. {
3.    permission java.awt.AWTPermission "showWindowWithoutWarningBanner";
4.    permission javax.security.auth.AuthPermission "createLoginContext.Login1";
5.    permission javax.security.auth.AuthPermission "doAsPrivileged";
6.    permission javax.security.auth.AuthPermission "modifyPrincipals";
7.    permission java.io.FilePermission "password.txt", "read";
8. };
9.
10. grant principal SimplePrincipal "role=admin"
11. {
12.    permission java.util.PropertyPermission "*", "read";
13. };
```

Listing 9–14 jaas.config

```
1. Login1
2. {
3.    SimpleLoginModule required pwfile="password.txt";
4. };
```

API **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.

API **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.

API **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 of useful stuff in the JDK 1.0 security model. For example, because applets under JDK 1.0 were so closely supervised, they couldn't do much good on a corporate intranet, even though relatively little risk attaches to executing an applet from your company's secure intranet. It quickly became clear to Sun that for applets to become truly useful, it was important for users to be able to assign *different* levels of security, depending on where the applet originated. If an applet comes from a trusted supplier and it has not been tampered with, the user of that applet can then decide whether to give the applet 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 ensuring the integrity of data and for electronic signatures. The `java.security` package contains implementations of many of these 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 and how 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 algorithm #1) condenses any data block, no matter how long, into a sequence of 160 bits (20 bytes). As with real fingerprints, one hopes 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 of duplication occurring is negligible. How negligible? According to James Walsh in *True Odds: How Risks Affect Your Everyday Life* (Merritt Publishing 1996), the chance that you will die from being struck by lightning is about one in 30,000. Now, think of nine other people, for example, your nine least favorite managers or professors. The chance that you and *all of them* will die from lightning strikes is higher than that of a forged message having the same SHA1 fingerprint as the

original. (Of course, more than ten people, none of whom you are likely to know, will die from lightning strikes. However, 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 message digest as the original.

The second property is again a matter of probabilities, of course. Consider the following message by the billionaire father:

> *"Upon my death, my property shall be divided equally among my children; however, my son George shall receive 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, suppose George can bribe the lawyer holding the message. He wants to change the message so that Bill gets nothing. Of course, that 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 computers from the time the Earth was formed, each computing a million messages a second, he would not yet have found a message he could substitute.

A number of algorithms have been designed to compute these message digests. The two best-known are SHA1, the secure hash algorithm developed by the National Institute of Standards and Technology, and MD5, an algorithm invented by 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 2005). Note that recently, subtle regularities have been discovered in both algorithms. At this point, most cryptographers recommend avoiding MD5 and using SHA1 until a stronger alternative becomes available. (See `http://www.rsa.com/rsalabs/node.asp?id=2834` for more information.)

The Java programming language implements both SHA1 and MD5. The `MessageDigest` class is a *factory* for creating objects that encapsulate the fingerprinting algorithms. It has a static method, called `getInstance`, that returns an object 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 `update` method. For example, the following code passes all bytes in a file to the `alg` object just created to do the fingerprinting:

```
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 digested from a file, or you can type a message in the text area. Figure 9–11 shows the application.

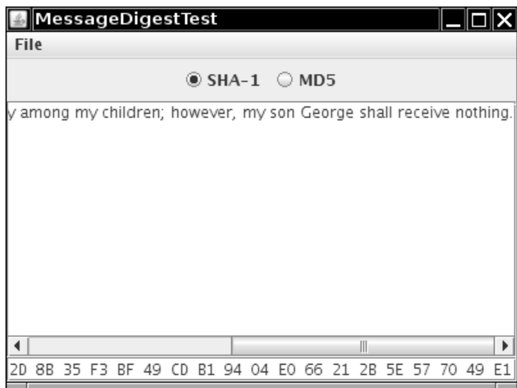


Figure 9–11 Computing a message digest

Listing 9–15 MessageDigestTest.java

```
1. import java.io.*;
2. import java.security.*;
3. import java.awt.*;
4. import java.awt.event.*;
5. import javax.swing.*;
6.
7. /**
8.  * This program computes the message digest of a file or the contents of a text area.
9.  * @version 1.13 2007-10-06
10.  * @author Cay Horstmann
11.  */
12. public class MessageDigestTest
13. {
```

Listing 9–15 MessageDigestTest.java (continued)

```
14.    public static void main(String[] args)
15.    {
16.       EventQueue.invokeLater(new Runnable()
17.          {
18.             public void run()
19.             {
20.                JFrame frame = new MessageDigestFrame();
21.                frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
22.                frame.setVisible(true);
23.             }
24.          });
25.    }
26. }
27.
28. /**
29.  * This frame contains a menu for computing the message digest of a file or text area, radio
30.  * buttons to toggle between SHA-1 and MD5, a text area, and a text field to show the
31.  * message digest.
32.  */
33. class MessageDigestFrame extends JFrame
34. {
35.    public MessageDigestFrame()
36.    {
37.       setTitle("MessageDigestTest");
38.       setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
39.
40.       JPanel panel = new JPanel();
41.       ButtonGroup group = new ButtonGroup();
42.       addRadioButton(panel, "SHA-1", group);
43.       addRadioButton(panel, "MD5", group);
44.
45.       add(panel, BorderLayout.NORTH);
46.       add(new JScrollPane(message), BorderLayout.CENTER);
47.       add(digest, BorderLayout.SOUTH);
48.       digest.setFont(new Font("Monospaced", Font.PLAIN, 12));
49.
50.       setAlgorithm("SHA-1");
51.
52.       JMenuBar menuBar = new JMenuBar();
53.       JMenu menu = new JMenu("File");
54.       JMenuItem fileDigestItem = new JMenuItem("File digest");
55.       fileDigestItem.addActionListener(new ActionListener()
56.          {
57.             public void actionPerformed(ActionEvent event)
58.             {
59.                loadFile();
60.             }
61.          });
```

Listing 9–15 MessageDigestTest.java (continued)

```
62.       menu.add(fileDigestItem);
63.       JMenuItem textDigestItem = new JMenuItem("Text area digest");
64.       textDigestItem.addActionListener(new ActionListener()
65.          {
66.             public void actionPerformed(ActionEvent event)
67.             {
68.                String m = message.getText();
69.                computeDigest(m.getBytes());
70.             }
71.          });
72.       menu.add(textDigestItem);
73.       menuBar.add(menu);
74.       setJMenuBar(menuBar);
75.    }
76.
77.    /**
78.     * Adds a radio button to select an algorithm.
79.     * @param c the container into which to place the button
80.     * @param name the algorithm name
81.     * @param g the button group
82.     */
83.    public void addRadioButton(Container c, final String name, ButtonGroup g)
84.    {
85.       ActionListener listener = new ActionListener()
86.          {
87.             public void actionPerformed(ActionEvent event)
88.             {
89.                setAlgorithm(name);
90.             }
91.          };
92.       JRadioButton b = new JRadioButton(name, g.getButtonCount() == 0);
93.       c.add(b);
94.       g.add(b);
95.       b.addActionListener(listener);
96.    }
97.
98.    /**
99.     * Sets the algorithm used for computing the digest.
100.    * @param alg the algorithm name
101.    */
102.   public void setAlgorithm(String alg)
103.   {
104.      try
105.      {
106.         currentAlgorithm = MessageDigest.getInstance(alg);
107.         digest.setText("");
108.      }
```

Listing 9–15 MessageDigestTest.java (continued)

```
109.       catch (NoSuchAlgorithmException e)
110.       {
111.          digest.setText("" + e);
112.       }
113.    }
114.
115.    /**
116.     * Loads a file and computes its message digest.
117.     */
118.    public void loadFile()
119.    {
120.       JFileChooser chooser = new JFileChooser();
121.       chooser.setCurrentDirectory(new File("."));
122.
123.       int r = chooser.showOpenDialog(this);
124.       if (r == JFileChooser.APPROVE_OPTION)
125.       {
126.          try
127.          {
128.             String name = chooser.getSelectedFile().getAbsolutePath();
129.             computeDigest(loadBytes(name));
130.          }
131.          catch (IOException e)
132.          {
133.             JOptionPane.showMessageDialog(null, e);
134.          }
135.       }
136.    }
137.
138.    /**
139.     * Loads the bytes in a file.
140.     * @param name the file name
141.     * @return an array with the bytes in the file
142.     */
143.    public byte[] loadBytes(String name) throws IOException
144.    {
145.       FileInputStream in = null;
146.
147.       in = new FileInputStream(name);
148.       try
149.       {
150.          ByteArrayOutputStream buffer = new ByteArrayOutputStream();
151.          int ch;
152.          while ((ch = in.read()) != -1)
153.             buffer.write(ch);
154.          return buffer.toByteArray();
155.       }
156.       finally
```

Listing 9–15 MessageDigestTest.java (continued)

```
157.       {
158.          in.close();
159.       }
160.    }
161.
162.    /**
163.     * Computes the message digest of an array of bytes and displays it in the text field.
164.     * @param b the bytes for which the message digest should be computed.
165.     */
166.    public void computeDigest(byte[] b)
167.    {
168.       currentAlgorithm.reset();
169.       currentAlgorithm.update(b);
170.       byte[] hash = currentAlgorithm.digest();
171.       String d = "";
172.       for (int i = 0; i < hash.length; i++)
173.       {
174.          int v = hash[i] & 0xFF;
175.          if (v < 16) d += "0";
176.          d += Integer.toString(v, 16).toUpperCase() + " ";
177.       }
178.       digest.setText(d);
179.    }
180.
181.    private JTextArea message = new JTextArea();
182.    private JTextField digest = new JTextField();
183.    private MessageDigest currentAlgorithm;
184.    private static final int DEFAULT_WIDTH = 400;
185.    private static final int DEFAULT_HEIGHT = 300;
186. }
```

API **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 message is altered, then the fingerprint of the altered message will not match the fingerprint of the original. If the message and its fingerprint are delivered separately, then the recipient can check whether the message has been tampered with. However, if both the message and the fingerprint were intercepted, it is an easy matter to modify the message and then recompute the fingerprint. After all, the message digest algorithms are publicly known, and they don't require secret keys. In that case, the recipient of the forged message and the recomputed fingerprint would never know that the message has been 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 tell everyone in the world your public key. However, only you hold the private key, and it is important that you safeguard it and don't release it to anyone else. The keys are matched by mathematical relationships, but the exact nature of these relationships is not important for us. (If you are interested, you can look it up in *The Handbook of Applied Cryptography* at `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:

```
p:
fca682ce8e12caba26efccf7110e526db078b05edecbcd1eb4a208f3ae1617ae01f35b91a47e6df63413c5e12ed0899
bcd132acd50d99151bdc43ee737592e17

q: 962eddcc369cba8ebb260ee6b6a126d9346e38c5

g:678471b27a9cf44ee91a49c5147db1a9aaf244f05a434d6486931d2d14271b9e35030b71fd73da179069b32e29356
30e
1c2062354d0da20a6c416e50be794ca4

y:
c0b6e67b4ac098eb1a32c5f8c4c1f0e7e6fb9d832532e27d0bdab9ca2d2a8123ce5a8018b8161a760480fadd040b927
281ddb22cb9bc4df596d7de4d1b977d50
```

Private key:

```
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 your public key, they can't compute your private key in your lifetime, no matter how many computing resources they have available.

It might seem difficult to believe that nobody can compute the private key from the public keys, but nobody has ever found an algorithm to do this for the encryption algorithms that are in common use today. If the keys are sufficiently long, brute force—simply trying all possible keys—would require more computers than can be built from all the atoms in the solar system, crunching away for thousands of years. Of course, it is possible that someone could come up with algorithms for computing keys that are much more clever than brute force. For example, the RSA algorithm (the encryption algorithm invented by Rivest, Shamir, and Adleman) depends on the difficulty of factoring large numbers. For the last 20 years, many of the best mathematicians have tried to come up with good factoring algorithms, but so far with no success. For that reason, most cryptographers believe that keys with a "modulus" of 2,000 bits or more are currently completely safe from any attack. DSA is believed to be similarly secure.

Figure 9–12 illustrates how the process works in practice.

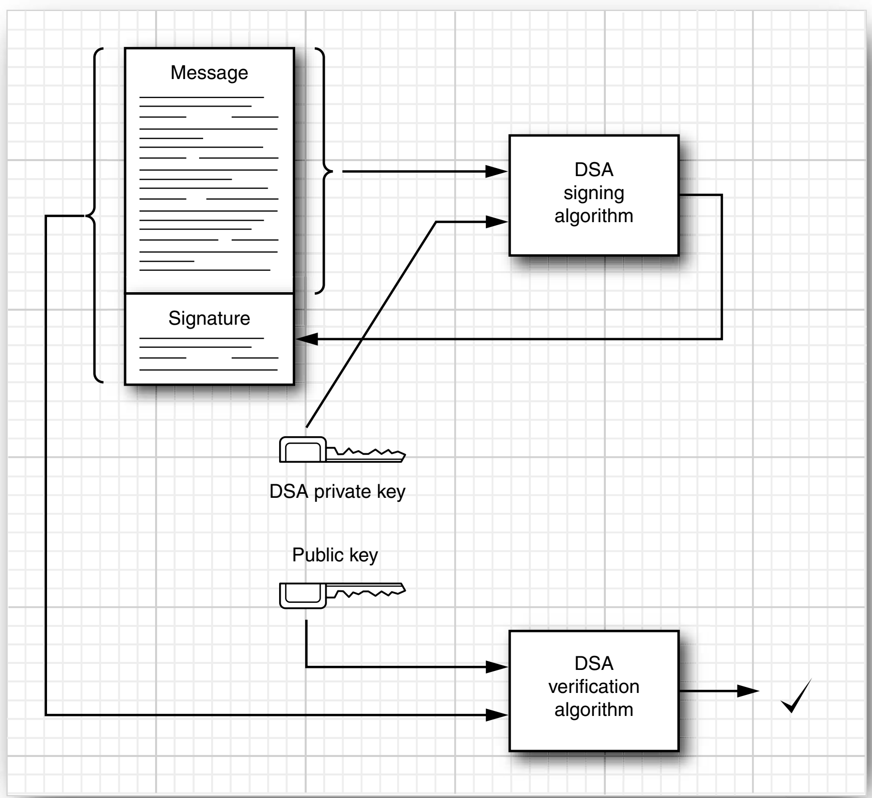


Figure 9–12 Public key signature exchange with DSA

Suppose Alice wants to send Bob a message, and Bob wants to know this message came from Alice and not an impostor. Alice writes the message and then *signs* the message digest with her private key. Bob gets a copy of her public key. Bob 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 for verification.

You can see why security for private keys is all-important. If someone steals Alice's private key or if a government can 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 distribution formats is called X.509. Certificates in the X.509 format are widely used by VeriSign, Microsoft, Netscape, and many other companies, for signing e-mail messages, authenticating program code, and certifying many other kinds of data. The X.509 standard is part of the X.500 series of recommendations for a directory service by the international telephone standards body, the CCITT.

The precise structure of X.509 certificates is described in a formal notation, called "abstract syntax notation #1" or ASN.1. Figure 9–13 shows the ASN.1 definition of version 3 of the X.509 format. The exact syntax is not important for us, but, as you can see, ASN.1 gives a precise definition of the structure of a certificate file. The *basic encoding rules,* or BER, and a variation, called *distinguished encoding rules* (DER) describe precisely how to save this structure in a binary file. That is, BER and DER describe how to encode integers, character strings, bit strings, and constructs such as `SEQUENCE`, `CHOICE`, and `OPTIONAL`.

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 certificates. We expect that ultimately the functionality of this tool will be embedded in other, more user-friendly programs. But right now, we use `keytool` to show how Alice can sign a document and send it to Bob, and how Bob can verify that the document 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 keystore 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
```

```
[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  }
```

Figure 9–13 ASN.1 definition of X.509v3

When creating or opening a keystore, you are prompted for a keystore password. For this example, just use `secret`. If you were to use the `keytool`-generated keystore for any serious purpose, you would need to choose a good password and safeguard this file.

When generating a key, you are prompted for the following information:

```
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:

```
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 the phone.

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

```
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
```

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 adds 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, and the 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 digital 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 you. You 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 page. Then, you can verify that the message was in fact authored by Alice and has not been tampered with. Now, suppose you get a message from a stranger who claims to represent a famous software company, urging you to run the program that is attached to the message. The stranger even sends you a copy of his public key so you can verify that he authored the message. You check that the signature is valid. This proves that the message was signed with the matching private key 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 private keys, 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 acquaintance you both trust. Suppose the stranger meets your acquaintance in person and hands over a disk with the public key. Your acquaintance later meets you, assures you that he met the stranger and that the stranger indeed works for the famous software company, and then gives you the disk (see Figure 9–14). That way, your acquaintance vouches for the authenticity of the stranger.

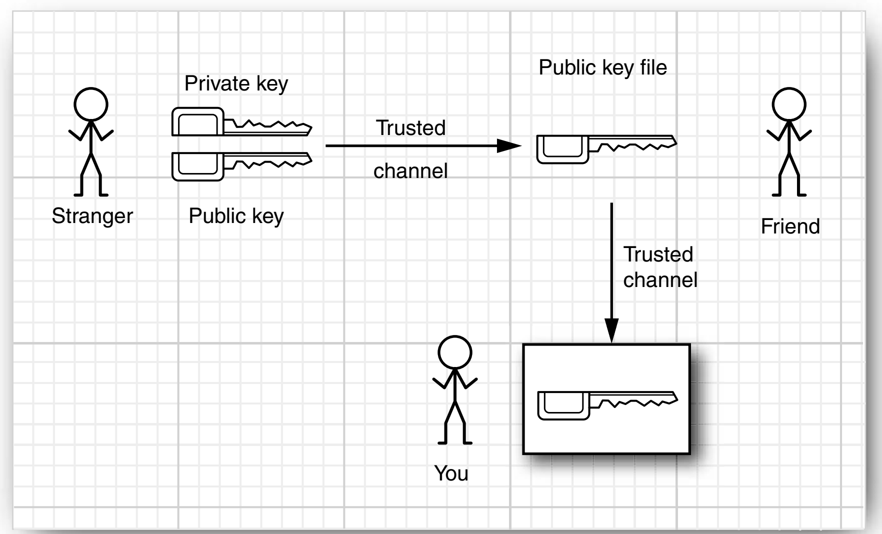


Figure 9–14 Authentication through a trusted intermediary

In fact, your acquaintance does not actually need to meet you. Instead, he can use his private key to sign the stranger's public key file (see Figure 9–15).

When you get the public key file, you verify the signature of your friend, and because you trust him, you are confident that 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 of trust"—a chain of mutual acquaintances—so that you trust every member of that chain. In practice, of course, that isn't always true. You might trust your friend, Alice, and you know that Alice trusts Bob, but you don't know Bob and aren't sure that you trust him. Other trust models assume that there is a benevolent big brother in whom we all trust. The best known of these companies is VeriSign, Inc. (`http://www.verisign.com`).

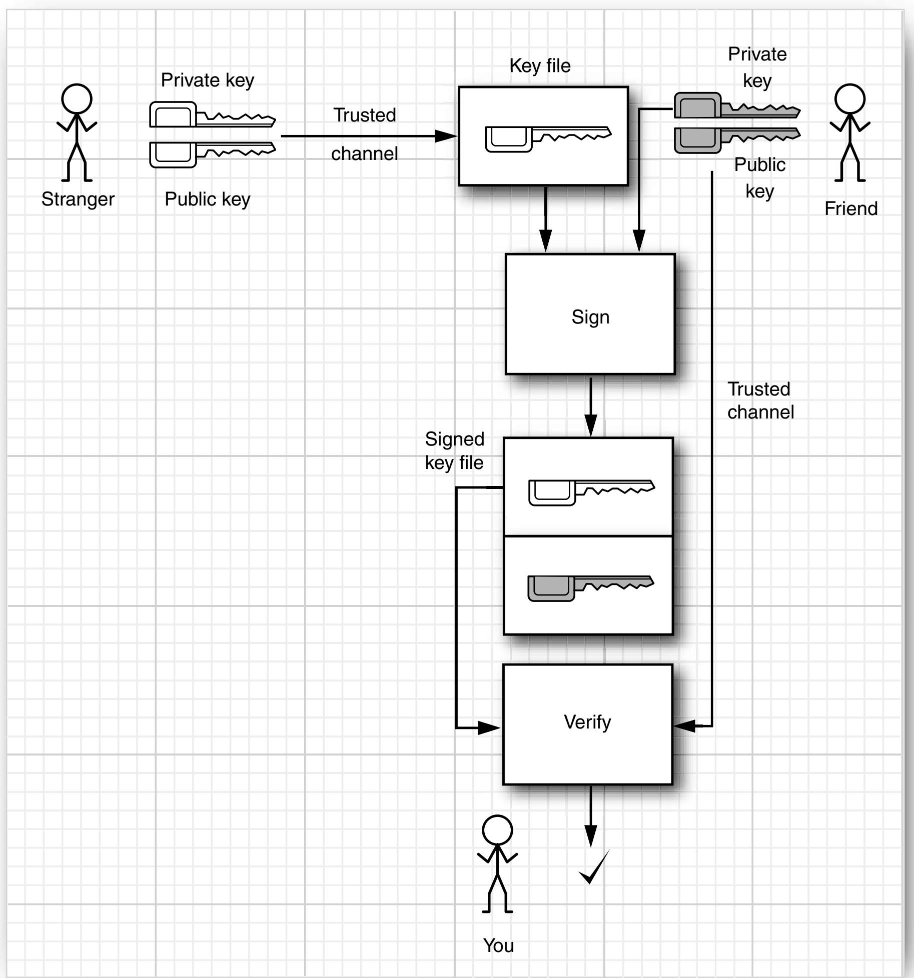


Figure 9–15 Authentication through a trusted intermediary's signature

You will often encounter digital signatures that are signed by one or more entities who will vouch for the authenticity, and you will need to evaluate to what degree you trust the authenticators. You might place a great deal of trust in VeriSign, perhaps because you saw their logo on many web pages or because you heard that they require multiple people with black attaché 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 does not personally meet every individual or company representative when authenticating a public key. You can get a "class 1" ID simply by filling out a web form and paying a small fee. The key is mailed to the e-mail address included in the certificate. Thus, you can be reasonably assured that the e-mail address is genuine, but the requestor could have filled in *any* name and organization. There are more stringent classes of IDs. For example, with a "class 3" ID, VeriSign will require an individual requestor to appear before a notary public, and it will check the financial rating of a corporate requestor. Other authenticators will have different procedures. Thus, when you receive an authenticated message, it is important that you understand what, in fact, is being authenticated.

Certificate Signing

In the section "Verifying a Signature" on page 814, you saw how Alice used a self-signed certificate to distribute a public 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 verifying lots of signature fingerprints. Now suppose that there is an entity that Cindy trusts to verify signatures. In this example, Cindy 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 employees. When they install each other's keys, then the keystore will trust them implicitly because they are signed by a trusted key.

Here is how you can simulate this process. Create a keystore `acmesoft.certs`. Generate a key par and export the public key:

```
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.

```
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 the Information Resources Department and have it signed. Unfortunately, this functionality is missing in the `keytool` program. In the book's companion code, we supply a `CertificateSigner` class to fill the gap. An authorized staff member at ACME 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 the 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. Alternatively, ACME Software can simply store the file in a company directory. Remember, this file contains Alice's public key and an assertion by ACME Software that this key really belongs to Alice.

Now Cindy imports the signed certificate into her keystore:

```
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 is *not* 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 never 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 run more sophisticated software to manage certificates, and they use slightly different formats for certificates. This section 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 and Mac 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 storage 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 Mail (PEM) format, not the DER format that the keystore accepts easily. Copy the file `demoCA/cacert.pem` to a file `acmeroot.pem` and 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 format:

```
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:

```
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.

```
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*.

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.

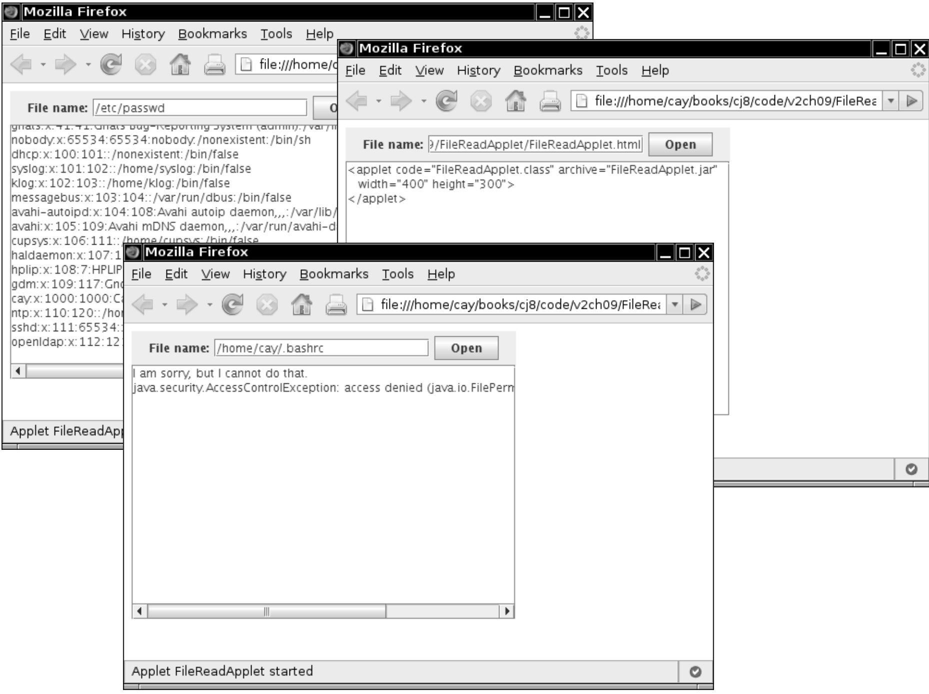


Figure 9–16 A signed applet can read local files

Listing 9–16 FileReadApplet.java

```
1. import java.awt.*;
2. import java.awt.event.*;
3. import java.io.*;
4. import java.util.*;
5. import javax.swing.*;
6.
7. /**
8.  * This applet can run "outside the sandbox" and read local files when it is given the right
9.  * permissions.
10.  * @version 1.11 2007-10-06
11.  * @author Cay Horstmann
12.  */
13. public class FileReadApplet extends JApplet
14. {
15.    public void init()
16.    {
17.       EventQueue.invokeLater(new Runnable()
18.          {
19.             public void run()
20.             {
```

Listing 9–16 FileReadApplet.java (continued)

```
21.             fileNameField = new JTextField(20);
22.             JPanel panel = new JPanel();
23.             panel.add(new JLabel("File name:"));
24.             panel.add(fileNameField);
25.             JButton openButton = new JButton("Open");
26.             panel.add(openButton);
27.             ActionListener listener = new ActionListener()
28.             {
29.                public void actionPerformed(ActionEvent event)
30.                {
31.                   loadFile(fileNameField.getText());
32.                }
33.             };
34.             fileNameField.addActionListener(listener);
35.             openButton.addActionListener(listener);
36.
37.             add(panel, "North");
38.
39.             fileText = new JTextArea();
40.             add(new JScrollPane(fileText), "Center");
41.          }
42.       });
43.    }
44.
45.    /**
46.     * Loads the contents of a file into the text area.
47.     * @param filename the file name
48.     */
49.    public void loadFile(String filename)
50.    {
51.       try
52.       {
53.          fileText.setText("");
54.          Scanner in = new Scanner(new FileReader(filename));
55.          while (in.hasNextLine())
56.             fileText.append(in.nextLine() + "\n");
57.          in.close();
58.       }
59.       catch (IOException e)
60.       {
61.          fileText.append(e + "\n");
62.       }
63.       catch (SecurityException e)
64.       {
65.          fileText.append("I am sorry, but I cannot do that.\n");
66.          fileText.append(e + "\n");
67.       }
68.    }
69.    private JTextField fileNameField;
70.    private JTextArea fileText;
71. }
```

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.)

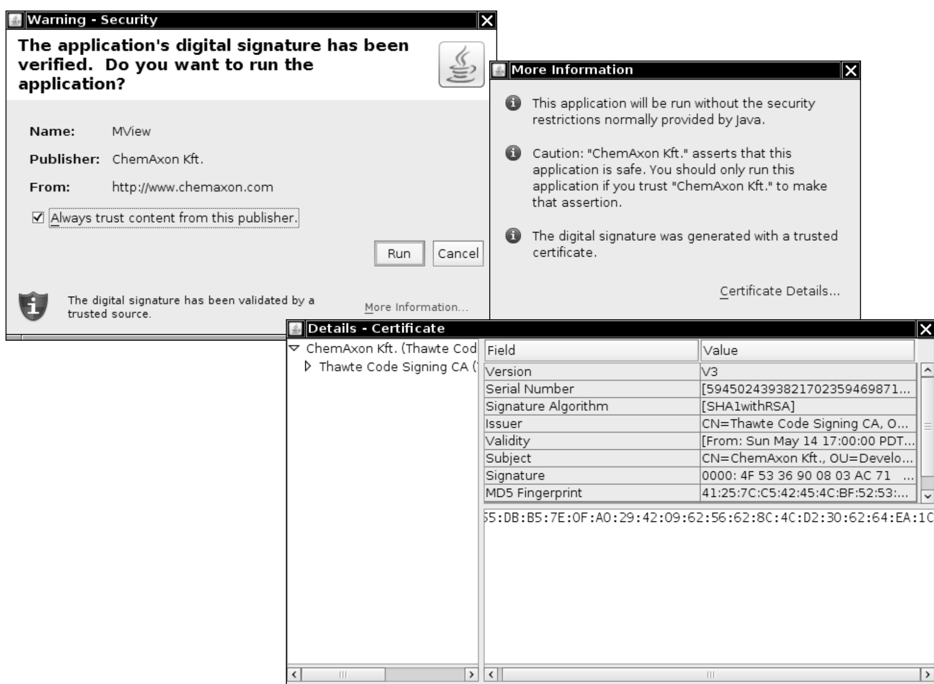


Figure 9–17 Launching a signed applet

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(algorithName, 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();
```

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

```
1. import java.io.*;
2. import java.security.*;
3. import javax.crypto.*;
4.
5. /**
6.  * This program tests the AES cipher. Usage:<br>
7.  * java AESTest -genkey keyfile<br>
8.  * java AESTest -encrypt plaintext encrypted keyfile<br>
9.  * java AESTest -decrypt encrypted decrypted keyfile<br>
10.  * @author Cay Horstmann
11.  * @version 1.0 2004-09-14
12.  */
13. public class AESTest
14. {
```

Listing 9–17 AESTest.java (continued)

```
15.    public static void main(String[] args)
16.    {
17.       try
18.       {
19.          if (args[0].equals("-genkey"))
20.          {
21.             KeyGenerator keygen = KeyGenerator.getInstance("AES");
22.             SecureRandom random = new SecureRandom();
23.             keygen.init(random);
24.             SecretKey key = keygen.generateKey();
25.             ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream(args[1]));
26.             out.writeObject(key);
27.             out.close();
28.          }
29.          else
30.          {
31.             int mode;
32.             if (args[0].equals("-encrypt")) mode = Cipher.ENCRYPT_MODE;
33.             else mode = Cipher.DECRYPT_MODE;
34.
35.             ObjectInputStream keyIn = new ObjectInputStream(new FileInputStream(args[3]));
36.             Key key = (Key) keyIn.readObject();
37.             keyIn.close();
38.
39.             InputStream in = new FileInputStream(args[1]);
40.             OutputStream out = new FileOutputStream(args[2]);
41.             Cipher cipher = Cipher.getInstance("AES");
42.             cipher.init(mode, key);
43.
44.             crypt(in, out, cipher);
45.             in.close();
46.             out.close();
47.          }
48.       }
49.       catch (IOException e)
50.       {
51.          e.printStackTrace();
52.       }
53.       catch (GeneralSecurityException e)
54.       {
55.          e.printStackTrace();
56.       }
57.       catch (ClassNotFoundException e)
58.       {
59.          e.printStackTrace();
60.       }
61.    }
62.
```

Listing 9–17 AESTest.java (continued)

```
63.    /**
64.     * Uses a cipher to transform the bytes in an input stream and sends the transformed bytes
65.     * to an output stream.
66.     * @param in the input stream
67.     * @param out the output stream
68.     * @param cipher the cipher that transforms the bytes
69.     */
70.    public static void crypt(InputStream in, OutputStream out, Cipher cipher)
71.          throws IOException, GeneralSecurityException
72.    {
73.       int blockSize = cipher.getBlockSize();
74.       int outputSize = cipher.getOutputSize(blockSize);
75.       byte[] inBytes = new byte[blockSize];
76.       byte[] outBytes = new byte[outputSize];
77. 
78.       int inLength = 0;
79.       boolean more = true;
80.       while (more)
81.       {
82.          inLength = in.read(inBytes);
83.          if (inLength == blockSize)
84.          {
85.             int outLength = cipher.update(inBytes, 0, blockSize, outBytes);
86.             out.write(outBytes, 0, outLength);
87.          }
88.          else more = false;
89.       }
90.       if (inLength > 0) outBytes = cipher.doFinal(inBytes, 0, inLength);
91.       else outBytes = cipher.doFinal();
92.       out.write(outBytes);
93.    }
94. }
```

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.

API `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.

API `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:

```
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:

```
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.

API **`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

```
1. import java.io.*;
2. import java.security.*;
3. import javax.crypto.*;
4.
5. /**
6.  * This program tests the RSA cipher. Usage:<br>
7.  * java RSATest -genkey public private<br>
8.  * java RSATest -encrypt plaintext encrypted public<br>
9.  * java RSATest -decrypt encrypted decrypted private<br>
10.  * @author Cay Horstmann
11.  * @version 1.0 2004-09-14
12.  */
13. public class RSATest
14. {
15.    public static void main(String[] args)
16.    {
17.       try
18.       {
19.          if (args[0].equals("-genkey"))
20.          {
21.             KeyPairGenerator pairgen = KeyPairGenerator.getInstance("RSA");
22.             SecureRandom random = new SecureRandom();
23.             pairgen.initialize(KEYSIZE, random);
24.             KeyPair keyPair = pairgen.generateKeyPair();
25.             ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream(args[1]));
26.             out.writeObject(keyPair.getPublic());
27.             out.close();
28.             out = new ObjectOutputStream(new FileOutputStream(args[2]));
29.             out.writeObject(keyPair.getPrivate());
30.             out.close();
31.          }
32.          else if (args[0].equals("-encrypt"))
33.          {
```

Listing 9–18 RSATest.java (continued)

```
34.             KeyGenerator keygen = KeyGenerator.getInstance("AES");
35.             SecureRandom random = new SecureRandom();
36.             keygen.init(random);
37.             SecretKey key = keygen.generateKey();
38.
39.             // wrap with RSA public key
40.             ObjectInputStream keyIn = new ObjectInputStream(new FileInputStream(args[3]));
41.             Key publicKey = (Key) keyIn.readObject();
42.             keyIn.close();
43.
44.             Cipher cipher = Cipher.getInstance("RSA");
45.             cipher.init(Cipher.WRAP_MODE, publicKey);
46.             byte[] wrappedKey = cipher.wrap(key);
47.             DataOutputStream out = new DataOutputStream(new FileOutputStream(args[2]));
48.             out.writeInt(wrappedKey.length);
49.             out.write(wrappedKey);
50.
51.             InputStream in = new FileInputStream(args[1]);
52.             cipher = Cipher.getInstance("AES");
53.             cipher.init(Cipher.ENCRYPT_MODE, key);
54.             crypt(in, out, cipher);
55.             in.close();
56.             out.close();
57.          }
58.          else
59.          {
60.             DataInputStream in = new DataInputStream(new FileInputStream(args[1]));
61.             int length = in.readInt();
62.             byte[] wrappedKey = new byte[length];
63.             in.read(wrappedKey, 0, length);
64.
65.             // unwrap with RSA private key
66.             ObjectInputStream keyIn = new ObjectInputStream(new FileInputStream(args[3]));
67.             Key privateKey = (Key) keyIn.readObject();
68.             keyIn.close();
69.
70.             Cipher cipher = Cipher.getInstance("RSA");
71.             cipher.init(Cipher.UNWRAP_MODE, privateKey);
72.             Key key = cipher.unwrap(wrappedKey, "AES", Cipher.SECRET_KEY);
73.
74.             OutputStream out = new FileOutputStream(args[2]);
75.             cipher = Cipher.getInstance("AES");
76.             cipher.init(Cipher.DECRYPT_MODE, key);
77.
78.             crypt(in, out, cipher);
79.             in.close();
80.             out.close();
81.          }
```

Listing 9–18 RSATest.java (continued)

```
82.       }
83.       catch (IOException e)
84.       {
85.          e.printStackTrace();
86.       }
87.       catch (GeneralSecurityException e)
88.       {
89.          e.printStackTrace();
90.       }
91.       catch (ClassNotFoundException e)
92.       {
93.          e.printStackTrace();
94.       }
95.    }
96.
97.    /**
98.     * Uses a cipher to transform the bytes in an input stream and sends the transformed bytes
99.     * to an output stream.
100.    * @param in the input stream
101.    * @param out the output stream
102.    * @param cipher the cipher that transforms the bytes
103.    */
104.   public static void crypt(InputStream in, OutputStream out, Cipher cipher)
105.         throws IOException, GeneralSecurityException
106.   {
107.      int blockSize = cipher.getBlockSize();
108.      int outputSize = cipher.getOutputSize(blockSize);
109.      byte[] inBytes = new byte[blockSize];
110.      byte[] outBytes = new byte[outputSize];
111.
112.      int inLength = 0;
113.      ;
114.      boolean more = true;
115.      while (more)
116.      {
117.         inLength = in.read(inBytes);
118.         if (inLength == blockSize)
119.         {
120.            int outLength = cipher.update(inBytes, 0, blockSize, outBytes);
121.            out.write(outBytes, 0, outLength);
122.         }
123.         else more = false;
124.      }
125.      if (inLength > 0) outBytes = cipher.doFinal(inBytes, 0, inLength);
126.      else outBytes = cipher.doFinal();
127.      out.write(outBytes);
128.   }
129.
130.   private static final int KEYSIZE = 512;
131. }
```

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.

Practical, realistic guide to building MIDP 2.0/MSA applications that are robust, responsive, maintainable, and fun

BASIC USER INTERFACE

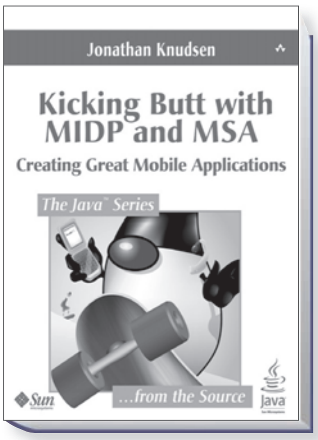
FROM

KICKING BUTT WITH MIDP AND MSA

Creating Great Mobile Applications

by Jonathan Knudsen

©2008 | 432 PAGES | ISBN: 0-321-46342-0



ALSO AVAILABLE

- SAFARI BOOKS ONLINE
- E-BOOK: 0137134002
- MOBI POCKET: 0137134010
- SONY READER: 0137134029

TABLE OF CONTENTS

SECTION I:
GETTING STARTED
1: Overview
2: Tools
3: Quick Start
4: Core APIs

SECTION II:
THE LIVES OF MIDLETS
5: The MIDlet Habitat
6: Starting MIDlets Automatically

SECTION III:
USER INTERFACE
7: Basic User Interface
8: More User Interface

SECTION IV:
GRAPHICS
9: Creating Custom Screen
10: Custom Items
11: Using the Game API
12: Scalable Vector Graphics
13: 3D Graphics

SECTION V:
STORAGE AND RESOURCES
14: Record Stores
15: Reading and Writing Files
16: Contacts and Calendars
17: Mobile Internationalization

SECTION VI:
NETWORKING
18: The Generic Connection Framework
19: Text and Multimedia Messaging
20: Bluetooth and OBEX
21: XML and Web Services
22: Session Initiation Protocol

SECTION VII:
MULTIMEDIA
23: Playing and Recording Sound and Video
24: Advanced Multimedia

SECTION VIII:
SECURITY AND TRANSACTIONS
25: Smart Cards and Cryptography
26: Mobile Payments
27: Know Where You Are
28: Application Architecture

FOR MORE INFORMATION
informit.com/title/9780321463425

Addison Wesley

Section III

User Interface

7

Basic User Interface

BUILDING a user interface in MIDP is entirely different from desktop or server application development. The MIDP user interface APIs, known as LCDUI, were designed for the unique constraints of a small environment:

- Screen sizes vary among devices. The screens you get with MIDP adjust smoothly to any screen size.
- Input methods vary among devices. LCDUI offers the abstraction of *commands*, which can adapt to the specific controls of a device.

This chapter and Chapter 8, “More User Interface,” discuss the screens that LCDUI supplies and their supporting baggage. If you want more control or are trying to do something fancy, you can do all the drawing yourself using `Canvas` or `GameCanvas`, which are covered in Chapter 9, “Creating Custom Screens,” and Chapter 11, “Using the Game API,” respectively. Another approach is to build an SVG Tiny user interface for your application. See Chapter 12, “Scalable Vector Graphics,” for information about using the JSR 226 SVG API to interact with the user.

LCDUI offers four ready-made screens, all descendants of `javax.microedition.lcdui.Displayable`:

- `TextBox` has a title and a text area that fills the whole screen.
- `Alert` is suitable for showing short messages to the user. In addition to text, it can show an icon and make an appropriate sound.

- `List` shows a list of items that can have associated icons. `List` can be used as a menu.
- `Form` can contain a combination of items, like checkboxes, text fields, images, and others.

This chapter introduces the fundamentals of LCDUI and its two simplest screens, `TextBox` and `Alert`. It also describes commands in full detail. The next chapter explores the more complex and useful screens, `List` and `Form`.

7.1 How to Show Screens

The device's display is represented by an instance of `javax.microedition.lcdui.Display`. You've already seen this class. To use it, pass your `MIDlet` object to the static `getDisplay()` method. Usually, you do this in the MIDlet's constructor or `startApp()`, something like this:

```
Display d = Display.getDisplay(this);
```

To show one of the four screen types, use one of `Display`'s `setCurrent()` methods. You've seen this kind of thing already:

```
Form f = new Form("Sign in");
d.setCurrent(f);
```

This snippet is how it works for `TextBox`, `List`, and `Form`. You just create the screen and put it up.

`Alert` is different. `Alert` is supposed to show a message to the user, but it is a temporary notice on the way to another screen. To show an `Alert`, you have to also tell `Display` about the next screen by using a different form of `setCurrent()`:

```
Alert a = new Alert("Empty alert");
d.setCurrent(a, f);
```

This code tells `Display` to show the `Alert a` first. When the `Alert` is finished, the `Form f` will become the current screen.

Keep in mind that `setCurrent()` does not immediately change what the user sees. It merely queues a request to the device to change the current screen next time it gets a chance. Suppose you call `setCurrent()` during the `startApp()` method of your application. The screen update might not happen until `startApp()` returns and relinquishes control of the system thread.

Even when you show screens on the display, you will not get to cover the *whole* display. Usually, devices reserve a status bar of themselves, something that shows battery life, signal strength, and other information. Even the emulator in your development tool will show this behavior. If you really want the whole display, use `setFullScreenMode()` in `Canvas` (see Chapter 9).

7.2 TextBox, the Runt of the Litter

`TextBox` is one of those things that sounds good on paper but isn't of much practical use, like some of those kitchen tools that are collecting dust in the back corners of your cabinets. Did you really think you would ever want to make your own beef jerky?

`TextBox` uses the entire screen to display text or allow the user to input text. This capability might have made sense a few years ago when many mobile phones had tiny displays, but nowadays displays are quite a bit larger. While it's possible you could use `TextBox` to display a lot of text (instructions, perhaps), you'd be better off using a `Form` instead, because you'll have more control over the text and be able to include images. On the flip side, you could use `TextBox` to allow the user to input a lot of text, but that's a bad idea too. Inputting text is painful on mobile devices. If your application calls for extensive text input on a mobile device, you should modify your application design. For text input, use a `TextField` in a `Form` instead. You'll find out all about these classes in the next chapter.

`TextBox` has a title and a large text area that usually takes up the rest of the screen. The actual appearance of `TextBox` is determined by the device.

Create a `TextBox` by specifying the title, the text, the maximum length, and constraints. The constraints are defined by constants in the `TextField` class, which is part of the `Form` family and are described in the next chapter. The constraints are as follows:

- `TextField.ANY` allows any type of input.
- `TextField.DECIMAL` allows numbers with decimal points.
- `TextField.EMAILADDR` limits input to valid characters in an e-mail address.
- `TextField.NUMERIC` is for numbers without decimal points.
- `TextField.PHONENUMBER` allows telephone numbers.
- `TextField.URL` is for entering URLs.

As if that weren't enough, you can make a constraint more specific by adding a *modifier* to it. Here are the modifiers:

- `INITIAL_CAPS_SENTENCE` makes the first letter a capital letter.
- `INITIAL_CAPS_WORD` makes the first letter of each word a capital letter.
- `NON_PREDICTIVE` is used to suppress whatever fancy text input guessing your device might be doing. Many phones let you tap away on the numeric keyboard and make educated guesses about the words you might be trying to form. Using the `NON_PREDICTIVE` modifier should turn off this behavior.
- `PASSWORD` usually shows characters as asterisks so that anyone looking over your shoulder won't be able to see your password. Sometimes password fields show you the last character that you're typing so you can make sure it's correct.
- `SENSITIVE` is for information that shouldn't be stored in a predictive input dictionary or cached in any other way.
- `UNEDITABLE` is for text that cannot be edited.

The constraints and modifiers are really only *suggestions* to the implementation. It's a lot like when you ask the plumber to take his muddy boots off before walking across your carpet. You can ask, but it doesn't mean anything is going to happen. The Sun Java Wireless Toolkit emulator is good about honoring constraints and modifiers, but my Motorola V3 doesn't seem to understand `INITIAL_CAPS_WORD` or `NON_PREDICTIVE`.

Some combinations do not work. If you try to create a `TextBox` with the constraint `NUMERIC` and initial text "Aardvark," the `TextBox` constructor throws `IllegalArgumentException`.

7.3 Input Modes

The constraints apply to the content of the `TextBox`, but you can also request an *input mode* to assist your users. You don't actually request the input mode, you just ask the device to make it easy to enter a certain set of characters. The device should select whatever input mode it has that corresponds.

You can request a Unicode character block or an input subset. The names are defined in the Java SE classes `java.lang.Character.UnicodeBlock` and `java.awt.im.InputSubset` respectively. Finally, MIDP defines the subsets `MIDP_UPPERCASE_LATIN` and `MIDP_LOWERCASE_LATIN` for the entry of uppercase and lowercase Latin letters. To request an appropriate input mode, pass the string

name of the corresponding constant to TextBox's setInitialInputMode() method.

The following example requests an input mode appropriate for entering Hebrew characters:

```
// TextBox tb = ...
tb.setInitialInputMode("UCB_HEBREW");
```

7.4 Using Alerts for Notifications

Alerts are good for displaying a message to the user. They have four attributes that control how they are presented:

- The *title* and *text* have the same meaning as TextBox, although the presentation is likely to be a little different.
- You can supply an *image* to be displayed in the Alert. Pass null to Alert's constructor to show no image.
- An *alert type* is a hint. If you tell the device that the Alert you want to show is an informational alert, the device might try to display it using a standard image and might even play a sound.

The alert types are defined as static constants in AlertType and have the self-explanatory names ALARM, CONFIRMATION, ERROR, INFO, and WARNING.

Alerts have a *timeout* that determines how long the alert is shown on the display before the next screen is shown. The timeout has a default value, which is returned from getDefaultTimeout(). You can set a new timeout value (in milliseconds) with setTimeout(). If you prefer that your users have to explicitly dismiss an alert, you call setTimeout(Alert.FOREVER). The system will provide some way to dismiss the alert (probably a **Done** or **OK** command).

You can add commands to an alert, just as you can to any other screen, but the rules about commands and command listeners are a little funky. Check the Alert API documentation for details. If you find yourself wanting to add commands to an alert, consider using a different type of screen.

Each device has some freedom in deciding how to show an alert. Some have icon images and sounds for the alert types. Some show the image you've supplied and some do not. The arguments you are passing to the Alert constructor are requests, not orders, although you can be sure that the title and message text will show up on the display.

7.5 A Very Quick Introduction to Images

You won't learn the whole story about images until Chapter 9, but for now just know that you can load an image from a resource file using this kind of code:

```
try {
  Image i = Image.createImage("/Minerva-t.png");
}
catch (IOException ioe) {
  // Handle the exception.
}
catch (Exception e) {
  // Handle other types of exceptions.
}
```

This code *does not* load an image from a file system. It looks for the resource file `Minerva-t.png` inside the MIDlet suite JAR file. If you place resource files in the right place in your development project, your development tool will package them into the MIDlet suite JAR file. In the Sun Java Wireless Toolkit, the correct place for resource files is the `res` directory of your project.

It's a good idea to catch `Exception` in addition to `IOException`. Errors in the image bits might cause `IllegalArgumentExceptions`. It's much better for you to catch these and handle them properly than to have your application fail unexpectedly in the field.

Depending on the device, an alert might have a preferred size for images, such that if you supply an image of the right size, it will display particularly well. You can ask `Display` for this size by passing `Display.ALERT` to `getBestImageWidth()` and `getBestImageHeight()`.

7.6 Putting It Together

This example shows how to use `TextBox` and `Alert`.

```
import javax.microedition.lcdui.*;
import javax.microedition.midlet.MIDlet;

public class UIOneMIDlet
    extends MIDlet
    implements CommandListener {
  private Display mDisplay;

  private TextBox mTextBox;
```

```
    private Command mExitCommand, mAlertCommand;

    public void startApp() {
      if (mTextBox == null) {
        mTextBox = new TextBox("Kick Butt with MIDP!",
            "Choose 'Show alert' to see an Alert.",
            128, TextField.UNEDITABLE);
        mExitCommand = new Command("Exit", Command.EXIT, 0);
        mAlertCommand = new Command("Show alert",
            Command.SCREEN, 0);
        mTextBox.addCommand(mExitCommand);
        mTextBox.addCommand(mAlertCommand);
        mTextBox.setCommandListener(this);
        mDisplay = Display.getDisplay(this);
      }
      mDisplay.setCurrent(mTextBox);
    }

    public void pauseApp() {}

    public void destroyApp(boolean unconditional) {}

    public void commandAction(Command c, Displayable d) {
      if (c == mExitCommand) {
        destroyApp(true);
        notifyDestroyed();
      }
      else if (c == mAlertCommand) {
        Image i = null;
        try { i = Image.createImage("/Minerva-t.png"); }
        catch (java.io.IOException ioe) {}
        catch (Exception e) {}
        Alert a = new Alert("This is the title.",
            "This is the message.",
            i, AlertType.INFO);
        a.setTimeout(5000); // Five seconds.
        mDisplay.setCurrent(a, mTextBox);
      }
    }
  }
```

On the Sun Java Wireless Toolkit emulator, the alert looks like Figure 7.1.

As you can see, it is very easy to create and display an alert. Play around with the parameters in `Alert`'s constructor to see the effects. In the Sun Java Wireless Toolkit emulator, `AlertType.ALARM` and `AlertType.CONFIRMATION` have associated sounds. Also try the `FOREVER` timeout to see how it behaves.

Figure 7.1 An alert in the toolkit emulator

7.7 Good for the Old Ticker

All of MIDP's screens can display a *ticker*, which is text that scrolls across the screen, just like that famous sign in Times Square in New York City. For example,

```
Display d = Display.getDisplay(this);
Form f = new Form("Kick Butt with MIDP!");
f.setTicker(new Ticker("I had two birds / " +
        "But I needed three / " +
        "To take me away / " +
        "From the A & P. "));
d.setCurrent(f);
```

In the Sun Java Wireless Toolkit emulator, it looks like the example in Figure 7.2.

7.8 The Whole Story on Commands

You already learned a little about commands in Chapter 3, "Quick Start." Now you'll get the whole story.

A command represents an action that a user can perform in your application. The device decides how the command is shown to the user, and the device figures out how the user invokes the command. Your application provides a listener object that is notified when a command is invoked.

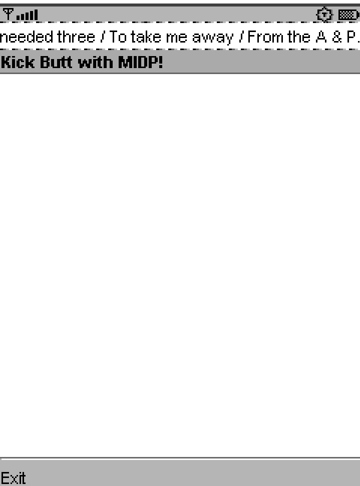


Figure 7.2 A ticker

Commands have a short label, a long label, a type, and a priority. The device decides which label to show. One possibility is that the short label is displayed on the screen next to a soft button and the long label is displayed in a menu of commands. The long label is used only if it is available; it's possible to create a `Command` with only a short label.

The command type helps the device figure out how to show the command. Command types are defined by constants in `Command`. The types for common commands are `BACK`, `CANCEL`, `EXIT`, `HELP`, `OK`, and `STOP`. Application-specific commands use the type `SCREEN`. Finally, the type `ITEM` is used for commands that are applied to `Form` items, but you won't learn about that until the next chapter.

A device is allowed to use a label other than the one you've specified, depending on the command type. For example, if you create a command using `new Command("Ooga booga", Command.EXIT, 0)`, the device could decide to use a label of "Exit" or "Quit" instead of the one you supplied, resulting in a more consistent user experience.

If there are more commands than space on the screen, some of the commands might be hidden in a menu. The Sun Java Wireless Toolkit emulator has space for two commands, one for each of its soft buttons. If you add more commands to a screen, one soft button still invokes a command, while the other soft button brings up a menu of all the others. Figure 7.3 shows a screen with four commands, with the menu of additional commands showing.

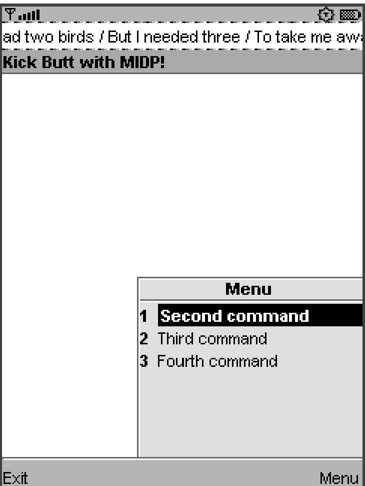


Figure 7.3 Commands that won't fit are stuck in a menu.

The command priority tells the device how important a command is relative to the other commands in a screen. Lower priorities are more important. Assuming you cared more about letting users spend money than viewing a license agreement, you might create command priorities like this:

```
Command exitCommand = new Command("Exit", Command.EXIT, 8);
Command nextCommand = new Command("Next item",
    "Show the next item", Command.SCREEN, 2);
Command addCommand = new Command("Add to cart",
    "Add this item to your shopping cart",
    Command.SCREEN, 2);
Command licenseCommand = new Command("View license",
    Command.SCREEN, 700);
```

Each screen has one command listener that is notified when a command is invoked. The command listener can be any object, but it is often convenient to make the MIDlet a command listener. The `CommandListener` interface has one method:

```
public void commandAction(Command c, Displayable d)
```

The `Displayable` is the screen whose command has just been invoked. You can register a single listener with multiple screens, in which case it is useful to know to which screen's command you are responding. The `Command` parameter is, of course, the command that the user has invoked.

Remember that the system thread that calls your `commandAction()` method does not belong to you. Be quick about your work in `commandAction()`, or create your own thread if you must do something that takes a long time.

7.9 Command Placement

The placement of commands is, in some cases, inscrutable. Even the simplest use of commands sometimes leads to an uneven user experience across a variety of devices.

One surprising example is a `List` (from the next chapter) that has a single command, **OK** (Figure 7.4). My Motorola V3 and Sun's emulator behave as you would expect, although it's probably just lucky that the **OK** command is on the right side in both of them.

Now try this on a Nokia Series 40 handset (Figure 7.5). Nokia adds its own **Mark** or **Unmark** command to this type of list, mapped to the center select button. The left and right soft buttons are both available, but for some reason the **OK** command gets stuck inside an **Options** menu. When you choose **Options**, you're taken to another screen with the **OK** command, a **Select** command, and a **Back** command. Yuck!

If you need to provide a user experience with more consistency across platforms, you'll have to build your own solution using `Canvas`, which is described in Chapter 9.

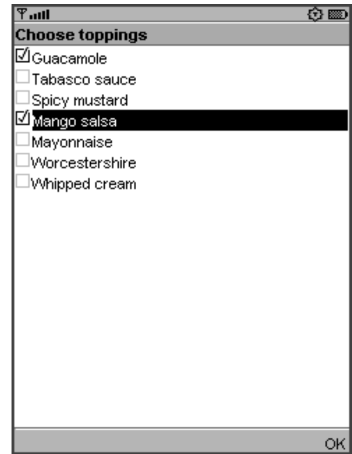


Figure 7.4 No surprises here.

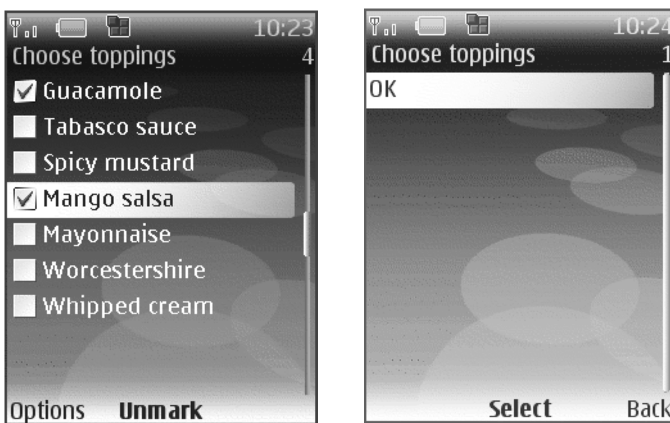


Figure 7.5 The **OK** command is hidden in an **Options** menu.

7.10 Summary

In this chapter, you got an introduction to MIDP's user interface APIs. You learned about the simplest screens, `TextBox` and `Alert`, and how to show them on a device's display. All `Displayables` can show a ticker, which is scrolling text. The user performs actions in an application by using commands. The device determines how commands look and how the user invokes them. A command listener object, which is often a MIDlet, responds to commands.

LearnIT at InformIT

Go Beyond the Book

Read IT
Transport IT
Absorb IT
Follow IT
Search IT
Watch IT
Research IT
Discuss IT
Cite IT
Engage IT
Catch IT

Books
eBooks
Short Cuts
Rough Cuts
Safari Books Online
Video
Articles
Blogs
Reference Guides
Conferences
Podcasts

informIT

11 WAYS TO LEARN IT at **www.informIT.com/learn**

The digital network for the publishing imprints of Pearson Education

Cisco Press

EXAM/CRAM