

Java Enterprise in a Nutshell, 3rd Edition

By William Crawford, Jim Farley

.....
Publisher: O'Reilly

Pub Date: November 2005

ISBN: 0-596-10142-2

Pages: 892

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

Overview

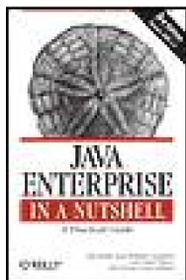
Nothing is as constant as change, and this is as true in enterprise computing as anywhere else. With the recent release of Java 2 Enterprise Edition 1.4, developers are being called on to add even greater, more complex levels of interconnectivity to their applications.

To do this, Java developers today need a clear understanding of how to apply the new APIs, use the latest open source Java tools, and learn the capabilities and pitfalls in Java 2 Enterprise Edition 1.4 - so they can plan a technology and implementation strategy for new enterprise projects.

Fortunately, this is exactly what they get with the new *Java Enterprise in a Nutshell*, 3rd Edition. Because most integrated development environments (IDE) today include API lookup, we took out the main API sections from our previous edition to make room for new chapters, among others, on Ant, Cactus, Hibernate, Jakarta Struts, JUnit, security, XDoclet, and XML/JAXP.

Revised and updated for the new 1.4 version of Sun Microsystems Java Enterprise Edition software, *Java Enterprise in a Nutshell*, 3rd Edition is a practical guide for enterprise Java developers.

← PREV



Java Enterprise in a Nutshell, 3rd Edition

By William Crawford, Jim Farley

.....
Publisher: O'Reilly

Pub Date: November 2005

ISBN: 0-596-10142-2

Pages: 892

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

— [Copyright](#)

— [Preface](#)

— [What's New in This Edition](#)

— [Contents of This Book](#)

— [Java Programming Resources](#)

— [Examples Online](#)

— [Conventions Used in This Book](#)

— [Using Code Examples](#)

— [Safari® Enabled](#)

— [Comments and Questions](#)

— [Acknowledgments](#)

— [Part I: The Java Enterprise APIs](#)

— [Chapter 1. Introduction](#)

— [Section 1.1. Enterprise Computing Defined](#)

— [Section 1.2. Enterprise Computing Demystified](#)

— [Section 1.3. Standard Java Enterprise APIs](#)

— [Section 1.4. De Facto Standard Enterprise Development Tools](#)

— [Section 1.5. An Enterprise Computing Scenario](#)

— [Section 1.6. Other Enterprise APIs](#)

— [Chapter 2. Application Assembly and Deployment](#)

— [Section 2.1. J2EE Application Assembly Model](#)

— [Section 2.2. Component Modules](#)

— [Section 2.3. Application Assemblies](#)

— [Section 2.4. Deploying J2EE Applications](#)

— [Chapter 3. Java Servlets](#)

— [Section 3.1. Getting a Servlet Environment](#)

— [Section 3.2. Servlet Basics](#)

— [Section 3.3. Web Applications](#)

— [Section 3.4. Servlet Requests](#)

— [Section 3.5. Servlet Responses](#)

— [Section 3.6. Custom Servlet Initialization](#)

— [Section 3.7. Security](#)

— [Section 3.8. Servlet Filters](#)

- [Section 3.9. Thread Safety](#)
- [Section 3.10. Cookies](#)
- [Section 3.11. Session Tracking](#)
- [Section 3.12. Databases and Non-HTML Content](#)
- [Chapter 4. JavaServer Pages](#)
 - [Section 4.1. JSP Basics](#)
 - [Section 4.2. JSP Actions](#)
 - [Section 4.3. The JSP Expression Language](#)
 - [Section 4.4. JSP Standard Tag Library](#)
 - [Section 4.5. Custom Tags](#)
 - [Section 4.6. Wrapping Up](#)
- [Chapter 5. JavaServer Faces](#)
 - [Section 5.1. The Sample Application](#)
 - [Section 5.2. Structure of a JSF Application](#)
 - [Section 5.3. Managed Beans](#)
 - [Section 5.4. The JSF Expression Language](#)
 - [Section 5.5. JSF Actions and Views](#)
 - [Section 5.6. Building Tables](#)
 - [Section 5.7. Validation](#)
 - [Section 5.8. Moving on with JSF](#)
- [Chapter 6. Enterprise JavaBeans](#)
 - [Section 6.1. What Version Is Covered Here?](#)
 - [Section 6.2. EJB Component Model Overview](#)
 - [Section 6.3. EJB Tutorial](#)
 - [Section 6.4. Deploying EJBs](#)
 - [Section 6.5. Using Enterprise JavaBeans](#)
 - [Section 6.6. Session Bean Specifics](#)
 - [Section 6.7. Entity Beans](#)
 - [Section 6.8. Message-Driven Beans](#)
 - [Section 6.9. Transaction Management](#)
 - [Section 6.10. EJB 3.0](#)
- [Chapter 7. Java and XML](#)
 - [Section 7.1. Using XML Documents](#)
 - [Section 7.2. Java API for XML Processing](#)
 - [Section 7.3. SAX](#)
 - [Section 7.4. DOM](#)
 - [Section 7.5. XSLT](#)
- [Chapter 8. JDBC](#)
 - [Section 8.1. JDBC Architecture](#)
 - [Section 8.2. Connecting to the Database](#)
 - [Section 8.3. Statements](#)
 - [Section 8.4. Results](#)
 - [Section 8.5. Handling Errors](#)
 - [Section 8.6. Prepared Statements](#)
 - [Section 8.7. BLOBs and CLOBs](#)
 - [Section 8.8. Metadata](#)

- [Section 8.9. Transactions](#)
- [Section 8.10. Stored Procedures](#)
- [Section 8.11. Escape Sequences](#)
- [Section 8.12. RowSets](#)
- [Chapter 9. JNDI](#)
 - [Section 9.1. JNDI Architecture](#)
 - [Section 9.2. A Simple Example](#)
 - [Section 9.3. Introducing the Context](#)
 - [Section 9.4. Looking Up Objects in a Context](#)
 - [Section 9.5. The NamingShell Application](#)
 - [Section 9.6. Listing the Children of a Context](#)
 - [Section 9.7. Creating and Destroying Contexts](#)
 - [Section 9.8. Binding Objects](#)
 - [Section 9.9. Accessing Directory Services](#)
 - [Section 9.10. Modifying Directory Entries](#)
 - [Section 9.11. Creating Directory Entries](#)
 - [Section 9.12. Searching a Directory](#)
 - [Section 9.13. Event Notification](#)
- [Chapter 10. J2EE Security](#)
 - [Section 10.1. Basic Security Concepts](#)
 - [Section 10.2. A Look at Java and J2EE Security Standards](#)
 - [Section 10.3. Declarative Security Versus Programmatic Security](#)
 - [Section 10.4. Web Component Security](#)
 - [Section 10.5. EJB Component Security](#)
 - [Section 10.6. Other J2EE Security Topics](#)
 - [Section 10.7. Limitations of J2EE Security](#)
- [Chapter 11. Java Message Service](#)
 - [Section 11.1. JMS in the J2EE Environment](#)
 - [Section 11.2. Elements of Messaging with JMS](#)
 - [Section 11.3. The Anatomy of Messages](#)
 - [Section 11.4. Point-to-Point Messaging](#)
 - [Section 11.5. Publish-Subscribe Messaging](#)
 - [Section 11.6. Unified Messaging](#)
 - [Section 11.7. Transactional Messaging](#)
- [Chapter 12. Web Services with JAX-RPC and SAAJ](#)
 - [Section 12.1. What's Covered Here?](#)
 - [Section 12.2. Brief Introduction to Web Services](#)
 - [Section 12.3. Java Web Services](#)
 - [Section 12.4. Writing Web Service Clients](#)
 - [Section 12.5. Writing Web Services](#)
 - [Section 12.6. Deploying Web Services](#)
- [Chapter 13. Remote Method Invocation](#)
 - [Section 13.1. What's Covered Here?](#)
 - [Section 13.2. Introduction to RMI](#)
 - [Section 13.3. Defining Remote Objects](#)
 - [Section 13.4. Creating the Stubs and Skeletons](#)

- [Section 13.5. Accessing Remote Objects as a Client](#)
- [Section 13.6. Dynamic Classloading](#)
- [Section 13.7. Remote Object Activation](#)
- [Section 13.8. RMI and Native Method Calls](#)
- [Section 13.9. RMI Over IIOP](#)
- [Chapter 14. Java IDL \(CORBA\)](#)
 - [Section 14.1. A Note on Evolving Standards](#)
 - [Section 14.2. The CORBA Architecture](#)
 - [Section 14.3. Creating CORBA Objects](#)
 - [Section 14.4. Putting It in the Public Eye](#)
 - [Section 14.5. Finding and Using Remote Objects](#)
 - [Section 14.6. What If I Don't Have the Interface?](#)
- [Chapter 15. JavaMail](#)
 - [Section 15.1. Email and JavaMail](#)
 - [Section 15.2. Creating and Sending Messages](#)
 - [Section 15.3. Retrieving Messages](#)
 - [Section 15.4. Multipart Messages](#)
- [Chapter 16. Transactions](#)
 - [Section 16.1. Transaction Overview](#)
 - [Section 16.2. Programmatic Transactions Versus Declarative Transactions](#)
 - [Section 16.3. Optimistic Concurrency](#)
 - [Section 16.4. EJB Transaction Management](#)
 - [Section 16.5. Some Common Programming Scenarios](#)
 - [Section 16.6. Transaction Best Practices](#)
- [Part II: Open Source Enterprise Tools](#)
 - [Chapter 17. Ant](#)
 - [Section 17.1. What Version Is Covered Here?](#)
 - [Section 17.2. Ant Overview](#)
 - [Section 17.3. Ant Fundamentals](#)
 - [Section 17.4. Core Tasks](#)
 - [Section 17.5. Enterprise Tasks](#)
 - [Section 17.6. Creating Portable Build Processes](#)
 - [Chapter 18. JUnit and Cactus](#)
 - [Section 18.1. What's Covered Here?](#)
 - [Section 18.2. Unit Testing Concepts](#)
 - [Section 18.3. JUnit Overview](#)
 - [Section 18.4. Using JUnit with Ant](#)
 - [Section 18.5. Testing Enterprise Components with Cactus](#)
 - [Chapter 19. Struts](#)
 - [Section 19.1. The Scope of Struts](#)
 - [Section 19.2. The Sample Application](#)
 - [Section 19.3. The Development Process with Struts](#)
 - [Section 19.4. The Struts Controller](#)
 - [Section 19.5. The Action Class](#)
 - [Section 19.6. Views in Struts](#)
 - [Section 19.7. Struts Tags](#)

- [Section 19.8. Struts Plug-ins](#)
- [Section 19.9. DynaActionForms and the Struts Validator](#)
- [Chapter 20. Hibernate](#)
 - [Section 20.1. The Sample Application](#)
 - [Section 20.2. Principles of Hibernate](#)
 - [Section 20.3. Configuration and Mapping](#)
 - [Section 20.4. The Hibernate API](#)
 - [Section 20.5. HQL \(Hibernate Query Language\)](#)
 - [Section 20.6. Hibernate Services](#)
 - [Section 20.7. Conclusion](#)
- [Chapter 21. Annotations with XDoclet and J2SE Metadata](#)
 - [Section 21.1. What's Covered Here?](#)
 - [Section 21.2. What Are Code Annotations?](#)
 - [Section 21.3. Annotation Tools](#)
 - [Section 21.4. XDoclet Tutorial](#)
 - [Section 21.5. J2SE Annotations Tutorial](#)
- [Part III: Appendixes](#)
 - [Appendix A. J2EE Deployment Descriptor Reference](#)
 - [Section A.1. Web Components \(web.xml\)](#)
 - [Section A.2. Enterprise JavaBeans \(ejb-jar.xml\)](#)
 - [Section A.3. Application Archives \(application .xml\)](#)
 - [Section A.4. Web Services \(webservices.xml\)](#)
 - [Section A.5. Web Service Java/WSDL Mappings](#)
 - [Appendix B. JavaServer Faces Tag Libraries](#)
 - [Section B.1. JSF Core Tags](#)
 - [Section B.2. JSF HTML Tags](#)
 - [Appendix C. Enterprise JavaBeans Query Language Syntax](#)
 - [Section C.1. Basic Structure of EJB QL Queries](#)
 - [Section C.2. FROM Clause](#)
 - [Section C.3. SELECT Clause](#)
 - [Section C.4. WHERE Clause](#)
 - [Section C.5. ORDER BY Clause](#)
 - [Appendix D. SQL Reference](#)
 - [Section D.1. Relational Databases](#)
 - [Section D.2. Data Types](#)
 - [Section D.3. Schema Manipulation Commands](#)
 - [Section D.4. Data Manipulation Commands](#)
 - [Section D.5. Functions](#)
 - [Section D.6. Return Codes](#)
 - [Appendix E. JMS Message Selector Syntax](#)
 - [Section E.1. Structure of a Selector](#)
 - [Section E.2. Identifiers](#)
 - [Section E.3. Literals](#)
 - [Section E.4. Operators](#)
 - [Section E.5. Expressions](#)
 - [Appendix F. FRMI Tools](#)

- [rmic: The Java RMI Compiler](#)
- [rmid: The RMI Activation Daemon](#)
- [rmiregistry: The Java RMI Object Registry](#)
- [serialver: The RMI Serial Version Utility](#)
- [Appendix G. IDL Reference](#)
 - [Section G.1. IDL Keywords](#)
 - [Section G.2. Identifiers](#)
 - [Section G.3. Comments](#)
 - [Section G.4. Basic Data Types](#)
 - [Section G.5. Constants and Literals](#)
 - [Section G.6. Naming Scopes](#)
 - [Section G.7. User-Defined Data Types](#)
 - [Section G.8. Exceptions](#)
 - [Section G.9. Module Declarations](#)
 - [Section G.10. Interface Declarations](#)
 - [Section G.11. Value Type Declarations](#)
- [Appendix H. HJava IDL Tools](#)
 - [idlj: The Java IDL Compiler](#)
 - [orbd: Naming Service Daemon](#)
 - [servertool](#)
 - [tnameserv: Transient Naming Service Daemon](#)
- [About the Authors](#)
- [Colophon](#)
- [Index](#)



Java Enterprise in a Nutshell, Third Edition

by Jim Farley and William Crawford with Prakash Malani, John G. Norman, and Justin Gehtland

Copyright © 2006 O'Reilly Media, Inc. All rights reserved. Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (safari.oreilly.com). For more information, contact our corporate/institutional sales department: (800) 998-9938 or corporate@oreilly.com.

Editor:	Mike Loukides and Debra Cameron
Production Editor:	A. J. Fox
Cover Designer:	Edie Freedman
Interior Designer:	David Futato

Printing History:

September 1999:	First Edition.
April 2002:	Second Edition.
November 2005:	Third Edition.

Nutshell Handbook, the Nutshell Handbook logo, and the O'Reilly logo are registered trademarks of O'Reilly Media, Inc. The *In a Nutshell* series designations, *Java Enterprise in a Nutshell*, Third Edition, the image of a sand dollar, and related trade dress are trademarks of O'Reilly Media, Inc.

Java™ and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc., in the United States and other countries. O'Reilly Media, Inc. is independent of Sun Microsystems.

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 O'Reilly Media, Inc. was aware of a trademark claim, the designations have been printed in caps or initial caps.

While every precaution has been taken in the preparation of this book, the publisher and authors assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

ISBN: 0-596-10142-2

[M]



[← PREV](#)

Preface

This book is both a practical guide and a quick reference for Java™ programmers who are writing enterprise applications. The world of enterprise development in the Java environment consists of a mix of standardized APIs and services and de facto standard tools and APIs that have been developed in the Java community. The first part of the book provides a fast-paced introduction to key standard Java enterprise principles and APIs: J2EE™ deployment, servlets, JavaServer Pages™ (JSPs), JavaServer™ Faces (JSF), Enterprise JavaBeans™ (EJB), XML, JDBC™, Java Naming and Directory Interface (JNDI), J2EE security principles, Java Message Service (JMS), SOAP web services, Remote Method Invocation (RMI), Java IDL (CORBA), JavaMail™, and transactions. The second part of the book covers a select set of key de facto tools commonly used in Java enterprise development: Ant, JUnit, Cactus, Struts, Hibernate, and XDoclet.

[← PREV](#)

What's New in This Edition

The landscape of Java enterprise tools and APIs has evolved quite a bit in the three years since the second edition. As we were planning the third edition, we realized that the set of enterprise tools that developers found useful had settled into a mix of standard and de facto standard solutions. The standard solutions are APIs within the Java 2 Enterprise Edition (J2EE) and Java 2 Standard Edition (J2SE™) set of specifications, like JavaServer Pages and JDBC. The de facto standard solutions are APIs and tools developed by the open source community, such as Hibernate and Ant.

We felt it was critical to cover both aspects of the enterprise toolbox, and so we decided to drop the API reference material typically found in the Nutshell series in favor of coverage of the de facto tools in the Java enterprise space. As you look through [Part II](#) of the book, you'll find tutorial chapters on a critical set of these community tools, such as Ant, Hibernate, and Struts, among others. This change is a recognition that enterprise developers have numerous ways to access API details, using tools integrated into their development environments. More importantly, it's a recognition that effective tutorials on these common community tools are critical to having a complete picture of the larger Java enterprise framework, a framework that, today, is much larger than just the standard APIs published as part of the J2EE specification.

In addition to this brand new material, we've also brought all the material up-to-date with the current versions of the various APIs and tools. The examples and tutorial material now cover J2EE 1.4 versions of the various standard APIs, and we also cover the current released versions of the community tools at the time of this writing. In key areas, we've also provided some previews of new developments coming in the near future.



Contents of This Book

This book is divided into three parts:

[Part I](#), *The Java Enterprise APIs*

The chapters in this part provide tutorials in the key standard Java Enterprise APIs and principles and provide enough information so that you can start using them right away.

[Part II](#), *Open Source Enterprise Tools*

[Part II](#) is similar to [Part I](#), except that the chapters cover important but nonstandard community tools in the Java enterprise space.

[Part III](#), *Appendixes*

This part of the book contains summary and reference information.

Java Programming Resources

This book is designed to provide tutorials in a wide variety of J2EE APIs and tools, both standard and de facto. It does not, and cannot, tell you everything you need to know about these APIs and tools. In fact, in almost every case, you'll find that there are one or more entire books devoted to the topic covered in any given chapter, and we refer you to those books for further reading. In addition to full-length treatments of these topics, there are several valuable (and free) electronic sources of information about Java programming.

Sun's web site for all things related to Java is <http://java.sun.com>. The web site specifically for Java developers is <http://developer.java.sun.com>. Much of the content on this developer site is password-protected, and access to it requires (free) registration.

Some of the enterprise APIs covered in this book are part of the core Java 2 platform, so if you have downloaded the JDK, you already have the classes for APIs such as JDBC, RMI, XML, JNDI, and Java IDL. Other APIs are standard extensions, however; so if you want to use, say, EJBs or servlets, you have to download the classes separately, either by themselves or as part of an application server (for example, the Apache Tomcat web container will provide the servlet and JSP APIs). The best way to get the latest APIs is to start on Sun's Products and APIs page at <http://java.sun.com/products/> and find the appropriate API. And of course, the various de facto standard tools discussed in the second part of the book, like JUnit, Ant, and Hibernate, each have their own download locations as described in their respective chapters.

Sun distributes electronic documentation for all Java classes and methods in its *javadoc* HTML format. Although this documentation is sometimes sparse, it is still an excellent starting point when you need to know more about a particular Java package, class, method, or field. If you don't already have the *javadoc* files with your Java distribution, see <http://java.sun.com/docs/> for a link to the latest available version.

Finally, don't forget O'Reilly's Java web site (<http://java.oreilly.com>), which contains Java news and commentary. In addition, the O'Reilly network offers [OnJava.com](http://www.onjava.com) at <http://www.onjava.com>, which contains insightful tips and tricks for every level of Java programmer.



Examples Online

The examples in this book are available online and can be downloaded from the home page for the book at <http://www.oreilly.com/catalog/javaentnut3/>. You may also want to visit this site for important notes about the book or errata.



Conventions Used in This Book

This section covers typographic conventions and the format used for XML structure diagrams.

Typographic Conventions

Italic

Used for emphasis and to signify the first use of a term. Italic is also used for commands and file and directory names.

Constant width

Used in all Java code and generally for anything that you would type literally when programming, including options, keywords, data types, constants, method names, variables, class names, and interface names.

Constant width italic

Used for the names of function arguments and generally as a placeholder to indicate an item that should be replaced with an actual value in your program.

Constant width bold

Used for user input on the command line.

Indicates a tip, suggestion, or general note.

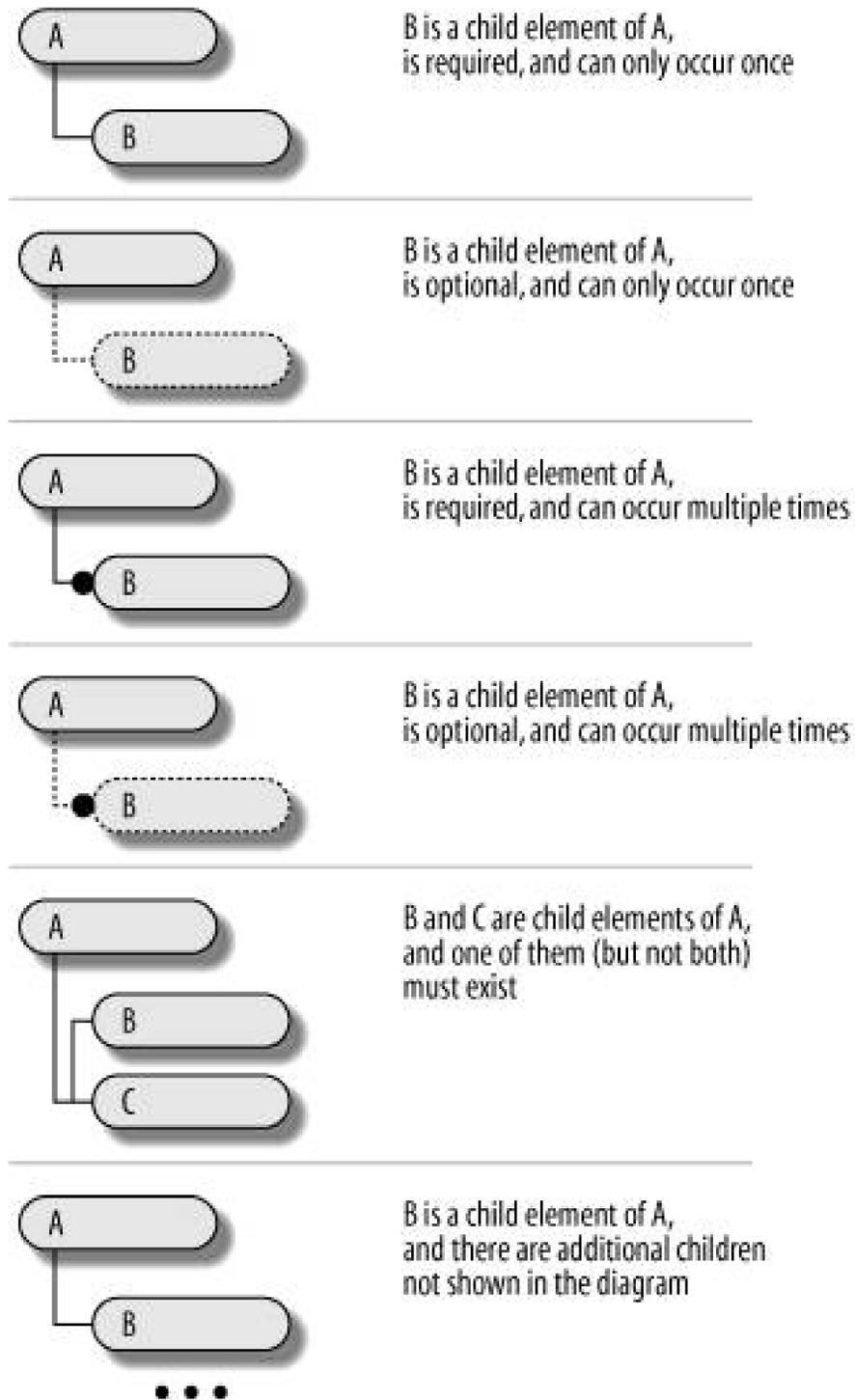
Indicates a warning or caution.

XML Schema/DTD Diagrams

In a number of places in the book, we need to describe the structure of specific XML formats (based

on either DTDs or XML schemas). We've chosen to use a simplified diagram format to show these XML structures graphically. This format is described in the key in [Figure P-1](#).

Figure P-1. Key to XML structure diagrams



[Figure P-2](#) shows a sample XML structure diagram. Here is sample XML data that follows this structure:

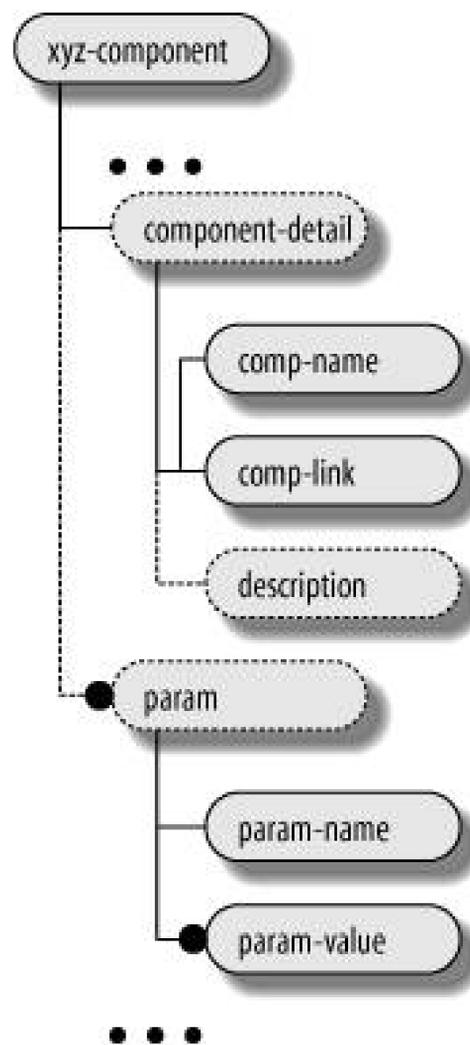
```
<xyz-component>
  . . .
  <component-detail>
    <comp-name>Name</comp-name>
  </component-detail>
```

```

<param>
  <param-name>param1</param-name>
  <param-value>value1</param-value>
  <param-value>value2</param-value>
</param>
. . .
</xyz-component>

```

Figure P-2. XML structure diagram



Note that these XML diagrams are simplified and don't describe all aspects of XML schemas or DTDs. They do not include element attributes, for example, nor do they describe element sequence constraints in XML schemas. They are meant to provide a high-level view of the XML structure only. For precise details, the corresponding DTD or schema should be consulted.

Using Code Examples

This book is here to help you get your job done. In general, you may use the code in this book in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Java Enterprise in a Nutshell*, Third Edition, by Jim Farley and William Crawford with Prakash Malani, John G. Norman, and Justin Gehtland. Copyright 2006 O'Reilly Media, Inc., 0-596-10142-2."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at permissions@oreilly.com.

← PREV

Safari® Enabled



When you see a Safari® Enabled icon on the cover of your favorite technology book, that means the book is available online through the O'Reilly Network Safari Bookshelf.

Safari offers a solution that's better than e-books. It's a virtual library that lets you easily search thousands of top tech books, cut and paste code samples, download chapters, and find quick answer: when you need the most accurate, current information. Try it for free at <http://safari.oreilly.com>.

← PREV

 PREVIOUS

Comments and Questions

Please address comments and questions concerning this book to the publisher:

O'Reilly Media
1005 Gravenstein Highway North
Sebastopol, CA 95472
(800) 998-9938 (in the United States or Canada)
(707) 829-0515 (international or local)
(707) 829-0104 (fax)

There is a web page for this book that lists errata, examples, and any additional information. You can access this page at:

<http://www.oreilly.com/catalog/javaentnut3/>

To comment or ask technical questions about this book, send email to:

bookquestions@oreilly.com

For more information about books, conferences, Resource Centers, and the O'Reilly Network, see the O'Reilly web site at:

<http://www.oreilly.com>

Acknowledgments

This book began six years ago as an extension of the series begun by *Java in a Nutshell*. The readers of the past two editions have provided us with plenty of constructive comments, criticism, suggestions, and validation over that time, and at the end of the day, they are the inspiration for this new edition. The authors would also like to extend their gratitude to the book's technical reviewers, whose constructive criticism has done much to improve this work: Ed Zarecor, Doug Heath, Pat Niemeyer, Max Cooper, Steve Ditlinger, Bruce Tate, and Roshni Malani.

Jim Farley

First and foremost, I have to thank my coauthors, because this book simply wouldn't exist without them. Many thanks to Will Crawford for revising and extending the bulk of the chapters on web components (servlets, JSP, and JSF), for the updated chapter on JDBC, and the reworked chapter on XML. Thanks to Prakash Malani for his contributions to the chapters on security and transactions, and for his gracious attitude and generous efforts in reviewing others' material; to Justin Gehtland for the much-needed chapter on Hibernate; and to John Norman for the new chapter on Struts, which rounds out the web components material. It's been a pleasure working with you all on this.

But a writing project of any kind requires a much larger cast of characters than those listed on the cover. Deb Cameron had the unenviable job of editing this edition, which turned out to be a much bigger job than she signed up for, due to the new direction we decided to take with the book. She also had to deal with five authors and managed to pull it off with little damage (emotional, physical, temporal, or otherwise) and a lot of grace.

My wife, Sandy Mallalieu, was supportive as always through this latest writing project. My many colleagues in the high-tech industry were invaluable sounding boards and critics for many of the new ideas in this edition. I can't possibly list them all here, but you know who you are, and I thank you for your thoughts on difficult and sometimes esoteric issues. A special round of thanks goes out to all of my Harvard students, who were willing and eager guinea pigs for some of the concepts that found their way into the book in one form or another. And, as usual, for late-night inspiration, my undying gratitude to Madeline and to Declan MacManus (and his new wife).

William Crawford

The usual acknowledgments, of course: my parents, my family, my friends. In the office, I had excellent support from my colleagues at Perceptive Informatics, especially Howard Foster, who did his best to make sure I had time to actually work on writing projects, and Dave Godwin, who bought a large box of books in the hope that it would impress the customers. The jury is still out on that one.

Jim Farley, my partner in crime for three editions now, has really been responsible for keeping this process rolling. Despite Jim's prodigious output, we probably wouldn't have made our final deadline without the help of Prakash Malani, John Norman (who got to port my JSF example over to Struts, in a project that was quite possibly the first and last of its kind), and Justin Gehtland. Welcome aboard.

I'd also like to thank the students in the various courses I've taught over the last year. It's easy to forget the kinds of questions we ask when we're just starting out. In particular, my April 2005 Enterprise Java class at Alliant Corporation gave me the opportunity to test out the JSF tutorial before we went to press, as well as showing me a great time in Hanoi.

The third *Java Enterprise in a Nutshell* Patient Editor award goes to Deb Cameron, with special distinction for this edition, and great appreciation for helping me work around the literally dozens of emergencies that afflicted this last year. Sorry there's no plaque, but thanks for a great job.

Prakash Malani

This writing project has been amazing but challenging at the same time. Deb and my fellow authors have always gone way out of their way to help! Working with these wonderful individuals has been a great pleasure.

I am in great debt to my friend Ron Hitchens for having the confidence to recommend me for this exciting project. I would sincerely like to thank Mike Loukides for trusting me to help out. I am grateful to my close friends Max Cooper and Steve Ditlinger as well as my sister Roshni Malani, who did an awesome job of carefully reviewing the chapters. However, any and all mistakes are solely mine.

My wife, Clare Zhang, and my son, Arun Malani, are always loving, patient, understanding, and encouraging. Without their help and support, this project would not have been possible. I dedicate the chapter on transactions to them.

I dedicate the chapter on security to my parents, Chandrakant and Rajukumari Malani, who have done an awesome job of bestowing everything upon me! Their dedication and sacrifice for their kids always inspires me to be a better person, and especially a better parent!

John G. Norman

Thank you, Jim and Deb, for bringing me on! And to Jim and Will for reading the chapter on Struts with attention and care. My students in the Enterprise Java course at Harvard Extension have been a great help in reminding me to be clear and eschew surplusage. At work, I am grateful to everyone at H3.com, and especially to Stan Ward and Hans Gieskes for not noticing too much when I ducked out occasionally to write. The usual thanks to the likes of Alex Chilton, Norman Blake, Carrie Brownstein and Corin Tucker, Peter Perrett, and Pete Shelley.

Finally, Juliette et Caroline: merci pour tous votre amour et dévotion.

Justin Gehtland

I'd like to thank Deb Cameron and Michael Loukides for thinking of me, the rest of the team for letting me join at the last second, and Gavin King, the founder of the Hibernate project, for launching such a cool tool. And, as always, Lisa and Zoe.



Part I: The Java Enterprise APIs

The chapters in [Part I](#) introduce you to key standard APIs within the Java Enterprise Edition framework. Each chapter provides a broad overview of a particular area of the J2EE environment, as well as tutorial material on the relevant APIs.

[Chapter 1](#), *Introduction*

[Chapter 2](#), *Application Assembly and Deployment*

[Chapter 3](#), *Java Servlets*

[Chapter 4](#), *JavaServer Pages*

[Chapter 5](#), *JavaServer Faces*

[Chapter 6](#), *Enterprise JavaBeans*

[Chapter 7](#), *Java and XML*

[Chapter 8](#), *JDBC*

[Chapter 9](#), *JNDI*

[Chapter 10](#), *J2EE Security*

[Chapter 11](#), *Java Message Service*

[Chapter 12](#), *Web Services with JAX-RPC and SAAJ*

[Chapter 13](#), *Remote Method Invocation*

[Chapter 14](#), *Java IDL (CORBA)*

[Chapter 15](#), *JavaMail*

[Chapter 16](#), *Transactions*

Chapter 1. Introduction

This book has arguably unreasonable goals. Its title alone invites raised eyebrows. Enterprise application development, on any platform, involves a wide array of tools to solve an equally wide array of issues and to manage several levels of complexity. How can you possibly hope to cover Java enterprise development tools in a single work, let alone in a nutshell?

Our approach to this lofty goal is simple. We have selected the key tools and APIs that enterprise architects and developers have found invaluable when working in the Java environment. Some of these are part of the core Java 2 Standard Edition (J2SE) environment; many of them are part of the larger Java 2 Enterprise Edition (J2EE)^[*] environment; many of them fall outside the Java standards but are useful and popular de facto standards within the Java community. For all of the tools covered we provide a short but effective tutorial while also striving to cover enough critical detail that the material can serve as a quick reference as well. Along the way, in keeping with the "nutshell" concept of this series, we've had to edit brutally. Only tools and APIs that are proven in the industry while also being critical to success made the cut. Some things, useful and otherwise, had to be left by the wayside. But we hope that sifting effort alone will benefit you, since the book also serves as a guide through the sometimes confusing swirl of competing APIs, tools, and frameworks that make enterprise development daunting at times.

[*] Sun recently announced a new naming scheme for the various Java platform editions. In the future, the standard edition, J2SE, will be referred to simply as "Java SE," and the enterprise edition, J2EE, will be called "Java EE." We've chosen to stick with the current nomenclature of "J2SE" (or just "Java") for the standard edition and "J2EE" for the enterprise edition. If the new naming scheme takes hold in the community, we're sure our readers are smart enough to make the translation. If you've been working in the Java environment for very long, you've already had a lot of practice in this regard.

We've divided the material in the book into two main sections: tools that are part of the Java and J2EE standards and are managed under the auspices of the Java Community Process and tools that are not part of these standards but have been proven in practice and are de facto standards in the Java development community. Before we summarize the contents of these sections of the book, let's make it clear what we mean by "enterprise computing."

1.1. Enterprise Computing Defined

The term "enterprise computing" raises many different images in people's minds: big, sterile-looking corporate data centers filled with humming servers; complex financial systems stretching across the globe; elaborate multilevel software systems running a stupefying amount of code distributed across many servers, locations, companies, countries, even (in the case of NASA's Mars Rover project) planets.

All of these images are true, to a degree. But they're all imprecise to a degree as well. Enterprise computing can be done on a single student's laptop as well as in a big corporate data center. It can be used in a small mom and pop business with a single storefront as well as in a huge multinational. It can be done just as effectively with a very small amount of code as it can with huge code repositories.

At its heart, enterprise computing is all about combining separate applications, services, and processes into a unified system that is greater than the sum of its parts. The rest is context that determines how you go about accomplishing this basic goal.

Anyone can write enterprise applications, not only because of the generalized nature of enterprise computing but also because the tools and techniques for developing enterprise systems have matured to the point that this domain is accessible to a much broader group of software developers. Many of the arcane details have been smoothed over by standards, well-defined APIs, and really powerful tools. This has made enterprise computing less scary and a lot more fun.

1.2. Enterprise Computing Demystified

Enterprise computing has a reputation for complexity and, for the uninitiated, is often shrouded with mystery. Here are some reasons enterprise computing can seem intimidating:

- Enterprise computing usually takes place in a heterogeneous network of systems, one in which the computers might range from large mainframes and supercomputers down to PCs, mobile laptops, and even handheld devices. The computers were purchased at different times from a variety of different vendors and run two, three, or more different operating systems.
- A variety of server applications run on top of the heterogeneous hardware. An enterprise might support application servers from three different companies, each of which covers a different functional subset of the enterprise and each of which includes different, incompatible interfaces and extensions.
- Enterprise computing involves the use of many different protocols and standards. Some standards overlap in small or significant ways. Many have been extended in various vendor-specific, nonstandard ways. Some are quite old and use a vocabulary and terminology that date back to an earlier era of computing. This creates a confusing alphabet soup of acronyms and abbreviations.
- Enterprise computing has only recently emerged as an integrated discipline of its own. Although enterprise development models are becoming more cohesive and encompassing, many enterprises are still left with lots of "legacy systems" that are aggregated in an ad hoc way.

Java helps to alleviate these intimidating aspects of enterprise computing. First, since Java is platform-independent, the heterogeneous nature of the network ceases to be an issue. Second, the J2EE framework forms a single, standard layer on top of various proprietary or vendor-enhanced APIs. For example, the Servlet API and the JavaServer Pages API comprise a single, standard, well-defined way to implement web interfaces, regardless of the application server, web server, operating system, or hardware being used. Third, the Java and J2EE frameworks provide an excellent foundation on which you can layer other, nonstandard but compatible tools. Standards can cover only part of the enterprise application landscape, and there will always be valuable utilities developed by the business and developer communities that fall outside of the standards. A standardized framework provides a stable, formally managed backdrop for using these utilities without compromising the portability promises of Java and J2EE. Finally, recall that many enterprise protocols and standards were developed before the days of object-oriented programming. The object-oriented power and elegance of the Java language make the Java Enterprise APIs simpler, easier to use, and easier to understand than the non-Java APIs and protocols upon which they are layered.

Our message to you, and the central tenet of this book, is the following:

- Enterprise computing is useful to everyone.
- Any programmer can write distributed applications using the Java Enterprise APIs.

With that said, it is important to understand that distributed enterprise computing actually is somewhat more complicated than local computing. Just as using threads in a program introduces complexities that don't exist in single-threaded programs, using distributed services in a program introduces complexities that don't exist in programs that run entirely on one computer. And just as developing a rapport with one person is much easier than arriving at a consensus with a crowd, dealing with the vagaries of an entire enterprise is much more challenging than satisfying the needs of a small department in an organization. Do not fear, however: the complexities of distributed, enterprise computing really don't have to be overwhelming, and with a little study, any programmer can master them.



1.3. Standard Java Enterprise APIs

The J2EE standard framework provides support for a number of the most commonly used distributed computing technologies and services. These APIs serve as building blocks for enterprise applications. In this section, we group them into rough functional categories and discuss how they are covered in this book. In the section that follows this one, we discuss some key, but nonstandard, enterprise APIs and tools that facilitate the assembly of these enterprise components into applications. At the end of the chapter, we describe an enterprise computing scenario that illustrates how these APIs and tools can be used together to produce an enterprise application.

1.3.1. Basic Tools

Two essential technologies for enterprise application development are XML and security services.

1.3.1.1. Java API for XML Processing (JAXP)

XML is not strictly an enterprise-focused tool. XML is useful in all kinds of contexts, enterprise and otherwise. We've included it in this book, however, because XML is now such a fundamental, all-purpose tool that it touches nearly every level of enterprise application development, from basic datafile formats to interprocess communications to user interface representations.

The Java API for XML Processing (JAXP) (defined in JSR 005)^[*] is the standard API for consuming and producing XML in Java. Application servers compliant with J2EE 1.3 and later must include a compliant JAXP implementation. JAXP provides a pluggable API that supports both SAX and DOM parsing of XML content, as well as XSLT-based transformations of XML. JAXP 1.2, the version specified in the J2EE 1.4 specification, supports the SAX 2 and DOM 2 parsing APIs, as well as the XSLT 1.0 transform API. JAXP is covered briefly in [Chapter 7](#). The JAXP APIs are also covered in *Java in a Nutshell* by David Flanagan (O'Reilly).

[*] Java standards, called Java Specification Requests or JSRs, are defined through the Java Community Process. A complete list of JSRs can be found at <http://www.jcp.org/en/jsr/all>.

1.3.1.2. J2EE security

Security is of paramount importance in enterprises today and is entrenched in many aspects of application development and deployment. In fact, security is a concern that must be considered in all aspects of the modern enterprise, well beyond the J2EE security APIs discussed in this book.

The J2EE security chapter ([Chapter 10](#)) provides a brief overview of fundamental security concepts such as authentication and authorization. Authentication refers to the process of a user identifying himself to an application, typically using a username and password. Authorization refers to granting

the authenticated user access to the resources within the application, as appropriate to that user.

[Chapter 10](#) explains security in the web tier as well as the Enterprise JavaBeans (EJB) tier with several examples. It elaborates on the important concept of identity propagation across tiers as well as within the EJB tier.

The chapter covers programmatic as well as declarative security approaches. In programmatic security, you program the security rules in the code, while in declarative security you specify the rules in the deployment descriptors and let the application server enforce the rules. The chapter explains the ways in which programmatic and declarative security in Java and J2EE can be used together to build effective security into your applications.

[Chapter 10](#) also discusses single sign-on, a technique for sharing authentication credentials across various applications.

1.3.2. Web UIs and Components

The J2EE framework supports web interfaces with the concept of *web components*. These components run within application servers and take advantage of several runtime services provided by these servers, including user session management, security services like authentication and encrypted communications, and so on. Web components are also able to access other J2EE resources and components, such as Java Database Connectivity (JDBC) `DataSources` and local and remote EJB components.

J2EE supports the development of web components and web UIs with a family of three standards: Java servlets, JavaServer Pages (JSPs) and, more recently, JavaServer Faces (JSF). Each of these supports different programming models and runtime models, and each is important in different ways and in different contexts. Struts, a nonstandard but popular web interface technology, is described in a later section.

1.3.2.1. Java servlets: Basic web-enabled components

A *servlet* is a Java object that runs within a server to provide a service to a client. The name "servlet" was originally meant to be a takeoff on "applet"—a servlet can be thought of, roughly, as a server-side applet. The Java Servlet API (JSR 053 and JSR 154) provides a generic mechanism for extending the functionality of any kind of server that uses a protocol based on requests and responses.

For the most part, servlets are used behind web servers for dynamic generation of HTML content. But more generally, a servlet is a Java object that can receive HTTP requests and provide HTTP responses. The most typical scenario is a web browser that issues a request to a web server and a Java servlet that receives the request, performs some operations (perhaps JDBC calls to retrieve information or a call to an EJB component to invoke some business logic), and sends an HTML response back to the browser. But servlets can also serve in other ways in an enterprise application. A remote application can issue an HTTP request to a servlet, and the servlet can respond with an XML document in response, for example.

A key advantage of servlets over some other similar technologies, such as Microsoft Active Server Pages, is that servlets are portable among operating systems and among application servers. The advantage of servlets over some script-based web technologies, like Perl CGIs, is that servlets are

also compiled objects that are persistent between invocations, which gives them major performance benefits over parsed scripts. Servlets also have full access to the rest of the Java platform, so features such as database access are automatically supported.

The Servlet API is a standard extension to the Java platform and is part of the J2EE specifications. The API is implemented in the `javax.servlet` and `javax.servlet.http` packages. The `javax.servlet` package defines classes that represent generic client requests and server responses, while the `javax.servlet.http` package provides specific support for the HTTP protocol, including classes for tracking multiple client requests that are all part of a single client session. See [Chapter 3](#) for details on servlet programming.

1.3.2.2. JavaServer Pages (JSPs): Dynamic web pages

JavaServer Pages (JSPs) (JSR 053 and JSR 152) are also part of the J2EE specifications family and closely related to Java servlets. You can think of JSPs as an alternative approach to creating servlets, in one sense. JSPs are similar to alternative technologies such as PHP and Microsoft Active Server Pages they all provide a way to insert dynamic elements directly into HTML pages. In the case of JSPs these dynamic elements invoke Java code using references and calls to Java beans, using custom tags that act as dynamic macros that are implemented by Java beans, or using raw Java code snippets. [Chapter 4](#) provides details on writing JSPs. In addition, this chapter provides some details on using the Java Standard Tag Library (JSTL), a standard set of handy JSP tags that can be used in any compliant JSP engine.

1.3.2.3. JavaServer Faces (JSF): Web UI components

JavaServer Faces (JSR 127) is a recently released Java standard for user interface (UI) components. JSF builds on the foundation provided by Java servlets and JSPs, providing a UI component model based on JSP tags. This UI component model allows you to create UIs for your application from a set of predefined components (similar to the role that Java Swing plays in the world of desktop GUIs). The JSF model also includes state management and page navigation facilities that help in the development of applications using customary design patterns like model-view-controller (MVC).

See [Chapter 5](#) for a short tutorial on JSF, its tag libraries, and application configuration schemes.

1.3.3. Accessing Information Systems

Access to databases and distributed systems is essential for enterprise applications. APIs in this area include JDBC and Java Naming and Directory Interface (JNDI).

1.3.3.1. JDBC: Working with relational databases

JDBC (JSR 054) is the Java API for working with relational database systems. JDBC allows a Java program to send SQL query and update statements to a database server and to retrieve and iterate through query results returned by the server. JDBC also allows you to get metadata about the

database and its tables from the database server.

The JDBC API is independent of vendor-specific APIs defined by particular database systems. The JDBC architecture relies upon a `Driver` class that hides the details of communicating with a database server. Each database server product requires a custom `Driver` implementation to allow Java programs to communicate with it. Major database vendors have made JDBC drivers available for their products. In addition, a "bridge" driver exists to enable Java programs to communicate with databases through existing ODBC drivers.

The JDBC API is found in the `java.sql` package, which was first introduced in Java 1.1. Java 1.2 updated the core APIs to include JDBC 2.0, which added a number of new classes to support advanced database features. JDBC 2.0 also provides additional features in the `javax.sql` standard extension package. JDBC includes classes for extracting data from databases in the form of Java objects, for pooling database connections, and for obtaining database connection information from a JNDI name service. The extension package also supports scrollable result sets, batch updates, and the storage of Java objects in databases. The latest versions of the J2SE, 1.4 and 5.0, include the JDBC 3.0 API, whose specification was released in February 2002. This version of JDBC adds support for transaction "savepoints," allowing you to programmatically roll back database transactions to multiple checkpoints, as well as some enhancements to connection pooling support. At this writing, however, driver support for JDBC 3.0 is still somewhat limited, and in several cases you may have only JDBC 2.x API support in your drivers.

The JDBC API is simple and well designed. Programmers who are familiar with SQL and database programming in general should find working with databases in Java very easy. See [Chapter 8](#) for a tutorial on JDBC and [Appendix D](#) for a quick reference to SQL.

1.3.3.2. Java Naming and Directory Interface (JNDI): Working with directory services

JNDI is the Java Enterprise API for working with networked naming and directory services. It allows Java programs to use name servers and directory servers to look up objects or data by name and search for objects or data according to a set of specified attribute values. JNDI is implemented in the `javax.naming` package and its subpackages as a core API in the Java platform.

The JNDI API is not specific to any particular name or directory server protocol. Instead, it is a generic API that is general enough to work with any name or directory server. To support a particular protocol, plug a service provider for that protocol into a JNDI installation. Service providers have been implemented for the most common protocols, such as LDAP, Active Directory, and Novell's NDS. Service providers have also been written to interact with the RMI and CORBA object registries.

JNDI also plays a key role in the J2EE framework since it is the API used to access runtime configuration information and resources by J2EE applications. Information specified in component deployment descriptors is made available to components by the application server through an internal name server, and the application components access this name server through JNDI.

JNDI is covered in detail in [Chapter 9](#).

1.3.3.3. Java Transaction API (JTA): Accessing transactional resources

The JTA, or Java Transaction API, is a Java Enterprise API for managing distributed transactions. Distributed transactions are one of the things that make distributed systems more complicated than nondistributed programs. To understand distributed transactions, you must first understand simple, nondistributed transactions.

A transaction is a group of several operations that must behave atomically, as if they constitute a single, indivisible operation. Consider a banking application that allows a user to transfer money from a checking account to a savings account. If the two account balances are stored in a database, the application must perform two database updates to handle a transfer: it must subtract money from the checking account and add money to the savings account. These two operations must behave atomically. To see why, imagine what would happen if the database server crashed after money had been subtracted from the checking account but before it had been added to the savings account. The customer would lose money!

The JTA defines a Java binding for the standard XA API for distributed transactions (XA is a standard defined by the Open Group). Using the JTA, we can write a program that communicates with a distributed transaction service and uses that service to coordinate a distributed transaction that involves, for example, a transfer of money between database records in two different databases.

[Chapter 16](#) provides a brief general tutorial on the JTA. Since EJBs and EJB containers provide direct support for distributed transactions using JTA, there is also brief coverage of the API (as it relates to EJB) in [Chapter 6](#). JDBC drivers can also optionally support JTA-enabled distributed transactions, so JTA (as it relates to JDBC) is also briefly mentioned in [Chapter 8](#).

1.3.4. Distributed Computing

Support for distributed computing is key to building enterprise applications. In this area, you'll find APIs for web services, remote method invocation (RMI), object request brokers (like CORBA), and, as but far from least, Enterprise JavaBeans, a component architecture that brings it all together.

1.3.4.1. JAX-RPC and SAAJ: Using and writing web services

Web services have created quite a stir in the enterprise development domain by promising a new idiom for writing and using distributed services. Web services refer to services that are callable using an XML-based protocol and transmitted over HTTP, SMTP, and other communication protocols. Web services are exciting because, like CORBA, they provide a platform-independent approach to distributed computing; since they depend on ubiquitous formats and protocols (such as XML and HTTP), the endpoints in web service architectures can be implemented in any platform or language that supports these protocols. In other words, web services can be produced and consumed by nearly all modern programming languages and on all platforms.

But web services have some additional advantages that RMI and CORBA do not. Since the overall distributed computing architecture is generic and extensible, web services provide much more flexibility in terms of the physical implementation of the web services and their surrounding runtime support. A web service engine can be a full-blown application server with failover and load-balancing or a tiny runtime process on a handheld device. Programming models, APIs, object models, and the like are left up to the implementation platform, allowing them to be optimized for the particular environment.

The fundamental standard protocols used in web services today are SOAP (Simple Object Access Protocol) for interprocess communications and WSDL (Web Services Descriptor Language) for describing web services. In the Java community, a suite of standard Java APIs have been defined to help developers build and use web services. We discuss the two most important ones in this book: the Java API for XML Remote Procedure Calls (JAX-RPC) (JSR 101, JSR 224) and the SOAP with Attachments API for Java (SAAJ). SAAJ defines a standard API for basic, low-level messaging using SOAP. On top of this, JAX-RPC provides a standard API for interacting with web services using a remote method invocation paradigm in which a request to a web service is expected to result in an immediate response. Both of these APIs are covered in the web services tutorial in [Chapter 12](#).

1.3.4.2. Java RMI: Java-to-Java remote objects

Remote method invocation (RMI) is a programming model that provides a high-level, generic approach to distributed computing. RMI extends the Java object-oriented programming paradigm to distributed client/server programming: it allows a client to communicate with a server by invoking methods on remote objects that reside on the server. RMI is implemented in the `java.rmi` package and its subpackages, which were introduced in Java 1.1 and were enhanced in later versions of the Java platform.

The Java RMI implementation is full-featured but still simple and easy to use. It gains much of its simplicity by requiring both client and server to be implemented in Java. This requirement ensures that both client and server share a common set of data types and have access to the object serialization and deserialization features of the `java.io` package, for example. On the other hand, this means that it is more difficult to use RMI with distributed objects written in languages other than Java, such as objects that exist on legacy servers. The default remote method communication protocol used by RMI will allow only Java code to interact with RMI objects. But you can also opt to use RMI/IIOP, which was made a standard part of the core Java APIs in Version 1.3 of the Java platform. RMI/IIOP is an optional communication protocol that allows RMI objects to interact with CORBA-based remote objects. Since CORBA objects can be implemented in many languages, this provides a bridge to systems implemented in other languages.

The `java.rmi` package makes it easy to create networked, object-oriented programs. Programmers who have spent time writing networked applications using lower-level protocols are usually amazed by the power of RMI. See [Chapter 13](#) for a tutorial on using RMI.

1.3.4.3. Java IDL: Using and writing CORBA remote objects

As we've just discussed, RMI is a distributed object solution that works especially well when both client and server are written in Java. It is more work, and therefore less attractive, in heterogeneous environments in which clients and servers may be written in arbitrary languages. For environments like these, the Java platform includes a CORBA-based solution for remote method invocation on distributed objects.

CORBA (Common Object Request Broker Architecture) is a widely used standard defined by the Object Management Group (OMG). The Java binding of this standard is implemented as a core part of the Java platform in the `org.omg.CORBA` package and its subpackages. The implementation includes a simple Object Request Broker (ORB) that a Java application can use to communicate (as both a client

and a server) with other ORBs, and thus with other CORBA objects.

The interfaces to remote CORBA objects are described in a platform- and language-independent way with the Interface Definition Language (IDL). Sun provides an IDL compiler that translates an IDL declaration of a remote interface into the Java stub classes needed for implementing the IDL interface in Java or for connecting as a client to a remote implementation of the interface from your Java code.

A number of Java implementations of the CORBA standard are available from various vendors. This book documents Sun's implementation, known as Java IDL. It is covered in detail in [Chapter 14](#). The syntax of the IDL language itself is summarized in [Appendix G](#), and the CORBA-related tools provided with Sun's JVM are described in [Appendix H](#).

1.3.4.4. Enterprise JavaBeans: Distributed business components

Enterprise JavaBeans (EJB) do for server-side enterprise programs what Java beans do for client-side GUIs. EJB is a component model for units of business logic and business data. Programming models that take business logic out of the client and put it on a server or in a middle tier have many advantages in enterprise applications. However, the task of writing this middleware has always been complicated by the fact that business logic must be mixed in with code for handling transactions, security, networking, and so on.

The EJB model attempts to separate high-level business logic from low-level housekeeping chores. And in the process, it attempts to leave the former to you, the enterprise developer, while the latter is delegated to the EJB server. A component in the EJB model is an object that implements business logic or represents business data. The difference between an enterprise bean and a run-of-the-mill RMI remote object or nonremote Java bean is that EJB components run within an EJB container, which in turn runs within an EJB server. The container and server provide features such as transaction management, resource pooling, lifecycle management, security, name services, distribution services, and so on. With all these services provided by the container and server, enterprise beans (and enterprise bean programmers) are free to focus purely on business logic.

The EJB specification is a document that specifies the contracts to be maintained and conventions to be followed by EJB servers, containers, and beans. Writing EJB components is straightforward: simply write code to implement your business logic, taking care to follow the rules and conventions imposed by the EJB model.

EJB components can also run within the larger J2EE framework. In addition to the standalone EJB component services, EJBs running within a J2EE server can be assembled with other components (like web components) into J2EE applications.

Unlike some of the other Java Enterprise APIs, the EJB API is just the beginning of understanding how EJB works. The key to understanding Enterprise JavaBeans lies in the interactions among beans, containers, and the EJB server. These interactions are described in detail in [Chapter 6](#). In addition, the syntax for the abstract persistence language used in entity EJBs, EJB QL, is described in [Appendix C](#).

1.3.5. Messaging

Messaging provides an essential link for enterprise applications, and Java's support for messaging includes Java Message Service (JMS) and JavaMail for email-based messaging.

1.3.5.1. Java Message Service (JMS): General enterprise messaging

JMS (JSR 914) is the Java Enterprise API for working with networked messaging services and for writing message-oriented middleware (fondly referred to as MOM).

The word "message" means different things in different contexts. In the context of JMS, a message is a chunk of data that is sent from one system to another in an asynchronous manner. The data serves as a kind of event notification and is almost always intended to be read by a computer program, not by a human. In a nondistributed system that uses the standard Java event model, an `Event` object notifies the program that some important event (such as the user clicking a mouse button) has occurred. In a distributed system, a message serves a similar purpose: it notifies some part of the distributed system that an interesting event has occurred. You can think of a networked message service as a distributed event notification system.

JMS is also a good complement to the synchronous communication provided by RMI, CORBA, and most web services built using JAX-RPC. When an RMI client, for example, makes a remote method call on a server object, the client will block until the remote method returns. JMS provides a way for you to communicate asynchronously with a remote process: you can send your message and carry on with useful work while the message is delivered and processed at the receiving end. If there's a response from the receiver(s), a callback can be invoked on your end, and you can deal with it then.

Like JNDI and JDBC, JMS is an API layered on top of existing, vendor-specific messaging services. In order to use JMS in your application, you need to obtain a JMS provider implementation that supports your particular message server. Some J2EE application servers bundle their own JMS providers that you can use as message servers; some of them provide easy ways to bridge their application servers to other message services like IBM MQSeries or SonicMQ; others provide neither and leave it to you to obtain and install a JMS provider.

[Chapter 11](#) provides a tutorial on using JMS, and [Appendix E](#) documents the syntax for JMS message selectors.

1.3.5.2. JavaMail: Email-based messaging

Email is another critical communication protocol for enterprise systems. Email is a widespread tool used for interpersonal messaging in a wide variety of contexts, from corporate communications to family reunion planning, as anyone reading this book is surely aware. In an enterprise application context, email can also be an important tool for end user event notifications (for example, a notice that a lower-priced flight has matched your travel profile), for content delivery (e.g., a weekly "ezine delivered automatically from a content management system), and for system monitoring (e.g., a notice to a system administrator that connectivity to a critical information system has been lost).

J2EE's tool for composing, sending, and receiving email is the JavaMail API, contained in the `javax.mail` package. For dealing with various types of content in MIME-based email messages, a companion API called the JavaBeans Activation Framework and provided in the `javax.activation` package is used in conjunction with JavaMail ("activation" in the name refers to activating a content

handler to deal with a particular type of content). [Chapter 15](#) is a tutorial on the use of these APIs.



1.4. De Facto Standard Enterprise Development Tools

The Java community members involved in defining the J2EE platform and its supporting standard enterprise APIs would be the first to admit that it was never their intent to create standards for every element of enterprise development. If there were only one right way to do everything, there wouldn't be much room for creativity (or profit, for that matter). J2EE covers a critical mass of the capabilities needed when building and running enterprise applications. It standardizes those things that are required in order to make enterprise applications portable across application servers and support systems, to allow vendors to integrate their services into the Java environment in a standard way and to prevent enterprise developers from wasting their valuable time re-creating utilities that really should be plumbing, enabling them to focus their efforts on the business logic and user interfaces of their systems.

In addition to the standard Java Enterprise APIs discussed earlier, an enterprise developer typically finds herself turning to a set of other support tools and extension APIs in order to be efficient and effective in building enterprise systems. These tools and APIs fill the gaps left by the standard enterprise Java APIs. Sometimes these gaps are intentional (e.g., "it doesn't make sense to standardize an API for X"). Other times the gaps are temporary, because a standard hasn't been defined yet for a particular area.

In the second part of this book, we provide tutorials on popular, effective tools and APIs that are typically used when building enterprise Java systems. Choosing the tools to include in this section was difficult in many cases, because so many viable, effective options are available in some areas. But we chose tools and APIs based on the size of their respective user communities, the effectiveness of their solutions, and the expected longevity of the tools and APIs relative to other options.

Readers will quickly realize that all of the tools chosen for this section of the book are open source tools (see <http://opensource.org>). That was an intentional decision on our part, for a few reasons. In nearly every functional area covered here, a family of solutions is available, with some commercial and some open source. The right tool for you depends on the context of your enterprise development scenario, but we wanted to give you practical exposure to the issues in each area, a sense of why extension tools exist in the first place, and what makes the tool effective and useful. In order to do that, we needed to choose tools that were accessible to all readers and that represented proven, effective solutions for the areas we cover here. For these reasons, open source tools were the obvious approach. While many good, effective commercial tools exist in virtually all of these areas, they would not necessarily be accessible to all readers to experiment with the examples provided here. We also realize that commercial tools are a necessary and effective part of many development environments that readers need to use. To that end, as we cover each of the tools included here, we've tried to highlight general concepts in each area that will help map the characteristics found in these open source tools back to their commercial counterparts and hopefully help you, in an indirect way, to understand and manage those tools.

1.4.1. Ant: Building and Deploying Applications

Ant is used to manage the process of building and deploying code, among other things. Ant is similar in purpose to age-old tools like `make`, `gmake`, and `imake`. It provides a way to codify the parameters needed to build your code (like dependent libraries, configuration files, and the like) and to define various tasks that can be done with the code (compile it, generate Javadoc pages from it, assemble it into a J2EE application archive, deploy it to an application server, and run unit tests on it, among other things).

Ant is an open source project managed by the Apache Software Foundation. It "ships" with a large set of core tasks that can be used to compose Ant "build scripts," which are written in XML. An Ant build script consists of a set of targets that you define. Each target consists of a set of tasks that are executed when that target is requested by the user or invoked by another target. These tasks can be core tasks included with Ant (like "compile this set of Java code" or "copy these files to that directory"), they can be custom tasks that you define, or they can be tasks that are imported from a third-party library. Many J2EE application servers and tools (including many of the open source tools discussed in this book) now include their own Ant tasks, to allow you to easily integrate them into your project Ant scripts.

Many Ant tasks and practices have been defined to help develop and manage large enterprise projects in the Java environment. [Chapter 17](#) provides both an overview tutorial of Ant in general and some details on Ant-related utilities that are particularly useful when developing J2EE applications. The chapter also includes some best practices in terms of designing Ant build processes for multiple developers and multiple environments.

1.4.2. JUnit and Cactus: Testing Your Objects, Components, and Applications

In any application development context, the practice of testing your code is a critical success factor. Testing enterprise applications involves a number of different dimensions. The user experience and the proper behavior of views (in the model-view-controller, or MVC, sense) can be tested using functional testing tools; the correct behavior of software at the code level, in terms of objects and components performing as expected, can be tested using unit testing tools; the throughput and overall behavior of a system when under heavy load (large numbers of users or transactions or both) can be tested using performance testing tools.

JUnit is an open source unit testing framework for Java. It is one in a series of such tools, built for a number of development environments using the same conceptual architecture. Other tools in this suite include PerlUnit for Perl and CppUnit for C++. JUnit includes a Java API that provides interfaces and base classes for defining and running unit tests as well as some tools that facilitate the configuration, running, and reporting of unit test suites. JUnit is used by developers to write suites of unit tests that exercise their code in critical ways and that verify the results to ensure that they are correct, according to the documented behavior of the code under test. These unit tests are themselves Java code, making it easy to use the same code management tools to manage tests for code along with the code itself.

Enterprise developers face unique challenges when unit testing their Java objects and components. The proper behavior of enterprise code can be tested only if a suitable simulation of its runtime environment can be achieved. A web component like a JSP tag handler, for example, can be tested only if the testing framework can operate within a web container, can issue simulated handle requests to the tag handler, and can interpret the responses that are generated to assess whether

the test succeeded or not.

To facilitate the task of enterprise unit testing, the good people at Apache extended JUnit with a framework called Cactus. Cactus allows J2EE developers to write unit tests for full enterprise components, like servlets and EJBs.

[Chapter 18](#) provides an introduction to the basic JUnit testing framework and also gives a tutorial on using Cactus to define and execute test suites for enterprise components.

1.4.3. Struts: Implementing Model-View-Controller Systems

The MVC paradigm for architecting UI-oriented applications was introduced more than 20 years ago in the Smalltalk environment. Since then, it's become a popular and effective pattern for designing and managing web applications, and many tools have been created to support it.

In the MVC pattern, the model represents the data and business logic at the heart of the application, views represent interactive user interface elements, and the controller links the views to the model and keeps the application flow moving. In an MVC-based application, users interact with one or more views in the user interface. Their actions are passed to the controller, which is responsible for handling them. Based on the user's actions, the controller can make changes to the application model, generate new views for the user, or all of the above. Changes in the application model can also generate notifications directly to views, causing them to change appearance in key ways or adjust their behavior, based on the new state of the model.

Java servlets and JavaServer Pages provide the essential tools for building web interfaces in Java, using an object-oriented programming model or a "scripted" web page model, respectively. Struts is an Apache project built on top of both of these standard Java technologies, using servlets to implement the controller, both JSPs and servlets to implement views, and standard Java technologies (such as Java beans and EJBs) to implement the application model. Struts further facilitates MVC development by providing a configuration scheme for specifying the page flow of the application based on user actions and an API for defining action handlers and other key MVC elements.

It's important to point out that the introduction of the JavaServer Faces standard provides many of the same elements that the Struts project covers. Put another way, Struts was created to fill a perceived gap between the Java standards at the time and the needs of enterprise developers. Since then, JavaServer Faces was defined to fill a part of that same gap. JSF was released in specification form in March 2004, and implementations are now readily available. The relationship that Struts will have in the future with Java servlets, JSPs, and JSF still remains to be seen, but in the meantime, Struts remains a popular and effective tool for MVC development.

A tutorial on Struts is provided in [Chapter 19](#). Because of the functional overlap between JSF and Struts, the code examples in the two chapters parallel each other and have identical user interfaces. As a result, you can compare and contrast the implementations to help you decide which solution is right for your application.

1.4.4. Hibernate: Object Relational Mappings

Many enterprise application scenarios involve interactions with relational databases. When operating within an object-oriented environment like Java, the issue of mapping relational constructs, like tables and columns and rows, into object-oriented concepts, like objects and methods and data members, arises to some degree in any database-enabled application. JDBC provides a relatively low level interface for interacting with databases allowing you to establish connections to database engines, construct SQL statements and issue them to the database, and retrieve the results in the form of basic Java data types. The work of integrating these operations into the overall object model is left up to the developer.

The approach to managing the object relational mapping (ORM) is an important one since it can make the overall development, testing, and long-term management of your code either very easy or very difficult indeed. For example, suppose that you developed a complicated application that relies heavily on a given database with its own rich, complex schema structure. Now suppose that the database needs to be changed in some significant ways (existing tables are restructured or eliminated, new tables and columns are introduced, and stored procedures are altered). If you have mapped the relational structures of the database into your application by sprinkling JDBC calls throughout your code in an ad hoc fashion, refactoring the application will be possibly painful and definitely costly. On the other hand, if you've used a consistent object relational mapping pattern that insulates the application from the database details, the refactoring job will be relatively straightforward, take less time, and introduce less instability into the system.

Since ORM mappings play such a potentially important role in enterprise development strategies, a number of tools and approaches have emerged to fill the gap between the low-level standard API provided by JDBC and the object-level persistence mapping needs of enterprise developers. One such tool is an open source project called Hibernate, currently managed by JBoss along with their open source application server and other related projects. Hibernate provides a framework for defining mappings from relational elements to Java object-oriented elements, for generating the SQL calls implied by these mappings, and for exposing the persistence capabilities to the rest of an enterprise application using a clean Java interface that effectively isolates the persistence details from the rest of the system. This makes database evolutions easier to manage, makes the overall application more portable (to different database schemas or vendors), and makes your life as an enterprise developer much happier in the long run.

It's important to note that efforts are afoot in the Java community to define standard ORM mapping approaches. The most notable of these efforts are the EJB 3.0 specification work (JSR 220) and the Java Data Objects (JDO) standard (JSR 012). JDO came about as a standards effort after Hibernate and other ORM tools had already taken hold in the Java community, with the same conceptual goal as many other Java standardization efforts to define a portable approach to ORM mappings that tools could support independently, giving the enterprise developer the ability to choose solutions and possibly switch solutions in the future without a lot of rework. The 1.0 version of JDO took a slightly different technical approach to managing the ORM mappings than Hibernate and other tools. The JDO approach involves postprocessing of Java bytecodes to enable their persistence capabilities, using an ORM mapping to drive the generation of persistence code at the bytecode level. The JDO approach did not take significant hold in the Java community for a variety of reasons, and a 2.0 version of the JDO specification is now being developed. Meanwhile, elements of the JDO and Hibernate approach to object relational mapping have had a significant influence on the upcoming EJB 3.0 specification, which is in development. The newer versions of Hibernate implement preliminary versions of the EJB 3.0 model, and it's expected that the relationship between the two will become even closer in the future.

Since this standards area is still in a state of flux, we chose to include an established but nonstandard ORM mapping tool in this edition, to give you exposure to a tool that can be immediately applied in

enterprise contexts and to provide a basis for understanding other ORM mapping tools that you may encounter in the Java community.

[Chapter 20](#) provides a tutorial on the use of Hibernate to create ORM mapping layers for your applications.

1.4.5. XDoclet and Java Annotations: Simplifying Development with Metadata

As enterprise applications and their corresponding development practices have matured and expanded their scope, the concept of code and artifact generation has become more and more important. In some cases, it can be more efficient to specify behavior using metadata and attributes rather than explicitly specifying the lines of code needed to perform a task. It's also possible to infer configuration parameters from attributes of source code rather than having to manually map the intentions of the code to configuration details, like deployment descriptor entries. A good example of this is object relational mapping scenarios, as discussed in the previous section. In many cases, it can save you a lot of time and effort if you can simply describe a mapping from relational tables to Java elements, using annotations inserted into the Java code itself, and then let a tool generate the corresponding implementation artifacts: the Java code and SQL statements that implement the mapping described in the metadata. Another example of this concept that is familiar to any Java developer is Javadoc: tags inserted into your Java code are used by Javadoc to generate HTML documentation for your source code, saving you the trouble of writing the HTML yourself.

XDoclet and the newer Java annotations (introduced in Java 5.0) are tools to assist with attribute-oriented programming. Attribute-oriented development is aimed at leveraging this metadata-driven approach to generating code and other implementation artifacts, but it is specifically aimed at using metadata inserted into Java code to drive the generation process.

XDoclet's approach uses Javadoc extension tags to drive the process of generating source code, deployment descriptors, configuration files, or whatever other implementation artifacts are needed to deploy and run your code. The typical example used to explain XDoclet's utility involves EJB components. Rather than writing several Java interfaces, Java classes, and a deployment descriptor, you can annotate a single Java class with special XDoclet metadata tags and use XDoclet to generate all the other required EJB artifacts for you.

The newer Java annotation features built into JDK 1.5/Java 5.0 provide similar capabilities through actual Java syntax extensions instead of Javadoc comments. The concept is the same (code is annotated with metadata that is used to generate other artifacts), but the annotation feature is a standard element of the Java language itself.

[Chapter 21](#) provides a tutorial on the general metadata models used by XDoclet and Java annotations and also provides details on some specific enterprise Java use cases, such as developing EJB components and web services. We also describe the integration of XDoclet and Java annotations into Ant build scripts. This chapter is an exception in the book since it covers both standard (Java annotations) and de facto standard (XDoclet) tools for the same functionality. In this case, we felt that there were significant parallels between the two solutions and presenting them side by side in the same chapter was much more effective for providing a comparison. Struts and JSF, on the other hand, use very different programming models to achieve somewhat similar functionality, so we felt they would be better served by separate chapters.

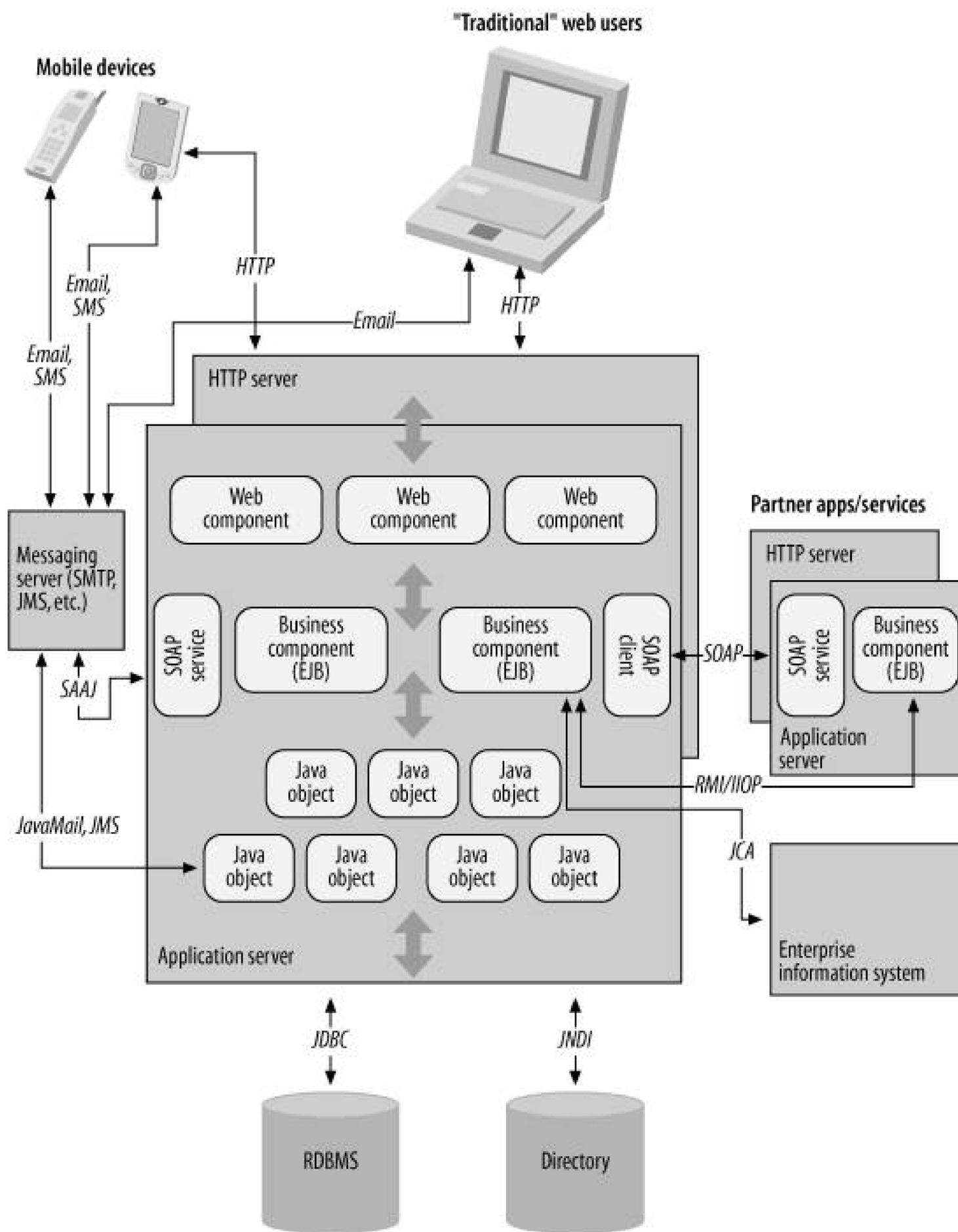


1.5. An Enterprise Computing Scenario

The previous sections have been rapid-fire introductions to the Java Enterprise APIs that are part of the J2EE framework as well as a set of key de facto standard tools used commonly in the Java Enterprise space. Don't worry if you didn't understand all the information presented there: the rest of the chapters in the following two parts of the book cover the APIs in more detail. The important message you should take from this chapter is that the Java Enterprise APIs are building blocks that work together to enable you to write distributed Java applications for enterprise computing. The computing infrastructure of every enterprise is unique, and the Java Enterprise APIs can be combined in any number of ways to meet the specific needs and goals of a particular enterprise.

[Figure 1-1](#) shows a schematic for a hypothetical enterprise. It illustrates some of the many possible interconnections among distributed services and shows some of the Java Enterprise APIs that facilitate those interconnections. The figure is followed by example scenarios that demonstrate how the Java Enterprise APIs might be used to solve typical enterprise computing problems. You may find it useful to refer to [Figure 1-1](#) while reading through the scenarios, but note that the figure doesn't illustrate the specific scenarios presented here.

Figure 1-1. The distributed computing architecture of a hypothetical enterprise



1.5.1. Enabling E-Commerce for a Mail-Order Enterprise

CornCo Inc. runs a successful catalog-based mail-order business selling fresh flavored popcorn. They want to expand into the exciting world of online sales. Here's how they might do it^[*]

[*] This example is intended to illustrate only how the Java Enterprise APIs can be used together. We have ignored efficiency

considerations, so the resulting design might not actually be practical for a large-scale e-commerce web site.

1. A customer visits the company's web site, <http://www.cornco.com>, and uses a web browser to interact with the company's web server. This allows the customer to view the company's products and make selections to purchase.
2. Before accessing protected or personalized sections of the web site, the customer must log in (authenticate) to identify himself to the system. Once logged in, he is authorized to use customer-related functions of the site and view his own data (and no one else's).
3. The web server uses a shopping cart servlet to keep track of the products the customer has chosen to buy. The HTTP protocol is itself stateless, but servlets can persist between client requests, so this shopping cart servlet can remember the customer's selections even while the customer continues to browse the web site.
4. When the customer is done browsing and is ready to purchase the selected products, the web server invokes a different checkout servlet. This servlet performs a number of important tasks, using several Java Enterprise APIs.
5. The checkout servlet uses JDBC to retrieve the list of products to be purchased (stored in a database by the shopping cart servlet).
6. Next, the servlet queries the customer for a shipping address, a billing address, and other required information and then uses JDBC again to store this information in a customer database. This database can be used, for example, by the CornCo marketing department for direct mail purposes.
7. The servlet then sends the customer's billing address and total purchase price to the billing server. This server is a legacy application, specific to CornCo, that has a nonstandard interface. Fortunately, however, the billing server exports itself as a CORBA object, so the servlet can treat the entire server as a CORBA remote object and invoke the necessary methods on it.
8. In order to ensure the very freshest product, CornCo maintains warehouses throughout the world. CornCo is a growing company, so the list of warehouses is frequently updated. The checkout servlet uses JNDI to contact a directory server and then uses the directory server to find a warehouse that is close to the customer and has the customer's requested products in stock.
9. Having located a warehouse that can fulfill the customer's order, the checkout servlet uses JMS to contact the company's enterprise messaging service. It uses this service to send the customer's order to the selected warehouse in the form of a message. This message is delivered to and queued up on the local computer at the warehouse.

1.5.2. Updating CornCo with Enterprise JavaBeans

You may have noticed a flaw in the previous scenario. The checkout servlet sends billing information to one server and then sends fulfillment information to another server. But it performs these two actions independently, without any attempt to maintain transactional integrity and make them behave atomically. In other words, if a network failure or server crash were to occur after the billing

information had been sent, but before the fulfillment information had been sent, the customer might receive a bill for popcorn that was never shipped.

The designers of the e-commerce system described in the previous section were aware of this problem, but since distributed transactions are complex and CornCo does not own a transaction management server, they simply chose to ignore it. In practice, the number of customers who would have problems would be small, and it was easier for the original programmers to let the customer service department sort out any irregularities.

But now, CornCo has hired a new vice president of information systems. She's tough as nails and likes all her i's dotted and her t's crossed. She won't stand for this sloppy state of affairs. As her first official act as VP, she buys a high-end J2EE application server and gives her e-commerce team the job of revamping the online ordering system to use it. The modified design might work like this:

- The customer interacts with the web server and the shopping cart servlet in the same way as before.
- The checkout servlet is totally rewritten. Now it is merely a frontend for an Enterprise JavaBeans component that handles the interactions with the ordering and fulfillment servers and with the marketing database. The servlet uses JNDI to look up the remote enterprise bean and then uses RMI to invoke methods on the bean (recall that all enterprise beans are RMI remote objects).
- The major functionality of the checkout servlet is moved to a new checkout bean. The bean stores customer data in the marketing database using JDBC, sends billing information to the billing server using CORBA, looks up a warehouse using JNDI, and sends shipping information to the warehouse using JMS. The bean doesn't explicitly coordinate all these activities into a distributed transaction, however. Instead, when the bean is deployed within the EJB container provided with the J2EE server, the system administrator configures the EJB component so that the server automatically wraps a distributed transaction around all of its actions. That is, when the `checkout()` method of the bean is called, it always behaves as an atomic operation.
- In order for this automatic distributed transaction management to work, another change is required in the conversion from checkout servlet to checkout bean. The checkout servlet managed all its own connections to other enterprise services, but enterprise beans don't typically do this. Instead, they rely on their server for connection management. Thus, when the checkout bean wants to connect to the marketing database or the enterprise messaging system, for example, it asks the EJB server to establish that connection for it. The server doesn't need to know what the bean does with the connection, but it does need to manage the connection if it is to perform transaction management on the connection.

1.5.3. Other Potential Directions

This hypothetical situation has hopefully highlighted some of the motivations for utilizing the enterprise APIs and tools we'll discuss in detail in the rest of the book. There are countless other scenarios we could reference to motivate exploration of other parts of the enterprise domain, such as:

1. CornCo expands to incorporate new partners whose services are accessible via SOAP services. The system now needs to include web service clients, making remote calls to these partner services.
2. Consequently, CornCo partners ask for integration points in the CornCo system to allow them to directly integrate their business processes, as well as desktop tools, into the CornCo environment. Since these partners use a variety of enterprise development platforms for their infrastructure, the development team decides to develop a suite of its own SOAP-based web services that expose the necessary integration points to these partners in a platform-neutral way.
3. Key CornCo clients have requested the ability to place and track orders from mobile devices, like mobile phones and personal digital assistants (PDAs), using simple email messages. To support these devices, inbound and outbound email capabilities need to be integrated into the system to allow these users to submit requests in the form of a formatted email and to receive email notifications of order status.

 PREVIOUS

1.6. Other Enterprise APIs

We have decided not to include several enterprise APIs in this nutshell book, for a variety of reasons. In most cases, it was due to prioritizing space for other tools that we felt were more important to a larger population of readers.

In the area of web services, the Java API for XML Registries (JAXR) provides a standard API for interacting with web service registries implemented using UDDI or ebXML, standard web service registry protocols. JAXR is not covered here, because despite the initial enthusiasm over web services and the current maturity of basic web service tools and APIs, the practice of creating web service registries remains a relatively niche concern. Large organizations creating broad service-oriented architectures (SOAs) may find effective use for UDDI and ebXML registries, but for many contexts, web services talking SOAP and described using WSDL are enough, so we've kept our coverage to JAX-RPC and SAAJ. Readers are encouraged to consult *Java Web Services* by Dave Chappell and Tyler Jewell (O'Reilly) for details on using JAXR.

The Java Management Extensions (JMX) provide a standard framework for creating distributed management and monitoring capabilities. The things being managed and monitored could be applications, devices, external services, or any critical enterprise capability. We've omitted JMX and related standards like the J2EE Management specification from the book because of their somewhat narrow domain of use. Full details on JMX can be found in *Java Management Extensions* by J. Steven Perry (O'Reilly).

Enterprise systems can be global not only in terms of server locations but also in terms of user population. Web applications, by definition, are potentially accessible worldwide, and in some cases it is important to consider the need to internationalize and localize your applications.

Internationalization refers to the ability of systems to behave properly when used with different languages and display standards. Localization refers to the ability of an application to adjust itself to local conditions, such as time zone and personal display preferences. In the interest of space, we do not discuss the topics of internationalization or localization of Java applications (enterprise or otherwise). The interested reader, however, should refer to *Java Internationalization* by Andy Deutsch and David Czarnecki, the *Java Servlet and JSP Cookbook* by Bruce W. Perry, or *Java Servlet Programming* by Jason Hunter, all from O'Reilly.

Chapter 2. Application Assembly and Deployment

In any enterprise framework, it's important to understand not just the APIs and tools that are needed to develop application code but also the model that the framework uses for assembling and deploying the components of applications. *Assembly* of an application or component is the process of collecting all of the elements required (source code, datafiles, and configuration files) into a structured format that can be deployed to the server environment. *Deploying* a component or application is the process of actually installing the runnable code in the runtime environment, which involves making the code available to the server and resolving resource references in the assembly with the resources actually available on the server.

Assembly and deployment are usually considered separate activities for several reasons. Perhaps most importantly, assembly is focused on the nature of the application itself while deployment is focused on the integration of the application with a specific runtime environment. Assembly of code, therefore, is more easily standardized within a given enterprise framework. Archive structures and configuration file formats can be specified in a generic way, including references to external resource like databases and messaging services. Deployment, on the other hand, is more context-dependent. The type and configuration of the runtime environment are going to determine how each deployment will be carried out. The assembly itself will need to be installed in the runtime environment in different ways, depending on the architecture and configuration of the server. If an application requires a specific resource (as specified in its assembly), and a suitable resource is not available in the runtime environment, it will need to be provisioned before the application can be deployed. The physical steps needed to perform these deployment tasks vary for different runtime environments although the same application assembly is being deployed to each.

J2EE is no different in this regard. As we'll see in later chapters of this book, J2EE supports various types of application components (web-based components built using the Java Servlet, JavaServer Pages, and JavaServer Faces APIs; business components built using the Enterprise JavaBeans APIs; and so on). The J2EE specifications include standard archive formats and configuration file formats for application and component assemblies. These configuration files, called *deployment descriptors*, allow you to configure both the elements that make up the components and external resources required by the components at runtime. These resources are specified based on other standard Java APIs, such as JDBC (*DataSources*) and JMS (*Destinations*).

With a J2EE component or application archive in hand, deployment is carried out in a server-dependent fashion. The resource references in the assembly will need to be resolved, in some cases requiring new resources to be provisioned in the J2EE server. The assembly also needs to be physically delivered to the server somehow, in whatever scheme the server requires. Deploying the same J2EE assembly to multiple servers simply means performing these deployment operations for each environment. Ideally, the J2EE assembly itself (i.e., the physical packaging and description of the application and components) remains the same, and the variations in the server environments are handled in the deployment steps.

In this chapter, we present the general model that J2EE supports for assembling components and

resources into full services or applications and how they are deployed to their runtime environments. As a complement to this tutorial material, [Appendix A](#) provides a reference for the key deployment descriptors defined in the J2EE family of specifications.



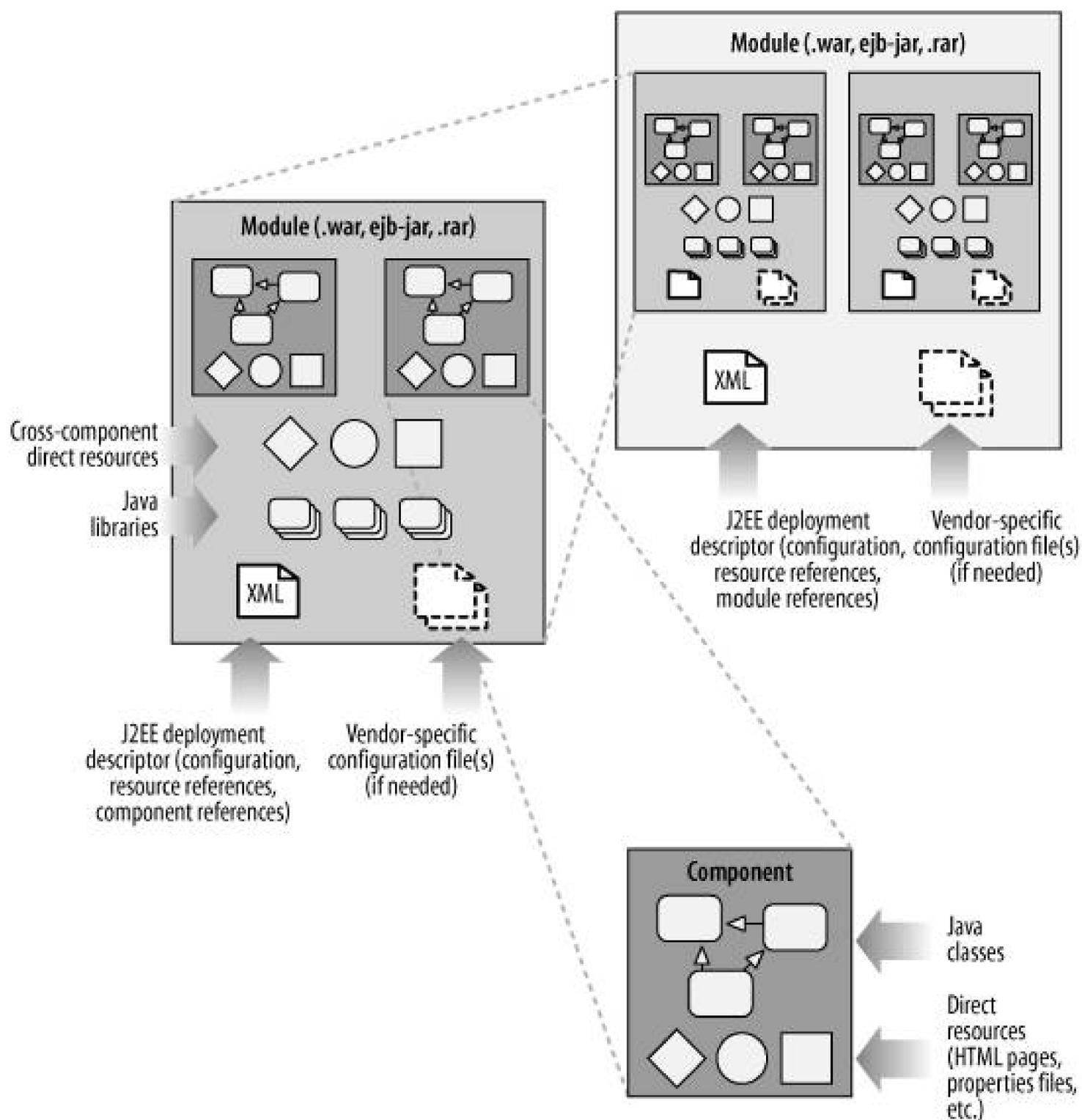
2.1. J2EE Application Assembly Model

First, let's get the J2EE terminology straight. [Figure 2-1](#) depicts the assembly model for J2EE applications. J2EE applications are composed of one or more *components*. A component is a functional element of an application that runs within a *container* and makes use of the runtime services provided by the container. These runtime services might be security management services, transaction management services, lifecycle services, or others, depending on the type of component and the type of container. A component consists of one or more Java classes that implement the component, along with any component-specific resources like properties files, images, and the like. J2EE components (and their containers) come in five flavors: *application clients*, *applets*, *web*, *EJB*, and *resource adapters*. ^[*] Each type of container is capable of providing a specific set of APIs and runtime services to its corresponding components.

[*] It's important to note that most references on this subject distinguish resource adapters as a different category of "component" in the J2EE model: a lower-level component that represents a resource to be used by the other, higher-order components, without a formal container model like the others. But for the sake of consistency of the assembly and deployment model, they are included here among the other J2EE component types, since at a general level, they are handled much the same as the other components.

J2EE components are grouped together and deployed in *modules*, which are collections of components of the same type, to be run inside the corresponding container type. A module contains all of the component-specific classes, plus any

Figure 2-1. J2EE assembly model (conceptual)



class libraries needed by the components and other cross-component resources, such as datafiles, HTML documents (for web components), and image files. Each module also contains a standard XML *deployment descriptor* that provides descriptive information, external resource references, and configuration details about each component it contains. If required, a module can also contain application server-specific details, such as configuration files, classes, and resources that are required by a particular application server. Most application servers, for example, allow you to specify additional configuration parameters, not included in the standard J2EE deployment descriptors, in separate configuration files. The application server vendor dictates the format for these files and where they should be included in the module archive. Modules are packaged in the form of jar files, with the deployment descriptor, classes, class libraries, and other resources placed in specific locations in the jar file.

An enterprise *application archive*, or *ear* file, is a set of one or more modules, and these modules collectively contain all of the components of an application. An application archive also contains a deployment descriptor, which specifies the modules that it contains and several other application-

wide configuration details. Like components, application archives are packaged into jar files with a particular layout, containing the deployment descriptor for the application, the module jar files, and any other application resources, such as documentation for the application.

In the remainder of this section, we'll talk about the general steps and concepts involved in assembling and deploying J2EE applications. Following this, we'll discuss component modules and application assemblies in more detail, using an example application as a guide.

2.1.1. Assembling Applications

As discussed in the introduction to this chapter, we've intentionally made the distinction between assembling and deploying applications because they really are two distinct stages in getting a J2EE application up and running. Assembling an application involves getting all of the elements of the application together in the right package structure and with the right configuration information included. Deploying an application is the act of taking the application assembly and plugging it into the application server environment so that it can actually run.

The first step in assembling an application is to assemble the component modules that make up the application. Once this is accomplished, you package up the component modules into an application archive.

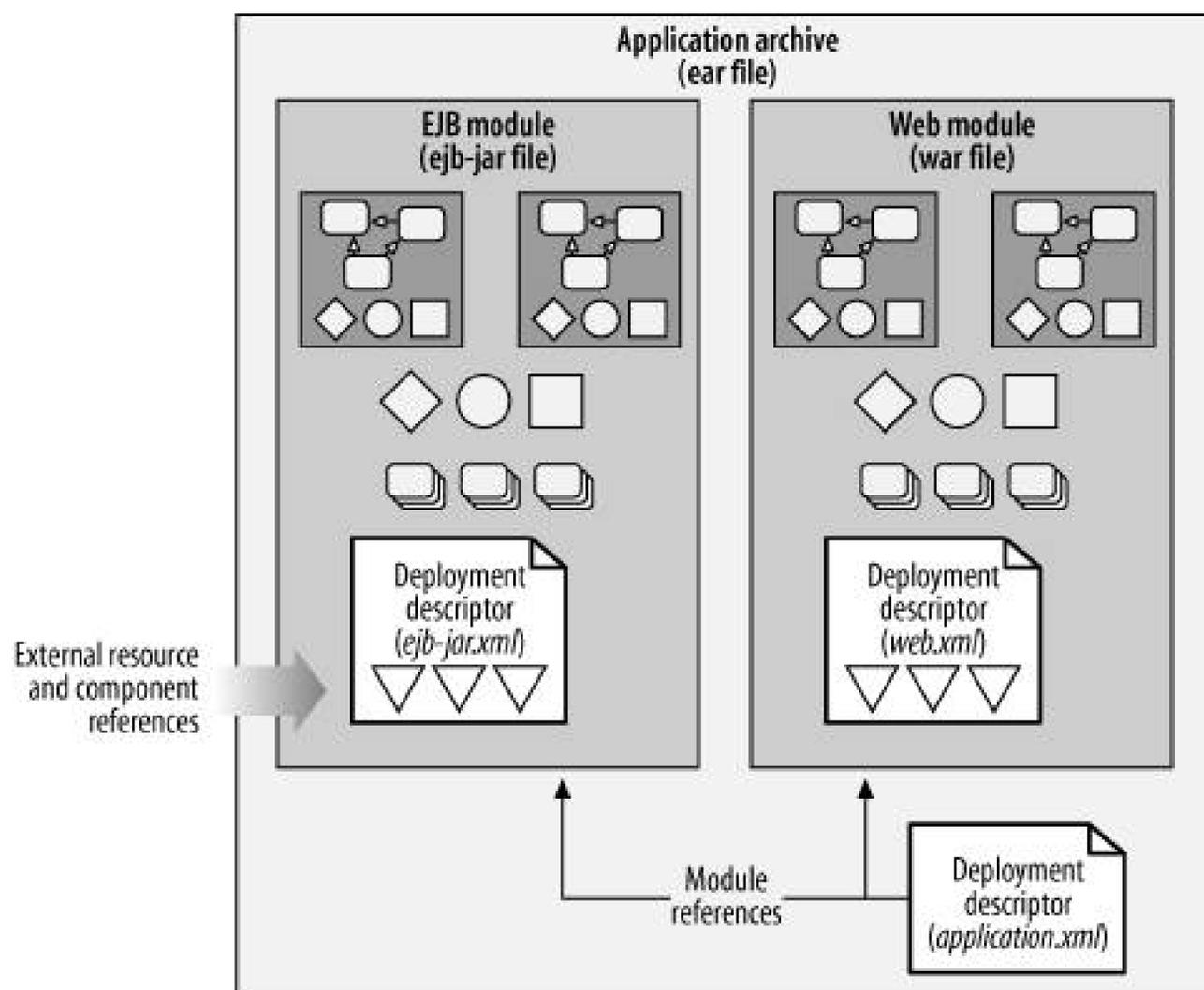
An application is fully assembled once all of its component modules have been pulled together into an application archive, as shown in [Figure 2-2](#). In the example shown, the application archive contains two component modules, a web module and an EJB module. Each component module contains a deployment descriptor describing the components themselves, how they should be managed by the container at runtime, and references to any resources or other components that the module's components require in order to operate correctly.

The application assembly itself also includes an *application.xml* deployment descriptor, which references all of the modules that make up the application.

At this point, the application is assembled but not deployed. All of the resource references in the component module descriptors are dangling, unresolved—we can't assign actual resources to these references until we put the application into a physical server environment. It is possible to resolve some internal component references at this point, if they reside in the same application assembly. For example, if one of the component modules in [Figure 2-2](#) references an EJB component and that EJB component is contained in the other component module, we can specify that internal reference directly in the deployment descriptor of the referring component. But for other resources, like JDBC `DataSources` and JavaMail `Sessions`, we won't be able to connect the deployment descriptor references to actual resources until we know the application server into which we're deploying and how its resources are configured internally.

This separation of assembly and deployment of components and applications, using deployment descriptors as the mediator between these two stages, helps you

Figure 2-2. Assembled application archive



to develop enterprise applications that migrate easily across environments. You may be writing a shopping cart application, for example, with an EJB component that requires a database connection in order to store and update order items. You might be developing this application in a development environment using a MySQL database but need to run it in a production environment using an Oracle database. The database requirement can be specified clearly as a reference to a `DataSource` in the EJB deployment descriptor when the application is assembled. When the application is deployed to the development environment, the `DataSource` reference can be bound to a `DataSource` configured against the MySQL database. When the application is eventually deployed to production, the same application assembly can be used, but the resource reference will now be bound to a `DataSource` configured to connect to the production Oracle database.

2.1.2. JNDI Configuration Services

Given all this, the configuration services provided by an application server are a critical element in deploying J2EE applications. Various runtime assets required by your components, such as database connections, JMS message queues, JCA resource adapters, or even other J2EE components, will be provided by the application server at runtime. These assets will be located at runtime using JNDI lookups, so it's critical that all of the assets referenced in the component deployment descriptors are configured properly in the JNDI namespace of the application server.

More complete details on JNDI and its use with general directory services can be found in [Chapter 9](#). Here, we'll discuss the naming convention typically used for different resources and components published in an application server's JNDI service and how names used in module deployment

descriptors are resolved against the real names of these resources and components.

2.1.2.1. Naming conventions

J2EE specifies some basic naming conventions for the assets published in the server's JNDI service for application components. In general, runtime assets are given names within the *java:comp/env* namespace in the JNDI service. The *java:comp* portion of the namespace indicates that the assets are targeted for J2EE components, and the */env* branch indicates assets that are part of the component runtime environment. Some application-wide assets, such as CORBA ORBs, are stored directly in the *java:comp* namespace. [Table 2-1](#) shows the various assets that can be stored in a JNDI service on behalf of J2EE components and the conventional namespace that is used for the names of these assets.

Table 2-1. JNDI naming conventions for J2EE runtime assets

Asset category	Java types supported	JNDI name/namespace
Database connections	<code>javax.sql.DataSource</code>	<i>java:comp/env/jdbc/*</i>
	<code>javax.jms.Queue</code>	
General messaging connections and destinations	<code>javax.jms.QueueConnectionFactory</code>	<i>java:comp/env/jms/*</i>
	<code>javax.jms.Topic</code>	
	<code>javax.jms.TopicConnectionFactory</code>	
Email messaging connections	<code>javax.mail.Session</code>	<i>java:comp/env/mail/*</i>
HTTP URL connections	<code>java.net.URL</code>	<i>java:comp/env/url/*</i>
Transactions	<code>javax.transaction.UserTransaction</code>	<i>java:comp/UserTransaction</i>
CORBA object request broker	<code>org.omg.CORBA.ORB</code>	<i>java:comp/ORB</i>
Resource adapter connections	<code>javax.resource.cci.ConnectionFactory</code>	<i>java:comp/env/eis/*</i>
	<code>javax.ejb.EJBHome</code>	<i>java:comp/env/ejb/*</i>
EJB components	<code>javax.ejb.EJBLocalHome</code>	

Asset category	Java types supported	JNDI name/namespace
General environment entries	Java basic data types (<code>Boolean</code> , <code>Byte</code> , <code>Character</code> , <code>Double</code> , <code>Float</code> , <code>Integer</code> , <code>Long</code> , <code>Short</code> , <code>String</code>)	<i>java:comp/env/*</i>

These naming conventions require coordination among the component code, the deployment descriptors, and the application server. In the component source code, you perform JNDI lookups to access runtime assets. You might access a JDBC `DataSource` from a component like this:

```
Context context = new InitialContext( );
DataSource ds = (DataSource)context.lookup("java:comp/env/jdbc/myDS");
```

We're assuming here that a `DataSource` is available under the name *myDS* within the standard J2EE namespace for JDBC resources, *java:comp/env/jdbc*.

The deployment descriptor for the component making the JNDI reference must declare its dependence on this JNDI asset. Assuming that this code resides in the implementation code for an entity EJB component, the EJB deployment descriptor would also need to include a segment declaring the dependence of the component on the `DataSource`, like this:

```
. . .
<resource-ref>
  <res-ref-name>jdbc/myDS</res-ref-name>
  <res-type>javax.sql.DataSource</res-type>
  <res-auth>Container</res-auth>
</resource-ref>
. . .
```

This entry declares that the component looks for a `javax.sql.DataSource` under the name *jdbc/myDS* within the standard J2EE namespace, *java:comp/env*. The standard namespace prefix is not included in the deployment descriptor declaration; it's assumed for all asset declarations. The application server is responsible for creating the default *java:comp/env* namespace in its internal JNDI namespace.

2.1.2.2. Two-level name resolution

Many application servers support two-level name bindings for the linking of JNDI names to actual assets. This capability offers flexibility and, in some cases, efficiency when operating a number of components and applications in the same server.

Two-level name binding allows you to assign an independent name to an asset when configuring it within the application server and then link this asset name to the name specified in the component deployment descriptors. Suppose, for example, that a deployment descriptor declares a requirement for a JavaMail `Session` to be deployed on the server under the name

java:comp/env/mail/AppSession. If the application server supports two-level name bindings, we can configure a `Session` on the server under an independent name, such as *java:comp/mail/GenericSession*, and link the asset name to the name required by the component module. In effect, this linking ensures that the runtime JNDI lookup at *java:comp/env/mail/AppSession* actually resolves to the `Session` deployed under *java:comp/mail/GenericSession*. The mechanics of making this linkage is application-dependent it can be done in a server-specific configuration file, in a web administration tool, or in something else entirely. Later in this chapter, we'll see a concrete example of configuring an asset this way in JBoss.

Not only does this two-level binding feature allow you to manage your namespace more flexibly, but it can also make it possible to share assets among components and applications running in the same server. Following on to the `Session` example we just used, suppose this `Session` were configured with fairly generic settings and that other components also needed a `Session` configured similarly. We can reuse the same `Session` asset with other components by simply linking the names that these other components declare for the asset with the same `Session` that is deployed at *java:comp/mail/GenericSession*.

2.1.3. Deploying Applications

Deploying an application involves several steps, including resolving resource and component references, managing the classloader used by the application server or servers, and finally, physically deploying the application. We describe this process next.

2.1.3.1. Resolving resource and component references

Once we move our application assembly into an application server environment (i.e., deploy our application), we need to settle all of the resource and component references contained in the various deployment descriptors. In the J2EE model, application servers make resources available to applications through an internal directory, accessed at runtime using the Java Naming and Directory Interface (JNDI). This internal directory could be a full LDAP server, a CORBA Naming Service, or any other directory service supported by JNDI.

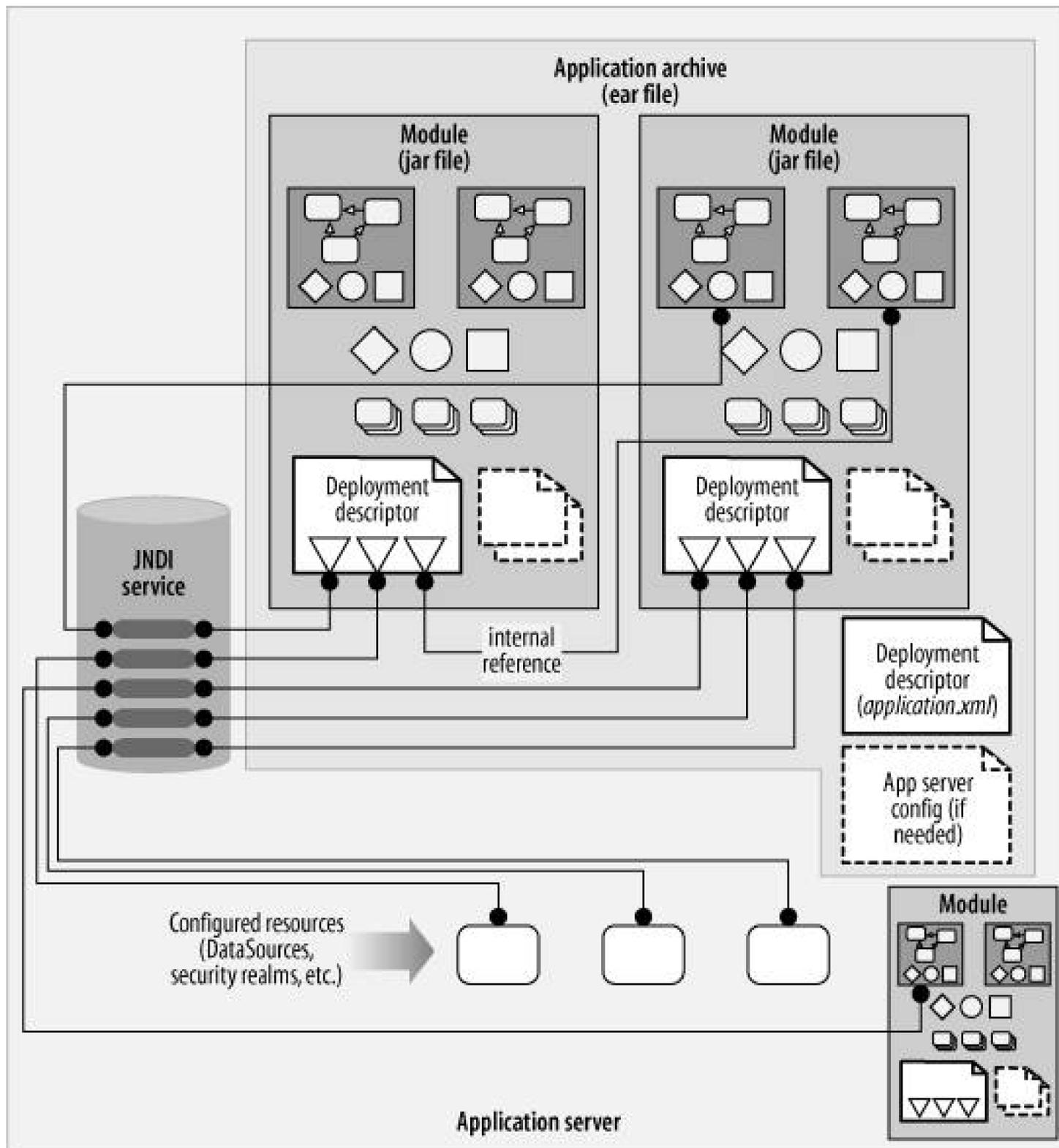
[Figure 2-3](#) shows the situation after a J2EE application assembly has been successfully deployed to an application server environment. The resource and component references in the module descriptor have been linked to deployed resources and components in the application server. These references are linked through the application server's JNDI service: components and resources have been deployed in the application server and "published" in its JNDI service under specific names. These components and resources are then linked to the references in the deployment descriptors in our deployed application, and the application is fully deployed.

If a component reference in a deployment descriptor needs to be linked to a component residing in the same application assembly, it's possible to make that link directly in the module deployment descriptor, and not through the server's JNDI service. This is depicted in [Figure 2-3](#) by the "internal reference" link between the two modules in the application. Otherwise, the resource and component references need to be resolved to real assets through the application server's JNDI service.

2.1.3.2. Runtime classloading models

Another key consideration when deploying applications is the classloading model used by the application server. Typically, a J2EE application server runs within a parent Java virtual machine, and that parent JVM may spawn other JVMs or non-Java processes to run specific runtime elements (messaging services, directory services, etc.). Within the application server's JVM, the management of classloaders can be done in a variety of ways. The most simplistic approach would be to have all applications share the same classloader. In a multiple application scenario, this is usually not desirable since various applications, or even different versions of the same application, may depend on different versions of Java

Figure 2-3. Deployed application



libraries and classes, which would cause conflicts in a single-classloader situation (the first version of a class loaded at runtime would be used by all applications in the server). If an application server runs all applications within the same classloader, then you may find yourself forced to run multiple server instances in order to run each application with the proper versions of libraries and classes.

A more realistic runtime model would be to assign a separate classloader to each deployed application. This allows each application to load its own versions of specific classes, independent of other applications. Assuming that the server is using the standard Java classloader implementation, each application classloader would be a child of the server's main classloader, so any libraries that you truly want to share across all applications (i.e., the same version used by all applications) will be loaded by the main server at startup. This is typically done by either adjusting the server's startup classpath or putting the shared libraries in a special library directory.

Some application servers impose a single classloading scheme on deployed applications, while others allow you to configure the classloading scheme at a server or an application level. It's important to understand the classloading schemes supported by your application server and to adjust these settings if needed.

2.1.3.3. Physical deployment

The physical steps that you take to deploy an application in an application server are a vendor-specific detail. The resource and component references might be resolved using server-specific configuration files, as depicted in [Figure 2-3](#), or an application server may provide a web-based administration tool that allows you to graphically link references in deployment descriptors to actual resources and components running in the application server. The documentation for your application server should explain the details.

Now that we've seen the J2EE assembly and deployment model at a conceptual level, let's run through the technical details, starting with the creation of component modules, then application assemblies, and then deployment. Along the way, we'll work with a concrete example application consisting of web and EJB components.

2.2. Component Modules

A *component module* is the smallest unit of deployment in the J2EE deployment model. A *module* is simply a jar file containing all of the bits and pieces that define a set of components. Each module contains components of the same type so that they can be easily delegated to the containers that handle them.

2.2.1. Module Jar Files

Each component module is packaged as a jar file with a specific layout for the elements that make up the components. [Table 2-2](#) summarizes the different module jar files and deployment descriptors used for the different J2EE component types:

Table 2-2. Component module format summary

Component type	Module jar file suffix	Deployment descriptor filename and location
Web	<i>.war</i> ("web archive")	<i>WEB-INF/web.xml</i>
Enterprise JavaBeans	<i>.jar</i>	<i>META-INF/ejb-jar.xml</i>
Applet	<i>.jar</i>	Not applicable
Application client	<i>.jar</i>	<i>META-INF/application-client.xml</i>
Resource adapter	<i>.rar</i> ("resource archive")	<i>META-INF/rar.xml</i>

In addition to the deployment descriptor locations specified [Table 2-2](#), each component module jar file also has a specific file layout that's required for the elements that make up the components. In a war file, for example, all individual class files included in the module must be placed in the *WEB-INF/classes* directory, all class libraries are placed in the *WEB-INF/lib* directory, and so on.

2.2.2. Deployment Descriptors

Each component module (except for applet jar files) includes a standard J2EE deployment descriptor. These descriptors are key to the assembly and deployment process in J2EE. The deployment descriptor knits together all the bits and pieces in the module jar file into actual components and describes to the container how each component should be managed. Without the deployment descriptor, the module jar just looks like a bag of Java classes and other files to the container.

Component deployment descriptors specify these major sets of information for the container:

Module descriptive information

In most cases, a component module as a whole can be given some descriptive information, such as a name, a description, and so on. This information can be used by application server management tools and server logs to provide human-readable information about the components deployed in the server.

Component elements

Each component (EJB, web, or resource connector) is made up of one or more parts. A web component is implemented as either a servlet class or a JSP file. An entity EJB is implemented as a bean implementation class, one or more client interfaces, one or more home interfaces, and a primary key class. For each component in the module jar file, the container needs to be told the parts that make up the component.

Configuration data for component services

Each type of component makes use of specific services provided by the container. Some of these services are optional; some are part of the standard runtime management provided by the container. Web components use basic instantiation and lifecycle services, optional security services, and so on. EJBs use more elaborate lifecycle services, security services, persistence services, transaction management services, and so on. Where there are options in terms of how these services are managed at runtime, the deployment descriptor allows you to specify these options.

Resource and component dependencies

A component may make use of various resources that are expected to be deployed in its runtime environment in the application server. These may include JDBC `DataSources` used to make database connections, JavaMail `Sessions` used to create and send/receive email messages, and so on. Also, components can make use of other components. A web component might make use of a session EJB to perform some key tasks on behalf of users of its interface, and the session EJB may in turn make use of an entity EJB to access persistent data. These dependencies on external resources and components need to be declared in the deployment descriptor as well.

The full format of each of the various component deployment descriptors is described in [Appendix A](#) and the details of their use are demonstrated in the examples in [Chapters 3, 4, 6, and 12](#). The following example demonstrates the general assembly process for component modules, including the use of deployment descriptors.^[*]

[*] You may also want to take a look at [Chapter 21](#), which describes XDoclet and annotations. One important use of these tools is to automate the process of generating deployment descriptors.

2.2.3. Component Module Assembly Example

Suppose we have a web application that maintains profiles for each user, and a profile contains various preferences for the behavior of the application. The profile data is stored in a database, and an EJB component is used to represent each profile within the application. The application also includes a JSP web component that provides an editor for profiles, where each user can specify his preferences. The JSP makes use of the profile EJB to access the profile data stored in the database.

Given this architecture, our application needs a web component module to deploy the JSP and an EJB module to deploy the profile EJB. Each of these modules will need its own deployment descriptor and will be packaged into a jar file to be assembled into the final application.

2.2.3.1. EJB module descriptor

Moving from the bottom up (from the database to the EJB component to the web UI), we can look at the deployment descriptor for our profile EJB component. The deployment descriptor is listed in [Figure 2-4](#). If you look through the XML descriptor, you'll notice that it includes all of the categories of information we mentioned in the previous section: module descriptive information, component element declarations, component service configuration, and resource/component references.

This module has only one EJB component, and we declare its name and the classes that make up its implementation within the `<enterprise-beans>` section of the descriptor. We also configure the container services for this component. In this case, we specify that the EJB will be using bean-managed persistence, that it will not be reentrant, and that the security management should adopt the identity of the caller when determining access rights to the component. The meaning of these service configurations is explained in [Chapter 6](#). Finally, we declare the external resources and components that the EJB component requires in order to operate. In this case, our EJB will need a JDBC `DataSource` to access the persistent profile data stored in the database. Here, we declare that our EJB code looks for the `DataSource` under the JNDI name `jdbc/ProfileDB`. This implies that, somewhere in the Java code that implements the EJB, we perform a JNDI lookup similar to the following:

```
Context ctx = new InitialContext( );
DataSource profileDB =
    (DataSource)ctx.lookup( "java:comp/env/jdbc/ProfileDB" );
```

Figure 2-4. Profile EJB deployment descriptor

```

<?xml version="1.0"?>

<!--EJB deployment descriptor for bean-managed profile EJB-->

<!DOCTYPE ejb-jar PUBLIC
    "-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans 2.0//EN"
    'http://java.sun.com/dtd/ejb-jar_2_0.dtd'>

<ejb-jar>
  <description>
    EJB components supporting the profile app
  </description>
  <display-name>Profile EJB Components</display-name>
</ejb-jar>

<enterprise-beans>
  <!--Declare the Profile EJB-->
  <entity>
    <ejb-name>BMP Profile Bean</ejb-name>
    <home>oreilly.jent.ejb.beanManaged.ProfileHome</home>
    <remote>oreilly.jent.ejb.Profile</remote>
    <local-home>
      oreilly.jent.ejb.beanManaged.ProfileLocalHome
    </local-home>
    <local>oreilly.jent.ejb.beanManaged.ProfileLocal</local>
    <ejb-class>oreilly.jent.ejb.beanManaged.ProfileBean</ejb-class>

    <persistence-type>Bean</persistence-type>

    <prim-key-class>java.lang.String</prim-key-class>

    <reentrant>False</reentrant>
    <security-identity>
      <use-caller-identity />
    </security-identity>

    <!--Declare the DataSource we require to access the profile db-->
    <resource-ref>
      <res-ref-name>jdbc/ProfileDB</res-ref-name>
      <res-type>javax.sql.DataSource</res-type>
      <res-auth>Container</res-auth>
    </resource-ref>

  </entity>
</enterprise-beans>
</ejb-jar>

```

Module description

Component elements

Component service config

Component elements

Component service config

Resource/component references

Our resource declaration in the EJB deployment descriptor instructs the container about this dependence so that the container can ensure that there is indeed a `DataSource` configured at that name in its JNDI service when we deploy the module. If there isn't, the container can stop the deployment with an error, allowing you to adjust the server configuration and redeploy the module. This is preferable to deploying an application and finding out at runtime that something is broken.

2.2.3.2. Web module descriptor

The deployment descriptor for the web component is listed in [Figure 2-5](#). This descriptor would be included in the web module jar in the *WEB-INF/web.xml* file. This sample deployment descriptor also demonstrates the four categories of information found in component descriptors. After the module description, we declare web components and the code that defines them using `<servlet>` elements. In this case, we have a single web component, our profile viewing JSP, and we declare it by giving it a name and specifying the JSP file in the module jar file that serves as the component. Following that, we specify some configuration parameters to the web container, telling it how to manage our component. In this case, we simply tell the container what URLs should be mapped to our web component and how long user sessions should last. Finally, we declare any resources that our components require to operate at runtime. As we mentioned earlier, these references are specified using the JNDI name that the component code uses to locate them at runtime, and they will be resolved to actual resources and components in the application server's JNDI service. In this case, we declare the EJB component that the JSP uses to access the profile data. We've specified that the JSP will look for the EJB home interface using the JNDI name *ejb/Profile*, which implies that somewhere in the component code we perform a JNDI lookup like this:

```
cntx = new InitialContext( );
ProfileHome profileHome =
    (ProfileHome)cntx.lookup("java:comp/env/ejb/Profile");
```

2.3. Application Assemblies

Once we have our components assembled into the appropriate modules, each with a deployment descriptor, we can assemble an application by bundling the component modules along with an application deployment descriptor into an enterprise application archive (ear) file.

Like component modules, an application archive is just a jar file with files stored in specific locations, as dictated by the J2EE specifications. The deployment descriptor is stored as *META-INF/application.xml* in the jar file, and component modules are placed in the jar and referenced in the application descriptor using their location relative to the top of the application archive. The full syntax of the application deployment descriptors can be found in [Appendix A](#).

Application deployment descriptors are simpler in structure, because they primarily serve to assemble component modules that have their own deployment information. The application deployment descriptor contains the following categories of information:

Application descriptive information

Similar to component modules, applications can also have descriptive information in their deployment descriptors. The application can be given a name and a description that can be displayed in management tools, for example.

Figure 2-5. Web module deployment descriptor

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app
  PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
  'http://java.sun.com/dtd/web-app_2_3.dtd'>

<web-app>
  <display-name>Profile app web components</display-name>
  <description>
    Web components supporting the profile app
  </description>
  <!--The profile view web component (a JSP file)-->
  <servlet>
    <servlet-name>Profile view</servlet-name>
    <jsp-file>/viewprofile.jsp</jsp-file>
  </servlet>
  <!--Map a URL (within our context root) to the JSP-->
  <servlet-mapping>
    <servlet-name>Profile view</servlet-name>
    <url-pattern>/viewprofile</url-pattern>
  </servlet-mapping>
  <!--Configure the session timeout for components in this module-->
  <session-config>
    <session-timeout>300</session-timeout>
  </session-config>
  <!--Resource declarations. All resources declared here must be
    "resolved" by the application server in order to deploy the
    application.-->
  <!--Declare EJB components that our web components use-->
  <ejb-local-ref>
    <ejb-ref-name>ejb/Profile</ejb-ref-name>
    <ejb-ref-type>Entity</ejb-ref-type>
    <local-home>
      oreilly.jent.ejb.beanManaged.ProfileLocalHome
    </local-home>
    <local>oreilly.jent.ejb.beanManaged.ProfileLocal</local>
  </ejb-local-ref>
</web-app>

```

Module description

Component element declarations

Component service config

Resource/ component references

Module references

Each component module to be included in the application must be referenced in its own `<module>` element in the *application.xml* file. The module reference indicates the location of the module jar file in the application archive. In the case of web modules, the module can also be given a web context root in the application descriptor. This context root is used as the base of URLs mapped to the components in the module.

Application security roles

Each component module can declare security roles that are used to control access to components and their operations. You can also declare security roles at the application level to indicate application-wide roles that must be mapped to actual users by the application server. Details on the J2EE security model can be found in [Chapter 10](#).

2.3.1. Application Assembly Example

Returning to our example application, we now need to create an application archive containing our application's two component module archives. One module is an EJB module that contains the profile EJB component, and the other is a web module that contains the UI elements, including the JSP used for viewing profiles.

[Figure 2-6](#) shows our application deployment descriptor. The first section of the file contains some descriptive information about the application as a whole. Following this, we declare the two module archives that contain our components. These jar files must be found in the application archive when the application is deployed at the relative locations used here in the `<module>` elements. Finally, our example descriptor declares some security roles to potentially be used by the application at runtime. As with any roles declared at the component level, the application server should verify that these roles have been configured within its runtime environment when the application is deployed.

Figure 2-6. Application deployment descriptor

```
<?xml version="1.0" encoding="UTF-8"?>  
  
<!DOCTYPE application PUBLIC  
    '-//Sun Microsystems, Inc.//DTD J2EE Application 1.3//EN'  
    'http://java.sun.com/dtd/application_1_3.dtd'
```

```
<application>
```

```
    <display-name>JEnt Profile Application</display-name>  
    <description>  
        Profile view/management application  
    </description>
```

← Application
descriptive info

```
    <!--Declare the web component module as part of the app-->  
    <module>  
        <web>  
            <web-uri>deploy-web.war</web-uri>  
            <!--Give the web compoments a URL root-->  
            <context-root>deploy</context-root>  
        </web>  
    </module>  
    <ejb>ProfileEJB module as part of the app-->  
    <module>  
        <ejb>ProfileEJB-generic-ejb.jar</ejb>  
    </module>
```

← Component
module references

```
    <security-role>  
        <role-name>admin</role-name>  
    </security-role>  
    <security-role>  
        <role-name>user</role-name>  
    </security-role>  
</application>
```

← Application
security roles

```
</application>
```

2.4. Deploying J2EE Applications

Once all of the J2EE modules have been packaged into their respective jar files, each with a deployment descriptor, and an application archive has been created with its own deployment descriptor, you still need to actually deploy the application to a J2EE application server. The deployment of an application (or even a single module) typically involves the steps described in the remainder of this section. In each case, we'll demonstrate the task using a JBoss application server and discuss how these tasks might be done in different ways in different application servers.

2.4.1. Adjust Deployment Descriptors

When you develop and package a set of components, you might imagine that the module should remain unchanged when it's deployed as part of an application. But there can be local environmental issues that force an application deployer to adjust the deployment descriptors. Some configuration parameters placed in environment entries may need to be adjusted, for example. A shopping cart web component might be delivered as part of an application with a JNDI environment entry that specifies the maximum number of line items the cart allows, with a default setting. This default value may need to be changed at deployment time to meet the needs of a particular user population (e.g., the application is being deployed for departmental administrators who require a much larger number of line items on average when ordering).

In addition to these configuration values, you may want to resolve component references directly in the module deployment descriptors. In our profile application, for example, we have a web component (our JSP for viewing profiles) that references the profile EJB component in its deployment descriptor because it makes use of it at runtime. As we mentioned earlier in this chapter, it's possible to directly resolve these references without setting up a link to a named asset in the server's JNDI service. In this case, we would alter the web module deployment descriptor to specify the component that the EJB reference should resolve to:

```

. . .
<!-- Declare EJB components that our web components use -->
<ejb-local-ref>
  <ejb-ref-name>ejb/Profile</ejb-ref-name>
  <ejb-ref-type>Entity</ejb-ref-type>
  <local-home>
    com.oreilly.jent.ejb.beanManaged.ProfileLocalHome
  </local-home>
  <local>com.oreilly.jent.ejb.beanManaged.ProfileLocal</local>
  <!-- The EJB module will be assembled as part of our app, so we
       can make a direct link to the Profile EJB using its ejb-name -->
  <ejb-link>BMP Profile Bean</ejb-link>
</ejb-local-ref>
</web-app>

```

Notice that the name used in the `<ejb-link>` element must match the value given to the EJB component in the `<ejb-name>` element in its deployment descriptor. You can verify our component name by checking [Figure 2-4](#).

Either of these changes can be made directly to the deployment modules packaged with the modules or edited deployment descriptors can be created and referred to using `<alt-dd>` elements^[*] in the application assembly deployment descriptor. If, for example, we wanted to take this approach with our edited EJB deployment descriptor, we could create the file as described before, include it in our application archive under a suitable filename (like *alt-Profile-ejb-jar.xml*), and reference it as an alternative descriptor for the EJB module in our application deployment descriptor:

[*] "alt-dd" stands for "alternative deployment descriptor."

```
<application>
. . .
<!-- Declare the EJB module as part of the app -->
<module>
  <ejb>ProfileEJB.jar</ejb>
  <alt-dd>alt-Profile-ejb-jar.xml</alt-dd>
</module>
. . .
</application>
```

2.4.2. Perform Server-Specific Configuration

As we've seen in the previous sections, each J2EE module's deployment descriptor allows you to specify the runtime configuration of the components in the module. You can do things like declare resources (JDBC `DataSources`, JMS `Destinations`, general environment variables, and so on) that the components reference in their source code. You can also declare other components that are referenced by the components in the module, like EJB components used by web components or other EJB components. But there are almost always other configuration details that need to be specified within the application server itself. If, for example, you look back at the example deployment descriptors in the previous sections, you'll notice that while resource and component references allow you to specify what JNDI name the components use to look up these things, there is no place in the various deployment descriptors to define how a component itself should be published in the server's JNDI space or to define the parameters or the JNDI name for the various types of resources used by J2EE components. In addition, there are several places in the deployment descriptors where security-related parameters (such as application roles) are referenced, but there is no place in the deployment descriptors to specify actually how these roles are satisfied by the application server (Are they mapped to users in an LDAP directory? Are they mapped to groups defined in a relational database somewhere?).

The physical configuration of all of these application runtime assets, and the resolution of these assets against the things named in the component deployment descriptors, is left up to the application server and is not specified in any J2EE specification. This gap in the specifications is intentional: application servers implement these services in many different ways depending on their internal architecture and the business model of their vendors. So there needs to be some room in thi

regard to support different server implementation models.

Looking at our example application that deals with user profiles, we have the following assets and components declared in the web and EJB deployment descriptors:

Profile EJB component

This is referenced in the web module's *web.xml*/file (Figure 2-5) using an `<ejb-local-ref>` element. The JNDI name used by the code is specified as `java:comp/env/ejb/Profile`.

JDBC DataSource

This is referenced in the EJB module's *ejb-jar.xml*/file (Figure 2-4) using a `<resource-ref>` element. The JNDI name used by the code is specified as `java:comp/env/jdbc/ProfileDB`.

Security roles

The application archive's *application.xml*/file (Figure 2-6) references two security roles used by the application, `admin` and `user`, using `<security-role>` elements.

As we've hinted at earlier in the chapter, typically you configure the required assets in the application server and link the references in the deployment descriptors to the assets configured in the application server. As we've also mentioned several times already, the mechanics of this task are server-specific. Once you've decided which application server you plan to use to deploy your application, your next step would be to read up on deployment details in the server's documentation. But to demonstrate the concepts, let's walk through these details for the JBoss application server.

Let's start with the `DataSource` reference made in the EJB module. First, we'll need to configure an actual `DataSource` in the application server that connects to the database required by the application. In JBoss, this is done using an XML configuration file that is placed in the server's deployment directory. The structure of this XML file is specific to JBoss. The following is the beginning of a JBoss configuration file that configures a `DataSource` in the server:

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

<datasources>
  <local-tx-datasource>
    <!-- The JNDI name of the DataSource. JBoss prefixes this name -->
    <!-- with "java:/" when the DataSource reference is created in -->
    <!-- the JNDI service. -->
    <jndi-name>ProfileDS</jndi-name>

    <!-- The JDBC URL used to create connections in this DataSource -->
    <connection-url>jdbc:hsqldb:hsqldb://localhost:1701</connection-url>

    <!-- The JDBC driver class required for the connections. -->
    <!-- Specifying this here allows the server to load the -->
    <!-- driver classes and verify that they are available. -->
```

```

<driver-class>org.hsqldb.jdbcDriver</driver-class>

<!-- The username and password to be used to connect -->
<user-name>sa</user-name>
<password></password>

<!-- Various other parameters, such as connection pool -->
<!-- parameters, would normally be specified next. -->
. . .

```

This `DataSource` configuration uses the Hypersonic database bundled with the JBoss server, specifies a database account named `sa` to connect to the database, and names the `DataSource` `java:/ProfileDS` when it is published in the server's JNDI service. These details would be placed in a file in the server's deployment directory whose name ends with `-ds.xml`, which is the convention used by JBoss's internal deployment engine for distinguishing `DataSource` configurations from other service configuration files.

Now we need to connect the `DataSource` reference made in the EJB module descriptor to the `DataSource` that we just configured on the server. In the JBoss server environment, we do that using an additional EJB module configuration file specific to JBoss named `jboss.xml` that is placed within the EJB module archive in the same location as the standard J2EE `ejb-jar.xml` file. This configuration file can be used for several different configuration tasks. In our case, we need to do only two things: give our profile EJB a JNDI name in the server's JNDI service, and specify that the `DataSource` referenced in the `ejb-jar.xml` file should point to the `DataSource` we just configured. Giving the profile EJB a JNDI name is necessary so that other components can reference it at runtime. In our case, we resolved the web component reference directly without using the JNDI service, but it's good practice to specify JNDI names for EJB components regardless. You never know when it might come in handy, in your application or in someone else's.

The `jboss.xml` file for our EJB module is shown in [Example 2-1](#). It follows the format of the `ejb-jar.xml` file somewhat, but the information provided here is complementary. First, we have an `<entity>` element that references the profile EJB component by specifying its `ejb-name`, `BMP Profile Bean`. Then we specify the JNDI name to be used for the component using a `<jndi-name>` element. Finally, we link the resource reference in the deployment descriptor, `jdbc/ProfileDB`, to the JNDI name of the `DataSource` we just configured, `java:/ProfileDS`.

Example 2-1. jboss.xml

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE jboss PUBLIC
    "-//JBoss//DTD JBOSS 4.0//EN"
    "http://www.jboss.org/j2ee/dtd/jboss_4_0.dtd">
<jboss>
  <enterprise-beans>
    <entity>
      <!-- Specify the JNDI name of the EJB home interface -->
      <ejb-name>BMP Profile Bean</ejb-name>
      <jndi-name>ejb/bmpProfile</jndi-name>
      <!-- Resolve the DataSource reference in the EJB deployment descriptor -->

```

```

    <resource-ref>
      <res-ref-name>jdbc/ProfileDB</res-ref-name>
      <jndi-name>java:/ProfileDS</jndi-name>
    </resource-ref>
  </entity>
</enterprise-beans>
</jboss>

```

At this point, we've resolved the EJB component reference and the `DataSource` reference in the deployment descriptors. We'll leave the details concerning the resolution of the security roles to [Chapter 10](#). But there are still a few server-specific configuration details to handle.

Our web components, for example, need to be given a URL context, the base to be used to access all of the mapped web components. We specified mappings for the web components in our `web.xml` file, but the context of these mappings needs to be specified somewhere as well. Assuming that we may deploy multiple applications in our server, and possibly include the same web components in multiple applications, we need a way to specify different URL bases on an application-by-application or module-by-module basis. In the case of JBoss, we use a JBoss configuration file for web modules, similar to the one we just saw for EJB modules. This configuration file `jboss-web.xml`, is also placed in the web module archive in the same location as the standard J2EE standard `web.xml` file. The `jboss-web.xml` file for our example is shown in [Example 2-2](#). Its only entry is the `<context-root>` element that specifies the URL base for our web components. In our case, assuming that the JBoss server is running on port 80 on a server named [jent-apps.org](#), users would access our profile view JSP using the URL <http://jent-apps.org/jent-deploy/viewprofile>. The first portion comes from the hostname of the physical server, the `jent-deploy` segment is specified in the `jboss-web.xml` file, and the `viewprofile` segment comes from the `<servlet-mapping>` element in the `web.xml` file in [Figure 2-5](#).

Example 2-2. jboss-web.xml

```

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

<!DOCTYPE jboss-web
  PUBLIC "-//JBoss//DTD Web Application 2H//EN"
  "http://www.jboss.org/j2ee/dtd/jboss-web_4_0.dtd">

<jboss-web>
  <context-root>/jent-deploy</context-root>
</jboss-web>

```

2.4.3. Generate Container-Specific Classes

The various APIs used to implement components in the J2EE framework often leave some implementation details up to the application server. JSP files need to be converted into runnable Java classes, concrete implementations of EJB client and home interfaces need to be generated, and so forth.

The process for generating these container-specific classes is not defined in the J2EE specification or

in the associated component API specifications (EJB, servlet, JSP, etc.). Each J2EE server provides its own tools or processes for generating these classes. JBoss, for example, automatically generates EJB interface implementations when you deploy an EJB module to the server, and the generation of Java classes from JSP pages is done at runtime. BEA's WebLogic application server, on the other hand, provides an optional command-line tool called `weblogic.appc` for generating container-specific classes for EJB components. This tool takes an `ejb-jar` file as an argument, parses its contents, and generates an augmented `ejb-jar` file that contains all of the WebLogic-specific classes for the EJB components in the module. Consult your J2EE vendor's documentation for details on its approach for generating these classes.



Chapter 3. Java Servlets

Over the last few years, Java has become the predominant language for server-side programming. This is due in no small part to the Java Servlet API, which provides a standard way to extend web servers to support dynamic content generation. In terms of enterprise computing, servlets are a natural fit if you are using the Web as your deployment platform. You can take advantage of web browsers as universally available thin clients using the web server as middleware for running application logic. Under this model, the user makes a request of the web server, the server invokes a servlet designed to handle the request, the servlet fulfills the request, and the result is returned to the user for display in the web browser.

While this sounds like every other dynamic content technology (such as CGI, ISAPI, ASP, PHP, and the like), servlets have some major advantages. For one, servlets are persistent between invocations, which dramatically improves performance relative to CGI-style programs. Servlets are also 100% portable across operating systems and servers, unlike any of the alternatives. Finally, servlets have access to all the APIs of the Java platform, so, for example, it is easy to create a servlet that interacts with a database, using the JDBC API (see [Chapter 8](#)).

The Servlet API also provides the basis for programming with JavaServer Pages (see [Chapter 4](#)), Struts (see [Chapter 19](#)), and Java Server Faces (see [Chapter 5](#)). In fact, if you use one of these frameworks to structure your application, you may never need to write a servlet at all. But since JSP, JSF, and Struts are all designed to work as components of larger applications, you can mix your own servlets, filters, and session management into your framework-based applications, and that makes a good knowledge of the Servlet API essential for advanced enterprise development.

Version 2.3 of the Servlet API is a required component of J2EE Version 1.3. A small update, Version 2.4, came out in 2003 and is part of the J2EE Version 1.4 specification. Since the two versions are very similar (and there are still a lot of Version 2.3 applications out there), you can assume that anything in this chapter applies equally to both versions unless we specifically say otherwise.

This chapter demonstrates the basic techniques used to write servlets using the Servlet API, including some common web development tasks such as cookie manipulation and session tracking. This chapter assumes that you have some experience with web development; if you are new to web development, you may want to brush up on web basics by consulting *Webmaster in a Nutshell* by Stephen Spainhour and Robert Eckstein (O'Reilly). For a more complete treatment of servlets, check out *Java Servlet Programming* by Jason Hunter with William Crawford (O'Reilly).

Other chapters in this book also cover topics relevant to servlet development, and you may want to consult them after digesting the following material on developing servlets. [Chapter 2](#) discusses the general J2EE deployment model for applications and components and goes into detail about deploying web components. [Chapter 10](#) provides an overview of J2EE security concepts and provides details on integrating security features into web applications. [Chapter 21](#) discusses code annotations that can be used in your servlet code to automatically generate deployment descriptors and other artifacts for your web applications, and [Appendix A](#) provides a reference for web deployment descriptors.

 PREY

3.1. Getting a Servlet Environment

You need a *servlet container* to run servlets. A servlet container uses a Java virtual machine to run servlet code as requested by a web server. The servlet container is also responsible for managing other aspects of the servlet lifecycle: user sessions, classloading, servlet contexts, servlet configuration information, servlet persistence, and temporary storage.

Because Tomcat is the reference implementation for the Servlet API, all the examples in this chapter have been tested with it. Tomcat 4.x supports the 2.3 API, and Tomcat 5.x supports the 2.4 API. Since Tomcat falls under the Apache umbrella, distribution is free, and you can download a copy (including, if you like, full source code) from <http://jakarta.apache.org>. Binary installations are available for Windows and several Unix flavors.

 PREY

3.2. Servlet Basics

The Servlet API consists of two packages, `javax.servlet` and `javax.servlet.http`. The `javax` is left over from an earlier stage of Java package naming conventions. As mentioned (and as indicated by the fact that this chapter appears in [Part I](#) of this book), servlets are a standard part of J2EE.

3.2.1. The Servlet Lifecycle

When a client makes a request involving a servlet, the server loads and executes the appropriate Java classes. Those classes generate content, and the server sends the content back to the client. In most cases, the client is a web browser, the server is a web server, and the servlet returns standard HTML. From the web browser's perspective, this isn't any different from requesting a page generated by a CGI script or, indeed, a static HTML file. On the server side, however, there is

Figure 3-1. The servlet lifecycle

an important difference: persistence.^[*] Instead of shutting down at the end of each request, the servlet can remain loaded, ready to handle subsequent requests. [Figure 3-1](#) shows how this all fits together.

[*] Note that we use "persistent" to mean "enduring between invocations," not "written to permanent storage."

The request processing time for a servlet can vary, but it is typically quite fast when compared to a similar CGI program. The real performance advantage of a servlet is that you incur most of the startup overhead only once. Most of the I/O-intensive resources (such as database connection pools) your application will need can be created by the servlet container at startup and shared by all your servlets. Instead of connecting to the database or just loading your code thousands of times a day, the container loads it only once. When the container loads a servlet for the first time, it calls the `init(ServletConfig)` method, which is allowed to complete before the servlet is asked to respond to

any requests.^[1] After the `init()` method runs, the servlet container marks the servlet as available. For each incoming connection directed at a particular servlet, the container calls the `service()` method on the servlet to process the request. The `service()` method can have access to all the resources created in the `init()` method. The servlet's `destroy()` method is called to clean up resources when the server shuts down.

[1] The `init()` method used to be an important way to create I/O-intensive resources. In more recent versions of the Servlet API, it has largely been supplanted by resource setup at the application level (via `ServletContextListener`, discussed later in this chapter) and at the container level.

Because servlets are persistent, you can actually remove a lot of filesystem and/or database access altogether. For example, to implement a page counter, you can simply store a number in a static variable rather than consulting a file (or database) for every request. Using this technique, you need to read and write to the disk only occasionally to preserve state. Since a servlet remains active, it can perform other tasks when it is not servicing client requests, such as running a background processing thread (i.e., where clients connect to the servlet to view a result) or even acting as an RMI host, enabling a single servlet to handle connections from multiple types of clients. For example, if you write an order processing servlet, it can accept transactions from both an HTML form and an applet using RMI.

The Servlet API includes numerous methods and classes for making application development easier. Most common CGI tasks require a lot of fiddling on the programmer's part; even decoding HTML form parameters can be a chore, to say nothing of dealing with cookies and session tracking. Libraries exist to help with these tasks, but they are, of course, decidedly nonstandard. You can use the Servlet API to handle most routine tasks, thus cutting development time and keeping things consistent for multiple developers on a project.

3.2.2. Writing Servlets

The three core elements of the Servlet API are the `javax.servlet.Servlet` interface, the `javax.servlet.GenericServlet` class, and the `javax.servlet.http.HttpServlet` class. Normally, you create a servlet by subclassing one of the two classes, although if you are adding servlet capability to an existing object, you may find it easier to implement the interface.

The `GenericServlet` class is used for servlets that don't implement any particular communication protocol. Here's a basic servlet that demonstrates servlet structure by printing a short message:

```
import javax.servlet.*;
import java.io.*;

public class BasicServlet extends GenericServlet {

    public void service(ServletRequest req, ServletResponse resp)
        throws ServletException, IOException {

        resp.setContentType("text/plain");
        PrintWriter out = resp.getWriter( );
        // We won't use the ServletRequest object in this example
        out.println("Hello.");
    }
}
```

```

    }
}

```

`BasicServlet` extends the `GenericServlet` class and implements one method: `service()`. Whenever a server wants to use the servlet, it calls the `service()` method, passing `ServletRequest` and `ServletResponse` objects (we'll look at these in more detail shortly). The servlet tells the server what type of response to expect, gets a `PrintWriter` from the response object, and transmits its output.

Why Two Classes?

The designers of the original Servlet API expected that it would be used for a variety of protocols above and beyond simple HTTP, hence the decision to create both a `GenericServlet` and an `HttpServlet` class. In practice, this never happened: APIs for other protocols use different mechanisms for content management. Prior to Version 2.3, generic servlets were often used to filter content from other servlets, a role now assumed by the `Filter` interface. Most servlets subclass `HttpServlet`.

3.2.3. HTTP Servlets

The `HttpServlet` class is an extension of `GenericServlet` that includes methods for handling HTTP-specific data.^[*] `HttpServlet` provides a number of methods, such as `doGet()`, `doPost()`, and `doPut()`, to handle particular types of HTTP requests (GET, POST, and so on). These methods are called by the default implementation of the `service()` method, which figures out what kind of request is being made and then invokes the appropriate method. Here's a simple `HttpServlet`:

[*] `HttpServlet` is an abstract class, implemented by your servlet classes.

```

import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;

public class HelloWorldServlet extends HttpServlet {

    public void doGet(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException {

        resp.setContentType("text/html");
        PrintWriter out = resp.getWriter( );

        out.println("<html>");
        out.println(
            "<head><title>Have you seen this before?</title></head>");
        out.println(

```

```

        "<body><h1>Hello, World!</h1><h6>Again.</h6></body></html>");
    }
}

```

`HelloWorldServlet` demonstrates many essential servlet concepts. First, `HelloWorldServlet` extends `HttpServlet`. This is standard practice for an HTTP servlet. `HelloWorldServlet` defines one method, `doGet()`, which is called whenever anyone requests a URL that points to this servlet. The `doGet()` method is actually called by the default `service()` method of `HttpServlet`. The `service()` method is called by the web server when a request is made of `HelloWorldServlet`; the method determines what kind of HTTP request is being made and dispatches the request to the appropriate `doXXX()` method (in this case, `doGet()`). `doGet()` is passed two objects, `HttpServletRequest` and `HttpServletResponse`, that contain information about the request and provide a mechanism for the servlet to produce output, respectively.

The `doGet()` method itself does three things. First, it sets the output type to `text/html`, which indicates that the servlet produces standard HTML as its output. Second, it calls the `getWriter()` method of the `HttpServletResponse` parameter to get a `java.io.PrintWriter` that points to the client. Finally, it uses the stream to send some HTML back to the client. This isn't really a whole lot different from the `BasicServlet` example, but it gives us all the tools we'll need later on for more complex web applications. We do have to explicitly set the content type, as there is no default setting, even for HTTP servlets where one might reasonably expect `text/html`.

If you define a `doGet()` method for a servlet, you may also want to override the `getLastModified()` method of `HttpServlet`. The server calls `getLastModified()` to find out if the content delivered by a servlet has changed. The default implementation of this method returns a negative number, which tells the server that the servlet doesn't know when its content was last updated, so the server is forced to call `doGet()` and return the servlet's output. If you have a servlet that changes its display data infrequently (such as a servlet that verifies uptime on several server machines once every 15 minutes), you should implement `getLastModified()` to allow browsers to cache responses. `getLastModified()` should return a `long` value that represents the time the content was last modified as the number of milliseconds since midnight, January 1, 1970, GMT (returned by calling the `getTime()` method `java.util.Date`).

A servlet should also implement `getServletInfo()`, which returns a `String` that contains information about the servlet, such as name, author, and version (just like `getAppletInfo()` in applets). This method is called by the web server and generally used for logging purposes. It's rarely implemented in the real world, but it really is good practice to do so.



3.3. Web Applications

Now that we've seen a basic servlet, we can step back for a moment and talk about how servlets are integrated into the servlet container. A web application consists of a set of resources, including servlets, static content, JSP files, and class libraries, installed in a particular path on a web server. This path is called the *servlet context*, and all servlets installed within the context are given an isolated, protected environment to operate in, without interference from (or the ability to interfere with) other software running on the server.

A servlet context directory tree contains several different types of resources. These include class files and jar files (which aren't exposed to clients connecting via web browsers), JSP files (which are processed by the JSP servlet before being fed back to the client), and static files, such as HTML documents and JPEG images, which are served directly to the browser by the web server.

The context has a virtual component, too. For each context, the servlet container will instantiate separate copies of servlets and will create a private address space that can be accessed via the `ServletContext` class. Servlets running in the same context can use this class to communicate with each other. We'll discuss this more later.

The simplest servlet installations create just a single context, rooted at `/`, which is the top of the web server path tree. Servlets and static content are installed within this context. Most of the time, you'll create multiple servlet contexts, rooted lower down on the directory tree. A catalog application, for example, could be rooted at `/catalog`, with all of the application paths below the context root.

If you write a web application that will be installed on multiple web servers, it isn't safe to assume the context root will be fixed. If the path of a resource within your application is `/servlet/CatalogServlet`, and it's installed within the `/catalog` context, rather than writing:

```
out.println("<a href=\"\"/catalog/servlet/CatalogServlet\">");
```

you should write:

```
out.println(
    "<a href=\"\" + request.getContextPath( ) + "/servlet/CatalogServlet\">");
```

This approach works regardless of the context path installed within the web server. Of course, in most cases, you shouldn't be writing HTML directly out of the servlet anyway see [Chapters 4, 5, and 19](#) on JSP, JSF, and Struts for more on the separation of content and business logic in web applications.

3.3.1. Structure of Web Applications

On disk, a web application consists of a directory. The directory contains a subdirectory called *WEB-INF* and whatever other content is required for the application. The *WEB-INF* directory contains a *classes* directory (containing application code), a *lib* directory (containing application jar files), and a file called *web.xml*. The *web.xml* file contains all of the configuration information for the servlets within the context, including names, path mappings, initialization parameters, and context-level configuration information.

The procedure for installing a web application into a servlet container varies from product to product, but it generally consists of selecting a context root and pointing the server to the directory containing the web application.^[*]

[*] Web applications can be packaged into jar file equivalents called war files. Just use the *jar* utility that comes with the JDK to pack up the web application directory (including the *WEB-INF* subdirectory) and give the resulting file a *.war* extension, or use the *war* target in an Ant script.

3.3.2. Mapping Requests with a Context

Servlets are installed within the servlet container and mapped to URIs. This is done either via global properties that apply to all servlets or by specific, servlet-by-servlet mappings. In the first case, a client invokes a servlet by requesting it by name. Some servers map servlets to a */servlet/* or */servlets/* URL. If a servlet is installed as *PageServlet*, a request to */servlet/PageServlet* would invoke it. Servlets can also be individually mapped to other URIs or to file extensions. *PageServlet* might be mapped to */pages/page1* or to all files with a *.page* extension (using **.page*).

All of these mappings exist below the context level. If the web application is installed at */app*, the paths entered into the browser for the examples earlier would be */app/servlet/PageServlet*, */app/pages/page1*, or */app/file.page*. Of course, if they are entered into a browser, these paths represent only the end of the full URL, which must also include the protocol, host, and port information (e.g., <http://myserver.org:8080>).

To illustrate, imagine the servlet mappings (all are below the context root) in [Table 3-1](#).

Table 3-1. Example servlet mappings

Mapping	Servlet
<i>/store/furniture/*</i>	<i>FurnitureServlet</i>
<i>/store/furniture/tables/*</i>	<i>TableServlet</i>
<i>/store/furniture/chairs</i>	<i>ChairServlet</i>
<i>*.page</i>	<i>PageServlet</i>

The asterisk serves as a wildcard. URIs matching the pattern are mapped to the specified servlet, provided that another mapping hasn't already been used to deal with the URL. This can get a little tricky when building complex mapping relationships, but the Servlet API does require servers to deal with mappings consistently. When the servlet container receives a request, it always maps it to the appropriate servlet in the following order:

1. *Exact path matching*

A request to `/store/furniture/chairs` is served by `ChairServlet`.

2. *Prefix mapping*

A request to `/store/furniture/sofas` is served by `FurnitureServlet`. The longest matching prefix is used. A request to `/store/furniture/tables/dining` is served by `TableServlet`.

3. *Extension*

Requests for `/info/contact.page` are served by `PageServlet`. However, requests for `/store/furniture/chairs/about.page` is served by `FurnitureServlet` (since prefix mappings are checked first, and `ChairServlet` is available only for exact matches).

If no appropriate servlet is found, the server returns an error message or attempts to serve content on its own. If a servlet is mapped to the `/` path, it becomes the default servlet for the application and is invoked when no other servlet is found.

3.3.3. Context Methods

Resources within a servlet context (such as HTML files, images, and other data) can be accessed directly via the web server. If a file called `index.html` is stored at the root of the `/app` context, then it can be accessed with a request to `/app/index.html`. Context resources can also be accessed via the `ServletContext` object, which is accessed via the `getResource()` and `getResourceAsStream()` methods. A full list of available resources can be accessed via the `getResourcePaths()` method. In this case, an `InputStream` containing the contents of the `index.html` file can be retrieved by calling `getResourceAsStream("/index.html")` on the `ServletContext` object associated with the `/app` context.

The `ServletContext` interface provides servlets with access to a range of information about the local environment. The `getInitParameter()` and `getInitParameterNames()` methods allow servlets to retrieve context-wide initialization parameters. `ServletContext` also includes a number of methods that allow servlets to share attributes. The `setAttribute()` method allows a servlet to set an attribute that can be shared by any other servlets that live in its `ServletContext`, and `removeAttribute()` allows a servlet to remove an attribute from the application context. The `getAttribute()` method, which previously allowed servlets to retrieve hardcoded server attributes, provides access to attribute values, while `getAttributeNames()` returns an `Enumeration` of all the shared attribute names.

The servlet container is required to maintain a temporary working directory on disk for each servlet context. This directory is accessed by retrieving the `javax.servlet.context.tempdir` attribute, which consists of a `java.io.File` object pointing to the temporary directory. Each servlet must have its own temporary directory. The servlet container is not required to maintain its contents across restarts, so

if you store something there, it may not be there if the servlet container restarts.



3.4. Servlet Requests

When a servlet handles a request, it typically needs specific information about the request so that it can respond appropriately. Most frequently, a servlet retrieves the value of a form variable and uses that value in its output. A servlet may also need access to information about the environment in which it is running. For example, a servlet may need to authenticate the user who is accessing the servlet.

The `ServletRequest` and `HttpServletRequest` interfaces provide access to this kind of information. When a servlet is asked to handle a request, the servlet container passes it a request object that implements one of these interfaces. With this object, the servlet can determine the actual request (e.g., protocol, URL, type), access parts of the raw request (e.g., headers, input stream), and get any client-specific request parameters (e.g., form variables, extra path information). For instance, the `getProtocol()` method returns the protocol used by the request, while `getRemoteHost()` returns the name of the client host. The interfaces also provide methods that let a servlet get information about the server (e.g., `getServername()`, `getServerPort()`). As we saw earlier, the `getParameter()` method provides access to request parameters such as form variables. There is also the `getParameterValues()` method, which returns an array of strings that contains all the values for a particular parameter. This array generally contains only one string, but some HTML form elements (as well as non-HTTP-oriented services) do allow multiple selections or options, so the method always returns an array, even if it has a length of 1.

`HttpServletRequest` adds a few more methods for handling HTTP-specific request data. For instance, `getHeaderNames()` returns an enumeration of the names of all the HTTP headers submitted with a request, while `getHeader()` returns a particular header value. Other methods handle cookies and sessions, as we'll discuss later.

[Example 3-1](#) shows a servlet that restricts access to users who are connecting via the HTTPS protocol, using digest-style authentication and coming from a government site (a domain ending in `.gov`). The restrictions are implemented by checking information pulled from the servlet request using many of the methods described earlier.

Example 3-1. Checking request information to restrict servlet access

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;

public class SecureRequestServlet extends HttpServlet {

    public void doGet(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException {

        resp.setContentType("text/html");
```

```

    PrintWriter out = resp.getWriter( );

    out.println("<html>");
    out.println("<head><title>Semi-Secure Request</title></head>");
    out.println("<body>");

    String remoteHost = req.getRemoteHost( );
    String scheme = req.getScheme( );
    String authType = req.getAuthType( );

    if((remoteHost == null) || (scheme == null) || (authType == null)) {
        out.println("Request Information Was Not Available.");
        return;
    }

    if(scheme.equalsIgnoreCase("https") && remoteHost.endsWith(".gov")
        && authType.equals("Digest")) {
        out.println("Special, secret information.");
    }
    else {
        out.println("You are not authorized to view this data.");
    }

    out.println("</body></html>");
}
}

```

3.4.1. Forms and Interaction

The problem with creating a servlet like our earlier `HelloWorldServlet` is that it doesn't do anything we can't already do with HTML. If we are going to bother with a servlet at all, we should do something dynamic and interactive with it. In many cases, this means processing the results of an HTML form. To make our example less impersonal, let's have it greet the user by name. The HTML form that calls the servlet using a GET request might look like this:

```

<html>
<head><title>Greetings Form</title></head>
<body>
<form method=get action="/servlet/HelloServlet">
What is your name?
<input type=text name="username" size=20>
<input type=submit value="Introduce Yourself">
</form>
</body>
</html>

```

This form submits a form variable named `username` to `/servlet/HelloServlet`. The `HelloServlet` itself

does little more than create an output stream, read the `username` form variable, and print a nice greeting for the user. Here's the code:

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;

public class HelloServlet extends HttpServlet {

    public void doGet(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException {

        resp.setContentType("text/html");
        PrintWriter out = resp.getWriter( );

        out.println("<html>");
        out.println("<head><title>Finally, interaction!</title></head>");
        out.println("<body><h1>Hello, " + req.getParameter("username")
            + "!</h1>");
        out.println("</body></html>");
    }
}
```

All we've done differently is use the `getParameter()` method of `HttpServletRequest` to retrieve the value of a form variable.^[*] When a server calls a servlet, it can also pass a set of request parameters. With HTTP servlets, these parameters come from the HTTP request itself in this case, in the guise of URL-encoded form variables. Note that a `GenericServlet` running in a web server also has access to these parameters using the simpler `ServletRequest` object. When `HelloServlet` runs, it inserts the value of the `username` form variable into the HTML output, as shown in [Figure 3-2](#).

[*] For those interested in ancient history, in Java Web Server 1.1, the `getParameter()` method was deprecated in favor of `getParameterValues()`, which returns a `String` array rather than a single string. However, after an extensive write-in campaign, Sun took `getParameter()` off the deprecated list for Version 2.0 of the Servlet API, so you can safely use this method in your servlets. This is not an issue with later versions of the API, and the fact that we even talk about it dates us.

Figure 3-2. Output from HelloServlet

3.4.2. POST, HEAD and Other Requests

As mentioned earlier, `doGet()` is just one of a collection of enabling methods for HTTP request types. `doPost()` is the corresponding method for POST requests. The POST request is designed for posting information to the server, although in practice it is also used for long parameterized requests and larger forms, to get around limitations on the length of URLs.

If your servlet is performing database updates, charging a credit card, or doing anything that takes an explicit client action, you should make sure this activity is happening in `doPost()` method because POST requests aren't *idempotent*, which means that they aren't safely repeatable, and web browsers treat them specially. For example, a browser can't bookmark or, in some cases, reload a POST request. On the other hand, GET requests are idempotent, so they can safely be bookmarked, and a browser is free to issue the request repeatedly without necessarily consulting the user. You can see why you don't want to charge a credit card in a GET method!

To create a servlet that can handle POST requests, all you have to do is override the default `doPost()` method from `HttpServlet` and implement the necessary functionality in it. If necessary, your application can implement different code in `doPost()` and `doGet()`. For instance, the `doGet()` method might display a postable data entry form that the `doPost()` method processes. `doPost()` can even call `doGet()` at the end to display the form again.

The less common HTTP request types, such as HEAD, PUT, TRACE, and DELETE, are handled by other `doXXX()` dispatch methods. A HEAD request returns HTTP headers only, PUT and DELETE allow clients to create and remove resources from the web server, and TRACE returns the request headers to the client. Since most servlet programmers don't need to worry about these requests, the `HttpServlet` class includes a default implementation of each corresponding `doXXX()` method that either informs the client that the request is unsupported or provides a minimal implementation. You can provide your own versions of these methods, but the details of implementing PUT or DELETE functionality go rather beyond our scope.

3.5. Servlet Responses

In order to do anything useful, a servlet must send a response to each request that is made to it. In the case of an HTTP servlet, the response can include three components: a status code, any number of HTTP headers, and a response body.

The `ServletResponse` and `HttpServletResponse` interfaces include all the methods needed to create and manipulate a servlet's output. We've already seen that you specify the MIME type for the data returned by a servlet using the `setContentType()` method of the response object passed into the servlet. With an HTTP servlet, the MIME type is generally `text/html`, although some servlets return binary data: a servlet that loads a GIF file from a database and sends it to the web browser should set a content type of `image/gif`, while a servlet that returns an Adobe Acrobat file should set it to `application/pdf`.

`ServletResponse` and `HttpServletResponse` each define two methods for producing output streams, `getOutputStream()` and `getWriter()`. The former returns a `ServletOutputStream`, which can be used for textual or binary data. The latter returns a `java.io.PrintWriter` object, which is used only for textual output. The `getWriter()` method examines the content type to determine which charset to use, so `setContentType()` should be called before `getWriter()`.

`HttpServletResponse` also includes a number of methods for handling HTTP responses. Most of these allow you to manipulate the HTTP header fields. For example, `setHeader()`, `setIntHeader()`, and `setDateHeader()` allow you to set the value of a specified HTTP header, while `containsHeader()` indicates whether a certain header has already been set. You can use either the `setStatus()` or `sendError()` method to specify the status code sent back to the client. `HttpServletResponse` provides a long list of integer constants that represent specific HTTP status codes (we'll see some of these shortly). You typically don't need to worry about setting a status code, as the default code is 200 ("OK"), meaning that the servlet sent a normal response. However, a servlet that is part of a complex application structure (such as serving XML content to an AJAX-based dynamic web interface) may need to use a variety of status codes. Finally, the `sendRedirect()` method allows you to issue a page redirect. Calling this method sets the `Location` header to the specified location and uses the appropriate status code for a redirect.

3.5.1. Request Dispatching

Request dispatching allows a servlet to delegate request handling to other components on the server. A servlet can either forward an entire request to another servlet or include bits of content from other components in its own output. In either case, this is done with a `RequestDispatcher` object that is obtained from the `ServletContext` via the `getRequestDispatcher()` method (also available via the `HttpServletRequest` object). When you call this method, you specify the path to the servlet to which you are dispatching the request. The path should be relative to the servlet context. If you want to dispatch a request to `/servlet/TargetServlet` within the `/app` context (which is accessed from a user's browser by `/app/servlet/TargetServlet`), request a dispatcher for `/servlet/TargetServlet`.

When you dispatch a request, you can set request attributes using the `setAttribute()` method of `ServletRequest` and read them using the `getAttribute()` method. A list of available attribute names is returned by `getAttributeNames()`. Rather than taking only `String` objects (like parameters), an attribute may be any valid Java object.

`RequestDispatcher` provides two methods for dispatching requests: `forward()` and `include()`. To forward an entire request to another servlet, use the `forward()` method. When using `forward()`, the `ServletRequest` object is updated to include the new target URL. If a `ServletOutputStream` or `PrintWriter` has already been retrieved from the `ServletResponse` object, the `forward()` method throws an `IllegalStateException`.

The `include()` method of `RequestDispatcher` causes the content of the dispatchee to be included in the output of the main servlet, just like a server-side include. To see how this works, let's look at part of a servlet that does a keep-alive check on several different servers. The `ServerMonitorServlet` referenced in this example relies on the `serverurl` attribute to determine which server to display monitoring information for:

```
out.println("Uptime for our servers");
// Get a RequestDispatcher to the ServerMonitorServlet
RequestDispatcher d = getServletContext( ).
    getRequestDispatcher("/servlet/ServerMonitorServlet");
req.setAttribute("serverurl", new URL("http://www1.company.com"));
d.include(req, res);
req.setAttribute("serverurl", new URL("http://www2.company.com"));
d.include(req, res);
```

Request dispatching is obviously different from issuing an HTTP redirect (via the `sendRedirect()` method of `HttpServletResponse`) since everything takes place within a single server request. Using a forward rather than a redirect gives a better user experience since content is displayed in the browser without two round-trips to the server. However, when you forward a request, the apparent URL (from the browser's perspective) remains the original request URL. This can break relative links and sometimes leads to confusing caching behavior on the browser. It is a good practice to avoid relative links wherever possible and encode the full server-side path, starting with the context name, instead.

3.5.2. Error Handling

Sometimes things just go wrong. When that happens, it's nice to have a clean way out. The Servlet API gives you two ways of to deal with errors: you can manually send an error message back to the client or you can throw a `ServletException`. The easiest way to handle an error is simply to write an error message to the servlet's output stream. This is the appropriate technique to use when the error is part of a servlet's normal operation, such as when a user forgets to fill in a required form field.

3.5.2.1. Status codes

When an error is a standard HTTP error, you should use the `sendError()` method of `HttpServletResponse` to tell the server to send a standard error status code. `HttpServletResponse`

defines integer constants for all the major HTTP status codes. [Table 3-2](#) lists the most common status codes. For example, if a servlet can't find a file the user has requested, it can send a 404 ("File Not Found") error and let the browser display it in its usual manner. In this case, we can replace the typical `setContentType()` and `getWriter()` calls with something like this:

```
response.sendError(HttpServletResponse.SC_NOT_FOUND);
```

If you want to specify your own error message (in addition to the web server's default message for a particular error code), you can call `sendError()` with an extra `String` parameter:

```
response.sendError(HttpServletResponse.SC_NOT_FOUND,
    "It's dark. I couldn't find anything.");
```

Table 3-2. Some common HTTP status codes

Constant	Code	Default message	Meaning
<code>SC_OK</code>	200	OK	The client's request succeeded, and the server's response contains the requested data. This is the default status code.
<code>SC_NO_CONTENT</code>	204	No Content	The request succeeded, but there is no new response body to return. A servlet may find this code useful when it accepts data from a form, but wants the browser view to stay at the form. It avoids the "Document contains no data" error message.
<code>SC_MOVED_PERMANENTLY</code>	301	Moved Permanently	The requested resource has permanently moved to a new location. Any future reference should use the new location given by the <code>Location</code> header. Most browsers automatically access the new location.
<code>SC_MOVED_TEMPORARILY</code>	302	Moved Temporarily	The requested resource has temporarily moved to another location, but future references should still use the original URL to access the resource. The temporary new location is given by the <code>Location</code> header. Most browsers automatically access the new location.
<code>SC_UNAUTHORIZED</code>	401	Unauthorized	The request lacked proper authorization. Used in conjunction with the <code>WWW-Authenticate</code> and <code>Authorization</code> headers.
<code>SC_NOT_FOUND</code>	404	Not Found	The requested resource is not available.

Constant	Code	Default message	Meaning
<code>SC_INTERNAL_SERVER_ERROR</code>	500	Internal Server Error	An error occurred inside the server that prevented it from fulfilling the request.
<code>SC_NOT_IMPLEMENTED</code>	501	Not Implemented	The server doesn't support the functionality needed to fulfill the request.
<code>SC_SERVICE_UNAVAILABLE</code>	503	Service Unavailable	The server is temporarily unavailable, but service should be restored in the future. If the server knows when it will be available again, a <code>Retry-After</code> header may also be supplied.

3.5.2.2. Servlet exceptions

The Servlet API includes two `Exception` subclasses, `ServletException` and its derivative, `UnavailableException`. A servlet throws a `ServletException` to indicate a general servlet problem. When a server catches this exception, it can handle the exception however it sees fit.

`UnavailableException` is a bit more useful, however. When a servlet throws this exception, it is notifying the server that it is unavailable to service requests. You can throw an `UnavailableException` when some factor beyond your servlet's control prevents it from dealing with requests. To throw an exception that indicates permanent unavailability, use something like this:

```
throw new UnavailableException(this,
    "This is why you can't use the servlet.");
```

`UnavailableException` has a second constructor to use if the servlet is going to be temporarily unavailable. With this constructor, you specify how many seconds the servlet is going to be unavailable, as follows:

```
throw new UnavailableException(120, this, "Try back in two minutes");
```

One caveat: the servlet specification does not mandate that servers actually try again after the specified interval. If you choose to rely on this capability, you should test it first with the container you plan to deploy on.

3.5.2.3. A file-serving servlet

[Example 3-2](#) demonstrates both of these error-handling techniques, along with another method for reading data from the server. `FileServlet` reads a pathname from a form parameter and returns the associated file. Note that this servlet is designed to return only HTML files. If the file can't be found, the servlet sends the browser a 404 error. If the servlet lacks sufficient access privileges to load the file, it sends an `UnavailableException` instead. Keep in mind that this servlet is a teaching exercise:

you should not deploy it on your web server. (For one thing, any security exception renders the servlet permanently unavailable, and for another, it can serve files from the root of your hard drive.)

Example 3-2. Serving files

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;

public class FileServlet extends HttpServlet {

    public void doGet(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException {

        File r;
        FileReader fr;
        BufferedReader br;
        try {
            r = new File(req.getParameter("filename"));
            fr = new FileReader(r);
            br = new BufferedReader(fr);
            if(!r.isFile( )) { // Must be a directory or something else
                resp.sendError(resp.SC_NOT_FOUND);
                return;
            }
        }
        catch (FileNotFoundException e) {
            resp.sendError(resp.SC_NOT_FOUND);
            return;
        }
        catch (SecurityException se) { // Be unavailable permanently
            throw(new UnavailableException(
                "Servlet lacks appropriate privileges."));
        }

        resp.setContentType("text/html");
        PrintWriter out = resp.getWriter( );
        String text;
        while( (text = br.readLine( )) != null)
            out.println(text);

        br.close( );
    }
}
```

3.6. Custom Servlet Initialization

At the beginning of this chapter, we talked about how a servlet's persistence can be used to build more efficient web applications. This is accomplished via class variables and the `init()` method. When a server loads a servlet for the first time, it calls the servlet's `init()` method and does not make any service calls until `init()` has finished. In the default implementation, `init()` simply handles some basic housekeeping, but a servlet can override the method to perform whatever one-time tasks are required. This often means doing some sort of I/O-intensive resource creation, such as opening a database connection. You can also use the `init()` method to create threads that perform various ongoing tasks. For instance, a servlet that monitors the status of machines on a network might create a separate thread to periodically ping each machine. When an actual request occurs, the service methods in the servlet can use the resources created in `init()`. Thus, the status monitor servlet might display an HTML table with the status of the various machines.

The default `init(ServletConfig)` implementation is not a do-nothing method, so you should remember to always call the `super.init(ServletConfig)` method. If you override the parameterless version, you don't have to invoke the superclass's method; the server calls the parameterized version, which calls the parameterless method. You can access the `ServletConfig` object using the `getServletConfig()` method.

The server passes the `init()` method a `ServletConfig` object, which can include specific servlet configuration parameters (for instance, the list of machines to monitor). `ServletConfig` encapsulates the servlet initialization parameters, which are accessed via the `getInitParameter()` and `getInitParameterNames()` methods. `GenericServlet` and `HttpServlet` both implement the `ServletConfig` interface, so these methods are always available in a servlet and they access the servlet's `ServletConfig` to retrieve these parameters. Init parameters are set using the `<init-param>` elements for the `<servlet>` in the `web.xml`/deployment descriptor (see [Appendix A](#)).

Every servlet also has a `destroy()` method that can be overwritten. This method is called when, for whatever reason, a server unloads a servlet. You can use this method to ensure that important resources are freed or that threads are allowed to finish executing unmolested. Unlike `init()`, the default implementation of `destroy()` is a do-nothing method, so you don't have to worry about invoking the superclass's `destroy()` method.

[Example 3-3](#) shows a counter servlet that saves its state between server shutdowns. It uses the `init()` method to first try to load a default value from a servlet initialization parameter. Next the `init()` method tries to open a file named `/data/counter.dat` and read an integer from it. When the servlet is shut down, the `destroy()` method creates a new `counter.dat` file with the current hit count for the servlet.

Example 3-3. A persistent counter servlet

```
import javax.servlet.*;
import javax.servlet.http.*;
```

```
import java.io.*;

public class LifecycleServlet extends HttpServlet {

    int timesAccessed;

    public void init(ServletConfig conf) throws ServletException {

        super.init(conf);

        // Get initial value
        try {
            timesAccessed = Integer.parseInt(getInitParameter("defaultStart"));
        }
        catch(NullPointerException e) {
            timesAccessed = 0;
        }
        catch(NumberFormatException e) {
            timesAccessed = 0;
        }

        // Try loading from the disk
        DataInputStream ds = null;
        try {
            File r = new File("./data/counter.dat");
            DataInputStream ds = new DataInputStream(new FileInputStream(r));
            timesAccessed = ds.readInt( );
        }
        catch (FileNotFoundException e) {
            // Handle error
        }
        catch (IOException e) {
            // This should be logged
        }
        finally {
            if (ds !=null)
                try {
                    ds.close( );
                } catch (IOException ignored) {}
        }
    }

    public void doGet(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException {

        resp.setContentType("text/html");
        PrintWriter out = resp.getWriter( );

        timesAccessed++;

        out.println("<html>");
        out.println("<head>");
        out.println("<title>Life Cycle Servlet</title>");
    }
}
```

```

        out.println("</head><body>");

        out.println("I have been accessed " + timesAccessed + " time[s]");
        out.println("</body></html>");
    }

    public void destroy( ) {

        // Write the Integer to a file
        File r = new File("./data/counter.dat");
        DataOutputStream dout = null;
        try {
            dout = new DataOutputStream(new FileOutputStream(r));
            dout.writeInt(timesAccessed);
        }
        catch(IOException e) {
            // This should be logged
        }
        finally {
            try dout.close( ); catch (IOException ignored) {}
        }
    }
}

```

3.6.1. Servlet Context Initialization

Version 2.3 of the Servlet API added support for application-level events using a listener-style interface. Classes that implement the `ServletContextListener` interface can be associated with a servlet context and are notified when the context is initialized or destroyed. This provides programmers with the opportunity to create application-level resources, such as database connection pools, before any servlets are initialized and to share single resources among multiple servlets using the `ServletContext` attribute functionality.

`ServletContextListener` contains two methods, `contextInitialized()` and `contextDestroyed()`, which take a `ServletContextEvent`. Context listeners are associated with their context in the *web.xml* file for the web application (see [Appendix A](#) for details on configuring contexts in the *web.xml* file). [Example 3-4](#) defines a listener that creates a hashtable of usernames and unencrypted passwords and associates it as a context attribute. We use it in a later example.

Example 3-4. A servlet context listener

```

import javax.servlet.ServletContextListener;
import javax.servlet.ServletContextEvent;

public class ContextResourceLoader implements ServletContextListener {

    public void contextInitialized(ServletContextEvent sce) {
        java.util.Hashtable users = new Hashtable( );
    }
}

```

```
        users.put("test", "test");
        users.put("admin", "bob3jk");
        sce.getServletContext( ).setAttribute("enterprise.users", users);
    }

    public void contextDestroyed(ServletContextEvent sce) {
        // This is where we clean up resources on server shutdown/restart
    }
}
```

Obviously, a real application would manage usernames and passwords in a more robust fashion, such as by storing them in a database. In this case, we can count on the JVM to properly garbage-collect the `Hashtable` object. If we do something more complex (such as maintaining a pool of connections to a relational database), we would use the `contextDestroyed()` method to make sure those resources are properly freed.

[← PREVIOUS](#)

3.7. Security

Servlets don't have to handle their own security arrangements. Instead, they can rely on the capabilities of the web server to limit access where required. The security capabilities of most web servers are limited to basic on-or-off access to specific resources, controlled by username and password (or digital certificate), with possible encryption using SSL. Most servers are limited to basic authentication, which transmits passwords more or less in the clear, while some support the more advanced digest authentication protocol, which works by transmitting a hash of the user's password and a server-generated value rather than the password itself. Both of these approaches look the same to the user; the familiar "Enter Username and Password" window pops up in the web browser.

Recent versions of the Servlet API take a much less hands-off approach to security. The *web.xml* file can define which servlets and resources are protected and which users have access. The user access model is the J2EE User-Role model, in which users can be assigned one or more roles. Users with a particular role are granted access to protected resources. A user named Admin might have both the Administrator role and the User role while users Bob and Ted might have only the User role. (See [Chapter 10](#) for more details about J2EE security.)

In addition to basic, digest, and SSL authentication, the web application framework allows for HTML form-based logins. This approach allows the developer to specify an HTML or JSP page containing a form like the following:

```
<form method="post" action="j_security_check">
<input type="text" name="j_username">
<input type="password" name="j_password">
<input type="submit" value="Log In">
</form>
```

Note that form-based authentication is insecure and works only if the client session is being tracked via cookies or SSL signatures.

The `HttpServletRequest` interface includes a pair of basic methods for retrieving standard HTTP user authentication information from the web server. If your web server is equipped to limit access, a servlet can retrieve the username with `getRemoteUser()` and the authentication method (basic, digest, or SSL) with `getAuthType()`. Version 2.2 of the Servlet API added the `isUserInRole()` and `getUserPrincipal()` methods to `HttpServletRequest`. `isUserInRole()` allows the program to query whether the current user has a particular role (useful for dynamic content decisions that cannot be made at the container level). The `getUserPrincipal()` method returns a `java.security.Principal` object identifying the current user.

3.8. Servlet Filters

Version 2.3 of the Servlet API introduced a new method of handling requests using the `javax.servlet.Filter` class. When filters are used, the servlet container creates a *filter chain*, which consists of zero or more `Filter` objects and a destination resource, either a servlet or another resource available on the web server (such as an HTML or JSP file).

Filters are installed in the server and associated with particular request paths (just like servlets). When a filtered resource is requested, the servlet constructs a filter chain and calls the `doFilter()` method of the first filter in the filter chain, passing a `ServletRequest`, a `ServletResponse`, and the `FilterChain` object. The filter can then perform processing on the request. Filters are often used to implement logging, control security, or set up connection-specific objects. A filter can also wrap the `ServletRequest` and `ServletResponse` classes with its own versions, overriding particular methods. For instance, one of the example filters included with the Tomcat server adds support for returning compressed output to browsers that support it.

After the filter has processed the response, it can call the `doFilter()` method of the `FilterChain` to invoke the next filter in the sequence. If there are no more filters, the request is passed on to its ultimate destination. After calling `doFilter()`, the filter can perform additional processing on the response received from farther down the chain.

In the event of an error, the filter can stop processing, returning to the client whatever response has already been created or forwarding the request to a different resource.

[Example 3-5](#) provides a form-based authentication filter that could be customized to provide additional functionality. It works by intercepting each request and checking the `HttpSession` for an attribute called `enterprise.login`. If that attribute contains a `Boolean.TRUE`, access is permitted. If not, the filter checks for request parameters named `login_name` and `login_pass` and searches for a match in a hashtable containing valid username/password pairs. If valid login credentials are found, filter chain processing continues. If not, the user is served a login page located at `/login.jsp`, retrieved via a `RequestDispatcher`.^[*]

[*] This isn't a highly secure system. Unless the client has connected via SSL, the username/password combination is transmitted unencrypted over the Internet. Also, successful logins leave the `login_name` and `login_pass` parameters in the request when processing it, potentially making them available to a malicious JSP file or servlet. This can be an issue when designing a shared security scheme for dynamic content created by a group of different users (such as at an ISP). One way to get around this is to create a custom `HttpServletRequest` wrapper that filters out the `login_name` and `login_pass` parameters for filters and resources further down the chain.

Astute readers will note that we try to retrieve the user's hashtable from a servlet context attribute. We showed how to set this attribute at the web application level earlier in the chapter. In case you don't have that set up, the filter's `init()` method will create its own if it can't find one in the context.

Example 3-5. AuthenticationFilter

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.Hashtable;

public class AuthenticationFilter implements Filter {

    private Hashtable users = null;
    public void init(FilterConfig config)
        throws javax.servlet.ServletException {

        users = (Hashtable)config.getServletContext( ).getAttribute(
            "enterprise.users");

        if(users == null) {
            users = new Hashtable(5);
            users.put("test", "test");
        }
    }

    public void doFilter(
        ServletRequest req, ServletResponse res, FilterChain chain)
        throws java.io.IOException, javax.servlet.ServletException {

        HttpServletRequest request = (HttpServletRequest)req;
        HttpSession sess = request.getSession(true);

        if(sess != null) {
            Boolean loggedIn = (Boolean)sess.getAttribute("enterprise.login");
            if (loggedIn != Boolean.TRUE) {
                String login_name = request.getParameter("login_name");
                String login_pass = request.getParameter("login_pass");
                if((login_name != null) && (login_pass != null))
                    if(users.get(login_name).toString( ).equals(login_pass)) {
                        loggedIn = Boolean.TRUE;
                        sess.setAttribute("enterprise.login", Boolean.TRUE);
                        sess.setAttribute("enterprise.loginname", login_name);
                    }
            }

            if (loggedIn == Boolean.TRUE) {
                chain.doFilter(req, res);
            } else {
                request.setAttribute("originaluri", request.getRequestURI( ));
                request.getRequestDispatcher("/login.jsp").forward(req, res);
            }
        }
    }

    public void destroy( ) {
        // Code cleanup would be here
    }
}
```

Here's the JSP page used to display the login form. The important thing to note is that the form submits back to the original URI. The filter uses the `setAttribute()` method of `HttpServletRequest` to specify the URI to post the form back to; the filter is then reapplied, and if the user has provided appropriate credentials, access to the resource is granted. For more on JSP, see [Chapter 4](#).

```
<html><body bgcolor="white">

<% out.print ("<form method=post
action=\""+request.getAttribute("originaluri").toString( ) +"\">"); %>

Login Name: <input type=text name="login_name"><br>
Password: <input type=password name="login_pass">
<input type=submit value="Log In">
</form>

</body></html>
```

When configuring the filter (we'll see an example in the next section), map it only to the paths you wish to protect. Mapping it to `/*` will not work since that would also protect the `login.jsp` file. If you did want to protect your whole application, you could build the login form into the filter, but that's no good practice—you have to recompile for every change.

3.8.1. Filters and Request Dispatchers

In the original filter specification, a filter would run only in response to an original request from a remote client. Requests created internally via the `RequestDispatcher` class would not trigger new filter chains. The example in the previous section relies on this behavior; otherwise the filter would never be able to display the `login.jsp` file.

Servlet API 2.4 gives us a little more flexibility here. When declaring a filter mapping in `web.xml`, we can indicate whether it runs on requests only (the default) or whether it runs on includes, forwards, or any combination of the three. For example:

```
<filter-mapping>
  <filter-name>Authentication Filter</filter-name>
  <url-pattern>/secrets/*</url-pattern>
  <dispatcher>FORWARD</dispatcher>
  <dispatcher>REQUEST</dispatcher>
</filter-mapping>
```

This configuration causes the filter to run on request forwards as well as request dispatches, ensuring that security is applied even if an unprotected portion of the application redirects the user to a protected portion. However, use this power with care: many applications, including the various web application frameworks, make extensive use of request forwarding, which could cause the filter to run several times in the course of a single user request.



3.9. Thread Safety

In a typical scenario, only one copy of any particular servlet or filter is loaded into the server's runtime at any given time. Each servlet might, however, be called upon to deal with multiple requests at the same time. This means that a servlet needs to be threadsafe. If a servlet doesn't use any class variables (that is, any variables with a scope broader than the service method itself), it is generally already threadsafe. If you are using any third-party libraries or extensions, make sure that those components are also threadsafe. However, a servlet that maintains persistent resources needs to make sure that nothing untoward happens to those resources. Imagine, for example, a servlet that maintains a bank balance using an `int` in memory.^[*] If two servlets try to access the balance at the same time, you might get this sequence of events:

[*] Hey, bear with us on this. It's an example.

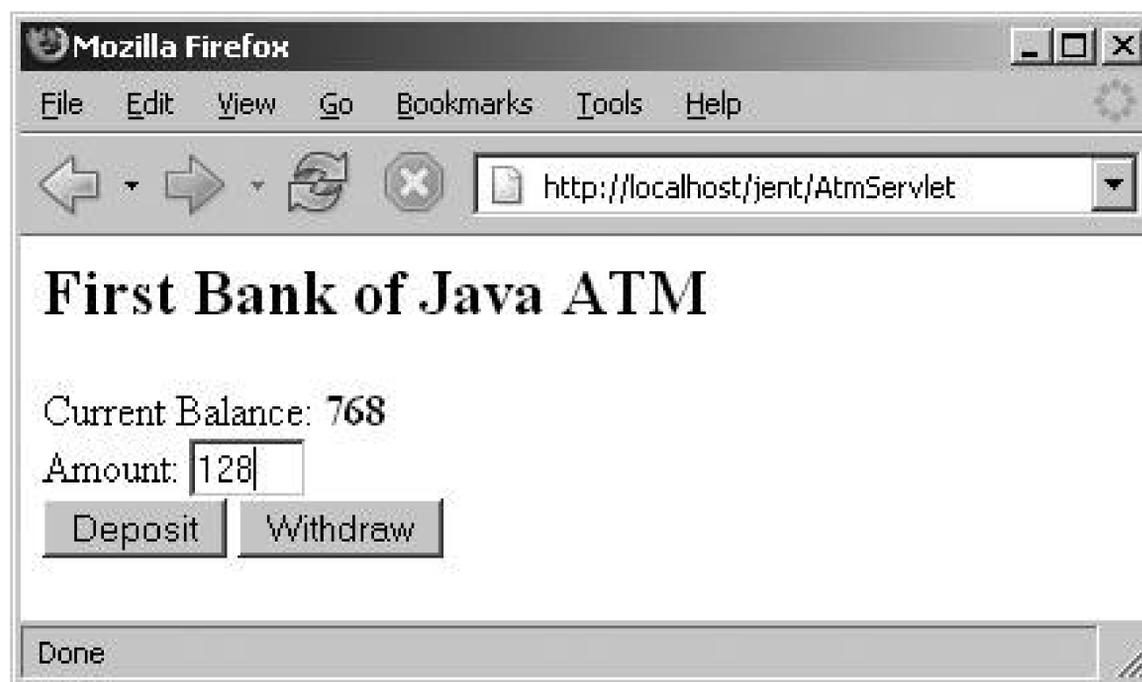
1. User 1 connects to the servlet to make a \$100 withdrawal.
2. The servlet checks the balance for User 1, finding \$120.
3. User 2 connects to the servlet to make a \$50 withdrawal.
4. The servlet checks the balance for User 2, finding \$120.
5. The servlet debits \$100 for User 1, leaving \$20.
6. The servlet debits \$50 for User 2, leaving -\$30.
7. The programmer is fired.

Obviously, this is incorrect behavior, particularly that last bit. We want the servlet to perform the necessary action for User 1, and then deal with User 2 (in this case, by giving him an insufficient funds message). We can do this by surrounding sections of code with synchronized blocks. While a particular synchronized block is executing, no other sections of code that are synchronized on the same object (usually the servlet or the resource being protected) can execute. For more information on thread safety and synchronization, see *Java Threads* by Scott Oaks and Henry Wong (O'Reilly).

[Example 3-6](#) implements the ATM display for the First Bank of Java. The `doGet()` method displays the current account balance and provides a small ATM control panel for making deposits and withdrawals, as shown in [Figure 3-3](#).^[1]

[1] Despite the fact that Java is a very large island, there's still only one account.

Figure 3-3. The First Bank of Java ATM display



The control panel uses a POST request to send the transaction back to the servlet, which performs the appropriate action and calls `doGet()` to redisplay the ATM screen with the updated balance.

Example 3-6. An ATM servlet

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.*;
import java.io.*;

public class AtmServlet extends HttpServlet {

    Account act;

    public void init(ServletConfig conf) throws ServletException {
        super.init(conf);
        act = new Account( );
        act.balance = 0;
    }

    public void doGet(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException {

        resp.setContentType("text/html");
        PrintWriter out = resp.getWriter( );

        out.println("<html><body>");
        out.println("<h2>First Bank of Java ATM</h2>");
        out.println("Current Balance: <b>" + act.balance + "</b><br>");
        out.println("<form method=post>");
        out.println("Amount: <input type=text name=AMOUNT size=3><br>");
        out.println("<input type=submit name=DEPOSIT value=\"Deposit\">");
    }
}
```

```

        out.println("<input type=submit name=WITHDRAW value=\"Withdraw\">");
        out.println("</form>");
        out.println("</body></html>");
    }

    public void doPost(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException {

        int amt=0;
        try {
            amt = Integer.parseInt(req.getParameter("AMOUNT"));
        }
        catch (NullPointerException e) {
            // No Amount parameter passed
        }
        catch (NumberFormatException e) {
            // Amount parameter was not a number
        }

        synchronized(act) {
            if((req.getParameter("WITHDRAW") != null) && (amt < act.balance))
                act.balance = act.balance - amt;
            if((req.getParameter("DEPOSIT") != null) && (amt > 0))
                act.balance = act.balance + amt;
        } // End synchronized block

        doGet(req, resp);                // Show ATM screen
    }

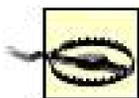
    public void destroy( ) {
        // This is where we would save the balance to a file
    }

    class Account {
        public int balance;
    }
}

```

The `doPost()` method alters the account balance contained within an `Account` object `act` (since `Account` is so simple, we've defined it as an inner class). In order to prevent multiple requests from accessing the same account at once, any code that alters `act` is synchronized on `act`. This ensures that no other code can alter `act` while a synchronized section is running.

The `destroy()` method is defined in the `AtmServlet`, but it contains no actual code. A real banking servlet would obviously want to write the account balance to disk before being unloaded. And if the servlet were using JDBC to store the balance in a database, it would also want to destroy all its database-related objects.



Servlet API 2.4 deprecates the `SingleThreadModel` interface, which was a tag interface to identify a servlet that could serve only one request at a time. Since `SingleThreadModel` could not resolve all potential threading issues, it has been deprecated and should not be used in new development.

← PREV

3.10. Cookies

Very few software features have caused as much public confusion and outcry as the HTTP cookie. Ethical and moral considerations aside, cookies allow a web server to store small amounts of data on client systems. Cookies are generally used to store basic user identification or configuration information. Because a cookie's value can uniquely identify a client, cookies are often used for session tracking (although, as we'll see shortly, the Servlet API provides higher-level support for this).^[*]

[*] The cookie standard is spelled out in RFC 2965, available at <http://rfc.net/rfc2965.html>.

To create a cookie, the server (or, more precisely, a web application running on the server) includes a `Cookie` header with a specific value in an HTTP response. The browser then transmits a similar header with that value back to the server with subsequent requests, which are subject to certain rules. The web application can use the cookie value to keep track of a particular user, handle session tracking, and so forth. Because cookies use a single `Cookie` header, the syntax for a cookie allows for multiple name/value pairs in the overall cookie value.

The Servlet API includes a class, `javax.servlet.http.Cookie`, that abstracts cookie syntax and makes cookies easy to work with. In addition, `HttpServletResponse` provides an `addCookie()` method and `HttpServletRequest` provides a `getCookies()` method to aid in writing cookies to and reading cookies from the HTTP headers, respectively. To find a particular cookie, a servlet needs to read the entire collection of values and look through it:

```
Cookie[] cookies;
cookies = req.getCookies( );
String userid = null;

for (int i = 0; i < cookies.length; i++)
    if (cookies[i].getName( ).equals("userid"))
        userid = cookies[i].getAttribute( );
```

A cookie can be read at any time but can be created only before any content is sent to the client. This is because cookies are sent using HTTP headers. These headers can be sent to the client before the regular content. Once any content has been written to the client, the server can flush the output and send the headers at any time, so you can't create any new cookies safely. You must create new cookies before sending any content. Here's an example of creating a cookie:

```
String userid = createUserID( ); // Create a unique ID
Cookie c = new Cookie("userid", userid);
resp.addCookie(c); // Add the cookie to the HTTP headers
```

Note that a web browser is required to accept only 20 cookies per site and 300 total per user, and

the browser can limit each cookie's size to 4096 bytes.

Cookies can be customized to return information only in specific circumstances. In particular, a cookie can specify a particular domain, a particular path, an age after which the cookie should be destroyed and whether the cookie requires a secure (HTTPS) connection. A cookie is normally returned only to the host that specified it. For example, if a cookie is set by server1.company.com, it isn't returned to server2.company.com. You can get around this limitation by setting the domain to company.com with the `setDomain()` method of `Cookie`. By the same token, a cookie is generally returned for pages only in the same directory as the servlet that created the cookie, or it's returned under that directory. We can get around this limitation using `setPath()`. Here's a cookie that is returned to all pages on all top-level servers at company.com:

```
String userid = createUserID( ); // Create a unique ID
Cookie c = new Cookie("userid", userid);
c.setDomain(".company.com"); // *.company.com, but not *.web.company.com
c.setPath("/"); // All pages
resp.addCookie(c); // Add the cookie to the HTTP headers
```

 PREVIOUS

3.11. Session Tracking

Very few web applications are confined to a single page, so having a mechanism for tracking users through a site can often simplify application development. The Web, however, is an inherently stateless environment. A client makes a request, the server fulfills it, and both promptly forget about each other. In the past, applications that needed to deal with a user through multiple pages (for instance, a shopping cart) had to resort to complicated dodges to hold onto state information, such as hidden fields in forms, setting and reading cookies, or rewriting URLs to contain state information.

The Servlet API provides classes and methods specifically designed to handle session tracking. A servlet can use the session-tracking API to delegate most of the user-tracking functions to the server. The first time a user connects to a session-enabled servlet, the servlet simply creates a `javax.servlet.http.HttpSession` object. The servlet can then bind data to this object, so subsequent requests can read the data. After a certain amount of inactive time, the session object is destroyed.

A servlet uses the `getSession()` method of `HttpServletRequest` to retrieve the current session object. This method takes a single `boolean` argument. If you pass `TRUE` and there is no current session object, the method creates and returns a new `HttpSession` object. If you pass `false`, the method returns `null` if there is no current session object. For example:

```
HttpSession thisUser = req.getSession(true);
```

When a new `HttpSession` is created, the server assigns a unique session ID that must somehow be associated with the client. Since clients differ in what they support, the server has a few options that vary slightly depending on the server implementation. In general, the server's first choice is to try to set a cookie on the client (which means that `getSession()` must be called before you write any other data back to the client). If cookie support is lacking, the API allows servlets to rewrite internal links to include the session ID, using the `encodeURL()` method of `HttpServletResponse`. This is optional, but recommended, particularly if your servlets share a system with other, unknown servlets that may rely on uninterrupted session tracking. However, this on-the-fly URL encoding can become a performance bottleneck because the server needs to perform additional parsing on each incoming request to determine the correct session key from the URL. (The performance hit is so significant that many servers disable URL encoding by default.)

To use URL encoding, run all your internal links through `encodeURL()`. If you have a line of code like this:

```
out.println("<a href=\"/servlet/CheckoutServlet\">Check Out</a>");
```

you should replace it with:

```
out.print("<a href=\"");
out.print(resp.encodeURL("/servlet/CheckoutServlet"));
out.println("\>Check Out</a>");
```

When cookies can't be used, the server will encode the session ID into the path by appending `;jsessionid=` and the session ID, like this:^[*]

[*] Earlier versions of the Servlet API left the session encoding algorithm up to the container developer. Later releases standardized the system to prevent portability problems for applications that also use path encoding.

```
<a href="/app/CheckoutServlet;jsessionid=12AB39J">Check Out</a>
```

In addition to encoding your internal links, you need to use `encodeRedirectURL()` to handle redirects properly. This method works in the same manner as `encodeURL()`.^[1]

[1] These methods were introduced in Servlet API 2.1, replacing two earlier methods named `encodeUrl()` and `encodeRedirectUrl()`. This was done to bring the capitalization scheme in line with other Java APIs.

You can access the unique session ID via the `getID()` method of `HttpSession`. This is enough for most applications, since a servlet can use some other storage mechanism (i.e., a flat file, memory, or a database) to store the unique information (e.g., hit count or shopping cart contents) associated with each session. However, the API makes it even easier to hold onto session-specific information by allowing servlets to bind objects to a session using the `setAttribute()` method of `HttpSession`. Once an object is bound to a session, you can use the `getAttribute()` method to retrieve it.

Objects bound using `setAttribute()` are available to all servlets running on the server. The system works by assigning a user-defined name to each object (the `String` argument); this name is used to identify objects at retrieval time. In order to avoid conflicts, the general practice is to name bound objects with names of the form `applicationname.objectname`. For example:

```
session.setAttribute("myservlet.hitcount", new Integer(34));
```

Now that object can be retrieved with:

```
Integer hits = (Integer)session.getAttribute("myservlet.hitcount")
```

[Example 3-7](#) demonstrates a basic session-tracking application that keeps track of the number of visits to the site by a particular user. It works by storing a counter value in an `HttpSession` object and incrementing it as necessary. When a new session is created (as indicated by `isNew()`, which returns `true` if the session ID has not yet passed through the client and back to the server) or the counter object is not found, a new counter object is created.

Example 3-7. Counting visits with sessions

```

import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;

public class VisitCounterServlet extends HttpServlet {

    public void doGet(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException {

        PrintWriter out = resp.getWriter( );
        resp.setContentType("text/html");
        HttpSession thisUser = req.getSession(true);
        Integer visits;

        if(!thisUser.isNew( )) { // Don't check newly created sessions
            visits = (Integer)thisUser.getAttribute("visitcounter.visits");
            if(visits == null)
                visits = new Integer(1);
            else
                visits = new Integer(visits.intValue( ) + 1);
        }
        else
            visits = new Integer(1);

        // Put the new count in the session
        thisUser.setAttribute("visitcounter.visits", visits);

        // Finally, display the results and give them the session ID too
        out.println("<html><head><title>Visit Counter</title></head>");
        out.println("<body>You have visited this page " + visits + " time[s]");
        out.println("since your last session expired.");
        out.println("Your Session ID is " + thisUser.getId( ));
        out.println("</body></html>");
    }
}

```

3.11.1. HttpSessionBindingListener

Sometimes it is useful to know when an object is being bound to or unbound from a session object. For instance, in an application that binds a JDBC `java.sql.Connection` object to a session (something that, by the way, is ill-advised in all but very low traffic sites), it is vitally important that the `Connection` be explicitly closed when the session is destroyed.

The `javax.servlet.http.HttpSessionBindingListener` interface handles this task. It includes two methods, `valueBound()` and `valueUnbound()`, that are called whenever the object that implements the interface is bound or unbound from a session, respectively. Each of these methods receives an `HttpSessionBindingEvent` object that provides the name of the object being bound or unbound and the session involved in the action. The following is an object that implements the `HttpSessionBindingListener` interface in order to make sure that a database connection is closed

properly:

```
class ConnectionHolder implements HttpSessionBindingListener {  
  
    java.sql.Connection dbCon;  
  
    public ConnectionHolder(java.sql.Connection con) {  
        dbCon = con;  
    }  
  
    public void valueBound(HttpSessionBindingEvent event) {  
        // Do nothing  
    }  
  
    public void valueUnbound(HttpSessionBindingEvent event) {  
        dbCon.close( );  
    }  
}
```

Programmers should also be aware of the `HttpSessionListener` interface, which responds to session creation and destruction events.

3.12. Databases and Non-HTML Content

Most web applications need to communicate with a database, either to generate dynamic content or collect and store data from users, or both. With servlets, this communication is easily handled using the JDBC API described in [Chapter 8](#). Thanks to JDBC and the generally sensible design of the servlet lifecycle, servlets are an excellent intermediary between a database and web clients.

Most of the general JDBC principles discussed in [Chapter 8](#) apply to servlets. In our example, we create a database connection within the servlet's `init()` method. Larger applications will generally prefer to use a database connection pool, managed by the servlet container. Connection pools and `DataSource` objects (for container-managed database connections) are discussed in [Chapter 8](#). Another option would be to use the object relational capabilities provided by Hibernate in your servlet. Hibernate is discussed in [Chapter 20](#).

So far, all our servlets have produced standard HTML content. Of course, this is all most servlets ever do, but it's not all that they can do. Say, for instance, that your company stores a large database of PDF documents within an Oracle database, where they can be easily accessed. Now say you want to distribute these documents on the Web. Luckily, servlets can dish out any form of content that can be defined with a MIME header. All you have to do is set the appropriate content type and use a `ServletOutputStream` if you need to transmit binary data. [Example 3-8](#) shows how to pull Adobe Acrobat documents from an Oracle database.

Example 3-8. A servlet that serves PDF files from a database

```
import java.io.*;
import java.sql.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class DBPDFReader extends HttpServlet {

    Connection con;

    public void init(ServletConfig config) throws ServletException {
        super.init(config);
        try {
            Class.forName("oracle.jdbc.driver.OracleDriver");
            con = DriverManager.getConnection("jdbc:oracle:thin:@DBHOST:1521:ORCL",
                                           "user", "passwd");
        }
        catch (ClassNotFoundException e) {
            throw new UnavailableException("Couldn't load OracleDriver");
        }
        catch (SQLException e) {
```

```

        throw new UnavailableException("Couldn't get db connection");
    }
}

public void doGet(HttpServletRequest req, HttpServletResponse res)
    throws ServletException, IOException {

    try {
        res.setContentType("application/pdf");
        ServletOutputStream out = res.getOutputStream( );

        Statement stmt = con.createStatement( );

        /* This is dangerous for production code. Request
           parameters should never be injected into SQL without
           preprocessing. We're keeping syntax simple for this
           example. */
        ResultSet rs = stmt.executeQuery(
            "SELECT PDF FROM PDF WHERE PDFID = " + req.getParameter("PDFID"));

        if (rs.next( )) {
            BufferedInputStream pdfData =
                new BufferedInputStream(rs.getBinaryStream("PDF"));
            byte[] buf = new byte[4 * 1024]; // 4K buffer
            int len;
            while ((len = pdfData.read(buf, 0, buf.length)) != -1) {
                out.write(buf, 0, len);
            }
        }
        else {
            res.sendError(res.SC_NOT_FOUND);
        }
        rs.close( );
        stmt.close ( );
    }
    catch(SQLException e) {
        // Report it
    }
}
}

```

Chapter 4. JavaServer Pages

In [Chapter 3](#), we looked at Java servlets, J2EE's primary mechanism for communicating with web browsers. Servlets are a great technology, but they don't solve every problem faced by web developers, and they introduce a few issues of their own. One major problem is that developing complex HTML-based user interfaces with servlets is time-consuming. Embedding strings of `println()` statements is tedious and error-prone and requires that pages be assembled by a fully qualified programmer who, just perhaps, should be off doing other things. Configuration is also a problem, although it has been simplified a great deal since the original Servlet API. Adding a new servlet to a web application involves editing the deployment XML file and typically reloading the application or restarting the server.^[*] Changes to a servlet create the same issues, turning rapid prototyping into regular prototyping.

[*] Dynamic class reloading and change detection solve some of these problems, but not for all servers and not always reliably.

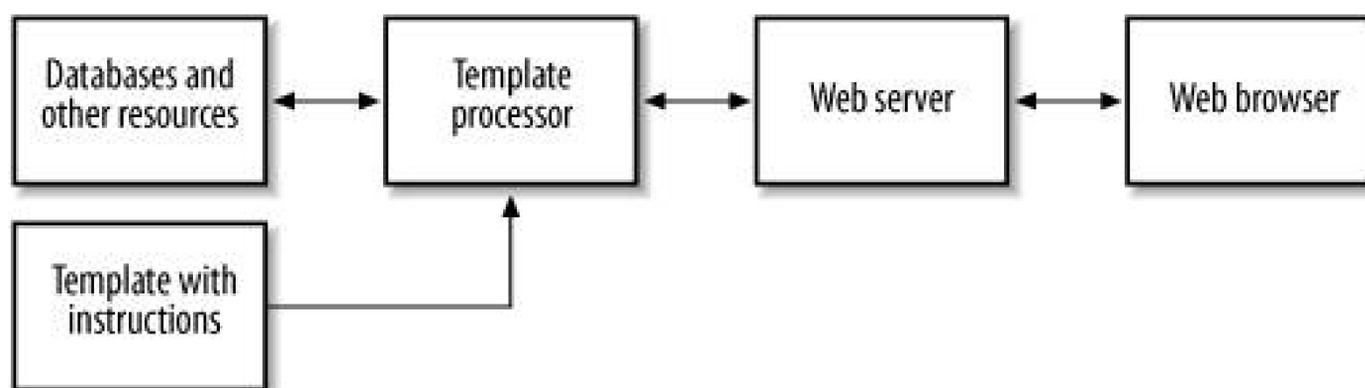
On the design side, servlets can also blend application logic with presentation logic, undoing one of the primary benefits of the client/server architectures that they often replace. In a canonical J2EE environment, where business logic is abstracted into Enterprise JavaBeans or other middleware, this is not much of a concern. But in real life, full J2EE systems aren't always appropriate. Many applications don't need the full weight of an application server, and often developers lack the interest or time to create a full J2EE implementation. And even within a J2EE application, there are substantial benefits to separating flow control (accomplished by servlets) from actual pages.

One obvious approach is to use static HTML content for most pages, reserving servlets for forms processing with a redirect or forward to another HTML page upon completion. Such an approach breaks down fairly quickly, as most pages in a web application have at least some dynamic element. A better solution, however, is to embed dynamic elements directly into the static content. This is the reverse of the approach servlets use, where static content is embedded within the dynamic framework.^[*] Static web pages with a few dynamic content extensions can be edited using any HTML authoring tool and previewed live in the browser, either with or without the dynamic elements (obviously, the efficacy of this last technique depends on the complexity of the page).

[*] Since the dawn of civilization, humankind has sought to embed code within content. Great, if virtual, wars have been fought over the advisability of actually doing so. At this point, very few people consider it wise to embed all of their presentation HTML directly within a servlet, but there is wide divergence of opinion on how much code is appropriate for a JSP. In truth, the answer depends on your application. It's safe to say that the larger or longer lived you expect your application to be, the more likely you will benefit from a clean separation of code and content.

There have been a few different approaches to creating dynamic web pages without a full servlet. Generally, these involve inserting either custom tags or control statements within HTML document templates and then running the documents through a server-side processor that replaces tags and directives with dynamically generated content. The resulting pure HTML is delivered to the client. This type of templating system is shown in [Figure 4-1](#). In a servlet environment, the template processor will be implemented as one or more servlets. Examples of templating engines include WebMacro (<http://www.webmacro.org>) and Velocity (<http://jakarta.apache.org/velocity>).

Figure 4-1. Generic templating engines



Sun's templating solution is called JavaServer Pages, or JSP, which is currently at Revision 2.0. JSP is a highly integrated dynamic content system that is built on top of the Java Servlet API. In fact, JSPs are compiled into Java servlets before being executed by the web server. This allows JSP developers to obtain all the advantages of Java servlets (efficient lifecycles, integration with Java middleware, and so on) without spending hours integrating HTML into actual Java code. JSP also allows page developers to access properties of Java beans without writing code. Finally, custom JSP tags can be created by developers for use by page designers, removing Java code from the JSPs entirely.

Along with the Servlet API, the JSP API is part of the J2EE specification. J2EE 1.4 incorporates JSP 2.0, which was released in 2003. Earlier versions of J2EE supported JSP 1.2. We will indicate changes when possible, but this chapter focuses on the 2.0 version of the standard.

JSP is also closely integrated with the JavaServer Faces and Struts application frameworks, which are described in [Chapters 5](#) and [19](#), respectively. In both systems, JSPs provide the user interface, collaborating with extensions provided by the application framework. Other parts of this book that you may find especially relevant include [Chapter 2](#), where we provide details on assembling and deploying web components, and [Chapter 21](#), where we discuss code annotation systems that can be used to automatically generate deployment descriptors and other artifacts for web applications.

In this chapter, we are going to look at JSP from a Java programmer's perspective as opposed to that of a web site designer. Of course, one of the great things about JSPs, and other templating technologies, is that you don't have to be a programmer to use them. However, for a more thorough introduction to JSP, as well as some substantial additional detail for experienced developers, we suggest *JavaServer Pages* by Hans Bergsten (O'Reilly).

4.1. JSP Basics

The JSP specification is part of the full J2EE specification, and JSP engines are included with most major servlet engines. The Tomcat 5.5 server from the Apache project is the reference implementation for both the Servlet 2.4 and JSP 2.0 specifications, and is freely available. For information on downloading and installing Tomcat, see <http://jakarta.apache.org/tomcat>. The rest of this chapter assumes some familiarity with the concept of a web application and servlets. [Chapter 3](#) covers servlets and web applications in more detail. For now, it is enough to know that a servlet is a compiled Java class that responds to a web request, and a web application is a collection of servlets and other resources.

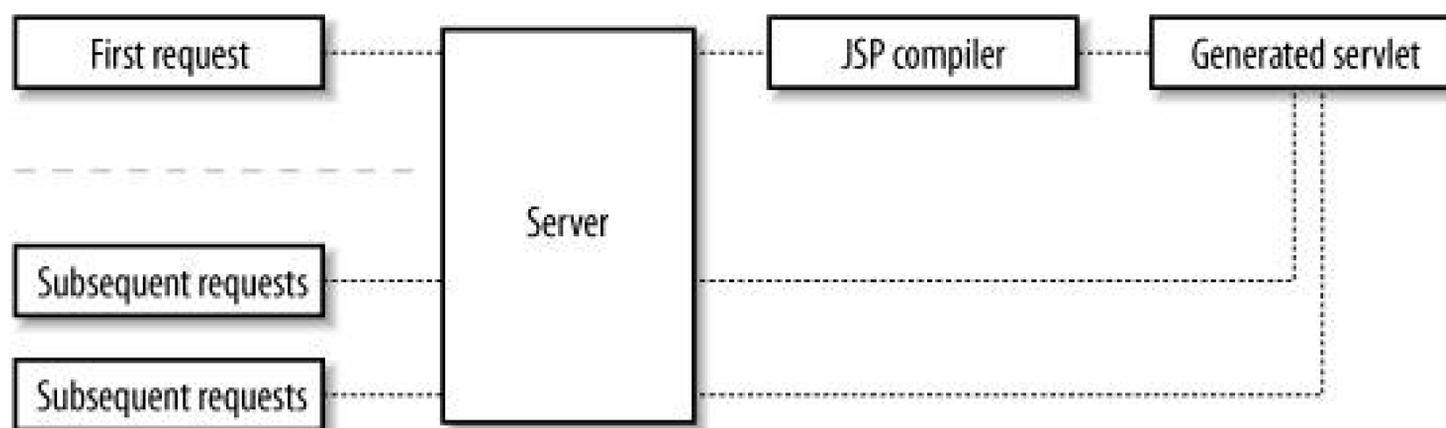
Since the JSP architecture is based on the servlet architecture, JSP support within a web server provides a translation layer. Individual JSPs are text files stored on the web server and accessed via their real path. For example, if a JSP named *index.jsp* resides at the root of the "enterprise" web application, it would be accessed by a request to <http://localhost:8080/enterprise/index.jsp>, assuming that the web server is running on port 8080 of localhost. When the JSP is first requested, the JSP engine uses the JSP file to generate the source code for a servlet. The generated source is then compiled, installed into the servlet engine, and used to service the request. Once the JSP has been compiled, the compiled version is saved and used to service additional requests according to the standard servlet lifecycle. [Figure 4-2](#) shows the process in more detail.

When the JSP file is modified, the server detects the change and rebuilds the corresponding servlet. The compilation phase imposes a slight delay the first time the page is retrieved. If necessary, many servers allow precompilation of JSPs to get around this problem.

Now is a good time to look at a simple JSP, so here's an example that displays the current date:

```
<html>
<body>
  Hello, visitor. It is now <%= new java.util.Date( ).toString( ) %>
</body>
</html>
```

Figure 4-2. JSP lifecycle



This looks easy enough. The only part of the example that is not regular HTML is the `<% %>` tag. JSP elements come in two forms: the simple `<% %>` entity, used for directly embedding Java code and issuing JSP specific directives, and XML style "action tags," which we'll see a little later. The `<%= %>` tag is an *expression* tag, which inserts the value of a single Java expression into the page's output. For space reasons, we're not going to include the Java code produced by the system, but if you want to look at it, you can find it in the web server's working areas. For Tomcat 5, the path is `/<tomcat_path>/work/Catalina/<servername>/<webappName>`. [Figure 4-3](#) shows the output sent to the client browser.

Figure 4-3. Output from date display JSP

Now, let's try something a little more complicated. The `<% %>` element can insert regular Java code, which can control the flow of the page. The following example checks whether it's before or after noon (we'll count nighttime as the afternoon for the purpose of this sample) and displays an appropriate greeting:

```

<html>
<body>
<%
java.util.Date theDate = new java.util.Date( );
%>
<% if (theDate.getHours( ) < 12) { %>
Good morning,

```

```

<% } else { %>
Good afternoon,
<% } %>
  visitor. It is now <%= theDate.toString( ) %>
</body>
</html>

```

Two things should be noted about this example. The first is the declaration of a local variable in the first pair of `<% %>` tags. Unlike the `%=` tags, nothing is displayed. The next JSP element includes a Java `if` statement that operates on the variable. Note that we don't finish the `if` statement in the JSP tag, but return to template text. This text will be displayed if the `if` clause evaluates to `true`. The next JSP element closes the `if` and begins an `else`. It is followed by the text to be displayed in the `else` case, and a final JSP element to close the `else`. The rest of the file is the same as in the first example, except we use the `Date` object created earlier rather than instantiating a new one.

When viewed in the afternoon, the JSP-generated servlet produces HTML that looks like this:

```

<html>
<body>

Good afternoon,

  visitor. It is now Mon Dec 24 14:50:39 EST 2007
</body>
</html>

```

Note that there is some whitespace where some of the JSP tags were. This is because we placed whitespace around them in the source JSP file. The JSP processor removes the tags themselves, but has no way of determining whether it should also remove the carriage returns that follow them. This is not a problem for HTML, since the web browser compresses all whitespace characters into a single space character, but you should be aware of this behavior when debugging or producing plain-text output.

4.1.1. Directives and Declarations

JSP also allows the insertion of *JSP directives*, which control behavior for the entire page. Directive tags begin with `<%@` rather than `<%`. There are three directives. The first, `include`, simply includes a static file into the page output. So to include a copyright notice at the bottom of each JSP, you can just add the tag:

```

<%@ include file="copyright.html"%>

```

The second directive, `page`, is more flexible. It defines parameters that relate to the current JSP and is valid across all invocations of that page. [Table 4-1](#) displays the possible attributes of the page

directive.

Table 4-1. JSP page directive values

Name	Description
<code>autoFlush</code>	Set to <code>true</code> or <code>false</code> , identifying whether the page buffer should be flushed when full. The default is <code>true</code> .
<code>buffer</code>	Set the buffer size, in kilobytes. The default is <code>8kb</code> , and values must be in the format of <code>number kb</code> or <code>none</code> .
<code>contentType</code>	Set the MIME type or MIME type and charset. The default is <code>text/html</code> .
<code>errorPage</code>	An optional path to a custom error-handling page.
<code>extends</code>	An optional fully qualified class name of a class implementing <code>javax.servlet.jsp.JspPage</code> or <code>javax.servlet.jsp.HttpJspPage</code> .
<code>import</code>	A comma-separated list of Java classes or packages to import.
<code>info</code>	Text describing this JSP. The text entered here can be used by a server administration tool.
<code>isErrorPage</code>	Identifies this page as an error page, making the exception object (see next section) available to script elements.
<code>isThreadSafe</code>	Set to <code>true</code> or <code>false</code> , indicating whether this JSP should implement the servlet <code>SingleThreadModel</code> .
<code>language</code>	The scripting language for this page. The default is <code>java</code> , and the JSP specification doesn't require support for any other language, although some application servers support JavaScript.
<code>session</code>	Set to <code>true</code> or <code>false</code> depending on whether the page should participate in user sessions. The default is <code>TRue</code> .

The final directive, `taglib`, is discussed later in this chapter.

JSP supports another tag sequence that globally affects all instances of a page. This is the declarator element, which begins with `<%!` and declares a global variable within the JSP. Unlike variables declared in a standard `<% %>` tag pair, variables declared with `<%!` will be available for all invocations of the JSP. For example:

```
<%! int globalHitCounts = 0; %>
This page has been accessed <%= ++globalHitCounts %> times.
```

However, use of declarations introduces thread safety and lifecycle issues. For example, there is no guarantee that the JSP will not be rebuilt and reloaded. If this happens, the hit count will return to 0.

Therefore, for most applications, you should use other approaches for maintaining global data.

4.1.2. Built-in Objects

In addition to objects declared within a JSP file itself, Java code running within a JSP has access to a set of classes provided by the web server. These classes allow for communication between JSPs, interaction with the JSP container, and support for sessions and elaborate output. In servlets, for instance, request parameters are retrieved from the browser via the `getParameter()` method of an `HttpServletRequest` object. JSPs can do the same thing. The following JSP fragment uses the built-in request object (which maps to `HttpServletRequest`) and the `out` object (which provides a `PrintWriter`) to echo the form parameters submitted by the client):

```
<ul>
<%
  java.util.Enumeration e = request.getParameterNames( );
  while(e.hasMoreElements( )) {
    String name = (String)e.nextElement( );
    out.println("<li>" + name + ":" +
      request.getParameter(name));
  }
%>
</li>
```

[Table 4-2](#) lists the objects available to a JSP.

Table 4-2. Objects available in JSPs

Name	Type	Description
<code>application</code>	<code>javax.servlet.ServletContext</code>	The servlet context for this web application.
<code>config</code>	<code>javax.servlet.ServletConfig</code>	The <code>ServletConfig</code> object associated with this page.
<code>exception</code>	<code>java.lang.Throwable</code>	For error pages only, the exception that triggered the error.
<code>out</code>	<code>javax.servlet.jsp.JspWriter</code>	A <code>PrintWriter</code> subclass that writes to the page's output stream.
<code>page</code>	<code>java.lang.Object</code>	The implementation class instance currently processing the request. When Java is the scripting language, <code>page</code> is synonymous to <code>this</code> .

Name	Type	Description
<code>pageContext</code>	<code>javax.servlet.jsp.PageContext</code>	The page context for this JSP.
<code>request</code>	<code>javax.servlet.ServletRequest</code> or <code>javax.servlet.http.HttpServletRequest</code>	The protocol-specific request object for this request.
<code>response</code>	<code>javax.servlet.ServletResponse</code> or <code>javax.servlet.http.HttpServletResponse</code>	The protocol-specific response object for this request.
<code>session</code>	<code>javax.servlet.http.HttpSession</code>	For pages retrieved over HTTP, the current user's <code>HttpSession</code> object.

4.1.3. Sharing Data Between JSPs, Other JSPs, and Servlets

HTTP is a stateless protocol, and by extension, the Web is inherently a stateless environment. If a client makes a series of connections to a web server, there is no built-in foolproof mechanism for the web server to associate the sequential requests with the same user. Similarly, a JSP or servlet will treat each request individually, regardless of whether the user has made other requests in the recent past.

The simplest form of communication between JSPs is a hyperlink. A JSP produces an HTML-formatted link, which the user can then click on to request another JSP. HTTP allows name=value parameters to be associated with an HTTP request. So, for example, if JSP 1 needs to inform JSP 2 of the user's language, it might use a JSP fragment like this one to produce a link:

```
<a href="page2.jsp?language=<%= userLanguage %>">Go to page 2</a>
```

This HTML fragment includes a JSP expression that passes a variable called `userLanguage` as a request parameter to `page2.jsp`, which can then retrieve it from the session.

This is a clumsy approach, however, for all but the simplest applications. For one thing, as soon as the user accesses a page that doesn't pass the language parameter on, such as a static HTML page or a link off-site, the information is gone and must somehow be retrieved again. A better place to store this information is the user's session, accessible via the session object made available to all JSPs served over HTTP. For more on the session object, see [Chapter 5](#). Information that is global to the entire application can be stored in the `ServletContext` object, which is accessible via the application object.

All information loaded into the session and application contexts will be available to any JSP that participates in the session or is part of the application. Servlets within the application will also be able to access the data. In the next section, we'll look at some other ways of using these mechanisms to share data across JSPs.

4.2. JSP Actions

If you've spent any time working with servlets, our examples up to this point really haven't been anything new. Essentially, we've replaced reams of `println()` calls with template text in a JSP. This certainly saves time, but we aren't doing anything different. For one thing, at the beginning of this chapter, we promised that JSP allowed nonprogramming web designers and content creators to help create dynamic content. The techniques we've just seen let you do that, but you have to teach them Java first, which pretty much defeats the purpose.

HTML developers may not be confident tackling server-side Java code, but they're certainly comfortable with markup tags. Web browsers, after all, just treat tags as instructions. If a browser sees a `` tag, it turns the running text to boldface until it sees another `` tag. Scriptlets in JSP do the same thing, except in two steps: the server processes the script, possibly producing more HTML, and the browser then views it. This is not a difficult concept, but we haven't gotten around the fact that the first set of instructions is provided as Java code and hence requires a Java programmer with some time on her hands.

JSP solves this problem with *action tags*. An action tag looks like a regular HTML tag and doesn't follow the `<% %>` syntax conventions we've seen before. JSP actions are divided into two categories: built-in functions, which we'll discuss in this section, and custom tags, which we'll discuss later in the chapter. Built-in JSP actions provide access to many of the features we've already seen with scriptlets for instance, there is another type of `include` we can perform with actions:

```
<jsp:include page="/headers/header.jsp"/>
```

If you think this looks like an XML tag, you're right: the syntax requirements are the same as for namespace-enabled XML. The `jsp` namespace is reserved for JSP built-in actions (we'll see the real value of this in the next section). The tag name is specified after the namespace and the action to be performed. Tags can then have zero or more attributes (such as `page`, `name`, `id`). Tags can be closed with a standard closing tag (such as `</jsp:include>`) or with a `/` before the final `>`, for tags that don't have any elements within them.

The difference between the `<%@ include %>` directive and the `<jsp:include>` action is that the directive happens once at compile time for the page while the action happens at request time, each time the page is processed for a client.

The `<jsp:include>` tag itself includes the contents of the specified relative URL (it must be from the same server) in the output of the current page.^[*] Parameters for the included page can be included by nesting a `<jsp:param>` tag into the include:

[*] In pure servlets, this is the functionality handled by the `RequestDispatcher` object.

```
<jsp:include page="/headers/header.jsp">
  <jsp:param name="company" value="The Company"/>
```

```
</jsp:include>
```

The `<jsp:forward>` tag works similarly, but transfers control to the destination page rather than including its content:

```
<jsp:forward page="/login/access.jsp" />
```

or, with parameters:

```
<jsp:forward page="/login/access.jsp">
  <jsp:param name="redirect" value="<%= request.getRequestURI( )%>" />
</jsp:forward>
```

The preceding example uses a JSP expression to generate the parameter value. One caution on `<jsp:forward>`: if any output has been written to the client (either because the output buffer was flushed or because buffering was turned off via a page directive), an attempt to use `<jsp:forward>` produces an `IllegalStateException`.

4.2.1. JSP Tags and JavaBeans

JSP actions combine with Java beans to provide data to page designers. Programmers can write a Java bean that handles data access and business logic (including accessing Enterprise JavaBeans) and make that information available to a JSP, without embedding all of the necessary JDBC (or whatever) code within the JSP itself.

As a Java class, Java beans can be instantiated in scriptlets. If we want to use a bean named `ProductBean` in the `com.company` package, we could write a JSP like this:

```
<% com.company.ProductBean product =
new com.company.ProductBean( );
product.setProductId("DH2309-AX");
%>
```

The `ProductBean`, after setting the product ID, retrieves the necessary product information, which is exposed via other properties. So if the bean has a `Price` property, we could retrieve it with:

```
<%= product.getPrice( ) %>
```

However, JSPs offer another way to access the bean, using the `<jsp:useBean>` tag:

```
<jsp:useBean id="product" class="com.company.ProductBean" />
```

It's that simple. Once the bean is created, properties can be set using the `<jsp:setProperty>` tag:

```
<jsp:setProperty name="product" property="ProductID" value="DH2309-AX" />
```

The `<jsp:setProperty>` tag can be nested within the `<jsp:useBean>` tag or used later. Properties are retrieved in the same way, using the `<jsp:getProperty>` tag:

```
<jsp:getProperty name="product" property="Price" />
```

There's one more trick to the `<jsp:setProperty>` tag. If the value attribute is not included, the JSP will search the incoming request parameters for a name=value pair with the same name as the bean property currently being set and will use the value from the request. For example, the request <http://shop.company.com/schlock/product.jsp?ProductId=DH2309-AX> has a `ProductId` parameter containing the ID for the product we want to display. If the `product.jsp` file contains these lines:

```
<jsp:useBean id="product" class="com.company.ProductBean" />
<jsp:setProperty name="product" property="ProductId" />
```

the `ProductId` property is set to DH2309-AX. For further convenience, we can use the special property name (*), which populates every property of the bean from the request parameters.

4.2.2. Scoping Beans

The Java bean used in the last example existed within the scope of the current page. This means that it was created when the client requested the page and destroyed when the page was destroyed. For some applications, this is fine: a Java bean that retrieves and displays information about a particular product in an online catalog doesn't need to hang around, although it should use efficient strategies to speed its interior execution.^[*] But what about a bean that retrieves information about the current user—display preferences, identity, and so on? This information is used by every page the user accesses, but doesn't change except in very specific situations. We want to create the bean once and store it in the user's session, where it can remain until the user leaves the site.

[*] For instance, if the bean retrieves product information from a SQL database, it should use a connection-pooling strategy, possibly via JDBC 2.0, to avoid the overhead of creating a new connection for each database access.

One way to accomplish this is to use a scriptlet to add the bean directly to the user's session using the implicit JSP session object and then to retrieve it manually for each page that uses it. However, the `<jsp:useBean>` tag offers a way around this, via the `scope` argument, which will apply to every instance of that particular Java bean class with the specified ID. By default, beans have a *page* scope, but they also can be set to have a *user* scope, so the same instance is used whenever a `<jsp:useBean>` tag is invoked for that class and ID. The *application* scope goes even further, applying

to the web application as a whole by inserting the bean instance into the `ServletContext` object associated with the web application. This allows servlets to access beans created by JSPs and vice versa.

The other available scope is *request*. Elements scoped to the request level will be available for the duration of the current request. This means they can be accessed by other JSPs referenced via `<jsp:include>` or `<jsp:forward>`.



4.3. The JSP Expression Language

JSP 2.0 introduced the JSP Expression Language (the EL), which is designed to make it easier for page authors to access and write logical tests. Expressions written in the EL can be embedded into static text or passed into JSP tag parameters. In both cases, the JSP engine evaluates the expression and embeds the result in the page.



Everything in this chapter up until this point will work with JSP 1.2 or JSP 2.0. The JSP EL is available only in JSP 2.0.

EL expressions begin with a `$` and are enclosed by curly bracket characters. Here's a simple example, evaluating the expression `3 > 4` (to false, of course) and also doing the sum:

```
<%@ page contentType="text/html; charset=UTF-8" language="java" %>
<html>
  <body>
    Is three greater than four? That's ${3 > 4}.
    The sum is ${3+4}, though.
  </body>
</html>
```

Expressions can also retrieve values from the page, request, session, and application scopes, in that order. Let's say that you have the following Java bean:

```
public class Product {
    private String name = null;
    private double price = 0.0d;

    public String getName( ) {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

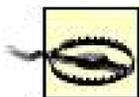
    public double getPrice( ) {
        return price;
    }

    public void setPrice(double price) {
```

```

        this.price = price;
    }
}

```



For the purpose of this example, we've used a `double` to represent the item price. We've done this to keep the examples short, and since we're only displaying data it doesn't cause any problems. However, you should not do it in a real system since `floats` and `doubles` are not accurate for certain kinds of calculations: try writing a program that multiplies 59999 by .01 and you'll see that the answer is not quite 599.99. A better option for dealing with currency values is the `java.math.BigDecimal` class.

We can load the object into the request scope with a servlet like this (note that the servlet isn't responsible for generating any HTML content, just for creating and packaging the `Product` object):

```

import javax.servlet.http.*;
import javax.servlet.ServletException;
import java.io.IOException;

public class ProductServlet extends HttpServlet {
    protected void doGet(HttpServletRequest request,
                        HttpServletResponse response)
        throws ServletException, IOException {
        Product product = new Product( );
        product.setName("Hyperwidget");
        product.setPrice(599.99d);
        request.setAttribute("product", product);
        request.getRequestDispatcher("/product.jsp")
            .forward(request, response);
    }
}

```

Now that the object is in the request scope, we can access it via the EL within the `product.jsp` page:

```

<%@ page contentType="text/html; charset=UTF-8" language="java" %>
<html>
  <head><title>Product Description</title></head>
  <body>
    <h1>${product.name}</h1>
    Price: $$${product.price}
  </body>
</html>

```

The only trick here is that we included two `$` characters before `{price}`: one to display to the user and one to indicate the start of the expression.

All of the built-in objects described in [Table 4-2](#) are available within the EL. For instance, to display the application context page, you can use the expression `${pageContext.request.contextPath}`, which pulls the `HttpServletRequest` object from the `PageContext` object and evaluates the `contextPath` bean property.

The EL also allows retrieval of values from maps using the following syntax:

```
mapname[ 'key' ]
```

Using the built-in `param` convenience object (which contains the form parameters for the current request), the standard "Hello, Joe" application looks like this:

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<html>
  <head><title>Generic Seasonal Greetings!</title></head>
  <body>Hello, ${param['name']}!</body>
</html>
```

Accessing this JSP via the URI [/context/hello.jsp?name=Joe](#) provides an appropriate greeting to Joe.

4.3.1. Using the EL with Tags

JSP EL expressions can also be used with JSP tag attributes. Simply replace the tag literal with the EL expression. For example, we can use the `<c:if>` tag (which we'll look at a little more in the next section) along with an EL expression to display special shipping information if the cost of an item on our product page is more than, say, \$400:

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core"
  prefix="c" %>

<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<html>
  <head><title>Product Description</title></head>
  <body>
    <h1>${product.name}</h1>
    Price: $$${product.price}

    <c:if test="${product.price > 400}">
      <br/><font color="red">Eligible for Saver Shipping!</font>
    </c:if>
  </body>
</html>
```

Of course, we really shouldn't have the logic for Saver Shipping eligibility embedded in the JSP itself. For more information on techniques for abstracting business logic away from content, see the chapter on Struts ([Chapter 19](#)), the chapter on JavaServer Faces ([Chapter 5](#)), or the book *J2EE Design Patterns* by William Crawford and Jonathan Kaplan (O'Reilly).



4.4. JSP Standard Tag Library

Many developers criticized the original JSP specification because it lacked some important basic capabilities. As we saw earlier in the chapter, you can implement complex logic within JSPs by embedding scriptlets. This approach works (at least, it's functional and produces web pages that look about right), but it makes JSPs look a lot less like HTML and a lot more like Java code. As a result, the range of people who can work on them is sharply limited. To make things even worse, because developers were virtually forced into embedding large amounts of Java code into each page (at least if the page were in any way complex), business logic had a tendency to creep down into the presentation layer.

The JSP EL removes some of the need for scriptlets (you can now display a variable without embedding Java code directly) but doesn't help implement page logic. That's where the JSP Standard Tag Library (JSTL) comes in. JSTL is not directly part of the JSP standard, but it's an associated standard, and JSTL Version 1.1 (which is compatible with JSP 2.0) is also included in J2EE 1.4. In this section, we'll discuss the most commonly used JSTL tags for variable handling, flow control, and internationalization.

The JSTL tags are easy to incorporate into your web applications. J2EE 1.4-compatible containers include all the necessary jar files. Standalone servlet/JSP containers, such as Tomcat, don't, so you'll need to obtain both the *jstl.jar* file and an implementation of the tags themselves separately. The reference implementation for JSTL grew out of the Jakarta taglibs project, and can be downloaded from <http://jakarta.apache.org/taglibs/index.html>. Put both jar files (the implementation from Jakarta is in *standard.jar*) into your *WEB-INF/lib* directory.

Once the JSTL tags are available to the web application, you need to map them to a namespace for each JSP. This is easy enough; just add the following line to the top of each page:

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core"
    prefix="c" %>
```

By convention, the JSTL tags are given the prefix `c`.

4.4.1. Variables and Flow Control

The `<c:set>` and `<c:remove>` tags provide basic variable handling for JSPs. In general, these tags should be used carefully: you very rarely need to manipulate variables within the JSP itself, other than possibly for page layout purposes, and in most cases the EL will suffice. But if you must, the following JSP file retrieves a variable from the request parameter (using the built-in `param` object and an EL expression), sets it to a session variable, displays the variable, and then removes the variable from the session:

```

<%@ taglib uri="http://java.sun.com/jsp/jstl/core"
    prefix="c" %>
<%@ page contentType="text/html; charset=UTF-8" language="java" %>

<html>
    <c:set var="username" scope="session" value="{param.username}"/>
    <c:out value="{username}" escapeXML="false"/>
    <c:remove var="username"/>
</html>

```

This JSP also uses the `<c:out>` tag to display the variable. This tag was, before JSP 2.0, the only nonscriptlet method to display a variable in a JSP. Now that we have the EL, the use of `<c:out>`, which is much clunkier, can be reduced but not quite eliminated. Unlike a regular EL expression, the `<c:out>` tag escapes all XML characters into XML/HTML-safe entities. The `>` character becomes `<`; and so on. If you expect to display variables that might include HTML or XML reserved characters, you should use `<c:out>` rather than embedding the bare expression into the page.

4.4.1.1. Flow control

JSTL includes tags for loops and conditional logic, freeing the page author from the tyranny of scriptlets. We saw one of these tags, `<c:if>`, in the previous section. The `<c:if>` tag evaluates an EL expression to true or false and, if true, processes the JSP code contained within the tag.

There is no corresponding `else` tag you need to use another `<c:if>` with an inverse condition or, if there are multiple conditions, the `<c:choose>` tag. In the following extremely contrived example, we display one message if a variable is 1 or below, another if it is 2 or 3, and a final message if it's greater than 3. The first two determinations are made using the `<c:when>` tag, which evaluates a conditional within a `choose` block, and the final message is displayed using a `<c:otherwise>` tag, which provides an optional fall-through action:

```

<%@ taglib uri="http://java.sun.com/jsp/jstl/core"
    prefix="c" %>

<%@ page contentType="text/html; charset=UTF-8" language="java" %>

<c:set var="niftylevel" value="5"/>

<c:choose>
    <c:when test="{niftylevel < 2}">
        Not terribly nifty.
    </c:when>
    <c:when test="{niftylevel > 1 and niftylevel < 4}">
        Nifty, but could be niftier.
    </c:when>
    <c:otherwise>
        Remarkably nifty.
    </c:otherwise>
</c:choose>

```

Processing stops when a condition is met so setting the `niftyLevel` variable to 3 after printing out the first message does not cause the second message to print.

Finally, JSTL includes a very robust loop tag, `<c:forEach>`. This tag accepts any standard `Collection` implementation, any standard `Map` implementation, arrays of primitive types, and even strings containing comma-separated values.^[*]

[*] `Iterator` and `Enumeration` objects are also supported, but with limits, since both types of objects can be processed only once.

In this example, the request attribute `catalog` contains an `ArrayList` holding a set of `Product` objects. The JSP works through the list, creating a temporary page variable named `product` and printing out each list entry.

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core"
    prefix="c" %>

<%@ page contentType="text/html; charset=UTF-8" language="java" %>
<html>
  <head><title>Widget Catalog</title></head>
  <body>
    <ul>
      <c:forEach var="product" items="{catalog}">
        <li>${product.name}: ${product.price}</li>
      </c:forEach>
    </ul>
  </body>
</html>
```

4.4.1.2. URL management

JSTL provides basic support for URL management. This functionality is important if you are using URL rewriting to track sessions and can also simplify page design if you're assembling complex URLs. The `<c:url>` tag simply accepts a (content-relative) URL and an optional set of request parameters and displays a properly formatted URL.

For example, in our catalog example, we could replace the contents of the `<c:forEach>` tag with the following code to produce a URL that would link to a product detail page, passing in the product's Stock Keeping Unit (SKU) code as a URL parameter named `productID`:

```
<c:url var="producturl" value="/product" >
  <c:param name="productID" value="{product.SKU}" />
</c:url>
<li><a href="{producturl}">${product.name}</a> - ${product.price}</li>
```

4.4.2. Internationalization

JSTL includes tags that support the Java internationalization standards. Unlike the tags we've looked at so far, the internationalization formatting tags reside in the `fmt` package rather than the `core` package.

Java's internationalization functions, in a proverbial nutshell, let developers move the display constants from an application to an external properties file, also known as a *resource bundle*. The properties file can then be translated into other languages, making it easy to write multilingual applications. The strings are kept in the standard properties file format:

```
first.name=First Name
last.name=Last Name
```

To use these two properties in a JSP file, we first make the properties file accessible to the web application by placing it in the classpath. If we name the file *messages.properties* and place it in *WEB-INF/classes*, we have created a resource bundle named `messages`. To provide other languages, we provide other files with the same base name followed by the two-character language code for the language in question. A German language resource bundle would be named *messages_dk.properties* and an explicitly English language bundle would be called *messages_en.properties*.

The `<fmt:message>` and `<fmt:bundle>` tags handle message display. Consider the JSP in [Example 4-1](#).

Example 4-1. message.jsp

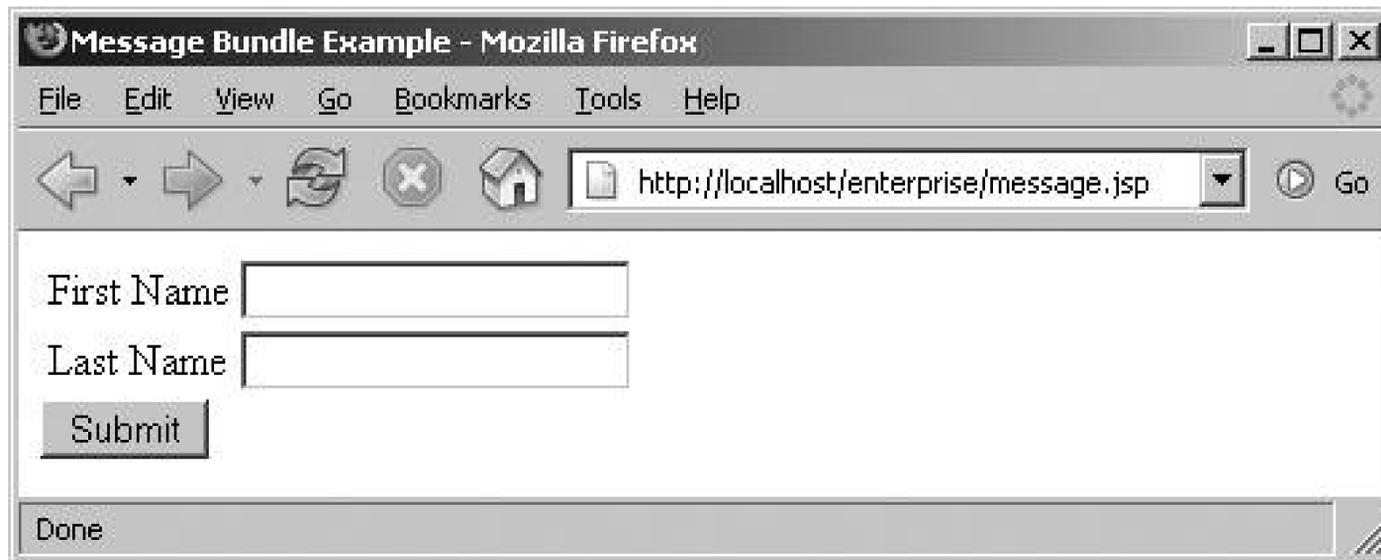
```
<%@ taglib uri="http://java.sun.com/jsp/jstl/fmt"
    prefix="fmt" %>

<%@ page contentType="text/html; charset=UTF-8" language="java" %>
<html>
  <head><title>Message Bundle Example</title></head>
  <body>
    <form>
      <fmt:bundle basename="messages">
        <table>
          <tr><td>
            <fmt:message key="first.name"/>
          </td><td><input type="text" name="firstname"></td></tr>
          <tr><td>
            <fmt:message key="last.name"/>
          </td><td><input type="text" name="lastname"></td></tr>
        </table>
      </fmt:bundle>
      <input type="submit" value="Submit">
    </form>
  </body>
```

```
</html>
```

The `<fmt:bundle>` tag sets the message bundle using the `basename` attribute. The two `<fmt:message>` tags retrieve the messages from the resource bundle by name. The resulting page looks like [Figure 4-4](#).

Figure 4-4. Internationalized screen (in English)



It's possible to skip the `<fmt:bundle>` tag, either by using the `<fmt:setBundle>` tag (which sets the bundle for the entire page) or by including a `bundle` attribute on the `<fmt:message>` tag itself.

4.5. Custom Tags

JSP and Java beans provide a nice way to abstract data access away from the page itself, but it still doesn't get code entirely out of the system. Page designers still must use script elements for control functions such as loops or branching logic. In addition, Java beans sometimes aren't enough to fully encapsulate a web application's internal logic, leaving large chunks of code embedded in JSPs. And of course, there's the matter of code reuse. While utility methods can be abstracted out into classes and made available to JSPs, scripting code embedded within a particular JSP tends not to lend itself to abstraction.

The JSP specification addresses this by allowing the creation of customtags. Tags are organized into tag libraries, which can be loaded by pages as needed. Each tag library gets its own namespace and can be shared by as many pages as necessary. Properly designed custom tags can cut most, if not all, of the remaining pure Java code out your JSPs. We've already seen tags in action, since JSTL is just a collection of JSP tags that are a standard part of the environment.

Imagine a simple tag that displays the current date and time. Let's call the tag `Date`. After creating and configuring it, we can access it within a JSP page:

```
<enterprise:Date/>
```

Note that the namespace is no longer `jsp`. Instead, we have a custom specified namespace that identifies a *tag library*, or *taglib*, containing a set of custom tags. We declare the tag library at the beginning of the JSP file with the `<%@ taglib %>` directive:

```
<%@ taglib
    uri="http://oreilly.com/enterprise/examples-taglib"
    prefix="enterprise"%>
```

The taglib containing the `Date` tag is identified by a specific URI. First, the server looks in the *web.xml* file for a mapping from the URI to a taglib file in the web archive. If there is not a mapping, then the server tries to find the taglib using the URI as a direct path. A taglib mapping is made in the *web.xml* file using a `<taglib>` element, like so:

```
<taglib>
<taglib-uri>http://oreilly.com/enterprise/examples-taglib</taglib-uri>
<taglib-location>
    /WEB-INF/jsp/enterprise-taglib.tld
</taglib-location>
</taglib>
```

The `<taglib-location>` tag points to a `.tld` file, which is an XML-formatted tag library deployment descriptor. Here's the `.tld` file that describes a single tag library:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<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">
  <description>A simple tag library for JEIAN</description>
  <tlib-version>1.0</tlib-version>
  <short-name>enterprise</short-name>
  <uri>http://oreilly.com/enterprise/examples-taglib</uri>
  <tag>
    <description>Display current date</description>
    <name>Date</name>
    <tag-class>com.oreilly.jent.jsp.DateTag</tag-class>
    <body-content>empty</body-content>
  </tag>
</taglib>
```

The key thing here is the `<tag>` tag with its subtags, `<name>` and `<tag-class>` (descriptions are optional). The `<name>` tag specifies the name that JSPs will use to refer to the tag itself, and the `<tag-class>` specifies the Java class that implements the tag functionality. In this case, it's a class named `DateTag` in the `enterprise` package. This class file must be available to the web application.

By this point, we've described a tag, put it in a library, associated the library with the web application, retrieved the library from the web application for use in a JSP page, mapped it to the `enterprise` namespace for the page, and invoked the tag itself. The only thing remaining is the actual implementing class.

4.5.1. Implementing Custom Tags

Custom tags come in two basic varieties: basic tags and body tags. Basic tags don't access their bodies (the content between the open and close tags), although they can control whether or not the JSP servlet processes their content. Body tags are just like basic tags, but have access to their body content. Support interfaces and classes for all custom tags are in the `javax.servlet.jsp.tagext` package.

Basic tags implement the `Tag` interface, which has two methods of interest to the tag developer: `doStartTag()` and `doEndTag()`. The `doStartTag()` method is called when the page processing reaches the opening of the tag (`<enterprise:Date>` in the previous example), and the `doEndTag()` method is called when execution reaches the end tag (`</enterprise:Date>`). These are called sequentially for empty tags.

The `doStartTag()` method can return `Tag.SKIP_BODY` or `Tag.EVAL_BODY_INCLUDE`. Skipping the body

causes execution to jump right to the end of the tag. Evaluating the body will cause the JSP to process the body normally, as if it were outside of the tag. The `doEndTag()` method can return `Tag.EVAL_PAGE` or `Tag.SKIP_PAGE`. If `EVAL_PAGE` is returned, the rest of the page will be processed; if `SKIP_PAGE` is returned, processing stops.

Here, finally, is the `DateTag` class. It extends the `TagSupport` class, which implements the `Tag` interface and provides stub versions of both `do-` methods, as well as the other methods in the interface (which are used by the JSP to provide resources to the tag). It implements the `doEndTag()` method:

```
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;

public class DateTag extends TagSupport {

    public int doEndTag( ) throws JspException {

        try {
            pageContext.getOut( ).println((new java.util.Date( )).toString( ));
        } catch (java.io.IOException e) {
            throw new JspException(e);
        }
        return Tag.EVAL_PAGE;
    }
}
```

The `pageContext` object is an instance of `PageContext` and is provided by the `TagSupport` class. The `PageContext` object provides access to the current output writer (via `getOut()`), as well as the `ServletRequest` and `ServletResponse` objects for the request, page attributes, `ServletConfig` and `ServletContext` objects, and the `HttpSession` object.

Shown below is another `TagSupport`-based tag, which is a little fancier. The `<enterprise:Evening>` tag encloses content and displays it only if the current server time is not within the workday. Here's how it works in a JSP:

```
<enterprise:Evening dayStart="9" dayEnd="17">
Sorry, we're closed for the day. Please try us tomorrow.
</enterprise:Evening>
```

The code is simple, but this time we extend the `doStartTag()` method instead and return `SKIP_BODY` if the current time is within the working day:

```
import java.util.*;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;

public class EveningTag extends TagSupport {
```

```

private int dayStart = 9;
private int dayEnd = 17; // 24 hour time
// In production, we should propagate errors with exceptions
public void setDayStart(String value) {
    dayStart = Integer.parseInt(value);
}
public void setDayEnd(String value) {
    dayEnd = Integer.parseInt(value);
}

public int doStartTag( ) throws JspException {
    int hour = GregorianCalendar.getInstance( ).get(Calendar.HOUR_OF_DAY);

    if((hour >= dayStart) && (hour <= dayEnd))
        return Tag.SKIP_BODY;

    return Tag.EVAL_BODY_INCLUDE;
}
}

```

Note that we also have two optional tag attributes: `dayStart` and `dayEnd`, which accept an hour value, defaulting to 9 A.M. to 5 P.M. These values are communicated to the `Tag` via JavaBean-style setter objects. The JSP implementation will call the setters for every supplied attribute before calling any of the `doXXX()` methods. Attributes also need to be declared in the tag library descriptor. Here's the extension to the previous tag library for the `Evening` tag:

```

<tag>
  <description>Display text if it's the evening</description>
  <name>Evening</name>
  <tag-class>com.oreilly.jent.jsp.EveningTag</tag-class>
  <body-content>JSP</body-content>
  <attribute>
    <name>dayStart</name>
    <required>>false</required>
  </attribute>
  <attribute>
    <name>dayEnd</name>
    <required>>false</required>
  </attribute>
</tag>

```

4.5.2. More About the Tag Lifecycle

JSP tags are frequently called upon to perform an action more than once. The JSP specification supports this capability via the `IterationTag` interface, which adds an additional lifecycle method, `doAfterBody()`. The `doAfterBody()` method is invoked after the tag body has been processed and can return `IterationTag.EVAL_BODY_AGAIN` to have the tag body reprocessed. The tag body is

reprocessed immediately, and the `doAfterBody()` method is called again. Returning `Tag.SKIP_BODY` stops the loop.

The next example creates a tag named `<enterprise:Loop>`, which processes its content a given number of times (specified via the `times` attribute).

```
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;

public class ForTag extends TagSupport {
    private int current = 1;
    private int times = 0;

    public void setTimes(int i) {
        times = i;
    }

    public int doStartTag( ) throws JspException {
        pageContext.setAttribute("current", new Integer(current));
        if(current < times)
            return Tag.EVAL_BODY_INCLUDE;

        return Tag.SKIP_BODY;
    }

    public int doAfterBody( ) throws JspException {
        if(current >= times)
            return Tag.SKIP_BODY;
        current ++;
        pageContext.setAttribute("current", new Integer(current));
        return IterationTag.EVAL_BODY_AGAIN;
    }
}
```

This tag also uses the `setAttribute()` method to create a variable in the page context named `current`. This allows us to see the loop count in our JSP body. Here's an example:

```
<enterprise:Loop times="4">
Loop <%= current %> <br>
</enterprise:Loop>
```

The output in the browser will be:

```
Loop 1
Loop 2
Loop 3
Loop 4
```

We can't just create variables at will, though. At the minimum, they have to be described in the tag library descriptor, so that the JSP compiler can create the appropriate references. Here's the descriptor for the `Loop` tag:

```
<tag>
  <name>Loop</name>
  <tag-class>com.oreilly.jent.jsp.ForTag</tag-class>
  <description>For Loop</description>
  <variable>
    <name-given>current</name-given>
    <variable-class>java.lang.Integer</variable-class>
    <scope>NESTED</scope>
  </variable>
  <attribute>
    <name>times</name>
    <required>>true</required>
    <rtexprvalue>>true</rtexprvalue>
  </attribute>
</tag>
```

The `<variable>` tag declares the `current` variable as a `java.lang.Integer`. Its scope is defined as `NESTED`, which means it will be available to code inside the tag only. The other legal values are `AT_BEGIN` and `AT_END`. If you want to create variables dynamically (for instance, to allow the user to choose the name of the counter variable), you can create a `TagExtraInfo` class and associate it with the tag using `<tei-class>`.

Also, note the `<rtexprvalue>` tag associated with the `times` attribute. This means that the attribute can accept a JSP expression as its value. So if you have created a variable named `count`, the `Loop` tag could be opened with:

```
<enterprise:Loop times="<%= count %>">
```

4.5.3. Body Tags

Tags that need to interact with their body content extend the `BodyTag` interface instead of the `Tag` interface. `BodyTag` introduces a new method, `doInitBody()`, which runs before the tag body is first processed. It also introduces the `BodyContent` object, which provides access to the tag body. `BodyContent` is an extension of `JspWriter`, with additional methods to retrieve its content in various ways, including as a `String` object. The `BodyTagSupport` class fills the same role for `BodyTag` that `TagSupport` does for `Tag` and `IterationTag`. All `BodyTag` objects are also `IterationTag` objects.

The next example shows a simple tag that retrieves the body content, processes it through the `java.net.URLEncoder` object, and outputs the result:

```

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

public class EncodeTag extends BodyTagSupport {

    public int doAfterBody( ) throws JspException {
        BodyContent c = getBodyContent( ); // Get the tag body
        JspWriter out = getPreviousOut( ); // Get the enclosing writer
        try {
            out.print(java.net.URLEncoder.encode(c.getString( )));
        } catch(java.io.IOException e) {
            throw new JspException(e);
        }
        return BodyTag.SKIP_BODY; // Do not process again
    }
}

```

The `getPreviousOut()` method of `BodyTagSupport` retrieves the `JspWriter` associated with the tag's parent. This might be the main `JspWriter` for the page, or it might be the `BodyContent` object of another JSP tag.

4.5.4. Intertag Communication

As just mentioned, it's possible for JSP custom tags to be nested within other JSP custom tags. The inner tags are executed between the `doStartTag()` and `doEndTag()` methods of the outer tags. Let's imagine an `<enterprise:Email>` tag. In the JSP file, it might be used like this:

```

<enterprise:Email>
<enterprise:EmailAddress addr="from">deb@company.com
</enterprise:EmailAddress>
<enterprise:EmailAddress addr="to">will@company.com
</enterprise:EmailAddress>
<enterprise:EmailText>Where's that JSP Chapter?</enterprise:EmailText>
</enterprise:Email>

```

The `Email` tag is implemented with either a `Tag` or a `BodyTag`, with JavaBean-style methods to set from and to addresses and email content. The `doEndTag()` method assembles and sends the email. Assuming that the `doStartTag()` method of the email tag class doesn't stop the tag's processing, each of the three interior tags will be processed in turn. These tags can access the parent tag using the `findAncestorWithClass()` method, which takes a `Tag` and a `class` reference and finds the first appearance of the specified class in the tag's hierarchy, working up from the specified tag.

Here's the code for the `EmailAddress` tag, which sets the to and from addresses of the `EmailTag` class (not included):

```
import javax.servlet.jsp.tagext.*;

public class EmailAddressTag extends BodyTagSupport {
    String addrType = null;
    public void setAddr(String s) {
        addrType = s;
    }
    public int doEndTag( ) {
        Tag t = findAncestorWithClass(this, enterprise.EmailTag.class);
        if(t != null) {
            EmailTag et = (EmailTag)t;
            if(addrType.equalsIgnoreCase("to"))
                et.setToAddress(getBodyContent( ).getString( ));
            else if (addrType.equalsIgnoreCase("from"))
                et.setFromAddress(getBodyContent( ).getString( ));
        }
        return Tag.EVAL_PAGE;
    }
}
```

And that's all there is to it. If the `EmailAddressTag` object doesn't find an `EmailTag` in its parent hierarchy, it exits without taking any action.

4.6. Wrapping Up

We've just gone through a great deal of JSP material, and you should now have a pretty good grasp of the range of capabilities provided by JSPs. Obviously, there is more depth and nuance than we can cover in a tutorial chapter, but the material we've discussed covers the full range of capabilities. As you begin to develop your JSPs, you may find it useful to consult [Chapter 2](#), where we provide details on assembling and deploying web applications, and [Appendix A](#), where we provide details on the structure of web deployment descriptors. In addition, you may want to become familiar with the code annotation facilities discussed in [Chapter 21](#), since these can help simplify the development of web components in general by automatically generating various code artifacts, such as deployment descriptors.

For information on JSP beyond the material in this book, we recommend *JavaServer Pages* (O'Reilly). This book provides more detail on some of the features we've flown rather lightly over and also addresses database access from JSPs and other important approaches to putting all the pieces together.

Chapter 5. JavaServer Faces

The JavaServer Faces API provides a standard framework for building presentation tiers for web applications. Unlike the Servlet API, which focuses on low-level request handling for the HTTP protocol, JSF provides developers with a set of prewritten user interface (UI) components and an event model for wiring interactions between the UI and the objects underlying your application.

The JSF team's goal was to streamline and standardize web UI development. JSF was also designed to be tool-friendly, allowing IDE vendors to support drag-and-drop user interface generation while retaining standardized, portable code and allowing third parties to develop UI components that any JSF developer can use. JSF is included in the J2EE 1.4 framework, but since it requires only the Servlet 2.3 and JSP 1.1 APIs, JSF applications can be deployed in a J2EE 1.3 environment as well.

This chapter provides a whirlwind introduction to programming with JavaServer Faces. Since JSF IDEs aren't yet very mature, and this is a Nutshell book anyway, we focus on the underlying code: if you start using a tool later, you'll have an understanding of how the API itself works, and, of course, you can always write a JSF UI by hand. While JSF might seem more conceptually daunting than other web frameworks, such as Struts, it's actually quite easy to work with by hand once you've gained some basic familiarity with the API.

JSF uses JSP tags to create user interfaces, so if you've never written a JSP page, you might want to take a quick look through [Chapter 4](#) and then come back. [Appendix B](#) provides a reference for all the standard JSP tags provided by the basic JSF implementation since we can't cover them all in any detail in a short tutorial. We've also elected not to discuss some advanced features, such as rendering for non-HTML output and developing custom components. For a much more extensive tutorial, please see *JavaServer Faces* by Hans Bergsten (O'Reilly).

5.1. The Sample Application

JSF is a big topic, and we can't cover it all here, but we did want to provide you with a complete application to look at. Our sample application lets us cover the most frequently used JSF functions. We'll reproduce bits of the code here, and you can download the whole thing from this book's web site, at <http://www.oreilly.com/catalog/javaentnut3/>.

The sample application itself is a simple library book management system that implements the following use cases:

- Log in as a user (with Borrower or Librarian privileges)
- Add a book (Librarians)
- View available books (everyone)
- Check out a book (Librarians, Borrowers)
- Check in a book (Librarians)
- Review all books

In order to compile the examples, you'll want the full JSF 1.1 reference implementation from Sun, which is available at <http://java.sun.com/j2ee/javaxserverfaces/>. You'll need a Servlet 2.3-compatible web container as well; we used JBoss 4.0, available from <http://jboss.org>, to develop and test the sample application. In the downloadable version, we've also included an Ant buildfile for building the sample application.

5.2. Structure of a JSF Application

JSF applications are web applications built on the Java Servlet framework, so they follow the same general packaging scheme as any other J2EE application (see [Chapter 2](#) for more information). The jar files for the JSF implementation must be available on the web application's classpath (usually accomplished by putting the appropriate jar files into the *WEB-INF/lib* directory).

For the reference implementation, the `FacesServlet` must be configured in the *web.xml* file for the application. This servlet provides the JSF engine: it's responsible for processing requests for JSF-based pages and returning the UI in HTML format. For our example, we're going to treat all resources with a *.faces* extension as a JSF target, so we configure the `FacesServlet` like this:

```
<servlet>
    <servlet-name>Faces Servlet</servlet-name>
    <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>

<!-- Faces Servlet Mapping -->
<servlet-mapping>
    <servlet-name>Faces Servlet</servlet-name>
    <url-pattern>*.faces</url-pattern>
</servlet-mapping>
```

Mapping **.faces* to the `FacesServlet` instructs the servlet container to delegate all requests ending with *.faces* to the JSF engine.

5.2.1. JSF Lifecycle

So what happens when a request is sent to the JSF engine? In short, the JSF engine creates a tree containing a set of "components," each corresponding to an element of the user interface for the page being submitted, and ensures that the correct user values are applied to those components. The component tree is used to process the request, and an appropriate response is generated for the user. The `FacesServlet` accomplishes this in six steps:

1. *Restore view*

The JSF engine looks at the request and matches it to a view definition, which is usually defined in a JSP file. By default, the view definition is found by looking for a *.jsp* file with the same base name as the request URI. A request for */view/books.faces* will use */view/books.jsp* as the view

definition. The engine then creates the component tree based on the view definition.

2. *Apply request values*

If any HTTP parameters were submitted along with the request, the JSF engine will try to map them to specific user interface components on the page. Otherwise, the engine skips to step 6, which renders the page and returns it to the client.

3. *Process validations*

After applying any request values to the page, the JSF engine will cycle through all of the input validators associated with each component. If any validations fail, the page is rerendered with the original data, and, depending on how the developer set the page up, warning messages may be displayed.

4. *Update model values*

If the validation process succeeds, the underlying Java beans associated with each UI component are updated.

5. *Invoke application*

After the model has been updated, the JSF engine invokes any business logic actions requested by the user.

6. *Render response*

The JSF engine renders a response for the user. Depending on the results of step 5, this may be the view associated with the page that was originally requested, or it may be a different view.

After steps 2 through 5, the JSF engine will process any events that were triggered during the processing activity. The event model allows you to override the lifecycle if necessary for instance, jumping from the Apply Request Values phase to the Render Response phase. We won't get into that in much detail in this chapter, but you should know the capability is there.

The Restore View phase is particularly important. This is where the JSF engine produces the *component tree*, which is an object hierarchy containing all of the JSF UI components within the requested view. This tree is used for all of the subsequent processing stages, and it maintains its state across multiple requests we'll see an example of how this works when we talk about user interface events in a few pages.

5.2.2. The JSF Configuration File

JSF relies heavily on a single configuration file, *faces-config.xml*, located in your web application's *WEB-INF* directory. This file is where you keep track of navigation rules, managed beans, localization information, and any other configuration data required by more advanced JSF features. Configuring web applications via XML will be familiar to users of Struts, although the JSF framework does more behind the scenes than Struts does, reducing the overall complexity of the file. For example, you don't have to define your action classes and the related forms as you do in Struts: the server loads Faces resources automatically based on URL patterns.

Here's an example of a very simple *faces-config.xml* file for the sample application. We'll look at it in more detail shortly.

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

<!DOCTYPE faces-config PUBLIC
  "-//Sun Microsystems, Inc.//DTD JavaServer Faces Config 1.1//EN"
  "http://java.sun.com/dtd/web-facesconfig_1_1.dtd">

<faces-config>

  <application>
    <locale-config>
      <default-locale>en</default-locale>
    </locale-config>
  </application>

  <!-- Navigation rules here -->

  <!-- Bean definitions here -->

</faces-config>
```

In the beginning of the file, we define the application's default localization setting, which is English. In the rest of this section, we'll show how to set up the various Java beans we want JSF to manage for us and the navigation rules our application should follow.

If the file gets too unwieldy, you can split it into parts, with each file defining configuration information for a subset of your application. To do this, add an initialization parameter named `javax.faces.CONFIG_FILES` to the `FacesServlet` configuration in *web.xml*, containing a comma-separated list of configuration files. Large projects with multiple developers will probably need to do this since it allows you to define settings for different modules of an application separately.

5.3. Managed Beans

As we've just seen, the JSF configuration file has two main sections: one for navigation rules and one for bean definitions. The JSF framework relies on *managed beans* to glue the user interface (defined primarily in JSP using the JSF tag libraries) to the business logic of your application (which can be whatever sort of code you want).

Managed beans are simply regular Java beans, configured within the *faces-config.xml* file. The bean class follows the standard JavaBeans rules: a zero-parameter constructor and a set of get/set methods for manipulating properties. In JSF, managed beans also contain *action methods*. Action methods are invoked by the JSF framework in response to a user action or an event and contain the code that actually manipulates the data model behind your application. In Struts, these methods would correspond to the `Action` classes that drive the application.

A JSF application usually has two types of managed beans: *model beans* and *backing beans*. The difference between the two is more architectural than technical. Model beans focus on integration with the data model of the application. In the Library application, we have model beans (in the `com.oreilly.jent.jsf.library.model` package) for tasks like managing the book collection itself and for keeping track of user accounts and privileges. Backing beans are linked more directly to the structure of the user interface. A backing bean contains action methods that manipulate the model beans on behalf of the UI and properties that relate directly to user interface elements embedded in HTML forms (we'll look at this in more detail in the next section). Remember that there's no technical difference between the two: you write and deploy them exactly the same way.^[*]

[*] Bean management is a little simpler in JSF than in Struts, where you have to associate request and session scope beans with particular forms via configuration XML. In JSF, beans are instantiated as they're called for within JSF pages no sooner. This means you can define as many request and session scope beans as you'd like and use them in multiple views rather than having to associate a particular `ActionForm` with a particular `Action`.

Beans have four possible scopes: application, session, request, and scopeless. If a managed bean is defined as application scope, the entire application shares a single instance. For session-scoped beans, the JSF framework creates a new instance of the bean for each user when the bean is first accessed by the user. For request-scoped beans, the framework creates a new instance for each request, assuming that the request uses the bean at some point. Scopeless beans are accessed by other beans, but not directly by the application. They are instantiated when requested and will be garbage-collected like any other object once they pass out of scope. Note that the four scope options are not quite the same as the equivalent Servlet/JSP scopes, and to ensure compatibility, you'll want to access them through JSF rather than trying to pull values out of request attributes or `HttpSession` objects.

[Example 5-1](#) shows a simple backing bean, which we use for keeping track of information about the current user's session. It includes properties (`currentUser` and `loggedIn`) and an action method, `logout()`.

Example 5-1. UserSession.java

```

import model.User;
import logic.UserManager;

public class UserSession {
    private User currentUser = null;

    public UserSession( ) {
        currentUser = UserManager.getGuestUser( );
    }

    public User getCurrentUser( ) {
        return currentUser;
    }

    public void setCurrentUser(User currentUser) {
        this.currentUser = currentUser;
    }

    public boolean isLoggedIn( ) {
        if (currentUser == null)
            return false;
        if (currentUser == UserManager.getGuestUser( ))
            return false;
        return true;
    }

    public String logout( ) {
        currentUser = UserManager.getGuestUser( );
        return "home";
    }
}

```

Managed beans are set up via the JSF configuration file. The following example sets up three beans, a model bean stored in the application scope, and backing beans stored in the session and request scopes:

```

<managed-bean>
    <managed-bean-name>library</managed-bean-name>
    <managed-bean-class>
        com.oreilly.jent.jsf.library.model.Library
    </managed-bean-class>
    <managed-bean-scope>application</managed-bean-scope>
</managed-bean>

<managed-bean>
    <managed-bean-name>usersession</managed-bean-name>
    <managed-bean-class>
        com.oreilly.jent.library.session.UserSession
    </managed-bean-class>

```

```

        <managed-bean-scope>session</managed-bean-scope>
    </managed-bean>

    <managed-bean>
        <managed-bean-name>loginform</managed-bean-name>
        <managed-bean-class>
            com.oreilly.jent.library.backing.LoginForm
        </managed-bean-class>
        <managed-bean-scope>request</managed-bean-scope>
    </managed-bean>

```

Each bean definition has a name, a class, and a bean scope (`application`, `session`, `request`, or `none`).

Sometimes we need to access managed beans directly within our code. We can do this using the `FacesContext` object, which provides access to a set of `Map` objects that are associated with the underlying JSF session. To retrieve the `usersession` managed bean from the session scope, we can use code like this at any time during request processing:

```

UserSession userSession =
    (UserSession)FacesContext.getCurrentInstance( )
    .getExternalContext( ).getSessionMap( ).get("usersession");

```

There's a major architectural problem with doing this too often, though you become reliant on the `usersession` literal rather than having a truly type-safe way to access session data. You can improve this in two ways. The first is to establish a centralized directory that different classes in your application can use to look up objects. The Library application uses a `Directory` class ([Example 5-2](#)) to provide easy access to the `Library` object that we store in the application's scope. This example also shows how to retrieve managed beans by dynamically executing an expression written in the JSF EL.

We'll look at the EL in the next section, so for now just be advised that `#{library}` means to retrieve a bean named `library` from the closest possible scope. There's no map maintained for application scope, so this technique is important.

Example 5-2. Directory class

```

package com.oreilly.jent.library;

import com.oreilly.jent.library.model.Library;

import javax.faces.context.FacesContext;
import javax.faces.el.ValueBinding;
import javax.faces.application.Application;

public class Directory {
    public static Library getLibrary( ) {
        FacesContext context = FacesContext.getCurrentInstance( );

```

```

        Application app = context.getApplication( );
        ValueBinding binding = app.createValueBinding("#{library}");
        Library library = (Library)binding.getValue(context);
        return library;
    }
}

```

The second way to reduce or eliminate dependencies on literal values in compiled code is to have the JSF engine populate the bean properties at runtime. If we want to give a managed bean access to another managed bean, we can also have the JSF framework provide the first bean with access to the second bean by passing the second bean to the first bean at runtime. In our application, we have a backing bean for the book display form that requires access to the library bean that we're managing via the JSF framework. The `BookDisplayForm` class just needs a simple setter method, like this:

```

public void setLibrary(Library library) {
    this.library = library;
}

```

With the setter in place, we can use the `<managed-property>` tag in the *Faces* configuration file to execute an expression and populate the `library` property whenever an instance of `BookDisplayForm` is created:

```

<managed-bean>
    <managed-bean-name>bookdisplayform</managed-bean-name>
    <managed-bean-class>
        com.oreilly.jent.jsf.library.backing.BookDisplayForm
    </managed-bean-class>
    <managed-bean-scope>session</managed-bean-scope>
    <managed-property>
        <property-name>library</property-name>
        <value>#{library}</value>
    </managed-property>
</managed-bean>

```

Remember to keep your scopes straight when setting up managed properties: the bean you're using as the property value must have an equal or broader scope. You can populate a request scope bean with session-scoped beans, but not vice versa. This is also where those scopeless beans (that is, beans with a scope of `none`) are useful: you can use them as injected property values for other managed beans without exposing them directly to the application.

5.4. The JSF Expression Language

Now that we've established the existence of managed beans, we can talk about the JSF EL. The EL, which is similar but not identical to the JSP EL described in [Chapter 4](#), allows you to write expressions that reference the managed beans in your application. These expressions serve two main purposes in JSF: retrieving values from beans and telling JSF components which bean properties to update. In some cases, the same expression does both, depending on where you are in the request lifecycle.

The general syntax of an EL expression is `#{identifiers and modifiers}`. The `#{` marks the beginning of the expression, and the `}` marks the end.

Here's an example of a simple tag that displays a value from a managed bean. In this case, we're displaying the `username` property, which is itself a property of the `currentUser` property of the `usersession` bean.

```
<h:outputText value="#{usersession.currentUser.username}"/>
```

The `<h:outputText>` tag is a built-in JSF tag, used to display the value of an expression. We'll look at more JSF tags in the next section. When working with input controls, the application uses the value expression first to prepopulate the control and then to store the results of the request. The following tag creates a text input control, associated with the `username` property of the `loginform` bean:

```
<h:inputText value="#{loginform.username}"/>
```

When the user first loads the JSF page containing this tag, the application displays an HTML input field containing the current value of the EL expression. In this case, the field will be blank since the login form bean is request scoped. When the form is submitted, the JSF framework will use the same EL expression to decide where to store the form values the browser sends: in this case, into the `username` property of the `loginform` bean.

The EL also supports lists and maps, allowing you to access more dynamic information. Many of the built-in EL objects (listed in [Table 5-1](#)) rely on this functionality. When referencing a `Map`, the identifier on the righthand side of the period is used as the key for retrieving a value from the map. For example, using the built-in attribute `sessionScope`, we can retrieve the `usersession` bean like this:

```
#{sessionScope.usersession.currentUser.username}
```

An equivalent expression would be:

```
#{sessionScope['usersession'].currentUser.username}
```

This is an alternative syntax for retrieving values from a map. The square-bracket syntax is generally used when retrieving values with (.) characters in their keys, which is often necessary when dealing with properties files and message bundles for internationalization.

Table 5-1. Built-in JSF Expression Language objects

EL object	Definition
<code>applicationScope</code>	A <code>Map</code> containing all application scope-managed beans
<code>cookie</code>	A <code>Map</code> of the HTTP cookies for the current request, by name
<code>facesContext</code>	The <code>FacesContext</code> instance associated with the current request
<code>header</code>	A <code>Map</code> with all HTTP headers submitted with the current request, containing one value for each header
<code>headerValues</code>	A <code>Map</code> with all HTTP headers submitted with the current request, containing a <code>String[]</code> array for each header with all submitted values
<code>initParam</code>	A <code>Map</code> containing each of the initialization parameters for the <code>FacesServlet</code>
<code>param</code>	A <code>Map</code> with all HTTP parameters submitted with the current request, containing one value for each parameter
<code>paramValues</code>	A <code>Map</code> with all HTTP parameters submitted with the current request, containing a <code>String[]</code> array for each parameter with all submitted values
<code>requestScope</code>	A <code>Map</code> containing all request scope-managed beans
<code>sessionScope</code>	A <code>Map</code> containing all session scope-managed beans
<code>view</code>	The <code>UIViewRoot</code> object associated with the current request

You can also perform calculations with EL expressions. The EL doesn't allow you to write value assignment statements directly, which frees page developers from the temptation to write too much business logic into the JSP files themselves. But as we'll see, many tags take Boolean parameters and it's often helpful to be able to express them in the EL rather than writing lots of extra methods in your beans.

The alphabetic versions of the relational operators, which are listed below the regular versions in [Table 5-2](#), exist to avoid conflict with JSP, XML, and HTML tags (having too many angle brackets running around gets confusing for the parsers!).

Table 5-2. JSF Expression Language operators

Operator type	Operators					
Arithmetic	+	-	/	%		
		div	mod			
Logical	&&		!			
and		not				
Relational	<	<=	>	>=	==	!=
lt	le	gt	ge	eq	ne	
Selection	?					

The `?` selection operator works the same way it does in Java. For example, we can use it for some simple formatting:

```
<h:outputText value="#{bookdisplayform.book.checkedOut ? 'Out' : 'In'}"/>
```

The selection operator is often useful for controlling component rendering on JSF pages when a bean method does not return true or false.

5.5. JSF Actions and Views

Now that we've set up our managed beans, we can start looking at the JSF pages themselves. Let's start with the home page for the Library application, which is shown in [Example 5-3](#).

Example 5-3. index.jsp

```
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>

<html>

  <f:view>
  <head>
    <title>JSF Library Home Page</title>
    <link rel="stylesheet" type="text/css" href='<%=
      request.getContextPath( ) + "/css/app.css" %>'>
  </head>
  <body>
    <h:form>
    <h1>JSF Library</h1>
    <hr>
    Welcome to the JSF library.
    <p>
    <h:outputText
      value="You are logged in as #{usersession.currentUser.username}"/>
    <p>
    <h:commandButton action="viewbooks" value="View Books"/>
    <h:commandButton action="addbook"
      rendered="#{usersession.currentUser.librarian}" value="Add Book"/>

    <hr>
    <h:commandLink rendered="#{not usersession.loggedIn}" action="login">
      <h:outputText value="Log In"/>
    </h:commandLink>
    <h:commandLink rendered="#{usersession.loggedIn}"
      action="#{usersession.logout}">
      <h:outputText value="Log Out"/>
    </h:commandLink>

    </h:form>
  </body>
</f:view>
```

```
<html>
```

This file lives in the root directory of our web application. But if we try and access it at <http://server/library/index.jsp>, we'll get a messy error. Instead, we access it via the `FacesServlet`, which is responsible for managing the JSF lifecycle. Earlier, we set up a mapping in `web.xml` that forwards all requests with a `.faces` extension to the `FacesServlet`. The `FacesServlet` sets up a `FacesContext` object and then dispatches to the appropriate view definitions such as the `index.jsp` file. The mapping is simple: a request for `/index.faces` uses the `/index.jsp` view.

In situations where you are using both JSF-enabled and standard JSPs, it can be useful to be able to easily distinguish between the two. One way to do this is to use the `javax.faces.DEFAULT_SUFFIX` context parameter. The value of this parameter determines the suffix used to find matching JSPs when the `FacesServlet` handles a request. If we used our earlier mapping for the `FacesServlet` and set the `DEFAULT_SUFFIX` to `*.jsf`, for example, then every URL ending with `.faces` will be resolved to a JSP with a `.jsf` suffix. Remember that context parameters are set using `<context-param>` elements in the `web.xml` file. See [Appendix A](#) for full details on `web.xml` syntax.

We've imported two tag libraries into this JSP file. The first, which we've given the `h:` prefix, is the JSF HTML tag library. The HTML tags provide access to the various UI components provided by the JSF libraries. The second tag library, which we've given the `f:` prefix, is the JSF core tag library. These are the standard prefixes, and we'll use them for the rest of the chapter.

All of the JSF content in a page must be contained within an `<f:view>` tag, which represents the root of the JSF component tree. [Example 5-3](#) uses the `<h:outputText>` tag to display text based on an EL expression, and the `<h:commandButton>` and `<h:commandLink>` tags to allow navigation within the application.

All of the HTML-related tags support the `rendered` attribute, which allows us to specify whether or not the UI component is displayed, based on the runtime value of a JSF expression. In [Example 5-3](#), we display a login link if the user isn't logged in and a logout link if he is. We also display an Add Book button for librarians only.

5.5.1. Navigation

Page-to-page navigation in JSF is built around *navigation rules*. A navigation rule has three basic parts: an originating view, an outcome, and a destination view. The two kinds of views are identified by URL patterns, and the outcome is a simple text string. When a user triggers a navigation event (by clicking on a button or a command link), the JSF framework selects a destination view by finding the navigation rule with the appropriate outcome that most closely matches the originating view.

Let's start with a simple definition. This rule states that the `addbook` outcome from the `/index.jsp` file should lead to the `/view/addbook.jsp` view:

```
<navigation-rule>
  <from-view-id>/index.jsp</from-view-id>
  <navigation-case>
    <from-outcome>addbook</from-outcome>
    <to-view-id>/view/addbook.jsp</to-view-id>
```

```

        </navigation-case>
    </navigation-rule>

```

Within the *index.jsp* file, we can reference this outcome directly by specifying the outcome name as the action associated with the command button:

```

<h:commandButton action="addbook"
    rendered="#{usersession.currentUser.librarian}" value="Add Book"/>

```

The result is a button that, when clicked, causes the server to display the */view/addbook.jsp* view.

Rather than linking views one by one, we can use wildcards and provide multiple navigation cases. Here's how we define a set of rules that applies to outcomes originating from any view:

```

<navigation-rule>
    <from-view-id>*</from-view-id>
    <navigation-case>
        <from-outcome>login</from-outcome>
        <to-view-id>/login/index.jsp</to-view-id>
    </navigation-case>
    <navigation-case>
        <from-outcome>home</from-outcome>
        <to-view-id>/index.jsp</to-view-id>
    </navigation-case>
    <navigation-case>
        <from-outcome>viewbooks</from-outcome>
        <to-view-id>/view/listbooks.jsp</to-view-id>
    </navigation-case>
</navigation-rule>

```

We use the wildcard character (*) in the `<from-view-id>` tag to declare that this rule should apply to all originating views unless a more exact match is available. With this rule in place, we can use the `login`, `home`, and `viewbooks` outcomes from anywhere in the application.

Wildcard mapping for navigation rules works almost exactly as it does for servlets. More precise mappings take precedence: if we defined a `home` outcome in a navigation rule governing the */view/addbook.jsp* view, it would override the default navigation rule as well as a rule governing (for example) */view/**.

5.5.1.1. Dynamic actions and navigation

Most applications need to control navigation based on the result of an action rather than just hardcoding outcomes directly to buttons. That's why they're called outcomes: they can differ based on the circumstances. We also need a way to execute code after clicking a button. Let's take a look at the library login functionality to see how this is done.

We start by defining the navigation rules in *faces-config.xml*. We want to display a success message if the user logs in successfully, and we want to redisplay the login page if the user can't log in for some reason. Here's the navigation rule:

```
<navigation-rule>
  <from-view-id>/login/index.jsp</from-view-id>
  <navigation-case>
    <from-outcome>success</from-outcome>
    <to-view-id>/login/success.jsp</to-view-id>
  </navigation-case>
  <navigation-case>
    <from-outcome>failure</from-outcome>
    <to-view-id>/login/index.jsp</to-view-id>
  </navigation-case>
</navigation-rule>
```

Next, we write a Java method to handle the login process. We include this method in the backing bean associated with the login activity. In this case, the method is `confirmCredentials()`. The only thing that's special about this method is that it returns a string indicating the outcome of the request in this example, `success` or `failure`. The code is in [Example 5-4](#).

Example 5-4. LoginForm.java

```
import logic.UserManager;
import session.UserSession;
import model.User;

import javax.faces.context.FacesContext;
import javax.faces.application.FacesMessage;

public class LoginForm {
    private String username;
    private String password;

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
```

```

        this.password = password;
    }

    public String confirmCredentials( ) {

        UserSession userSession =
            (UserSession)FacesContext.getCurrentInstance( )
                .getExternalContext( ).getSessionMap( ).get("usersession");

        if (UserManager.confirmLogin(username, password)) {
            User user = UserManager.getUser(username);
            userSession.setCurrentUser(user);
            return "success";
        }

        FacesMessage message = new FacesMessage( );
        message.setSummary("Login Error");
        message.setDetail("Unable to log you in -"
            + " username and password combination not found.");
        message.setSeverity(FacesMessage.SEVERITY_WARN);
        FacesContext.getCurrentInstance( ).addMessage(null,message);
        return "failure";
    }
}

```

Finally, as you'll see in [Example 5-5](#), we can wire the `<h:commandButton>` on the login form to the `confirmCredentials()` method on the backing bean. We use the same `action` property we used when we wired command buttons directly to outcomes.

Example 5-5. /login/index.jsp

```

<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>

<html>

    <f:view>
    <head>
        <title>JSF Library Login</title>
        <link rel="stylesheet" type="text/css"
            href='<%= request.getContextPath( ) + "/css/app.css" %>'>
    </head>
    <body>
        <h1>JSF Library Login</h1>
        <hr>
        In order to borrow or add books you must log into the Library
        application.
    </body>

```

```

    <h:messages showDetail="true" warnClass="loginWarning"
        layout="table" />

<h:form>

    <table>
    <tr>
    <td>Username:</td>
    <td><h:inputText value="#{loginform.username}" /></td>
    </tr>
    <tr>
    <td>Password:</td>
    <td><h:inputSecret value="#{loginform.password}" /></td>
    </tr>
    </table>

    <p>
    <h:commandButton value="Login"
        action="#{loginform.confirmCredentials}" />
    </p>

</h:form>
</body>
</f:view>
</html>

```

When the user clicks the button, the JSF server performs the following steps:

1. Obtains an instance of the `LoginForm` bean. Since we've declared it as a request scope bean, a new instance is always created.^[*]

[*] In Struts, request scope beans are reused. This is not the case in JSF.

2. Sets bean properties based on the form submission from the `<h:inputText>` and `<h:inputSecret>` controls on the login form.
3. Calls the `confirmCredentials()` method on the `LoginForm` instance, which attempts the login process and returns the string "success" or the string "failure" based on the outcome.
4. Displays the appropriate view based on the data in *faces-config.xml*.

We do a few special things with the failure case, which is shown in [Figure 5-1](#). First, the username input field remains populated with whatever the user originally submitted that's because the `LoginForm` bean instance created in step 1 is still in scope. The password field doesn't get filled back in, because it's an `<h:inputSecret>` component, which doesn't prepopulate. The original bean instance isn't thrown away until after the page is rendered. When the user resubmits the form, a new instance is created (however, the component tree is retained until the user accesses another view).

Figure 5-1. JSF Library Login failure



Second, the action method adds a `FacesMessage` object to the current instance of the `FacesContext` object. This message is then displayed via the `<h:messages>` tag. A JSF message has two parts: a summary and a detail. The `<h:messages>` tag lets us display both the summary and the detail (by default, only the summary is shown) for each message in the queue for the current page. We'll look at message display a bit more closely at the end of the chapter.

5.5.2. Sophisticated Views

Let's take a look at another page in the JSF Library, the View Books screen. This screen allows users to browse books and to check books in and out, provided that they have appropriate privileges. This screen illustrates how to use JSF select boxes and how to group display elements into panels for more control of the display. [Example 5-6](#) shows the `listbooks.jsp` file.

Example 5-6. `/view/listbooks.jsp`

```
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>

<html>
  <f:view>
  <head>
    <title>Library Books</title>
    <link rel="stylesheet" type="text/css"
```

```

        href='<%= request.getContextPath( ) + "/css/app.css" %>')>
</head>
<body>
    <h1>Library Collection</h1>
    <hr>
<h:form>
<table>
    <tr><td valign="top">

        <h:selectOneListbox size="10"
            value="#{bookdisplayform.currentBookID}"
            onchange="submit( )">
            <f:selectItems value="#{bookdisplayform.bookList}"/>
        </h:selectOneListbox>
        <p/>
        <h:commandButton value="Return Home" action="home"/>

</td><td valign="top">
    <h:panelGroup rendered="#{bookdisplayform.bookSelected}">
        <h:panelGrid columnClasses="rowLabel,rowContent" columns="2">
            <h:outputText value="Title:"/>
            <h:outputText value="#{bookdisplayform.book.title}"/>
            <h:outputText value="Author:"/>
            <h:outputText value="#{bookdisplayform.book.author}"/>
            <h:outputText value="ISBN:"/>
            <h:outputText value="#{bookdisplayform.book.ISBN}"/>
            <h:outputText value="Status:"/>
            <h:outputText value=
                "#{bookdisplayform.book.checkedOut ? 'Checked Out' : 'Checked In'}/>
        </h:panelGrid>

        <h:panelGroup id="buttonHolder">

        <h:panelGroup rendered="#{usersession.currentUser.borrower}">
            <h:commandButton value="Check Out"
                action="#{bookdisplayform.checkBookOut}"
                rendered="#{not bookdisplayform.book.checkedOut}"/>
        </h:panelGroup>

        <h:panelGroup rendered="#{usersession.currentUser.librarian}">
            <h:commandButton value="Check In"
                action="#{bookdisplayform.checkBookIn}"
                rendered="#{bookdisplayform.book.checkedOut}"/>
        </h:panelGroup>

    </h:panelGroup>
</td></tr>
</table>
<h:selectBooleanCheckbox onclick="submit( )"
    valueChangeListener="#{bookdisplayform.safeModeChange}"/>
<h:outputText value="Safe Mode"/>

```

```

</h:form>

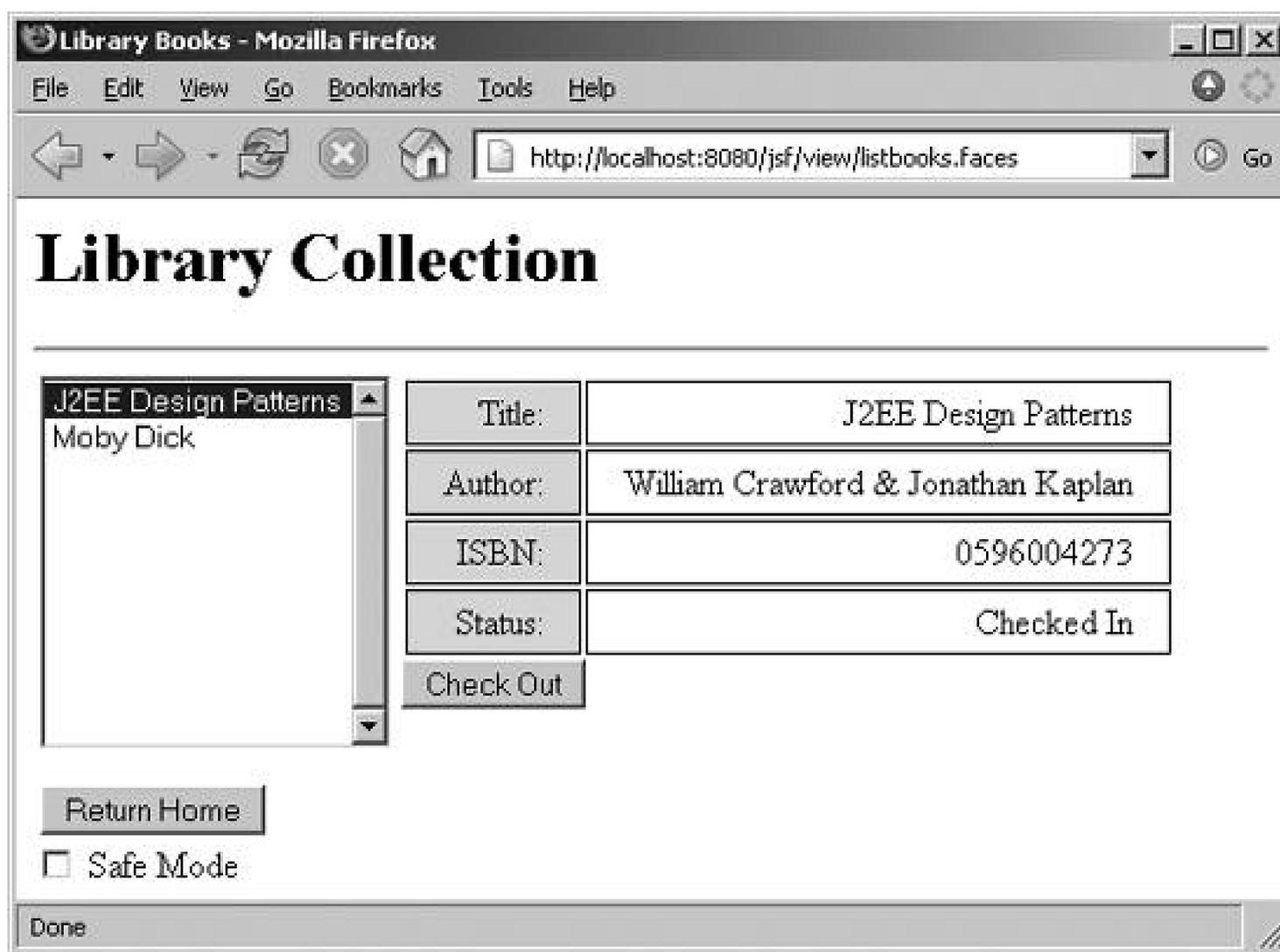
</body>

</f:view>
</html>

```

Figure 5-2 shows what the page looks like for Librarian users who have selected a particular book. The JSP produces a selection box based on the `booklist` property of the `BookDisplayForm` backing bean. Earlier we showed how the `BookDisplayForm` is configured as a managed bean in `faces-config.xml`. If a book has been selected, the page displays details about the book using the `<h:panelGrid>` and `<h:panelGroup>` tags. If the user has logged in, the page may also display a Check In or a Check Out button, depending on the book's status and whether the user is logged in as a Librarian or as a regular Borrower. Just ignore the Safe Mode checkbox for now; we'll look at it later.

Figure 5-2. Library Collection screen



Let's look at the select box first. The code in the JSP for creating this component is very simple:

```

<h:selectOneListbox size="10"
    value="#{bookdisplayform.currentBookID}"

```

```

        onchange="submit( )">
        <f:selectItems value="#{bookdisplayform.bookList}"/>
    </h:selectOneListbox>

```

We use the `<h:selectOneListbox>` component since we want the user to be able to select only one book at a time. We've also embedded some JavaScript in the tag, using the `onchange` attribute. This is a pass-through attribute to the HTML `<select>` tag that is produced when the page is rendered. It causes the form to be submitted when the user selects a new value.

The `<f:selectItems>` tag provides the list of items to include in the select box. In this case, we don't want to hardcode the options into the JSP tag (although this can be done using the `<f:selectItem>` tag). Rather, we want to get the list of books from the application's data model. So we've written a method on our backing bean for the form, which accesses our data model object to produce an array of `javax.faces.model.SelectItem` objects:

```

public SelectItem[] getBookList( ) {
    Library library = Directory.getLibrary( );
    List books = library.getBookList( );
    SelectItem[] items = new SelectItem[books.size( )];
    for (int i = 0; i < books.size( ); i++) {
        Book book = (Book)books.get(i);
        items[i] = new SelectItem(book.getTrackingID( ),
                                book.getTitle( ));
    }
    return items;
}

```

Since we don't want the data model itself to know about JSF objects (it doesn't need to) we haven't built the ability to create the `SelectItem` array into the `Library` class itself: we use the backing bean as the intermediary between the user interface and the data model.

The `SelectItem` constructor requires an object ID and a label. The object ID is used as the key when building the select box, and the label is used as the text displayed to the user. When an item is selected and the form is submitted, the object ID is what's made available to the application. For the `Library` application, the object ID is the book's tracking ID, which is a unique string that can be used to retrieve a `Book` object from the `Library` object.

The following three methods from the `BookDisplayForm` backing bean work together to translate the book ID bound to the `<h:selectOneListbox>` tag into the `book` property, which can then be accessed by any other tags in the UI that need access to the currently selected book:^[*]

[*] An obvious refactoring here would be to retrieve the `Book` object within the `setCurrentBookID()` method and to cache it for future use. We've kept it this way to make the division of responsibility clear and leave the refactoring as a simple exercise to the reader.

```

public String getCurrentBookID( ) {
    return currentBookID;
}

```

```

public void setCurrentBookID(String currentBookID) {
    this.currentBookID = currentBookID;
}

public Book getBook( ) {
    if (currentBookID == null) return null;
    return (Book)getLibrary( ).getBooks( ).get(currentBookID);
}

```

5.5.3. User Interface Events

In [Example 5-3](#), we used logic within a Java bean to modify the display based on selected values. This approach doesn't always make sense, so JSF provides another mechanism to respond to user actions that are focused primarily on the interface: *events*. By taking advantage of JSF's event model (which we mentioned briefly during our discussion of the JSF lifecycle earlier in this chapter), we can respond to changes made to user interface components without processing a full action.

Unlike actions, events have access to the `UIComponent` object that fired the event and through it to the rest of the component tree for the page. Events are, in general, not the place to implement business functionality: an event shouldn't be firing off emails or making model changes. That kind of business logic should be reserved for actions, which should not need to know the details of the user interface.

On the View Books page, we've added a Safe Mode checkbox that allows the user to turn the Check In/Check Out buttons on and off (OK, we'll fess up: this is a trivial example, but it shows how events can be used to modify the appearance of the user interface). We set up the Safe Mode checkbox like this, using the standard `<h:selectBooleanCheckbox>` UI component:

```

<h:selectBooleanCheckbox onclick="submit( )"
    valueChangeListener="#{bookdisplayform.safeModeChange}"/>

```

We use a JavaScript handler to submit the page when the value is changed, and we include a reference to a listener method on the backing bean for the form. One thing that we don't do is provide a value binding: all we're interested in is the change, so we don't bind the value of the checkbox to any JavaBean properties. This means the checkbox defaults false every time a user navigates to the view. The checkbox retains its value each time the view is reloaded, until the user navigates to a different view (at which point the JSF engine builds a new component tree for the new view, and any values that aren't bound to session- or application- scoped beans are lost).

The code for the listener function itself looks like this:

```

public void safeModeChange(ValueChangeEvent event) {

    UIComponent buttonHolderGrid =
        event.getComponent( ).findComponent("buttonHolder");

    if (buttonHolderGrid != null) {

```

```
Object newVal = event.getNewValue( );
if (newVal instanceof Boolean)
    buttonHolderGrid.setRendered(
        !((Boolean)newVal).booleanValue( ));
}
}
```

The listener starts out by retrieving the `UIComponent` that fired the event and uses that object to find the `buttonHolder` panel grid. Then, depending on the value the checkbox has been set to, it sets the `rendered` property for the panel grid, which either hides or shows the Check In and Check Out buttons.

Listeners can also be attached to command components such as buttons and links. This allows the application to respond to UI events before (or instead of) processing an action.

[← PREV](#)

5.6. Building Tables

JSF provides two mechanisms for generating tables. *Panel grids* allow you to organize multiple JSF components into an HTML table and are useful for laying out data entry forms and other pages for which you know the number of components but want to leave the mechanics of HTML formatting to the JSF engine. *Data grids* allow you to build tables based on variable numbers of objectlist objects, `ResultSets` and `RowSets`, and JSF `DataModel` objects.

5.6.1. Panel Grids

The Book Details form uses the `<h:panelGroup>` and `<h:panelGrid>` tags to group and display multiple JSF tags. The `<h:panelGroup>` tag is very simple; it groups a set of JSF components into a single unit. Decisions about which components to display can then be handled at the group level rather than component by component. Back in [Example 5-6](#), we grouped an `<h:panelGrid>` (for book information) and an `<h:panelGroup>` (containing two additional panel groups for control buttons) into a parent `<h:panelGroup>`. The top-level group controls rendering of all book information based on whether a book is selected.

The `<h:panelGrid>` component allows you to apply table formatting to a set of components. The tag allows the designer to specify a number of columns and then builds a table with the specified number of columns and as many rows as are required to accommodate the set of components provided by the page designer. In our example, we have two columns and eight components, so the JSF engine renders a table with two columns and four rows:

```
<h:panelGrid columnClasses="rowLabel,rowContent" columns="2">
  <h:outputText value="Title:" />
  <h:outputText value="#{bookdisplayform.book.title}" />
  <h:outputText value="Author:" />
  <h:outputText value="#{bookdisplayform.book.author}" />
  <h:outputText value="ISBN:" />
  <h:outputText value="#{bookdisplayform.book.ISBN}" />
  <h:outputText value="Status:" />
  <h:outputText value=
    "#{bookdisplayform.book.checkedOut ? 'Checked Out' : 'Checked In'}" />
</h:panelGrid>
```

To specify Cascading Style Sheets (CSS) styles, the `<h:panelGrid>` tag includes four CSS-related attributes: `columnClasses`, `rowClasses`, `headerClass`, and `footerClass`, each of which specifies a CSS style. The `columnClasses` and `rowClasses` attributes accept comma-separated lists of values that repeat if more columns or rows exist than there are CSS styles specified. This allows you to do things like alternate the color of each row by specifying something like:

```
<h:panelGrid rowClasses="oddRows,evenRows" columns="4">
    <!-- components would go here -->
</h:panelGrid>
```

The `<h:panelGrid>` tag works only with a known number of components. To display grids with unknown numbers of components, use the `<h:dataGrid>` tag, which we discuss next.

Remember that neither `<h:panelGroup>` nor `<h:panelGrid>` affects non-`UIComponent` page content. So if you want to use a group to control whether some text is displayed, you need to display the text via `<h:outputText>` rather than just entering it into the JSP file.

5.6.2. Data Grids

Panel grids aren't helpful when we don't know how many rows we need to display. For the Library Administration screen, shown in [Figure 5-3](#), we want to list all the books available and whether they're checked in and give librarians an easy way to check books in and out. Data grids allow us to iterate through a set of objects (such as a `java.util.List`) or the results of a SQL query and display the results in a table.

Figure 5-3. Library Administration table

[Example 5-7](#) shows the JSP file for the Library Administration page. The important tag is the `<h:dataTable>` tag, which contains a set of `<h:column>` tags that define the contents of each column. In this example, we're using the list of books from the shared library bean, so we don't even need a special backing bean for this form.

Example 5-7. /view/adminbooks.jsp

```

<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>

<html>
  <f:view>
    <head>
      <title>Library Administration</title>
      <link rel="stylesheet" type="text/css"
        href='<%= request.getContextPath( ) + "/css/app.css" %>'>
    </head>
    <body>
      <h1>Library Administration</h1>
      <hr>
      <h:form>
        <h:commandButton value="Return Home" action="home"/>

        <h:dataTable value="#{library.bookList}"
          headerClass="tableHeader" var="book">
          <h:column>
            <f:facet name="header">
              <h:outputText value="Title"/>
            </f:facet>
            <h:outputText value="#{book.title}"/>
          </h:column>
          <h:column>
            <f:facet name="header">
              <h:outputText value="Author"/>
            </f:facet>
            <h:outputText value="#{book.author}"/>
          </h:column>
          <h:column>
            <f:facet name="header">
              <h:outputText value="Checked Out"/>
            </f:facet>
            <h:selectBooleanCheckbox onclick="submit( )"
              value="#{book.checkedOut}"/>
          </h:column>
        </h:dataTable>
      </h:form>
    </body>
  </f:view>
</html>

```

The `value` attribute of the `<h:dataTable>` tag lets us specify the list of objects for the table, which are displayed one per row. As the table is rendered, the JSF engine iterates through each object on the list and creates a "temporary" variable using the name specified in the `var` attribute.

The `<h:column>` tags define the columns for the table. The body of the tag contains the components that will be included for that column in each row of the table. By including an `<h:outputText>` tag in the column definition, we can display a value for each object in the list.

Data tables also support *facets* via the `<f:facet>` tag. A facet allows you to define specific additional behavior for use under certain conditions. Data table columns support a `header` facet and a `footer` facet for each column. The renderer uses this information to generate header and footer rows where appropriate. In this case, we use header facets (in conjunction with the `headerClass` attribute) to display a set of header rows.

5.6.2.1. Editing data tables

We've also made this table editable by the extremely simple expedient of using an `<h:selectBooleanCheckbox>` tag to display the `checkedOut` property of each book. We use an `onclick` JavaScript handler to submit the form whenever the box is checked or unchecked. This is all we have to do to update the underlying modelJSF will take care of the rest, including associating the individual checkboxes with the appropriate Java beans. Of course, if this doesn't sound like good separation of the real data model and the application, you're right, and we'll look at that next.

If we want to make the administration view available to any user, but limit check in and check out capabilities to librarians, we can include two components in the column and use the `rendered` attribute to control which one is displayed:

```
<h:column>
  <f:facet name="header">
    <h:outputText value="Checked Out" />
  </f:facet>
  <h:selectBooleanCheckbox
    onclick="submit( )"
    value="#{book.checkedOut}"
    rendered="#{usersession.currentUser.librarian}" />
  <h:outputText value="#{book.checkedOut}"
    rendered="#{not usersession.currentUser.librarian}" />
</h:column>
```

5.6.2.2. Data model objects

Data tables don't actually access the underlying objects directly. Instead, the underlying objects (lists, JDBC result sets, and so forth) are wrapped inside an object that extends the `javax.faces.model.DataModel` class. The `<h:dataTable>` component interacts with the `DataModel` implementation, which handles interaction with the model itself.

Why is this important? In our example, we're accessing the library's book list directly. That makes it easy for us to support editing by clicking and unclicking checkboxes and keeps the code to a minimum. But what if we want to support sorting by title or by author? We can't simply sort the main list every time the user sorts since that list is shared by every user in the system. By writing a

custom data model (or a wrapper for one of the standard data model implementations), we can maintain a separate "view" into the underlying data for each session.



5.7. Validation

Most software requires some kind of data entry, and in order to be user-friendly, an application should support the user as much as possible throughout the data entry process. This means identifying invalid input and giving the user a chance to fix it sooner rather than later.

JSF, as you might expect, has built-in support for validation. All JSF user interface components support validation through both built-in and custom validation rules.

Let's take a look at the Add Books form ([Figure 5-4](#)) in the JSF Library, which needs to enforce minimum field lengths and mandatory fields. The Add Books function is available to librarians who have logged into the system. It allows them to fill out a simple form to create a new book.

5.7.1. Adding Validators to Forms

[Example 5-8](#) shows the JSP source for the Add Books form. We're using three different kinds of validation: required fields, built-in validators, and a custom validator (which we'll see in more detail in the next section). We're also using the `<h:message>` tag to display validation messages for each component.

Figure 5-4. Add Books form

Example 5-8. /view/addbook.jsp

```

<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>

<html>

<f:view>
<head>
  <title>Add Book to Library</title>
  <link rel="stylesheet" type="text/css"
        href='<%= request.getContextPath( ) + "/css/app.css" %>'>
</head>
<body>
  <h1>Add Books to Library</h1>
  <hr>
  Enter information below to add a book to the library.
<h:form>
  <table>
  <tr>
  <td>Title:</td>
  <td><h:inputText id="title" binding="#{addbookform.title}"
        required="true">
        <f:validateLength minimum="5"/>
        </h:inputText>
        <h:message styleClass="validationWarning" for="title"/>
  </td>
  </tr>
  <tr>
  <td>Author:</td>
  <td><h:inputText id="author" binding="#{addbookform.author}"
        required="true"/>
        <h:message styleClass="validationWarning" for="author"/>
  </td>
  </tr>
  <tr>
  <td>ISBN:</td>
  <td><h:inputText id="isbn" binding="#{addbookform.isbn}"
        required="true">
        <f:validator validatorId=
          "com.oreilly.jent.jsf.library.validator.ISBNValidator"/>
        </h:inputText>
        <h:message styleClass="validationWarning" for="isbn"/>
  </td>
  </tr>
  </table>

  <p>

```

```

    <h:commandButton value="Add Book" action="#{addbookform.addBook}"/>
    <h:commandButton value="Verify" action="verify"/>
  </p>

</h:form>

</body>

</f:view>
</html>

```

Each time a JSF form is submitted, the engine runs through a Process Validations phase before any other actions take place. The validation process goes through each component and checks for associated validators. The condition associated with each validator is evaluated, and if the validator fails, a message is added to the page's message queue. Regardless of success or failure of any particular validator, every validator on the page is checked. If any validator fails, processing of the request is aborted and the user is returned to the original form, which can display the error message produced by the validators and allow the user to try submitting again.

The `<f:validateLength>`, `<f:validateDoubleRange>`, and `<f:validateLongRange>` tags are all provided by the JSF implementation and allow validation of minimum lengths, floating-point double ranges, and long integer ranges, respectively. Each of these tags can be embedded within an input component to apply the specified validation behavior to that component. You can add multiple validators to a single component, and they will be processed in order.

Setting a field to `required` works a bit differently, as the various input components have a `required` attribute that can be set to `true` to make the fields mandatory (we do this for all three in [Example 5-8](#)). This check is also performed at the validation stage, and leaving out any required fields produces the same kind of message that any other validator would.

5.7.2. Displaying Validation Messages

The `<h:message>` and `<h:messages>` tags display the messages produced by the system during the validation phase. We saw the `<h:messages>` tag in [Example 5-5](#); it's used to show all of the messages produced during page processing.

Messages, as we saw earlier, have two components: a high-level message and a detailed message. Messages are also associated with particular components: in this case, with the components that failed validation. This lets us display a validation error message alongside the component that the message is associated with. The `<h:message>` tag displays the detailed message associated with a particular component:

```

<h:message styleClass="validationWarning" for="title"/>

```

The `for` attribute specifies the component ID for the component for which we want to display messages. Note that we have to specify component IDs manually rather than allowing the JSF engine to do it for us.

5.7.3. Custom Validators

The basic validators provided with the JSF API are useful but don't cover every possible situation. But not to fear you can create your own validator classes by implementing the `javax.faces.validator.Validator` interface. We've built a simple one in [Example 5-9](#) for validating ISBN numbers, using the simple ISBN checksum algorithm.^[*] With this validator, we made a design decision not to throw an error if the ISBN is null; we leave that decision to the `required` attribute on the input field and display an error only if the input provided isn't a valid ISBN.

[*] The ISBN algorithm is simple. ISBNs may contain the digits 0-9 as well as the character X, which means 10. Multiply each digit by its distance from the righthand side of the ISBN code (10 for the first digit, 9 for the second digit, etc). Add the 10 products and divide by 11. If there is no remainder, the code is valid.

Example 5-9. ISBNValidator.java

```
import javax.faces.validator.Validator;
import javax.faces.validator.ValidatorException;
import javax.faces.context.FacesContext;
import javax.faces.component.UIComponent;
import javax.faces.application.FacesMessage;

public class ISBNValidator implements Validator {

    public void validate(FacesContext facesContext,
                        UIComponent uiComponent, Object o)
                        throws ValidatorException {

        String val = o.toString( );
        if (val != null) {
            if (val.length( ) == 10) {
                if(!checkISBN(val))
                    throwValidatorException(
                        "ISBN does not pass validation");
            } else {
                throwValidatorException("ISBN must be 10 characters");
            }
        }
    }

    private boolean checkISBN(String ISBN) {
        int sum = 0;
        for (int i = 0; i < 10; i++) {
            char current = ISBN.charAt(i);
            int num = 0;
            if (current == 'X')
                num = 10;
            else
```

```

        num = current;
        sum += (num * (10-i));
    }
    return (sum % 11 == 0);
}

private void throwValidatorException(String detail)
    throws ValidatorException {
    FacesMessage message = new FacesMessage( );
    message.setSeverity(FacesMessage.SEVERITY_ERROR);
    message.setSummary("Invalid ISBN");
    message.setDetail(detail);
    throw new ValidatorException(message);
}
}

```

In order to use this validator, we need to register it in the *faces-config.xml* file, associating a validator ID with the validator class. Here's how we do it for the library:

```

<validator>
  <validator-id>
    ISBNValidator
  </validator-id>
  <validator-class>
    com.oreilly.jent.jsf.library.validator.ISBNValidator
  </validator-class>
</validator>

```

Once a custom validator is registered, we invoke it using the validator ID and the `<f:validator>` tag. Note that even though we've chosen to use the full package and class name as the validator ID (this is good practice in general, since it prevents confusion), we still have to register it before we can use it.

```

<f:validator
validatorId="ISBNValidator"/>

```

[← PREV](#)

5.8. Moving on with JSF

We couldn't cover the full range of JSF features in the few dozen pages available to us here. For a more in-depth look, *JavaServer Faces* (O'Reilly) delves deeper into the essential JSF functionality and provides an introduction to the art of creating your own custom JSF components.

[← PREV](#)

Chapter 6. Enterprise JavaBeans

RMI and JavaBeans bring a standard distributed object framework and a component model, respectively, to the general Java environment. The Enterprise JavaBeans architecture builds on these foundations to provide a standard *distributed component model* for the enterprise Java environment.

So, you may ask, how are EJB components different from "regular" RMI and CORBA distributed objects or nondistributed JavaBeans components? Well, in a nutshell (so to speak), an EJB includes the capabilities of both and adds an entire set of enterprise services. An EJB component has the remote capabilities of an RMI or CORBA object, in the sense that it can be exported as a remote object using RMI or RMI/IIOP. An EJB component is also a JavaBeans component since it has properties that can be introspected and it uses JavaBeans conventions for defining accessor methods for its properties. The EJB architecture provides a framework in which a developer can easily take advantage of transaction processing, security, persistence, and resource-pooling facilities provided by an EJB container. These facilities don't come for free, of course. You need to understand how they work and what rules your EJB object needs to follow in order to take advantage of these services.

Enterprise JavaBeans are useful in any situation in which regular distributed objects are useful. They excel, however, in situations that take advantage of the component nature of EJB objects and the other services that EJB objects can provide with relative ease, such as transaction processing and persistence. A good example is an online banking application. A user sitting at home wants to connect to all of her financial accounts, no matter where and with whom they may live, and see them tied together into one convenient interface. The EJB component architecture allows the various financial institutions to export user accounts as different implementations of a common `Account` interface, just as we would do with other distributed object APIs. But since the `Account` objects are also EJBs, we can use the EJB server's transaction management capabilities to implement the `Accounts` as transactional components, which allows the client to perform a number of account operations within a single transaction and then either commit them all or roll them back. This can be a critical feature in financial applications, especially if you need to ensure that a supporting transfer can be executed before a withdrawal request is submitted. The transactional support in EJB ensures that if an error occurs during the transfer and an exception is raised, the entire transaction can be rolled back and the client-side application can inform you of the reason. And of course, an application such as this may also take advantage of any of the other EJB-related component services.

The EJB component model insulates applications and beans (for the most part) from the details of the component services included in the specification. A benefit of this separation is the ability to deploy the same enterprise bean under different conditions, as needed by specific applications. The parameters used to control a bean's transactional nature, persistence, resource pooling, and security management are specified in separate *deployment descriptors*, and not embedded in the bean implementation or the client application code. So, when a bean is deployed in a distributed application, the properties of the deployment environment (client load levels and database configuration, for example) can be accounted for and reflected in the settings of the bean's deployment options.

The EJB API is a standard extension to Java, available in the `javax.ejb` package and its subpackages. You must install this extension API explicitly to write code against the EJB interfaces. You can find the latest version of the API at <http://java.sun.com/products/ejb>. You should also note that EJB is just a

specification for how distributed components should work within the Java environment. In order to actually create and use EJB objects, you need to install an EJB-enabled server. J2EE-compliant application servers provide this service and are bundled with the standard EJB API classes and interfaces.

Finally, it's important to note that this chapter provides only a basic introduction to Enterprise JavaBeans, which is an extensive specification with many features. For more complete coverage, see *Enterprise JavaBeans* by Richard Monson-Haefel (O'Reilly).



[← PREV](#)

6.1. What Version Is Covered Here?

As discussed in the introduction to this book, the current versions of virtually all the prominent J2EE servers support the 1.4 version of the J2EE specification. This translates to supporting the EJB 2.1 specification, the version required by J2EE 1.4. At the time of this writing, however, draft versions of the EJB 3.0 specification have been distributed, and various vendors and open source projects have announced support for the new version. Since the EJB 3.0 specification is still in development and likely to change before it is finalized, we've restricted our coverage in the chapter to EJB 2.1. But at the end of the chapter, we do provide a short summary of the new programming model being planned for EJB 3.0.

The code examples in this chapter have been tested in the EJB container in JBoss 4.

[← PREV](#)

6.2. EJB Component Model Overview

[Chapter 13](#) discusses the two fundamental roles in the RMI environment: the client of the remote object and the object itself, which acts as a kind of server or service provider. These two roles exist in the EJB environment as well, but EJB adds a third role, called the *container*. [Figure 6-1](#) shows a conceptual diagram of how the three EJB roles interact. The container is responsible for interceding between the client (any code that is invoking the EJB) and the EJB component itself. The container also provides all the extra services for an EJB object mentioned earlier: transaction processing, security, object persistence, and resource pooling. If you're familiar with CORBA, you can think of the EJB container as being roughly equivalent to the ORB in CORBA, with a few of the CORBA services thrown in as well. In EJB, however, the container is strictly a server-side entity. The client doesn't need its own container to use EJB objects, but an EJB object needs to have a container in order to be available for client use.

Figure 6-1. The basic roles in an EJB environment

6.2.1. The Enterprise JavaBeans Object

At the heart of an EJB component is the actual implementation object. The EJB implementation class is where the real value of the EJB lies—the business methods that clients want to invoke are implemented here.

The EJB object:

- Implements the business logic behind the operations in its client interfaces
- Interacts with the EJB container to make use of component services, like security and transaction management

6.2.1.1. Types of EJBs

The three fundamental types of Enterprise JavaBeans are *session*, *entity*, and *message-driven*. The key difference between these component models lies in how they are managed during their lifetime by the EJB container and what component services are available to them through the container. Before implementing an EJB component, it's important to understand these different types and decide which one is most appropriate for a particular situation.

Session bean

Typically accessed by one client at a time and is nonpersistent. It lives for a specific period of time (a session) and then gets removed by the server.

Entity bean

Represents a data entity in persistent storage (e.g., a database or filesystem). Entity EJBs that represent the same persistent entity can be accessed by multiple clients concurrently. The data that defines a particular entity can be persistent beyond a client session, beyond the life of a particular EJB component instance, and beyond the lifetime of the EJB server.

Message-driven beans

Seen by clients as a JMS destination, but are managed by the EJB container similar to a session EJB. Clients send messages to the JMS destination, the messages are mapped to operations on the EJB component, and the results, if any, are mapped back into JMS messages that are accessible by the client through the JMS destination.

To illustrate the differences between session, entity, and message-driven beans, suppose you're building an online banking system using EJB components. An automated teller machine, which reports on account balances and executes deposits and withdrawals on specified accounts, could be implemented as a session bean. A single client uses the teller bean to perform services on bank accounts that are maintained in some separate persistent store (the bank's database). An EJB object that directly represents a bank account, however, should be an entity bean. Multiple clients can access the account to perform transactions, and the state of the account entity should be persistent across the lifetime of the online banking server. If we want to provide wireless access to the banking system from PDAs or mobile phones, we might have these devices communicate with the system

using lightweight, asynchronous messaging, since their network connectivity is typically unstable and low in bandwidth. This may lead us to use message-driven EJBs, which can respond to these client messages but still take advantage of the lifecycle management capabilities of the EJB container.

6.2.1.2. EJB code artifacts

At a minimum, an EJB component consists of an implementation class that is deployed to the EJB container, plus a deployment descriptor that tells the EJB container how to manage the component. Depending on the specific type of EJB you are implementing, the EJB may also need to include one or more client interfaces and server-side support classes required for the EJB container to manage the component.

Specifically, a message-driven bean requires only an implementation class and a deployment descriptor. These beans are seen by clients as JMS destinations to which they send messages, so there's no need for client interfaces for these EJBs.

Session beans are created by clients using their home interface(s) and operated by clients using their client interfaces, so these interfaces (in addition to the implementation class and deployment descriptor) are required when implementing session EJBs.

Entity EJBs are the most complex to implement because they make the most use of the EJB container's services. In addition to the implementation class, home interface(s), client interface(s), and deployment descriptor, an entity bean may also require a primary key class. Entity beans also have a significant amount of additional information in their deployment descriptors, related to their persistence properties.

6.2.2. The EJB Client

An EJB client uses EJB objects to access data, perform tasks, and generally get things done. In the EJB environment, the first action a client performs is to find the home interface for the type of EJB object that it wants to use. This home interface is a kind of object factory, used to create new instances of the EJB type, look up existing instances (only when using entity EJB objects, discussed later), and delete EJB objects. This is a bit different from RMI, in which the client first has to get a direct handle to an existing RMI servant. In many RMI applications, this first RMI object is a kind of object factory that is used to create other RMI object references. So, in a sense, the use of home interfaces in EJB is just formalizing the role of factory objects in distributed component applications.

EJB home interfaces are located by clients using JNDI (described in [Chapter 9](#)), the same way that other J2EE resources like JDBC `DataSource`s and JMS `ConnectionFactory` resources are accessed. An EJB server publishes the home interface for a particular EJB object under a particular name in a JNDI namespace. The EJB client needs to connect to the JNDI server and look up the EJB home interface under the appropriate name in order to start things off.

We'll see concrete examples of EJB clients a bit later in the chapter, but here is a summary of the fundamental steps an EJB client performs:

1. Gets a JNDI context from the EJB/J2EE server

2. Uses this context to look up a home interface for the bean you want to use
3. Uses this home interface to create (or find) a reference to an EJB
4. Calls methods on the bean

6.2.3. The EJB Container

Most readers need to be familiar with EJB containers only from the perspective of an EJB client or an EJB object. A J2EE application server provides an internal EJB container, along with other required aspects of the J2EE framework. EJB-enabled application servers, with their own EJB containers and deployment tools, are available from BEA, JBoss, IBM, Sun, and many others.

The EJB container supplies the value-added features that EJB provides over standard remote objects built using RMI or CORBA. The EJB container manages the lifecycle of your EJBs and all the details of transaction processing, security, resource pooling, and data persistence. For components that require these services, the use of the EJB container reduces the burden on both client applications and EJB objects, allowing them to deal with just the business at hand.

An EJB container is the heart of an EJB environment, in the same way that an ORB is the heart of a CORBA environment or a web server is the heart of a web environment. As depicted in [Figure 6-1](#), all runtime interactions between the clients and the EJB objects themselves are mediated by the EJB container (although from the client's perspective, it seems as though they are interacting directly with the EJB component itself).

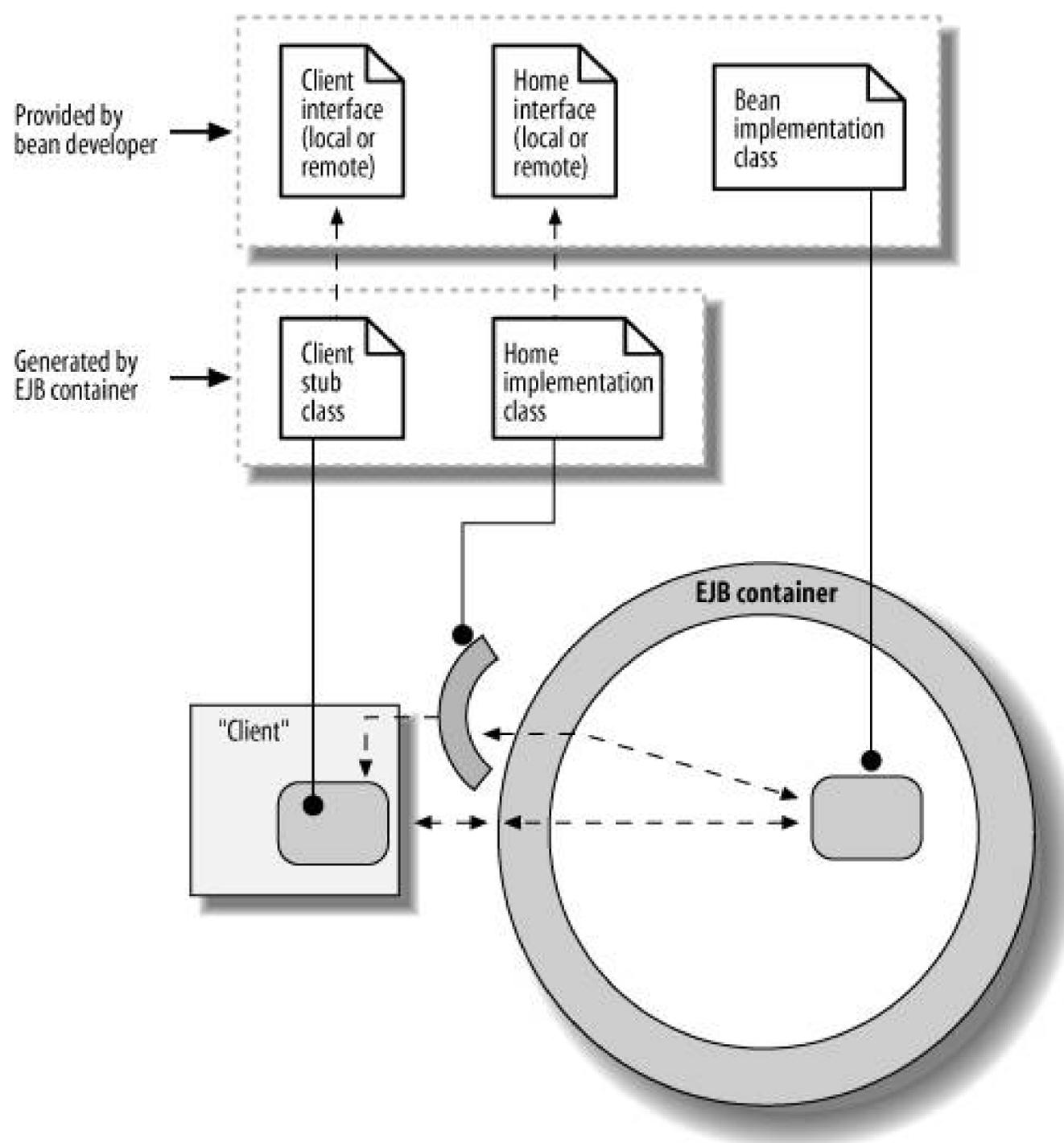
The EJB container takes all of the elements that make up your EJB (the implementation class, any client interfaces, etc.) and a deployment descriptor, and it uses these to generate the various support classes needed to manage and run the EJB component, as shown in [Figure 6-2](#). You provide any needed home and client interfaces, and the container generates both the client and the server-side implementations for these interfaces. As the figure shows, EJBs can support both local and remote clients. When a remote client looks up a session or entity bean's remote home interface through JNDI, it receives an instance of a remote stub class. All methods invoked on this stub are remotely invoked, via RMI or IIOP, on the corresponding home implementation object on the EJB server. Similarly, if the client creates or finds any beans through the remote home stub, the client receives remote object stubs, and methods invoked on the stubs are passed through RMI or IIOP to corresponding implementation objects on the server. These remote objects are linked, through the EJB container, to a corresponding enterprise bean object, which is an instance of your bean implementation class. Local clients interact with EJBs in a simplified, more efficient way. Local home objects and bean interfaces are obtained the same way as remote ones, but the objects obtained are nonremote objects that interact locally (i.e., from within the same Java virtual machine) with their EJB implementations.

Note that [Figure 6-2](#) demonstrates the situation for session and entity beans. Message-driven beans are accessed by clients using JMS messages sent to a JMS destination, and EJBs that are published as web services are invoked by clients using SOAP calls. So in both of these cases, clients do not need home or client interfaces. Details on JMS and SOAP clients can be found in [Chapters 11](#) and [12](#), respectively.

It's important to remember that all client requests to create, look up, delete, or call methods on EJBs are mediated by the EJB container. It either handles them itself or passes the requests to

corresponding methods on to the EJB object. Once the client obtains a reference to an interface for an EJB object, the container intercedes in all method calls to the bean to provide the bean with required services (transaction management, lifecycle management, and security) and to notify the bean of any events related to these services (for example, the bean needing to reload its persistent data or the bean instance being destroyed by the container).

Figure 6-2. Relationship of bean-provider classes and container-generated classes



When you deploy an EJB object within an EJB server, you use deployment descriptors to specify how the container should manage the bean during runtime, in terms of all the services the container provides. Deployment descriptors contain parameter settings for these various options. These settings can be customized for each deployment of an EJB object. You might purchase an EJB object from a vendor and deploy it on your EJB server with a particular set of container management options while someone else who purchased the same bean can deploy it with a different set of

deployment options. We discuss the details of the runtime options available in deployment descriptor and how to use them later in this chapter when we talk about deploying EJB components.



6.3. EJB Tutorial

Now let's look at the generic steps that you take to implement an Enterprise JavaBeans component. In all of the examples that follow, we're showing you the "traditional" approach to writing EJBs: you create the client and home interfaces (where needed), the implementation classes, and the deployment descriptor by hand. A number of tools are available to simplify EJB development by creating these elements for you. [Chapter 21](#) discusses XDoclet annotations that can be used to automatically generate interfaces and deployment descriptors for EJBs using special Javadoc comments in your code. At the end of this chapter we also give a preview of similar features being integrated into the EJB 3.0 specification. But even with current and upcoming development aids, it's important to understand the underlying EJB programming model so that you can apply the tools effectively.

As we discussed earlier, every type of EJB (session, entity, message-driven) must include a bean implementation class. This class houses the bean's business logic; it's the heart of the component. All the other trappings of the EJB either give clients ways to access the bean or help the container manage the bean.

If you are creating either an entity or a session bean, you also need to provide:

Home interfaces

A home interface is used by clients to create or find EJB objects of a specific type. An EJB can have both local and remote home interfaces, though in most cases only one or the other is needed, based on the context in which the EJB is being deployed.

Client stub interfaces

When a client creates or finds an EJB object through a home interface, the container provides the client with a reference to a stub that implements the client interface for the bean. The interface defines the methods the EJB object exports to clients. As with home interfaces, an EJB can have both local and remote bean interfaces.

The EJB object implementation must implement all the methods defined in its remote and local interfaces, provide methods that correspond to the methods on its home interface for creating or finding the bean, and also implement a set of required methods used by the EJB container to manage the bean.

Message-driven beans require only the bean implementation class. They don't have home interfaces or client interfaces since the client interface is implemented using JMS message passing (we'll give more details on creating and using message-driven beans later in the chapter).

To demonstrate the various components that make up an Enterprise JavaBeans object, we'll look at a simple example: a profile server. The *profile server* is a bean that provides profile information for

named users. This profile information consists of name/value pairs that might represent preferences in an application, historical usage patterns, and the like. You might see a profile server running behind an online information service, personalizing the content and appearance of the site. After we've gone through this general example of writing a bean, we'll look more closely at the differences between implementing session and entity beans.

6.3.1. Client Interfaces

Although the bean implementation is central to the functions of the EJB component, it's typical to start putting together an EJB by defining its client interface(s). Client interfaces contain declarations of the methods that are available to clients.

6.3.1.1. Remote client interfaces

A remote client interface for our `ProfileManager` is shown in [Example 6-1](#). A remote EJB client interface must extend the `javax.ejb.EJBObject` interface. `EJBObject` in turn extends the `java.rmi.Remote` interface, which makes the remote client interface an RMI remote interface as well.

Example 6-1. Remote interface for the profile manager bean

```
import javax.ejb.*;
import java.rmi.RemoteException;

public interface ProfileManager extends EJBObject {
    public ProfileBean getProfile(String acctName)
        throws NoSuchPersonException, RemoteException;
}
```

The `ProfileManager` interface defines a single method, `getProfile()`, that accepts an account name as its only argument. It returns a `ProfileBean` object containing the profile information for the person named. If the person's profile can't be found on the server, a `NoSuchPersonException` is thrown. This is an application-specific exception whose implementation isn't discussed in this chapter. Since the `ProfileManager` interface is an RMI remote interface, its methods must throw `RemoteException` in case some RMI communication problem occurs during a method call. Also, the arguments and return values for the methods have to be `Serializable`, or they need to be exportable RMI objects themselves. Our `getProfile()` method returns a `ProfileBean` object, which we'll implement as a `Serializable` object, as shown below in [Example 6-2](#). The `ProfileBean`, not shown here but included in the example source code download for the book, is a fairly straightforward Java bean: it simply has methods that allow you to get and set the profile entry values.

6.3.1.2. Local client interfaces

A local client interface for an EJB is defined in a similar way, but the rules and usage of the interface

differ from those of its remote counterpart. As we've mentioned, local interfaces are used by clients that reside in the same JVM as the EJB itself, so they follow the same argument-passing and return-value rules as normal, nonremote Java classes: objects are passed by reference, and basic data types are passed by value. No remote operations are involved in the use of a local interface, so there's no need to ensure that method arguments and return values are `Remote` or `Serializable` object types. And methods in a local interface aren't required to throw `RemoteException`. Local interfaces extend the `EJBLocalObject` interface, which is a simplified, nonremote interface. A local interface for our profile server is shown in [Example 6-2](#).

Example 6-2. Local interface for the profile server

```
import javax.ejb.*;

public interface ProfileManagerLocal extends EJBLocalObject {
    public ProfileBean getProfile(String acctName)
        throws NoSuchPersonException;
}
```

An EJB implementation class needs to provide implementations for all the methods exposed in its remote and local interfaces.

6.3.2. Home Interfaces

The client needs a way to create a reference to a profile server, so we have to provide a home interface for our bean. Since clients use home interfaces directly, and since clients can be either remote or local, an EJB can also have remote and local home interfaces.

[Example 6-3](#) shows a remote home interface for our EJB. It provides a single `create()` method that takes no arguments and returns the bean's remote interface type, `ProfileManager`.

Example 6-3. Remote home interface for the profile server bean

```
import javax.ejb.*;
import java.rmi.RemoteException;

public interface ProfileManagerHome extends EJBHome {
    public ProfileManager create( ) throws CreateException, RemoteException;
}
```

A remote home interface for an EJB object extends the `javax.ejb.EJBHome` interface. The remote home interface is also an RMI remote interface since `EJBHome` extends `java.rmi.Remote`. The home interface can contain multiple `create()` methods that take various initialization arguments to create the bean. A corresponding local home interface is shown in [Example 6-4](#).

Example 6-4. Local home interface for the profile server bean

```
import javax.ejb.*;

public interface ProfileManagerLocalHome extends EJBLocalHome {
    public ProfileManagerLocal create( ) throws CreateException;
}
```

For each `create()` method on a home interface, the EJB object implementation must have a matching `ejbCreate()` method that takes the same arguments. In either remote or local home interfaces, `create()` methods are required to throw `javax.ejb.CreateException`, in case some error occurs during the EJB creation process. Create methods (and all other methods) on remote home interfaces must also throw `java.rmi.RemoteException` (or one of its parent exceptions) since the home interface is an RMI remote interface and some sort of network/communication error could occur between the client and the home implementation on the server. If the corresponding `ejbCreate()` method on the bean implementation throws any other exceptions, the `create()` method has to include these in its `throws` clause as well. As we'll see in our bean implementation in the next section, the bean's `ejbCreate()` method doesn't throw any special exceptions, so we don't need to add any additional exceptions in the home interfaces.

Home interfaces for entity beans can also include finder methods, used to find previously created persistent entity beans. We'll discuss them when we talk about entity beans in detail.

6.3.3. The Bean Implementation

Now that we have a home interface that lets clients create EJB references and interfaces that describe what the EJB can do for the client, we need to actually implement the EJB object itself. Every bean implementation class can be divided into three sets of methods: business method implementations, container callback methods, and internal utility methods used by the bean class itself. A complete implementation class for the `ProfileManager` EJB can be found in the downloadable source code for the book; we'll just look at the highlights here in terms of these three categories of methods.

The business method implementations must match the business methods that are exposed in the EJB's client interfaces. For our `ProfileManagerBean`, we have only one business method in the client interfaces: `getProfile()`. Our implementation of this business method is simple enough:

```
public Profile getProfile(String name) throws NoSuchPersonException {
    // First validate the provided name, throw a NoSuchPersonException
    // if it's not valid
    if (!validateName(name)) {
        throw new NoSuchPersonException("No user found matching name \"" +
            name + "\"");
    }
}
```

```

    // If the name is valid, find/create the profile. Here, we simply
    // create a new profile Java bean (e.g., no persistence is provided)
    Profile profile = new Profile(name);
    return profile;
}

```

We validate the name argument and create a new `Profile` JavaBean object for the named user. Note that this highly simplified implementation provides no persistence features every time a username is given, a new, empty profile is created and returned. We'll see later in the tutorial how to integrate persistence into the bean implementation.

The `validateName()` method is an internal utility method: it's private, and it's not included in any of the client interfaces, so it's not available for clients to invoke.

Every implementation class for any EJB object must implement the `javax.ejb.EnterpriseBean` interface. This is typically done indirectly, through the `SessionBean`, `EntityBean`, or `MessageDrivenBean` interface. These interfaces include the required callback/notification methods that the EJB container needs in order to effectively manage the lifecycle of the EJB at runtime. Our sample EJB implementation is a session bean, so the `ProfileManagerBean` class implements the `SessionBean` interface.

The implementation class must follow a few other rules. The class must be declared as `public` to allow the container to introspect the class when generating the classes that hook the bean to the container and to allow the container to invoke methods on the bean directly where necessary. The bean class doesn't implement the bean's remote or local interface. This may seem a bit strange at first, especially if you're familiar with remote object systems like RMI and CORBA, since the purpose of the bean is to provide a concrete implementation of the EJB's interfaces, and in these other contexts this is done through direct inheritance. But in an EJB context, the EJB container always mediates between a client method request and the actual call on the corresponding method on a bean instance. As long as the container is explicitly told the EJB's various interfaces and its implementation class (and we'll see how to do that with deployment descriptors in a later section), it has all the information it needs to make this connection. When the EJB server generates the classes that bridge the bean to the container, it also provides a class that implements the remote interface and acts as a proxy to the EJB class itself.

The EJB container callbacks actually make up the bulk of the implementation class. These methods serve as the hooks the EJB container uses to manage the bean as a component. They also implement some of the functionality provided to the client through the client home and stub interfaces. For example, since the home interfaces for this EJB include a no-argument `create()` method, we must provide an equivalent `ejbCreate()` method that will be invoked by the container on a bean instance when the `create()` method is invoked by a client on a home interface. The `ejbCreate()` method is a container callback that gives the EJB implementation object a chance to initialize any resources it might need to operate properly. In our case, resources must be set up (because our implementation is so simple), but we still need to provide a method implementation for the container:

```

public void ejbCreate( ) {
    System.out.println("ProfileManagerBean created.");
}

```

The other remaining methods on the implementation class, `ejbActivate()`, `ejbRemove()`, `ejbPassivate()`, and `setSessionContext()`, are also container callbacks, invoked by the container to manage the EJB instance during its lifetime.

All session and entity bean implementations have to provide the following container callback methods, which are used in the following ways:

```
public void ejbCreate(...)
```

Called on a bean instance after a client invokes a corresponding `create()` method on a bean's home interface. Create methods are required for session and message-driven beans and optional for entity beans. The arguments, if any, in the create method indicate the identity or starting state of the bean. Note that the return type of `ejbCreate()` methods depends on the component model being used by the EJB. Specifics for each type of EJB model are given in the sections on session, entity, and message-driven beans.

```
public void ejbPassivate( )
```

Called by the container just before a bean is to be serialized and stored in passive storage (e.g., disk, database) on the server. This callback allows the bean to release any nonserializable resources (e.g., open files, network connections).

```
public void ejbActivate( )
```

Called by the container when the bean has been deserialized from passive storage on the server, and is becoming eligible to receive client method requests again. It allows the bean to reclaim any resources freed during passivation (e.g., file handles, network connections) or restore any other state not explicitly saved when the bean was serialized.

```
public void ejbRemove( )
```

Called by the container just before the bean is to be removed from the container and made eligible for garbage collection. Again, the exact semantics of this method depend on the component model of the EJB. "Removing" a session or a message-driven bean simply pulls the bean out of its active state before it is actually destroyed by the container. Removing an entity bean means to delete the bean's persistent state data from the underlying database.

In our example, the `ProfileManagerBean` doesn't need to perform any actions in these methods, so they are implemented as empty methods that simply print message to standard output, indicating that they have been called.

6.4. Deploying EJBs

As with web components (servlets and JavaServer Pages), EJB components in the J2EE environment are packaged into jar files, and the components are described and configured within an application server using deployment descriptors that are included in these jar files. These deployment descriptors are based on a XML DTD that is published as part of the EJB specification. An overview of J2EE deployment concepts can be found in Chapter 2. In this section, we discuss some EJB-specific deployment details that complement the material in Chapter 2.

Once you've written the home and remote interfaces and the implementation of your enterprise bean, you need to deploy your beans in an EJB container, which involves the following steps:

1. Specify the deployment information and options for your bean, in the form of an XML file called a deployment descriptor. The information in this deployment descriptor includes the names of classes that serve as the client interfaces, the home interfaces, and implementation for your EJB as well as any transaction support options, access control settings, and so on. In addition to the basic deployment information, different types of EJBs (session, entity, message-passing) require different sets of additional metadata (e.g., data mappings for container-managed entity beans and session timeout for session EJBs).
2. Provide any vendor-specific deployment information beyond the information in the standard J2EE deployment descriptor. Some EJB container implementations allow/require you to specify additional deployment parameters in a container-specific way, either in an additional configuration file for the container or through a server management console.
3. Generate the container-provided classes shown earlier in Figure 6-2.
4. Package your EJBs into an `ejb-jar` file.

As shown in Figure 6-2, the EJB container generates a set of classes that are needed to deploy your EJB object. It's up to the EJB container to provide a tool or tools for generating these classes. Some may be command-line tools that read a standard EJB deployment descriptor and generate the needed classes, while others may be GUI tools that let you control the deployment options of your bean using a visual interface.

6.4.1. EJB Deployment Descriptors

All EJB deployment descriptors include a listing of the Java classes that serve as the home, client, and implementation classes for a particular EJB. They also include various metadata about the bean, such as a description, a display name for the bean (useful in application server management interfaces), and so on. Appendix A provides a complete reference for EJB deployment descriptors. We'll simply give a short overview here. It might be useful to refer back to the schema structure diagrams in "Enterprise JavaBeans (ejb-jar.xml)" in Appendix A as you read through this section.

The general structure for an EJB deployment descriptor is as follows:

```

<? xml version="1.0" encoding="UTF-8" ?>
<ejb-jar 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/ejb-jar_2_1.xsd"
        version="2.
<!-- Metadata about the contents of the ejb-jar file ... -->
<!-- Provide an enterprise-bean element for each EJB component
        contained in the jar -->
<enterprise-beans>
  <!-- Describe a session bean -->
  <session>...</session>
  <!-- Describe an entity bean -->
  <entity>...</entity>
  <!-- Describe a message-driven bean -->
  <message-driven>...</message-driven>
</enterprise-beans>
<!-- Any additional metadata (inter-EJB relationships, assembly info,
        etc.) -->
<relationships>...</relationships>
<assembly-descriptor>...</assembly-descriptor>
<ejb-client-jar>...</ejb-client-jar>
...
</ejb-jar>

```

The EJB descriptor starts with the usual XML preface. The top-level element in the deployment descriptor is an `<ejb-jar>` element, and it references the standard XML schema for EJB deployment descriptors, in this case referencing the schema from the `java.sun.com` server (see Chapter 7 for more details on XML schemas and namespaces). The `<ejb-jar>` root element contains an `<enterprise-beans>` element that lists each EJB in the jar. Each entry within the `<enterprise-beans>` element contains the deployment information for a single EJB, including the various classes that make up the EJB's implementation, runtime management parameters, and so on.

A complete deployment descriptor that describes our `ProfileManager` EJB is shown in Example 6-5.

Example 6-5. Deployment descriptor for the profile server EJB

```

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

<ejb-jar 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/ejb-jar_2_1.xsd"
        version="2.1">
  <!-- Description of contents -->
  <description>Introductory EJB Example:
                Stateless ProfileManager Bean</description>
  <display-name>ProfileManager Bean Example (stateless)</display-name>
  <enterprise-beans>
    <!-- A stateless session profile server EJB -->

```

```

    <session>
      <display-name>ProfileManager Bean</display-name>
      <ejb-name>ProfileManagerBean</ejb-name>
      <!-- Remote home interface -->
      <home>
        com.oreilly.jent.ejb.stateless.ProfileManagerHome
      </home>
      <!-- Remote bean interface -->
      <remote>
        com.oreilly.jent.ejb.stateless.ProfileManager
      </remote>
      <!-- Local home interface -->
      <local-home>
        com.oreilly.jent.ejb.stateless.ProfileManagerLocalHome
      </local-home>
      <!-- Local bean interface -->
      <local>
        com.oreilly.jent.ejb.stateless.ProfileManagerLocal
      </local>
      <!-- Bean implementation class -->
      <ejb-class>
        com.oreilly.jent.ejb.stateless.ProfileManagerBean
      </ejb-class>
      <session-type>Stateless</session-type>
      <transaction-type>Container</transaction-type>
    </session>
  </enterprise-beans>
  <ejb-client-jar>ProfileManagerClient.jar</ejb-client-jar>
</ejb-jar>

```

We're using the `<description>` and `<display-name>` elements to provide some text descriptions of the contents within the top-level `<ejb-jar>` element. The information contained in these elements might be used by a management console on an application server or within an EJB-enabled IDE.

The `<enterprise-beans>` element contains one element for each EJB contained in an EJB jar. There are three possible subelements of `<enterprise-beans>`, one for each type of EJB: `<session>`, `<entity>`, and `<message-driven>`. Our `ProfileManager` EJB is a session bean, so we use the `<session>` element here. Within the `<session>` element, we provide a display name for the bean itself and an `<ejb-name>` element that provides a name that can be used to refer to this bean from other J2EE components. Then we specify all the classes that make up our EJB: the remote home and client interfaces (`<home>` and `<remote>`), the local home and client interfaces (`<local-home>` and `<local>`), and the EJB implementation class itself (`<ejb-class>`).

At the end of the `<session>` element comes configuration information that is specific to session EJBs. In this case, we are using the `<session-type>` element to specify that this is a stateless session bean and the `<transaction-type>` element to indicate that we want transactions managed by the EJB container for this EJB. This information is used primarily by the EJB container to determine how the EJB should be managed at runtime. The information that is specified in this section of the deployment descriptor depends on the type of EJB being described (session, entity, message-driven). Examples of the kind of metadata required for the various types of EJBs are provided later in this chapter.

6.4.2. Security-Related Deployment Attributes

Security management is another important service provided by the EJB container for your components. Essentially, the EJB container needs to provide some means for mapping a client identity to a named user or role defined on the EJB server itself. Then you, the bean provider, specify which users and/or roles can access each method on your bean. This approach to specifying security details keeps the access details out of the bean implementation code and allows the same bean to be deployed in different EJB containers, using different underlying security systems and access settings that suit the local context.

If you do need to resort to programmatic security checks in your EJB components, the `EJBContext` interface provides access to the user's identity and roles. Refer to "EJB Component Security" in Chapter 10 for further details on programmatic security in the EJB context.

6.4.2.1. Client security roles

The EJB specification allows you to specify security roles and associated access control levels in the EJB deployment descriptor. Suppose that we are deploying our EJB-based profile service and we want any client at all to be able to read profile entries, but we want only clients (individual users or client applications) identified as administrators to be able to set profile entries. To do this, we first have to define an appropriate security role in our EJB deployment descriptor. This is done in the `<assembly-descriptor>` element, which follows the `<enterprise-beans>` and `<relationships>` elements in the root `<ejb-jar>` element. Within the `<assembly-descriptor>`, you define a security role using the `<security-role>` element:

```
...
<assembly-descriptor>
    ...
    <security-role>
        <description>
            Profile administrators, allowed to update entries
        </description>
        <role-name>profile-admin</role-name>
    </security-role>
    ...
</assembly-descriptor>
...
```

In order for these security roles to actually mean anything, they need to be assigned to concrete principals and/or groups defined within the security realms of whatever J2EE application server you are using to deploy your EJBs. The specific details of how this is done depends on the particular vendor you are using. In BEA WebLogic, you would use the WebLogic administration interface to create new realms that refer to existing LDAP, RDBMS, native OS, and/or custom authentication services and then map your security roles to principals defined in these realms. Other application servers will provide other schemes for authenticating users and tying their identities into the EJB context.

6.4.2.2. Method permissions

So far we've declared some application-specific security roles, but we haven't given these roles any rights to invoke methods on our EJBs. To do this, we have to add one or more `<method-permission>` elements to the

`<assembly-descriptor>` . A `<method-permission>` element gives a particular role rights to access one or more methods on an EJB.

Suppose we defined a `Profile` EJB, with an implementation class called `BMPPProfileBean` , representing persistent data about a profile. This is an entity EJB version of the serializable `Profile` bean that we used in our stateless session example earlier (this EJB will be shown in detail in later sections). We might have a `setEntry()` method on this EJB that's used to alter entry values on the `Profile` , and we might want to restrict only the `profile-admin` role the rights to execute the `setEntry()` method on our `Profile` bean. If so, we would use a `<method-permission>` element such as:

```
...
<assembly-descriptor>
    ...
    <method-permission>
        <description>
            Only allow profile administrators to set entries on profiles
        </description>
        <role-name>profile-admin</role-name>
        <method>
            <ejb-name>BMPPProfileBean</ejb-name>
            <method-name>setEntry</method-name>
        </method>
    </method-permission>
    ...
</assembly-descriptor>
...
```

This allows any user who has the `profile-admin` role to invoke the `setEntry()` method on the `Profile` , while all other client methods on the bean are accessible to everyone. The `<role-name>` indicates which role should have access to this method, and the `method` element(s) indicates the EJB and method that this role can access. Here, we have a single method specified: any method named `setEntry` on the `BMPPProfileBean` EJB defined in this deployment descriptor. If `setEntry()` is overloaded on our bean, we can be more specific by including the method parameters of the single method we want made accessible:

```
...
<assembly-descriptor>
    ...
    <method-permission>
        <description>
            Only allow profile administrators to set entries on profiles
        </description>
        <role-name>profile-admin</role-name>
        <method>
            <ejb-name>BMPPProfileBean</ejb-name>
            <method-name>setEntry</method-name>
            <method-params>
                <method-param>java.lang.String</method-param>
                <method-param>java.lang.String</method-param>
            </method-params>
        </method>
    </method-permission>
    ...
</assembly-descriptor>
...
```

```

        </method>
    </method-permission>
    ...
</assembly-descriptor>
...

```

You can also give access to all methods on an EJB by using an asterisk (*) as the value of the `<method-name>` element. By default, all methods on all EJBs are accessible to everyone. Any methods not specified in a `<method-permission>` element in the deployment descriptor maintain this global accessibility.

6.4.2.3. Propagating identities

It's also possible in EJB to specify how identities are propagated from your EJB to other EJBs and resources by indicating which role your EJB will assume when its methods are executed. The identity used to execute the EJB methods will be passed forward when the EJB makes calls to other EJBs or accesses other J2EE resources. This identity will be used to determine access rights to these resources in the same way that caller's credentials are used to control access to the EJB itself and so on down the call chain.

To specify the runtime identity used to execute your EJB methods, use the `<security-identity>` element in the `<session>`, `<entity>`, and `<message-driven>` sections of the deployment descriptor. The `<security-identity>` element allows you to specify either that the methods on the bean will be invoked using the identity of the calling client or that a specific named security role will be used to invoke the bean methods.

The first option simply passes on the identity of the calling client when an EJB method invokes other EJB resources. To specify this option, include the following in the EJB's deployment descriptor section (a session bean is used as the example here):

```

...
<enterprise-beans>
    ...
    <session>
        ...
        <security-identity>
            <use-caller-identity />
        </security-identity>
        ...
    </session>
    ...
</enterprise-beans>
...

```

This scheme is most appropriate when the client operates in a context in which it can be authenticated against a shared authority (e.g., a servlet is the user entry point, and the user is challenged with an HTTP authentication prompt), and the user's identity can be propagated to the EJB container. Note that the `<use-caller-identity>` option can't be used with message-driven beans since there is no path for a user's credentials to be propagated to the EJB container via message destinations. A message-driven bean has to assume its own role when handling messages.

The other alternative is to specify a role under which all invocations of your EJB will be executed. For this

use the `<run-as>` option in the `<security-identity>` :

```

...
<enterprise-beans>
  ...
  <session>
    ...
    <security-identity>
      <run-as>
        <role-name>profile-admin</role-name>
      </run-as>
    </security-identity>
    ...
  </session>
  ...
</enterprise-beans>
...

```

Here, we're specifying that all method invocations on our EJB will run under the `profile-admin` role and we assume that role's access rights when invoking other EJBs and resources.

6.4.3. Packaging EJBs

An `ejb-jar` file is the standard packaging format for Enterprise JavaBeans. It is a normal Java archive (jar file), created using the standard `jar` utility, but it contains specific files that provide all the information needed for an EJB container to deploy the beans that are contained in the jar file. An `ejb-jar` file can contain one or more EJBs.

An `ejb-jar` file contains two types of files:

- The class files for each bean, including their home and client interfaces, and the bean implementation. An `ejb-jar` file can also include container-generated classes (concrete implementations of home and client interfaces, for example).
- Deployment descriptor files, as described in the previous sections. At a minimum, an `ejb-jar` file must include a standard `ejb-jar.xml` file. Different EJB container vendors may also require additional deployment descriptor files that allow you to control vendor-specific aspects of how your EJBs will be managed at runtime.

An `ejb-jar` file is a compact, efficient way to package a set of EJBs for deployment in an EJB container; it contains all the information an EJB container requires to manage your EJBs.

The compiled Java classes that make up your EJBs are included in the `ejb-jar` file in package-specific directories, just as in a regular jar file. The deployment descriptor that describes and configures the EJB must be included in the `ejb-jar` file as `META-INF/ejb-jar.xml`.

Some EJB container vendors include a utility to facilitate the creation of `ejb-jar` files from your bean class files. Ant also includes a task, `ejbjar`, that can be used to create `ejb-jar` files from your Ant buildfiles (see Chapter 17 for more details). It's a simple matter, however, to create an `ejb-jar` using the `jar` utility provided with

JDK. Assuming that you have created a valid *ejb-jar.xml* deployment descriptor file, such as the one shown earlier, simply compile all of your relevant classes into a given directory, create a *META-INF* directory in the same area, and put the *ejb-jar.xml* file in it, then use *jar* to create your *ejb-jar* file:

```
% jar cvf ProfileManager.jar com/oreilly/jent/ejb/*.class META-INF
added manifest
adding: com/oreilly/jent/ejb/ProfileManager.class(in = 344) (out= 225)(deflated 34%)
adding: com/oreilly/jent/ejb/ProfileManagerBean.class(in = 1254) (out= 598)(deflated 52%)
adding: com/oreilly/jent/ejb/ProfileManagerHome.class(in = 317) (out= 208)(deflated 34%)
adding: com/oreilly/jent/ejb/ProfileManagerLocal.class(in = 327) (out= 214)(deflated 34%)
adding: com/oreilly/jent/ejb/ProfileManagerLocalHome.class(in = 305) (out= 198)
(deflated 35%)
ignoring entry META-INF/
adding: META-INF/ejb-jar.xml(in = 0) (out= 0)(stored 0%)
```

This command creates an *ejb-jar* file named *ProfileManager.jar* in the current directory. Note that this *ejb-jar* file doesn't contain any container-generated classes; these need to be created using tools provided by the EJB vendor, as described in the next section.

6.4.3.1. Generating the container classes

So far we've seen the basic elements that make up an EJB (home interface(s), client interface(s), and an implementation class), how the configuration parameters for one or more beans are specified in an XML deployment descriptor, and how all of these can be packaged into a standard *ejb-jar* file. In order for an container to deploy your EJBs, it still needs to generate the container-specific classes depicted in Figure 6-1. How this is done depends on the EJB container you are using. If you are using BEA's WebLogic application server, for example, you can use the *appc* tool that is provided with the server to generate the runtime support classes needed by WebLogic. If we've packaged our EJB classes and deployment descriptor into an *ejb-jar* file named *ProfileManager.jar*, we can use the *appc* tool like so (for a Unix environment):

```
> java -classpath $WL_HOME/lib/weblogic.jar:$CLASSPATH weblogic.appc -output
wl-ProfileManager.jar ProfileManager.jar
```

If the EJB(s) contained in the specified *ejb-jar* file check out (the interfaces, implementation classes, and deployment descriptor meet the EJB specification requirements), this will generate a new *ejb-jar* file, *wl-ProfileManager.jar*, that contains all the classes needed to run your EJB(s) within the WebLogic server. This *ejb-jar* file can be directly deployed to the server.

Each EJB container will have its own scheme for generating support classes for your EJBs. Consult your vendor's documentation for specific details.

6.4.3.2. Delivering the package

Finally, after writing, configuring, and packaging one or more EJB components, you need to actually deliver the EJBs to the server running the EJB container. Again, this procedure is not specified by the EJB or J2EE specification; each server implementation is free to do this in whatever way is most appropriate for it. In some cases, EJB components are deployed by simply placing the *ejb-jar* file in a specific directory on the

server. In other cases, a server management tool (web-based or otherwise) needs to be used to upload ejb-jar file to the server to deploy it.



6.5. Using Enterprise JavaBeans

So far, we've seen the high-level details on implementing and deploying an EJB. Now let's look at how you use an enterprise bean as a client.

It's important to note that a "client" for a given EJB can be many things. In a fairly typical scenario, a client may be a Java servlet or JSP, running in the same or different J2EE server as the EJBs being used. A client can also be a standalone GUI client (in J2EE parlance, an "application client") that directly connects to a local or remote EJB container and makes requests of beans. A client can also be another EJB, invoking an EJB on the same or different EJB server in order to satisfy a client request. Regardless of which of these scenarios is the case, the following details apply.

6.5.1. Local Versus Remote Clients

A local client (i.e., a client that resides in the same Java virtual machine as the EJB container) can interact with EJBs using either the local or remote interfaces for the EJB. If the EJB has a local home and client interface, this provides an efficient way for the client to make requests, since their method calls don't need to be marshaled into a remote method call and transmitted over an RMI or IIOP connection to the EJB container. If the EJB doesn't have a local client and home interface, or if the local interfaces don't expose the functionality that the client needs (remember that local and remote interfaces for the same EJB don't need to be equivalent to each other), a local client must use its remote home and client interface. Local EJB interfaces were introduced in EJB 2.0, so if you are using EJB 1.1, your EJBs can have only remote interfaces.

Remote clients (i.e., clients that run in a different Java virtual machine than the EJB container) must use the EJB's remote client and home interfaces. Remote interfaces are RMI client stubs (see [Chapter 13](#) for full details on Java RMI), and the underlying remote method protocol being used can be RMI/JRMP (the "native" RMI protocol) or RMI/IIOP (the CORBA protocol). In order for a client to maintain portability across different EJB implementations, it's important that remote references obtained from the EJB container be cast to their interface types using the `narrow()` method on `javax.rmi.PortableRemoteObject`. This ensures that RMI/IIOP remote references will be safely cast to their expected type.

6.5.2. Finding Home Interfaces Through JNDI

Once an enterprise bean is deployed within an EJB container, the home interface(s) for the bean have been exported under a particular name using JNDI according to the settings in the deployment descriptor. As a client, you need to know how to connect to the JNDI context of the EJB/J2EE server and know the name for the bean home interface you're interested in. Full details on using JNDI and the options for creating `InitialContexts` can be found in [Chapter 9](#), but in general you need to set connection properties required for the JNDI provider you're connecting to and then create an `InitialContext` object that points to the root of the naming system:

```

Hashtable props = new Hashtable( );
// Specify the necessary properties
...
// Try looking up the context
javax.naming.Context ctx = null;
try {
    ctx = new javax.naming.InitialContext(props);
}
catch (NamingException ne) {
    System.out.println("Failed to connect to JNDI service for EJB server");
}

```

If you're running within a J2EE environment (e.g., the EJB client is actually a web component running within the same J2EE server), you typically won't need to specify any connection properties. The J2EE server is configured so that the default JNDI connection properties point to its internal JNDI service, and you can use the no-argument form of the `InitialContext` constructor:

```
ctx = new javax.naming.InitialContext( );
```

Now that we have a JNDI naming context from the EJB server, we can look up the home interface for the bean we're interested in. Assuming that we deployed our bean and specified to the EJB container that its JNDI name should be `ProfileManagerHome`, we can obtain a reference to a home interface to the bean with code such as the following:

```

ProfileManagerHome pHome = null;
try {
    Object pRef = ctx.lookup("ProfileManagerHome");
    pHome =
        (ProfileManagerHome)PortableRemoteObject.narrow(pRef,
                                                         ProfileManagerHome.class);
}
catch (NamingException ne) {
    System.out.println("Failed to lookup home for ProfileManager bean.");
}

```

Notice that we're using the `javax.rmi.PortableRemoteObject.narrow()` method to safely cast the remote reference obtained from the EJB container to the expected home interface type. This allows our client to work safely with EJB servers that provide RMI/IIOP remote references to their beans.

6.5.3. Creating and Finding Beans

As we saw in the previous sections, home interfaces for an EJB contain methods that allow a client to create new beans or find existing beans (for entity beans). Continuing our example client, assuming we're using our `ProfileManager` bean and its corresponding home interface shown in [Example 6-3](#), we

can create a reference to a `ProfileManager` bean as follows:

```
ProfileManager pServer = null;
try {
    pServer = pHome.create( );
}
catch (RemoteException re) {
    System.out.println("Remote exception while creating
ProfileManager: " +
re.getMessage( ));
}
catch (CreateException ce) {
    System.out.println("Error occurred during creation:
" + ce.getMessage( ));
}
```

We're using the `create()` method defined on our `ProfileManagerHome` interface. Since this is a remote home interface, we need to catch both `RemoteException` and `CreateException` in our client code, in case some error occurs while communicating with the remote EJB container or during the create process itself. Assuming all goes well, we can now use our `ProfileManager` bean, invoking its business methods as needed by our client.

In the sections that follow, we'll examine client usage of each type of EJB (session, entity, and message-driven) in more detail, after we discuss the implementation and deployment details of each EJB component type.

6.5.4. EJB Handles and Home Handles

Every bean's remote client interface (if it has one) extends the `EJBObject` interface. Among other things, this interface allows the client to obtain a serializable *handle* on the remote enterprise bean. This handle is a persistent reference to the bean that can be serialized and then stored in local storage on the client side or emailed as an attachment to other users, for example. Later, a client can deserialize the handle object and continue interacting with the bean it references. The handle contains all of the information needed to reestablish a remote reference to the enterprise bean it represents. Since this is useful only for beans that are still valid when the handle is reconstituted, it is usually applicable only to entity beans, since they are persistent. Handles to session beans can also be used this way, as long as the handle is serialized and deserialized during the lifetime of the session bean (e.g., the EJB container doesn't shut down in the meantime or the session bean is not removed from memory by the container).

You can get the handle for a bean using the `getHandle()` method on a remote bean object reference:

```
ProfileManager profileServ = ...;
Handle pHandle = pServ.getHandle( );
```

The `getHandle()` method returns a `javax.ejb.Handle` object. Typically, the `Handle` implementation for a particular EJB is provided by the EJB container and is generated automatically from the EJB remote client interface and bean implementation class that you provide. Every `Handle` implementation is a `Serializable` object, which allows it to be transmitted to remote clients and to be stored in serialized format, if needed:

```
ObjectOutputStream out = ...;
out.writeObject(pHandle);
```

Later, you can read the object back from its serialized state and obtain a reference to the same remote EJB, using the `getEJBObject()` method on the handle:

```
ObjectInputStream oin = ...;
Handle pHandleIn = (Handle)oin.readObject( );
EJBObject pRef = pHandleIn.getEJBObject( );
ProfileManager pIn =
    (ProfileManager)PortableRemoteObject.narrow(pRef, ProfileManager.class);
ProfileBean prof = pIn.getProfile("JohnSmith");
```

When we call `getEJBObject()` on the `Handle`, the `Handle` communicates with the source EJB container and attempts to construct a remote reference to the bean that the handle came from originally. As with other situations in which we receive a remote reference from the EJB container, we use the `PortableRemoteObject.narrow()` method to safely cast the remote EJB interface to the expected type, and then we can use the EJB.

It's also possible to get a handle for a remote home EJB interface. The same basic sequence of events can be followed for obtaining and serializing a home handle: obtain the handle by calling the `getHomeHandle()` method on a home stub instance, use the appropriate serialization technique (instantiate an `ObjectOutputStream`, store the handle in a binary field in a database, and so on) to store the handle:

```
ProfileManagerHome pHome = ...
HomeHandle pHomeHandle = pHome.getHomeHandle( );
ObjectOutputStream out = ...;
out.writeObject(pHomeHandle);
```

We can reconstitute the serialized `HomeHandle` later and use its `getEJBHome()` method to get a new remote home reference for the same EJB:

```
ObjectInputStream oin = ...;
HomeHandle pHomeHandleIn = (HomeHandle)oin.readObject( );
EJBHome pHomeRef = pHomeHandleIn.getEJBHome( );
ProfileManagerHome profileHomeIn =
    (ProfileManagerHome)PortableRemoteObject.narrow(
        pHomeRef, ProfileHome.class);
```

```
ProfileManager pServer = profileHomeIn.create( );
```

The EJB specification doesn't dictate the usable lifetime of a home handle. It may become invalid after the source EJB container restarts, it may expire after some timeout period, or it may remain valid for an extended period of time. Consult your EJB vendor documentation for specific details.



6.6. Session Bean Specifics

Now that we've seen all the basic elements of implementing, deploying, and using an EJB, let's look a bit at the specifics of implementing session beans.

Beans that simply provide a set of well-defined functionality to clients are best-suited to be session beans. A simplified view of session beans is to think of them as regular remotely callable objects (like RMI objects) that are managed by an EJB container and therefore have access to all the basic EJB services (such as lifecycle management and security). Another way to decide whether an EJB should be a session bean is to remember what a session bean isn't: session beans are not meant to represent persistent data entities. A client acquires a reference to a session bean and asks it to perform functional services by calling methods on the bean. If the chunk of functionality you want to package for clients fits this description, the session bean model is probably the right one.

A session bean gets its name from the fact that it doesn't live beyond the lifetime of its server in other words, it is not *persistent*. If an EJB client has a reference to a session bean and the host EJB server restarts, that session bean reference is no longer valid. You can reacquire a session bean of the same type from the same server after the restart, but it's not guaranteed to be in the same state as the bean you had before the server restart.

In terms of implementing session beans, the following sections cover specific requirements for session beans in addition to the general EJB requirements spelled out earlier.

6.6.1. Session Bean Implementation Classes

Session bean implementation classes must follow these rules:

- Session bean classes must implement the `javax.ejb.SessionBean` interface.
- The implementation class also must implement the `setSessionContext()` method specified in the `SessionBean` interface. The container calls this method on the session bean just after the bean has been created, passing in a `javax.ejb.SessionContext` object that represents the runtime context for the bean. This session context is valid for the life of the bean.

6.6.2. Stateless Versus Stateful Session Beans

Session beans can be stateful or stateless. A *stateless* session bean doesn't maintain client state across method calls. If a client makes a series of method calls or transactions with the stateless bean, the bean should be in the same effective state at the start of each method call or transaction. Our `ProfileManagerBean` is implemented as a stateless session bean the key signal of this is the use of the `Stateless` value for the `session-type` parameter in its deployment descriptor in [Example 6-5](#). The stateless session model makes sense for the `ProfileManager`: the component doesn't need to keep any client-specific state to perform its function. It just locates a profile for a named user and returns

it, and then it's out of the picture.

Since stateless beans don't have any client state, they are entirely interchangeable any instance of a stateless bean of a given type can be used to respond to a client request. And that's exactly how an EJB container manages stateless beans it keeps a pool of precreated beans on the server, and when a client request comes in, the container picks an instance at random, effectively, and invokes the required method on it. Stateless beans also don't need to be passivated since they have no state that needs to be restored when they're reactivated. The container simply destroys any stateless session beans it feels are no longer needed.

A *stateful* session bean, on the other hand, does maintain client-specific state that can be accessed and changed directly by a client. So a client that creates a stateful session bean owns it for the life of the bean, and no other clients ever get access to it. The bean stays active in the container until the client removes it or until the client's reference to the bean expires (e.g., the client dies and the lease on the remote reference expires), whichever comes first. Again, the container won't associate that particular bean instance with any other client the client that created it owns it for its lifetime.

To illustrate the difference between stateless and stateful session beans, let's take our stateless `ProfileManagerBean` and convert it to a stateful session bean. The `ProfileManagerBean` is stateless because all it does is accept requests for user profiles and return the profiles directly to the client as `Profile` objects, which are `Serializable` JavaBean objects. The client then interacts with the `Profile` directly, and the `Profile` holds the conversational state for the client, in the form of the values of the profile entries.

We can provide the same functionality by merging the stateless session `ProfileManager` EJB and the `Profile` Java bean into a single stateful session `Profile` EJB. Instead of creating a reference to a `ProfileManager` and calling its `getProfile()` business method to get a Java bean, the clients create a stateful `Profile` EJB directly using its home interface.

[Example 6-6](#) shows the remote interface for the `Profile` EJB (we won't implement a local interface for this example). It's similar to the interface for the `Profile` Java bean we used in the stateless `ProfileManager` example. It has `setEntry()` and `getEntry()` methods that access entries using their names. The `Profile` EJB also has accessors for the name of its user.

Example 6-6. Remote client interface for the Profile EJB

```
import java.rmi.RemoteException;
import javax.ejb.EJBObject;

public interface Profile extends EJBObject {
    // Get the name of the user associated with this profile
    public String getName( ) throws RemoteException;
    // Look up an entry by name
    public String getEntry(String name) throws RemoteException;
    // Set an entry value
    public void setEntry(String name, String value) throws RemoteException;
}
```

We'll need a home interface that clients can use to create these profiles. It's shown in [Example 6-7](#) we

provide only a single `create()` method, taking the name of the user who owns the profile. Notice that the `create()` method throws the exceptions required by the EJB standard (`RemoteException`, because this is a remote home interface, and `CreateException`), but it also throws an application-specific exception, `NoSuchPersonException`. We defined this exception for this application (it indicates that the named user could not be found on the server). We can add any application-specific exceptions that we want to `create()` methods on home interfaces, as long as we also include them in the signature of the corresponding `ejbCreate()` methods on the bean implementation class.

Example 6-7. Remote home interface for the Profile EJB

```
import java.rmi.RemoteException;

import javax.ejb.CreateException;
import javax.ejb.EJBHome;

public interface ProfileHome extends EJBHome {
    // Create a profile for a named person. Throws an exception if the person's
    // profile can't be found.
    public Profile create(String name)
        throws RemoteException, CreateException, NoSuchPersonException;
}
```

Creating a stateful session implementation of this `Profile` EJB is much the same as creating the implementation of the `ProfileManager` EJB. We need to provide all the same container callbacks because they are both session EJBs, we need to implement `ejbCreate()` methods for any `create()` methods on the home interface(s), and we need to implement the business methods exposed in the client interface(s). We won't show all the container callbacks here because they're much the same as the earlier `ProfileManager` example, and the full details can be found in the source code bundle for the book.^[*] Our home interface in [Example 6-7](#) has a single `create` method, so we need a corresponding `ejbCreate()` method:

[*] You can download the code bundle for this book from <http://www.oreilly.com/catalog/javaentnut3>.

```
// Create method with name of profile owner
public void ejbCreate(String name) throws NoSuchPersonException {
    mName = name;
    System.out.println("Profile EJB created for " + mName + ".");
}
```

The method implementation simply stores the username into an instance variable on the bean and prints a diagnostic message.

We also need to provide implementations of the business methods exposed in the client interface in [Example 6-6](#):

```
// Get the name associated with this profile
```

```

public String getName( ) {
    return mName;
}

// Get the value for a particular property in the profile
public String getEntry(String key) {
    return mEntries.getProperty(key);
}

// Change a value in the profile
public void setEntry(String key, String value) {
    mEntries.put(key, value);
}

```

Deploying this bean is very much like deploying our `ProfileManager` example. We need to put its details into the deployment descriptor as a `<session>` element within the `<enterprise-beans>` element of the `ejb-jar.xml` file. The key difference here is that we need to specify a value of `Stateful` for the `<session-type>` element, to instruct the container that it should manage this EJB using the stateful session bean model (e.g., clients own the beans they create, no other client gets access to an instance created by another client, and so on). We then package the `ejb-jar` file and perform the rest of the deployment steps as required by our particular EJB container.

Clients use the `Profile` bean much the same way as the `ProfileManager`. They first look up its home interface using a JNDI call to the EJB container, then create a `Profile` EJB using the home stub's `create()` method. This gives them a client stub that they can use to set and get profile entries. An example client use scenario is shown here:

```

// Get the Profile bean's home interface
ProfileHome pHome = (ProfileHome) context.lookup("ejb/ProfileHome");
// Create a profile for a person
System.out.println("Creating profile for " + name);
Profile profile = pHome.create(name);
// Get/set some entries in the profile
System.out.println("Setting profile entries for " + name);
profile.setEntry("favoriteColor", "blue");
profile.setEntry("language", "German");
System.out.println("Getting profile entries for " + name);
System.out.println("\tFavorite color: " +
    profile.getEntry("favoriteColor"));
System.out.println("\tLanguage: " + profile.getEntry("language"));

```

This example assumes that we've deployed the `Profile` bean's home interface under the name `ejb/ProfileHome` in the EJB container's JNDI context.

6.6.3. Container Management of Session Beans

Since stateless beans can be used by any client, the container pools stateless beans and doles them out to clients as needed. If new stateless beans are required, the container creates them; when they aren't (e.g., the rate of client requests decreases), they are simply destroyed. To allow the container to fill its pool, any stateless session bean must provide a single `ejbCreate()` method with no arguments. The container can call this whenever it decides another bean is needed in its ready pool.

Stateful beans are tied to the clients that create them, and they do maintain conversational state with their clients, so the container necessarily manages them differently. Stateful beans are created on demand when the container receives client requests for them. The container may still keep a pool of preconstructed instances of a stateful bean, but they aren't ready to accept method requests until a client calls one of the `create()` methods on the home interface, and the container subsequently calls the corresponding `ejbCreate()` method on one of the beans. At this point, the bean is "owned" by the client, and it's not removed until the client explicitly removes it by calling `remove()` on the client interface for the bean or until the client's reference to the bean goes away (the client quits or the bean reference goes out of scope on the client).

Stateful session beans can also be passivated and activated by the EJB container. The container may decide to do this in order to free up runtime resources on the server, for example. Passivating a session bean involves serializing its state and saving it to disk or some other storage area. When the container decides to passivate a stateful session bean, it invokes its `ejbPassivate()` method. The bean can do any cleanup of resources that it deems appropriate before it is serialized (close I/O streams or database connections, release file handles, and so forth). When the container decides to reactivate a passivated session bean, it first deserializes the bean, then invokes the bean's `ejbActivate()` method to give the bean a chance to restore any resources that it may need during runtime.

Stateless session beans aren't passivated or activated by the container since they aren't designed to maintain client state data. Instead, when the container decides a stateless session bean instance is no longer needed, it simply removes it entirely, and all of the stateless session bean's internal state is lost.

6.6.4. Optional Transaction Support

Stateful session beans can optionally receive notification of transaction boundaries from the EJB container. The container can notify the bean when a new client transaction is beginning and when the client transaction has either been completed or rolled back. For more information on EJB transaction management, see "[Message-Driven Beans](#)" later in this chapter.

Since session beans don't typically represent persistent shared data and stateful session beans can be accessed only by a single client at a time, user transaction boundaries may not be important in general to session beans. If, however, the session bean is managing database data for the user, it may want to know about the beginning and ending of user transactions, so that it can cache data at the start and commit its database updates at the end. For this reason, the EJB specification allows stateful session beans to optionally implement the `javax.ejb.SessionSynchronization` interface. By implementing this interface, the session bean indicates that it wants the container to notify it about the beginning and end of transactions.

In this case, the bean must implement the three methods that are declared on the `SessionSynchronization` interface: `afterBegin()`, `beforeCompletion()`, and `afterCompletion()`. The container calls the bean's `afterBegin()` method just after a new transaction begins. This lets the bean allocate any resources it might need during the transaction and cache database data, for

example. Just before the transaction completes, the container calls the bean's `beforeCompletion()` method. In this method, the bean can release any resources or cached data it may have initialized during the transaction. The `afterCompletion()` method is called just after the transaction has completed. The container passes in a `boolean` value that is `true` if the transaction was committed and `false` if the transaction was rolled back. The bean can use this notification to deal with rollbacks, for example, allowing the bean to undo any changes made during the transaction.



6.7. Entity Beans

An entity bean represents data that is stored in a database or some other persistent storage. Entity beans are persistent across client sessions and the lifetime of the server. No matter when or where you get a reference to an entity bean with a given identity, the bean should reflect the current state of the persistent data it represents. Multiple clients can access entity beans with the same identity at the same time. The EJB container manages these concurrent transactions for the underlying entity, ensuring that client transactions are properly isolated from each other, consistent, and persistent.

Returning to our running example of a profile service, we started with a session EJB, `ProfileManager`, that simply created profiles (represented as Java beans) for the client. Then we changed the design into a single `Profile` EJB, implemented as a stateful session bean. The major drawback in our stateful session `Profile` is that its profile data isn't persistent. A profile is created by a client and updated through client method calls, but once the `Profile` reference is given up by the client, or if the server crashes or shuts down for some reason, the accumulated profile data is lost. What we really want is a bean whose state is stored in a relational database or some other persistent storage that can be reloaded at a later time when the user reenters a profiled application. An entity EJB provides this functionality, and we can make our profiles persistent by implementing the `Profile` bean as an entity EJB. We'll use an entity bean implementation of the `Profile` EJB to demonstrate the features of entity EJBs.

At a basic level, entity beans are implemented similarly to session beans. You need to provide a local and/or remote home interface, a local and/or remote client interface, and a bean implementation. An entity bean, however, requires some additional methods in its home interface and bean implementation, to support the management of its persistent state and to allow clients to look up an entity in persistent storage. Entity beans must also provide a class that serves as its primary key, a unique identifier for entities.

The persistent data associated with an entity bean can be managed in two ways by the EJB container using bean-managed persistence (BMP) or container-managed persistence (CMP). With CMP, you leave the database calls to the container, and instead you tell the container (in the EJB's deployment descriptor) about bean properties and references that need to be persisted. The deployment tools provided with the EJB server are responsible for generating the corresponding database calls in the classes it uses to implement and deploy your bean. With BMP, you provide the actual database calls for managing your bean's persistent storage as part of your bean implementation.

If you can rely on the EJB container to handle your entity bean's persistence, this can be a huge benefit since it saves you from having to add (and debug) JDBC code to your beans and potentially makes your bean more portable across different persistent storage schemes (database vendors, schema variations, and even other persistence mechanisms like object databases). But even with the expanded CMP support provided in EJB 2.0 and 2.1, the automated persistence support in EJB is limited, and at times you'll need to manage persistence directly in your bean implementation. We discuss the pros and cons of each of these scenarios a bit later in this section.

6.7.1. Finder Methods

Since entity beans are persistent and can be accessed by multiple clients, clients have to be able to find them as well as create them. To this end, an entity bean's home interface(s) can provide finder methods, named with a `findXXX()` scheme, that a client can invoke to look up preexisting entities in persistent storage and have them returned in the form of an EJB reference. For each finder method, the bean implementation has to have a corresponding `ejbFindXXX()` method that takes the same arguments. The `findXXX()` methods on the home interface can have any name, as long as the method name begins with `find`. A person bean, for example, might define a `findBySSN()` method that lets you look up citizens of the U.S. using their Social Security number. The EJB implementation class must then have an `ejbFindBySSN()` method that implements this lookup function.

The entity version of our `Profile` bean has two finder methods defined in its remote home interface:

```
public Profile findByPrimaryKey(String name)
    throws RemoteException, FinderException;

public Collection findByEntryValue(String key, String value)
    throws RemoteException, FinderException;
```

The first one finds a `Profile` based on the name of the profile owner, which in our case is the unique identifier for a `Profile`. The second finder finds any `Profiles` that have a particular value for a named entry.

A client can use the `findXXX()` methods on the home interface to determine if an entity (or entities) with a given identity already exists in persistent storage. From the client's point of view, if a `findXXX()` method finds an appropriate entity or entities, entity bean instances are initialized and associated with these persistent entities, and references to the matching beans are returned to the client. If the identified entity is not found in persistent storage, a `javax.ejb.FinderException` is thrown. All `findXXX()` methods on the bean's home interface must declare that they can throw `FinderException` and, if it's a remote home interface, `RemoteException` (or one of the parents of `RemoteException`).

Each `findXXX()` method on the home interface must return either an instance of the bean's remote interface or a collection of these objects. If a finder method can potentially return multiple beans, it has to have a return type that is either an `Enumeration` or a `Collection`. In our profile bean example, the `findByPrimaryKey()` method can return only a single profile as a result, so it should be declared as returning an instance of the profile bean type. The `findByEntryValue()` method, on the other hand, can match multiple profiles, so it returns a `Collection`.

Every entity bean is required to have a `findByPrimaryKey()` method defined on each of its home interfaces. This finder method takes a single argument (an instance of the primary key for the entity bean) and returns the entity corresponding to that primary key. By definition, a primary key uniquely identifies a single entity, so this finder method should return a single bean instance. We discuss primary keys shortly in "Entity Bean Implementations."

6.7.2. Select Methods

In addition to finder methods, entity beans that make use of container-managed persistence (discussed later) can have *select* methods as well. Select methods are utility methods that are accessible only from the bean implementation class itself; they aren't visible to the client. One of the principles behind CMP is to encapsulate the persistence details and not expose them in the EJB implementation. However, it is sometimes necessary to perform lookups against the underlying persistent store from business methods in your EJB implementation. Select methods provide this lookup functionality without forcing you to put database details into your bean code. You can define any number of `ejbSelectXXX()` methods as abstract methods on your bean implementation class. The EJB container is responsible for providing implementations for these methods based on the persistence logic you provide for these methods in your deployment descriptors.

6.7.3. Entity Bean Implementations

Entity bean implementations must include all the basic container callbacks mentioned earlier: `ejbActivate()`, `ejbPassivate()`, `ejbRemove()`, and an `ejbCreate()` method for each `create()` method provided in the EJB's home interface(s). Entity beans must also provide several persistence-related callbacks as well as some additional callbacks related to the lifecycle model applied to entity beans. Before we discuss these, we should understand how primary keys play a role with entity beans.

6.7.3.1. Primary keys

Every type of entity bean must have an associated primary key class. The primary key serves as a unique identifier for the persistent entities represented by the entity bean. The primary key for a person's records in a database, for example, might be a first and last name, a Social Security number (for U.S. citizens), or some other identification number. The primary key used for entity beans is public, in that clients can see the primary key for an entity bean, and the primary key is used directly by the bean implementation to load or update its state from persistent storage.

A primary key can be either a custom class that we write as part of the EJB implementation, or it can be a basic Java object type, like a `String`, `Float`, or `Integer`. The name of the user uniquely identifies our `Profile` bean, for example. A custom primary key class for an entity bean implementation of our `Profile` might look something like the following:

```
public class ProfilePK implements java.io.Serializable {
    public String mName;
    public ProfilePK( ) {
        mName = null;
    }
    public ProfilePK(String name) {
        mName = name;
    }
    public boolean equals(Object other) {
        if (other instanceof ProfilePK &&
            this.mName.equals(((ProfilePK)other).mName)) {
            return true;
        }
    }
}
```

```

        else {
            return false;
        }
    }
    public int hashCode( ) {
        return mName.hashCode( );
    }
}

```

However, since the unique identifier for a `Profile` is simply a single `String`, we can just use `java.lang.String` as the primary key for our entity `Profile` bean and avoid having to write the primary key class in the first place.

A primary key class needs to satisfy the requirements of being a `Value` type in RMI-IIOP, because it may need to be transmitted to remote clients. In practical terms, this means that the primary key class has to be `Serializable` and must not implement the `java.rmi.Remote` interface. A primary key class also has to have a valid implementation of the `equals()` and `hashCode()` methods so that the container can manage primary keys internally using collections, for example. If your entity bean uses container-managed persistence, the primary key class must obey the following additional rules:

- It must be a public class.
- It must have a default constructor (one with no arguments) that is public.
- All its data members must be public.
- All of the names of the data members on the class must match the names of container-managed data members on the entity bean.

We've satisfied all of these requirements with our `ProfilePK` primary key class (note that the only data member, `mName`, has to have a matching `mName` data member on the entity implementation of the `Profile` bean).

6.7.3.2. Entity bean container callbacks

Instead of the single `setSessionContext()` callback method required on session EJBs, an entity EJB implementation is required to have the following two context-related callbacks:

```
public void setEntityContext(EntityContext ctx)
```

The container calls this method after a new instance of the bean has been constructed, but before any of its `ejbCreate()` methods are called. The bean is responsible for storing the context object. This method takes the place of the corresponding `setSessionContext()` method on session beans.

```
public void unsetEntityContext(EntityContext ctx)
```

The container calls this method before the entity bean is destroyed to disassociate the bean from its identity, represented by its primary key.

When an EJB container is managing an entity EJB, it requires several additional container callbacks related to persistence operations:

```
public primaryKeyType ejbFindByPrimaryKey( primaryKeyType ) throws FinderException
```

This is the only required finder method on an entity bean. Both the argument and the return type must be the bean's primary key type. The container is responsible for converting the returned primary key into a valid EJB reference for the client.

```
public void ejbPostCreate( )
```

If needed, an entity bean can optionally supply an `ejbPostCreate()` method for each `ejbCreate()` method it provides, taking the same arguments. The container calls the `ejbPostCreate()` method after the bean's `ejbCreate()` method has been called and after the container has initialized the transaction context for the bean.

```
public void ejbLoad( )
```

This method is called by the container to cause the bean instance to load its state from persistent storage. The container can call this bean method anytime after the bean has been created to do an initial load from persistent storage or to refresh the bean's state from the database (e.g., after a business method on the bean has completed or after the container has detected an update to the same persistent entity by another bean instance).

```
public void ejbStore( )
```

This method is called by the container to cause the bean to write its current runtime state to persistent storage. This method can be called anytime after a bean is created.

In addition to these entity-specific methods on bean implementations, the semantics of some of the other standard methods are slightly different for entity beans. Each `ejbCreate()` method, for example, is actually a request to create a new entity in persistent storage. The implementations of the create methods should not only assign any state data passed in as arguments but also create a record in persistent storage for the new entity bean described by the method arguments. In addition to this semantic difference, the signatures of `ejbCreate()` methods on entity beans can be different from session beans. For an entity bean that manages its own persistence (a bean-managed entity bean), the `ejbCreate()` methods return the primary key type for the bean. For a container-managed entity bean, the `ejbCreate()` methods return `void`, the same as for session beans.

An entity bean can be passivated by its container, but the meaning of being passivated is slightly different as well. A container passivates an entity bean (calling its `ejbPassivate()` method in the process) when it wants to disassociate the bean from the persistent data entity it has been representing. After being passivated, the bean may be put into the container's "wait" pool, to be associated with another client-requested entity at a later time, or it may be removed from the server altogether.

A client calling the `remove()` method on an entity bean reference or on the home interface for a bean is also interpreted differently: it's a request to remove the entity from persistent storage entirely. The entity bean implementation should therefore remove its state from the persistent storage in its `ejbRemove()` implementation.

6.7.4. Deployment Options for Entity Beans

When deploying entity EJBs, include an `<entity>` element in the `<enterprise-beans>` element within the deployment descriptor. A sample `<entity>` element for our `Profile` EJB is shown here:

```
...
<enterprise-beans>
  ...
  <entity>
    <display-name>BMP Profile Bean</display-name>
    <ejb-name>BMPProfileBean</ejb-name>
    <home>com.oreilly.jent.ejb.beanManaged.ProfileHome</home>
    <remote>com.oreilly.jent.ejb.Profile</remote>
    <ejb-class>com.oreilly.jent.ejb.beanManaged.ProfileBean</ejb-class>
    <persistence-type>Bean</persistence-type>
    <prim-key-class>java.lang.String</prim-key-class>
    <reentrant>false</reentrant>
    <resource-ref>
      <res-ref-name>jdbc/ProfileDB</res-ref-name>
      <res-type>javax.sql.DataSource</res-type>
      <res-auth>Container</res-auth>
    </resource-ref>
  </entity>
  ...
</enterprise-beans>
...
```

The `<entity>` element contains much of the same basic required information as a session element (`<ejb-name>`, `<home>`, `<remote>`, `<local-home>`, `<local>`, `<ejb-class>`), but it also contains several elements specific to entity beans. Three of these are required: `<persistence-type>`, `<prim-key-class>`, and `<reentrant>`.

The `<persistence-type>` element specifies how the persistence for your entity bean is to be managed. The allowed values for this element are `Bean` or `Container`, indicating the two fundamental ways that entity persistence can be handled. Our `ProfileBean` is written to use bean-managed persistence, so we set the persistence type to `Bean`.

The `<prim-key-class>` element specifies which class is being used as the primary key for this bean. We chose to use `Strings` rather than our custom `ProfilePK` class, so we specify `java.lang.String` as our primary key type.

The `<reentrant>` element indicates whether an EJB can be accessed by multiple clients concurrently. If an EJB is marked reentrant, it's eligible for concurrent access; otherwise, it's run in a single-

threaded manner only one client request is processed at a time by any given EJB instance within the container. In our example, we haven't written our `ProfileBean` to be threadsafe (a quick look at the `setEntry()` implementation in the source code will confirm that), so we've set the `<reentrant>` element to `false`. The `<reentrant>` property has other implications related to *loopback* calls (method implementations on the bean that result in calls back to the same bean instance). Refer to O'Reilly's *Enterprise JavaBeans* or the EJB specification for full details on loopback calls.

6.7.5. The Entity Context

The EJB container provides context information to an entity bean in the form of an `EntityContext` object. The container sets this object using the bean's `setEntityContext()` method and removes it when the bean is being removed by calling the bean's `unsetEntityContext()` method. Like `SessionContext`, `EntityContext` provides the bean with access to its corresponding remotely exported object through the `getEJBObject()` method. The `EntityContext` also gives an entity bean access to its primary key through `getPrimaryKey()`. The declared return type of this method is `Object`, but the object returned is of the bean's primary key type. Note that the data accessed through the `EntityContext` might be changed by the EJB container during the bean's lifetime, as explained in the next section. For this reason, you shouldn't store the EJB remote object reference or primary key in data variables in the bean object since they might not be valid for the entire lifetime of the bean. Our entity `ProfileBean`, for example, stores the `EntityContext` reference in an instance variable, where it can access the context data as needed during its lifetime.

6.7.6. Lifecycle of an Entity Bean

Before the first client asks for an entity bean by calling a `create()` or `findXXX()` method on its home interface, an EJB container might decide to create a pool of entity beans to handle client requests for beans. This potentially reduces the amount of time it takes for a client to receive an entity bean remote reference after it makes a request for an entity bean. To add a bean to its pool, the container creates an instance of your bean implementation class and sets its context using the `setEntityContext()` method. At this point, the entity bean hasn't been associated with a particular data entity, so it doesn't have an identity and therefore isn't eligible for handling client business method calls on bean references.

When a client calls a `create()` method on the bean's home interface, the container picks a bean out of this wait pool and calls the corresponding `ejbCreate()` method on the bean. If the `ejbCreate()` method is successful, it returns one or more primary key objects to the container. For each primary key, the container picks an entity bean out of its pool to be assigned to the entity represented by the key. Next, the container assigns the bean's identity by setting the properties in its `EntityContext` object (e.g., its primary key and remote object values). If the bean has an `ejbPostCreate()` method, that gets called after the bean's entity identity has been set. The `ejbCreate()` method should create the entity in persistent storage, if the bean is managing its own persistence.

Alternatively, the client might call a `findXXX()` method on the home interface. The container picks one of the pooled entity beans and calls the corresponding `ejbFindXXX()` method on it. If the finder method finds one or more matching entities in persistent storage, the container uses entity beans in the wait pool to represent these entities. It picks entity beans out of the pool and calls their `ejbActivate()` methods. Before calling `ejbActivate()`, the container sets the bean's context by assigning the corresponding primary key and remote object reference in its context.

After an entity bean has been activated (either by being created through one of its `ejbCreate()` methods or by being found and having its `ejbActivate()` method called), it is associated with a specific entity in persistent storage and with a specific client stub. At any point after this, the container can call the bean's `ejbLoad()` or `ejbStore()` method to force the bean to read or write its state from or to persistent storage. The bean's business methods can also be invoked by clients when it is in this state.

At some point, the container may decide to put the bean back into its internal pool. This might happen after all remote references to the bean have been released or after a certain period of inactivity with the bean. The container might also do this as a reaction to client-loading issues (e.g., time-sharing pooled beans between client requests). When the container wants to remove the association between the entity bean instance and the client stub but doesn't want the object's state removed from persistent store, it calls the bean's `ejbPassivate()` method. The bean can release any resources it allocated while in the active state, but it doesn't have to update persistent storage for the entity it represents, as this was done the last time the container invoked the bean's `ejbStore()` method.

The bean can also lose its association with an entity when the client decides to remove the entity. The client does this either by calling a `remove()` method on the bean's home interface or calling the `remove()` method directly on an EJB object. When one of these things happens, the container calls the bean's `ejbRemove()` method, and the bean deletes the data in persistent storage pertaining to the entity it represents. After the `ejbRemove()` method completes, the container puts the bean back into its internal pool.

6.7.7. Bean-Managed Persistence

Bean-managed persistence (BMP) is one option for dealing with the persistence of the state of entity EJBs. In BMP, all of the persistence management is done directly in the EJB implementation class, using custom JDBC calls or whatever API is appropriate for the persistent storage being used. The persistence management of an entity bean is handled in the `ejbCreate()`, `ejbRemove()`, `ejbLoad()`, and `ejbStore()` methods on the bean implementation. The persistent store is also accessed directly in the `ejbFindXXX()` methods on the implementation of BMP beans.

Let's look at a BMP entity implementation of our `Profile` EJB. We've already seen some details on the home and client interfaces for this entity bean in earlier sections. The purpose of the bean is the same as our stateful session version: it represents a profile for a named application user, maintaining a list of name/value pairs for various attributes and options. The difference is that this `Profile` EJB represents a profile entity that exists as a set of persistent data (stored in a database, in this case). The full source code for the implementation class can be found in the file `com/oreilly/jent/ejb/beanManaged/ProfileBean.java` in the code bundle for the book; we'll cover some highlights in this tutorial.

The structure of the entity `Profile` implementation is similar to the stateful session bean version. All of the EJB-required methods, including `ejbActivate()`, `ejbPassivate()`, `ejbCreate()`, and `ejbRemove()`, are present. The `ejbActivate()` and `ejbPassivate()` methods are called as the container moves the bean in and out of the entity bean "wait" pool, so it's a chance to free up any resources, such as file handles, before the bean is passivated and reclaim them if needed when it is activated again.

The `ejbCreate()` methods on the `ProfileBean` create a new profile entity in the database, so our `ejbCreate()` method (associated with the `create()` method in our home interface) is a bit more complicated than in our session beans because now it needs to make all the JDBC calls to create a profile entity in the underlying database:

```
public String ejbCreate(String name)
    throws DuplicateProfileException, CreateException {
    Connection conn = null;
    Statement s = null;
    try {
        conn = getConnection();
        s = conn.createStatement();
        s.executeUpdate("insert into profile (name) values ('"
            + name + "')");
    }
    catch (SQLException se) {
        System.out.println("Error creating profile, assuming duplicate.");
        try {
            StringWriter strw = new StringWriter();
            PrintWriter prntw = new PrintWriter(strw);
            se.printStackTrace(prntw);
            throw new DuplicateProfileException(
                "SQL error creating profile for " +
                name + ": " + se.toString() +
                "\n" + strw.toString());
        }
        catch (Exception e) {}
    }
    catch (NamingException ne) {
        System.out.println("Error accessing DataSource");
        throw new CreateException("Error accessing DataSource.");
    }
    finally {
        try { s.close(); } catch (Exception e) {}
        try { conn.close(); } catch (Exception e) {}
    }
    System.out.println("ProfileBean created for " + name + ".");
    return name;
}
```

Notice that the `ejbCreate()` method is using an internal utility method on our implementation, `getConnection()`, to get a JDBC connection to the database. This method looks for a `DataSource` stored in the JNDI context of the J2EE server and uses it to create a connection to the database. For details on configuring and using `DataSources`, see [Chapter 8](#).

Another difference in this `ejbCreate()` method is that it returns the EJB's primary key type (a `String`, in this case). The returned primary key value must represent the identity of the actual entity that was created by the `ejbCreate()` method. In our case, we just return the name of the profile owner, since that's our unique identifier for profiles. The EJB container is responsible for intercepting the primary key object returned by the `ejbCreate()` method, converting it to a reference to a live

`Profile` EJB, and returning a `Profile` stub to the client that originally called the `create()` method on the `ProfileHome` interface.

An entity bean can also optionally provide an `ejbPostCreate()` method for each `create()` method on its home interface(s). The container calls this method after the `ejbCreate()` method returns and after the container initializes the bean's transaction context. The `ejbPostCreate()` method has to match the arguments to its corresponding `create()` and `ejbCreate()` methods. In our case, nothing needs to be done in the `ejbPostCreate()` method, so we simply log that it's been called and return.

The `ejbRemove()` method on our `ProfileBean` deletes all the records for this profile entity from the database:

```
public void ejbRemove() {
    // Get this profile's name
    String key = (String)mContext.getPrimaryKey();
    Connection conn = null;
    Statement s = null;
    try {
        conn = getConnection();
        s = conn.createStatement();
        // Clear out any profile entries
        s.executeUpdate("delete from profile_entry where name = '"
            + key + "'");
        // Clear out the profile itself
        s.executeUpdate("delete from profile where name = '" + key + "'");

        System.out.println("ProfileBean removed.");
    }
    catch (SQLException se) {
        System.out.println("Error removing profile for " + key);
        se.printStackTrace();
    }
    catch (NamingException ne) {
        System.out.println("Error accessing DataSource");
        ne.printStackTrace();
    }
    finally {
        try { s.close(); } catch (Exception e) {}
        try { conn.close(); } catch (Exception e) {}
    }
}
```

In terms of the actual business methods of the `Profile` EJB, a `Properties` object holds the profile entries for the user, and the `getEntry()` and `setEntry()` remote method implementations access this `Properties` object for the client. The name is found in the primary key object, and the primary key is stored for us in the `EntityContext` that the container gives us through the `setEntityContext()` method. The `getName()` method on the `ProfileBean` implementation shows how we retrieve the username for the profile using the `getPrimaryKey()` method on the `EntityContext`:

```
public String getName( ) {
    return (String)mContext.getPrimaryKey( );
}
```

We've removed the `setName()` method from the entity version of our `Profile` EJB since we don't want to allow the client to change the name of an existing, active entity bean. The `Profile` client interface for this bean, not shown here, is similar to the `Profile` interface in [Example 6-6](#), with the `setName()` method removed. Since the `Profile` is now a persistent entity bean and the name is the primary key, or identifying attribute, of the bean, the name of the bean should be set only when the bean is created. While the entity bean is active, it is associated with a profile entity for a specific user and the client should read only the name associated with the profile.

Two additional persistence-related container callbacks are needed on entity bean implementations, `ejbLoad()` and `ejbStore()`. Since this bean is using BMP, our `ejbLoad()` and `ejbStore()` methods do all the JDBC calls directly, just like the other persistence callbacks in this version of the `Profile` EJB:

```
public void ejbLoad() {
    try {
        String key = (String)mContext.getPrimaryKey();
        loadFromDB(key);
    }
    catch (Exception e) {
        System.out.println("Failed to load ProfileBean: ");
        e.printStackTrace();
        throw new EJBException("ejbLoad failed: ", e);
    }
    System.out.println("ProfileBean load finished.");
}

public void ejbStore() {
    String key = (String)mContext.getPrimaryKey();
    Connection conn = null;
    Statement s = null;
    try {
        conn = getConnection();
        s = conn.createStatement();
        // Clear out old profile entries; replace with current ones
        s.executeUpdate(
            "delete from profile_entry where name = '" + key + "'");
        Enumeration pKeys = mEntries.propertyNames();
        while (pKeys.hasMoreElements()) {
            String pKey = (String)pKeys.nextElement();
            String pValue = mEntries.getProperty(pKey);
            s.executeUpdate(
                "insert into profile_entry (name, key, value) values "
                + "(" + key + "', '" + pKey + "', '" + pValue + "')");
        }
    }
    catch (Exception e) {
```

```

        System.out.println("Failed to store ProfileBean: ");
        e.printStackTrace();
        throw new EJBException("ejbStore failed: ", e);
    }
    finally {
        try { s.close(); } catch (Exception e) {}
        try { conn.close(); } catch (Exception e) {}
    }
    System.out.println("ProfileBean store finished.");
}

```

An `ejbFindXXX()` method in our entity `ProfileBean` corresponds to each `findXXX()` method in `ProfileHome`. The `ejbFindByPrimaryKey()` method simply takes the primary key passed in as an argument and attempts to load the data for the entity from the database.

```

public String ejbFindByPrimaryKey(String key) throws FinderException {
    loadFromDB(key);
    return key;
}

```

If successful, it returns the primary key to the container, where it is converted to a remote `Profile` object to be returned to the client. It's not necessary to actually load all the profile data here in the finder method; we need only to verify that the specified entity exists in the database and either return the primary key to signal success or throw an exception. Since we already have the internal `loadFromDB()` method used in `ejbLoad()`, it is a simple matter to reuse it in this finder method. If the performance hit for loading the profile data twice is too great, we would have to rewrite the finder method to simply check the `PROFILE` table for a record matching the name in the primary key.

The `ejbFindByEntryValue()` method takes key and value `String` arguments and attempts to find any and all profile entities with a matching key/value pair in the `PROFILE_ENTRY` table. Each name that has such a record is converted to a primary key object and returned to the container in a `Collection`.

```

public Collection ejbFindByEntryValue(String key, String value)
    throws FinderException {
    LinkedList userList = new LinkedList();
    // Get a new connection from the EJB server
    Connection conn = null;
    Statement s = null;
    try {
        conn = getConnection();
        s = conn.createStatement();
        // Issue a query for matching profile entries, grabbing
        // just the name
        s.executeQuery("select distinct(name) from profile_entry where " +
            " key = '" + key + "' and value = '" +
            value + "'");
        // Convert the results in primary keys and return an enumeration
        ResultSet results = s.getResultSet();
    }
}

```

```

        while (results.next()) {
            String name = results.getString(1);
            userList.add(name);
        }
    }
    catch (SQLException se) {
        // Failed to do database lookup
        throw new FinderException();
    }
    catch (NamingException ne) {
        // Failed to access DataSource
        throw new FinderException();
    }
    finally {
        try { s.close(); } catch (Exception e) {}
        try { conn.close(); } catch (Exception e) {}
    }
    return userList;
}

```

The container converts each primary key object in the returned `Collection` into a remote `Profile` object and returns the set to the client. If we encounter a database problem along the way, we throw a `FinderException`.

6.7.8. Container-Managed Persistence

In our first entity-based `Profile`, the persistent state of the profile entity is managed by the bean itself. Custom JDBC code in the `ProfileBean` implementation loads, stores, and removes the entity's database entries. The EJB container calls the appropriate callback methods on your entity bean, but your bean implementation is responsible for connecting to the database and making all of the necessary queries and updates to reflect the lifecycle of the data entity.

Container-managed persistence (CMP) allows you to define properties on your entity bean implementation that represent the state of the entity and tell the EJB container how to map these properties to persistent storage. With CMP, the container is responsible for loading, updating, and removing the entity data from persistent storage, based on the mapping you provide. The container invokes the callbacks on your bean implementation to notify it when it has performed these tasks for you, allowing your bean to do any follow-up work that may be required. The container also implements all the finder methods required by the bean's home interface.

If you want to take advantage of CMP, you have to indicate this to the EJB container when you deploy the bean, you need to provide a bean implementation that assumes that the container is performing all persistence management, and you need to supply the container with a data mapping at deployment time.

Implementing CMP-based entity beans involves describing the persistence properties of your bean rather than implementing them directly:

1. In your bean implementation class, declare abstract accessor methods for persistent properties that represent the state of the bean.
2. Declare finder and select methods in the home interface(s) for the EJB.
3. Map the persistent state properties to persistent storage in the EJB deployment descriptor.
4. Provide the container with the logic needed to implement the finder and select methods.

The EJB container is responsible for generating the relevant code to load, store, and update the state of the EJB to and from persistent storage and to implement the finder and select methods defined on your home interface(s). The various persistence-related container callback methods on your implementation class (`ejbLoad()`, `ejbStore()`, `ejbCreate()`, and `ejbRemove()`) are still used as callbacks by the EJB container, but in the case of CMP, they're invoked by the container just before and after it performs the corresponding persistence operation.

The complete source code for a CMP implementation of our `Profile` bean can be found in the source bundle for the book, in the file `com/oreilly/jent/ejb/containerManaged/ProfileBean.java`. We discuss the key features here.

In comparison to our earlier BMP-based `ProfileBean`, this bean is somewhat simpler since the `ejbRemove()`, `ejbLoad()`, and `ejbStore()` methods don't need to perform database calls.

The container handles the loading and storing of the bean's data and the removal of any entities from the database, so we don't need to do anything about these operations in our bean implementation. In our `Profile` bean case, all we do in our `ejbLoad()` and `ejbStore()` methods is map the persistence data managed by the container into our own internal data structures. This is not typical for CMP beans; we'll discuss the reasons that we need to do this in the next section when we discuss abstract persistence mappings. Here is the code for `ejbLoad()` and `ejbStore()`:

```
public void ejbLoad( ) {
    try {
        transferToProps( );
    }
    catch (IOException e) {
        System.out.println("Failed to load ProfileBean: ");
        e.printStackTrace( );
        throw new EJBException("ejbLoad failed: ", e);
    }
}

public void ejbStore( ) {
    try {
        transferToBytes( );
    }
    catch (IOException e) {
        System.out.println("Failed to store ProfileBean: ");
        e.printStackTrace( );
        throw new EJBException("ejbStore failed: ", e);
    }
}
```

The important thing to note here is that these callbacks aren't doing any database operation the container is doing the database calls, and we're just getting ready for this (`ejbStore()`) or cleaning up afterward (`ejbLoad()`).

The same is true of the `ejbRemove()` method it's called just before the container is about to remove this entity's data from the database. In our case, there's nothing to do, so we provide an empty method:

```
public void ejbRemove( ) {
    System.out.println("ProfileBean removed.");
}
```

6.7.8.1. Mapping container-managed fields: Abstract persistence schema

In the CMP model, the persistent state of your entity EJBs is specified in terms of an *abstract persistence schema*. The abstract schema consists of a set of named EJBs (the "tables" of the schema) and the persistent properties declared on your entity EJBs (the "columns" of the schema). You describe the components of this abstract schema in your EJB deployment descriptor and provide the container with a mapping from this abstract schema to a real persistence store (such as a relational database schema or an object database). The container takes all of this information and uses it to generate persistence management code for your EJB.

You indicate abstract persistence fields on your bean by providing abstract accessors (JavaBeans-style set and get methods) on the EJB implementation class. The EJB container provides concrete implementations for these abstract accessor methods in its generated classes. In our `ProfileBean` implementation class, we have two sets of accessors for the two abstract persistent fields needed by our bean: `get/setName()` for the `name` property and `get/setEntriesBytes()` for the `entriesBytes` property:

```
abstract public String getName( );
abstract public void setName(String name);

abstract public byte[] getEntriesBytes( );
abstract public void setEntriesBytes(byte[] entries);
```

The use of a byte array for our profile entries may seem like an odd implementation decision. It has to do with the limitations of automatic CMP persistence. We'll return to this later, but for now just take it on faith that our profile entries need to be maintained as a byte array.

Note that, since we need to define persistent field accessors as abstract in our implementation class, the class itself must be declared as abstract as well:

```
abstract public class ProfileBean implements EntityBean {
```

Next, we need to describe our abstract persistence schema in the EJB's deployment descriptor. This essentially amounts to specifying an abstract schema name for each EJB we are creating (thus defining the "tables") and listing the persistent properties of each EJB (defining the "columns"). A partial listing of the *ejb-jar.xml* file for our CMP `Profile` bean is shown in [Example 6-8](#).

Example 6-8. Section of *ejb-jar.xml* file for CMP profile EJB

```
<?xml version="1.0" encoding="UTF-8"?>
<ejb-jar 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/ejb-jar_2_1.xsd"
  version="2.1">
  <enterprise-beans>
    <!-- A Profile EJB using container-managed persistence -->
    <entity>
      <ejb-name>CMPProfileBean</ejb-name>
      <home>com.oreilly.jent.ejb.containerManaged.ProfileHome</home>
      <remote>com.oreilly.jent.ejb.containerManaged.Profile</remote>
      <local-home>
        com.oreilly.jent.ejb.containerManaged.ProfileLocalHome
      </local-home>
      <local>
        com.oreilly.jent.ejb.containerManaged.ProfileLocal
      </local>
      <ejb-class>
        com.oreilly.jent.ejb.containerManaged.ProfileBean
      </ejb-class>
      <persistence-type>Container</persistence-type>
      <prim-key-class>java.lang.String</prim-key-class>
      <reentrant>>false</reentrant>
      <cmp-version>2.x</cmp-version>
      <abstract-schema-name>ProfileBean</abstract-schema-name>

      <!-- Indicate which fields need to be managed persistently -->
      <cmp-field>
        <description>Name of the profile owner</description>
        <field-name>name</field-name>
      </cmp-field>
      <cmp-field>
        <description>Binary data containing profile entries.</description>
        <field-name>entriesBytes</field-name>
      </cmp-field>

      <!-- Since our primary key is simple (one-column), we need to specify
           to the container which field maps to the primary key -->
      <primkey-field>name</primkey-field>
    </entity>
    ...
  </enterprise-beans>
</ejb-jar>
```

```

    </enterprise-beans>
    ...
</ejb-jar>

```

Before listing our persistent fields, we provide an `<abstract-schema-name>` element, which labels our EJB with an abstract name within the abstract persistence schema we're defining. Think of this abstract name as a sort of virtual table name; it's a way to refer to it in the EJB query language statements that define finder and select methods.

Following this, we list the `<cmp-field>` elements that describe our EJB's persistent properties. Our `Profile` bean has two persistent properties: `name` and `entriesBytes`, represented by the corresponding accessor methods in the bean implementation. So we have two `<cmp-field>` elements that serve to "tag" these bean properties as ones that need to be treated as persistent state.

In addition, the container needs to know how the bean's primary key is mapped. The primary key has to be mapped to one or more of the `<cmp-field>` elements listed in the deployment descriptor. If it maps to a single `<cmp-field>`, the name of the field has to be provided in the deployment descriptor as a `<primkey-field>` element. If the primary key is a complex key that maps to multiple `<cmp-field>` elements on the EJB, no `<primkey-field>` is provided and instead the data members on the primary key class must have variable names that match the corresponding persistent fields in the EJB implementation class. The container automatically makes the mapping from primary key fields to persistent bean fields. In our case, the primary key is simply the bean's `name` property, so we've included a `<primkey-field>` element in our deployment descriptor to that effect.

All we've done so far in the `ejb-jar.xml` file is provide the basic deployment information for our EJB and list the fields in the bean implementation that need to be persisted. But the container doesn't know how to persist these fields yet. The EJB specification does not define a standard way to provide this mapping to physical data storage. One reason for not including this in the specification is that it would require assumptions about the type of persistence being used by the container. EJB containers can (ostensibly) use any kind of persistent storage to maintain the state of entity beans: relational databases, object databases, XML databases, even flat files. Undoubtedly, virtually all of the major J2EE vendors provide relational persistence support in their CMP implementations, but the EJB specification team did not want to build this assumption into the specification, so they left the mapping of abstract persistent fields to physical storage up to the container vendors.

If we're using BEA's WebLogic application server, for example, we would specify the CMP deployment parameters using some additional deployment files that are specific to WebLogic. All EJBs (session, entity, or message-driven) deployed in WebLogic need to have an additional deployment file named `weblogic-ejb-jar.xml`, where things such as the JNDI binding for the EJB home interface can be specified. For container-managed entity beans using relational persistence, an additional deployment file, `weblogic-cmp-rdbms-jar.xml`, is used to specify persistence management information for the bean.

Each EJB-enabled server has its own scheme, but the required information is roughly the same. Regardless of how you specify the CMP field mapping, the EJB container reads the information and generates the necessary code to load and store the bean's persistent data to and from the underlying persistent storage system. It's your responsibility to ensure that the underlying physical persistence schema (whether it's relational, object-oriented, or hierarchical) is compatible with the deployment information provided to the container.

6.7.8.2. Handling complex data structures

Now we can finally explain why we implemented our CMP `Profile` EJB using a byte array for the entry data.

Each EJB container is limited to some degree in the way that data on your bean implementation can be mapped into persistent data fields. There is no standard format defined for the data mapping a particular container supports, so it's possible a particular EJB provider won't support whatever complicated mapping you require for your bean. For the most part, however, you can expect EJB providers to limit the format to a single persistent data field being mapped to a single data member on your bean implementation. If your bean's data structures are too complicated for you to provide an explicit mapping to persistent data fields, you have to decide how to deal with this.

In our entity `Profile` example, the profile entries need to be a list of name/value pairs that represent the parameters in the profile. We don't know at deployment time how many entries there will be, so we can't enumerate a mapping to particular database fields. Ideally, we really want each entry in the profile to be stored in a separate entry table in our database, along with the name of the owner of the entry, which is exactly how we implemented our bean-managed implementation. But since we're using CMP, we can't count on the EJB container supporting this type of persistence mapping (and, in fact, chances are that it won't).

One option is to give up on CMP and manage it yourself in the bean implementation using BMP. Another is to make each entry in the profile its own EJB and store the entries as a list of `Entry` beans on the profile bean. In this case, you could use EJB relationships, described in a later section, to tell the EJB container how these two entity EJBs are related to each other (e.g., when a given profile EJB is loaded, all of its corresponding entry EJBs should also be found and loaded). This makes the implementation pretty complex, though, since we now have two entity EJBs required to manage the data associated with profiles, and the amount of deployment information we need to provide gets much more complicated. This could also turn out to be expensive in terms of interactions with the container and memory usage. Each entry in the profile would need to be managed separately by the container, with all of the relevant lifecycle notifications and the like. CMP is supposed to make things simpler and faster, not the reverse, so we would probably not take this path in a real-world situation.

Another option, and the one we used in our CMP `Profile` EJB, is to convert the data structure into something that can be persisted into a single database table. Since the EJB specification gives us well-defined container callbacks to mark persistence operations, we can use these to do the conversion of our internal data structures to and from simplified persistent properties. In our `Profile` EJB, we defined the profile entries as a persistent byte array property. In our deployment descriptors for the bean, we mapped this persistent property to the binary `PROFILE_BIN.ENTRIES_BYTES` column in the database.

This gives us a persistent binary field for our EJB, but we still have to serialize the `Properties` object into the byte array when it needs to be stored persistently and deserialize the byte array into the `Properties` object when the persistent data is loaded. As we mentioned before, when using CMP, the `ejbLoad()` and `ejbStore()` methods are used by the container as callbacks. The `ejbStore()` method is called just before the container performs an automatic store of the bean's persistent fields to the database, and `ejbLoad()` is called just after the container has loaded the persistent fields from the database. So we can use the `ejbStore()` method on our bean to copy the `Properties` object on our `ProfileBean` to the `mEntriesBytes` persistent data member. If you look at the `ejbStore()` method, you'll see that it simply calls our `transferToBytes()` method:

```

// Transfer the list of entries from our Properties member to the byte
// array
private void transferToBytes( ) throws IOException {
    // Serialize the Properties into a byte array using an ObjectOutputStream
    if (mEntries != null && !mEntries.isEmpty( )) {
        ByteArrayOutputStream byteOut = new ByteArrayOutputStream( );
        ObjectOutputStream objOut = new ObjectOutputStream(byteOut);
        objOut.writeObject(mEntries);
        setEntriesBytes(byteOut.toByteArray( ));
    }
    else {
        setEntriesBytes(null);
    }
}

```

This method uses I/O streams to serialize the `mEntries Properties` object into the `mEntriesBytes` byte array. After the container calls our `ejbStore()` method, it can write the `mEnTRiesBytes` data member on our bean to a raw data field in the database (e.g., a `LONG BINARY` field in a SQL database). On the reading end, the container will load the bytes stored in the database column into the `mEntriesBytes` byte array and call our `ejbLoad()` method. We can use the `ejbLoad()` method to convert the bytes loaded by the container into a `Properties` object; our `ejbLoad()` method simply calls `TRansferToProps()`:

```

// Convert the serialized byte array into our Properties entry list
private void transferToProps( ) throws IOException {
    // Take the raw byte array and deserialize it
    // back into a Properties object using an ObjectInputStream
    try {
        if (getEntriesBytes( ) != null) {
            ByteArrayInputStream byteIn =
                new ByteArrayInputStream(getEntriesBytes( ));
            ObjectInputStream objIn = new ObjectInputStream(byteIn);
            mEntries = (Properties)objIn.readObject( );
        }
        // If no entries in database, set properties
        // to a new, empty collection
        else {
            mEntries = new Properties( );
        }
    }
    catch (ClassNotFoundException cnfe) {
        System.out.println(
            "Properties class not found during de-serialization");
    }
}

```

This workaround is a good example of how the `ejbLoad()` and `ejbStore()` methods can be useful for CMP entity beans. It may also seem like an ideal solution for our `Profile` example since it allows us to deploy our entity `Profile` with container-managed persistence and still have a variable-sized lis

of entries on a profile. But this solution comes with a price tag: it makes our database records unusable for other, non-Java applications. The data stored in the `PROFILE_BIN.ENTRIES_ BYTES` column is a serialized Java object, so there's no way, for example, to check on a user's profile entries using a simple SQL query or to use a standard RDBMS reporting tool to show categories of users sorted by profile entries. As we'll see in the next section, the limitations of CMP also make it more difficult to implement certain complex finder methods.

6.7.8.3. Finder and select methods

When using container-managed persistence, the EJB container also generates all of the `ejbFindXXX()` methods required for the finder methods on the home interface(s). It can automatically generate an `ejbFindByPrimaryKey()` method, based on the data mapping and primary key information you provide at deployment time (although the EJB specifications are unclear as to whether the container is required to do this). But for any other `ejbFindXXX()` methods, you need to provide the container with the logic for the methods. The container can't infer the semantics of what you're trying to do based solely on method arguments. The same is true of select methods.

As an example, suppose we want to add a finder method for our `Profile` EJB, called `findEmptyProfiles()` that finds all the profiles in the persistent store that have no entries in them. We would add the finder method to the home interface(s) for the `Profile` bean:

```
public Collection findEmptyProfiles( )
    throws RemoteException, FinderException;
```

The finder method returns a `Collection` because it could potentially find more than one empty profile in the database. Now we need to tell the EJB container how to implement this method.

The persistence logic for both finder and select methods is provided using EJB QL, a standard query language defined as part of the EJB specification. EJB QL is similar in syntax to SQL, except that queries are defined using the abstract schema elements defined in the deployment descriptors for your EJBs.

Query logic for finder and select methods is included in the `ejb-jar.xml`/deployment descriptor, using `<query>` elements within the corresponding `<entity>` section. So to provide the query logic for the `findEmptyProfiles()` finder method on our `Profile` EJB, we would add the following stanza to our `ejb-jar.xml`/file, within the `<entity>` element for our `Profile` bean:

```
...
<enterprise-beans>
    ...
    <entity>
        ...
        <query>
            <query-method>
                <method-name>findEmptyProfiles</method-name>
                <method-params></method-params>
            </query-method>
        </ejb-ql>
```

```

                <![CDATA[SELECT OBJECT(p) FROM ProfileBean AS p
                        WHERE p.entriesBytes IS NULL]]>
            </ejb-ql>
        </query>
        ...
    </entity>
    ...
</enterprise-beans>
...

```

The full syntax for EJB QL is provided in [Appendix C](#), but it should be fairly obvious in our example what we're doing. The `findEmptyProfiles()` method takes no arguments, so we specify the `<query-method>` (using `findEmptyProfiles` as its `<method-name>`) and an empty `<method-params>` element. We want this finder to return all `Profiles` that have no entries in the database, so we specify an EJB QL statement that selects all `ProfileBeans` (using the abstract "table" name we defined earlier for our `Profile` EJB) whose `entriesBytes` persistent fields are null.

Select methods are defined using the same `<query>` element in the deployment descriptor. We simply specify the corresponding `ejbSelectXXX()` method name in the `<method-name>` element.

This persistence logic, in combination with the abstract schema information provided earlier in the deployment descriptor, is used by the container to generate the concrete implementations of the finder and select methods for our bean. The EJB QL might be "compiled" into SQL for a relational database, Object Query Language (OQL) for an object-oriented database, or some other native persistence query language.

You may have noticed in our CMP bean implementation (if you peruse the source code) that we've removed the `ejbFindByEntryValue()` finder method that we had in our BMP version. Because of the way we decided to implement the persistence of the entries of the `Profile` bean (using a single serialized byte array stored in a single binary database column), there isn't any way to implement the logic of `findByEntryValue()`. From the perspective of EJB QL, the `entriesBytes` "column" is simply an array of bytes, with no way to query for the data stored in the `Properties` object that it came from. Again, we could fix this problem by changing our persistence mapping—for example, we could create a new `Entry` EJB and have the `Profile` bean maintain a set of references to `Entry` beans. But as mentioned in the previous section, this would impose some significant resource needs on our profile service, since each entry in each profile would now be a full-blown EJB object with all of the requisite container management.

6.7.8.4. EJB relationships

In keeping with the idea of an abstract persistence schema, EJB relationships can be thought of as abstract foreign key constraints between our EJB "tables." Suppose, for example, that in addition to our `Profile` bean, we also defined a `Person` entity bean that provided access to more general information about a person (name, address, etc.). Within the `Profile` bean implementation, it might be useful to access the `Person` bean corresponding to the owner of a particular `Profile`. In the EJB 2.1 CMP model, we can do this using EJB relationships.

Since EJB relationships are a way to extend the abstract persistence schema of a set of CMP EJBs, they can be established only between EJBs that are using CMP. Session beans, message-driven

beans, and BMP entity beans can't participate in EJB relationships. EJB relationships are accessible only from within the bean implementation they aren't exposed directly to clients. EJB relationships can be one-to-one or one-to-many. In our case, we want a single `Person` to be related to a single `Profile`, so the relationship will be one-to-one. If we plan to change our `Profile` bean so that it uses a new `ENTRY` EJB to represent its entries and we want to establish a persistence relationship between the two, we would make the relationship one-to-many.

It's important to note that EJB relationships are created using the local interfaces for EJBs. If an EJB doesn't have a local interface, it can have one-way relationships to other beans, but other EJBs can't have relationships with it.

The process for establishing an EJB relationship is similar to defining persistent fields; you define the accessor method(s) on either end of the relationship, specify the relationship in the EJB deployment descriptor, and then provide the container with the information on how the relationship is mapped to the underlying persistent storage system. In our case, we want to define a one-to-one relationship between our `Profile` bean and a new `Person` bean. Let's suppose that the `Person` bean has a very simple local client interface:

```
import javax.ejb.*;

public interface PersonLocal extends EJBLocalObject {
    // Access the person's first and last name
    public String getFirstName( );
    public String getLastName( );
}
```

We won't show all of the details of the implementation of this `Person` EJB; we'll just highlight the details relevant to the EJB relationship.

The first step in creating the relationship is defining abstract accessor methods on the implementations of the beans involved. On our `ProfileBean` implementation class, we would add the following accessors:

```
// Get local person related to this profile
abstract public PersonLocal getPersonLocal( );
abstract public void setPersonLocal(PersonLocal person);
```

If we wanted the relationship to be bidirectional, we would also define accessor methods on the implementation class for our `Person` bean to read and write the `Profile` associated with the `Person`:

```
// Get/set local Profile using CMP relationships
abstract public ProfileLocal getProfileLocal( );
abstract public void setProfileLocal(ProfileLocal profile);
```

Notice that these accessors need to be defined in terms of the local interfaces of the beans involved.

Now we need to specify the relationship between these two beans in the EJB deployment descriptor.

Assuming that both beans are defined in the same *ejb-jar.xml* file, the relationship is defined like so:

```

...
<ejb-jar ...>
  <enterprise-beans>
    <!-- A Profile EJB using container-managed persistence -->
    <entity>
      <ejb-name>CMPProfileBean</ejb-name>
      ...
    </entity>
    <!-- A Person EJB using container-managed persistence -->
    <entity>
      <ejb-name>CMPPersonBean</ejb-name>
      ...
    </entity>
  </enterprise-beans>
  <!-- Establish EJB relationships between Person and Profile -->
  <relationships>
    <ejb-relation>
      <ejb-relation-name>Person-Profile</ejb-relation-name>
      <!-- Relation from person to profile -->
      <ejb-relationship-role>
        <ejb-relationship-role-name>
          Person-has-Profile
        </ejb-relationship-role-name>
        <!-- One profile per person -->
        <multiplicity>One</multiplicity>
        <relationship-role-source>
          <ejb-name>CMPPersonBean</ejb-name>
        </relationship-role-source>
        <cmr-field>
          <cmr-field-name>profileLocal</cmr-field-name>
        </cmr-field>
      </ejb-relationship-role>
      <!-- Relation from profile to person -->
      <ejb-relationship-role>
        <ejb-relationship-role-name>
          Profile-belongs-to-Person
        </ejb-relationship-role-name>
        <!-- One person per profile -->
        <multiplicity>One</multiplicity>
        <relationship-role-source>
          <ejb-name>CMPProfileBean</ejb-name>
        </relationship-role-source>
        <cmr-field>
          <cmr-field-name>personLocal</cmr-field-name>
        </cmr-field>
      </ejb-relationship-role>
    </ejb-relation>
  </relationships>
  ...

```

```
</ejb-jar>
```

We've added a `<relationships>` element to our `ejb-jar.xml` file, after the `<enterprise-beans>` section that describes the EJBs themselves. Each `<ejb-relation>` element in this section defines a relationship between two EJBs. Here we have a single `<ejb-relation>` that defines the relationship between the `Person` and `Profile` beans described earlier in the deployment descriptor. Each "side" of the relationship is described using an `<ejb-relationship-role>` element. The `<ejb-relationship-role>` element includes a name to assign to the role, the multiplicity of the role ("one" or "many"), the source EJB of the role (using the `<ejb-name>` associated with the EJB), and a `<cmr-field>` element that specifies the property on the bean that the role is associated with. In our case, we use the `profileLocal` and `personLocal` properties for the `<cmr-field>` elements to correspond to the accessor methods we defined earlier on our beans.

The last thing we need to do is provide the EJB container with the information specifying how this relationship is represented in persistent storage. Again, a format for specifying this information isn't defined in the EJB specification. You need to determine what form of persistence your J2EE server supports and how it requires the persistence mapping to be defined. The details of providing this mapping will be server-specific: JBoss will do it one way, WebLogic will do it another, and so on. Consult your application server's documentation for full details.

Since we defined both read and write accessors for either end of this EJB relationship, we can alter the persistent relationship between a `Person` and his `Profile` by simply using the `setPersonLocal()` or `setProfileLocal()` methods on the `Profile` or `Person` implementations, respectively. The data in persistent storage will be adjusted accordingly by the EJB container to reflect the new relationship. In the case of RDBMS persistent storage, the foreign key column(s) point to the appropriate row in the target table.

6.8. Message-Driven Beans

Message-driven EJBs are components that are managed by an EJB container like other EJBs, but they are invoked asynchronously by clients using JMS messages rather than method calls. In terms of their runtime lifecycle and container management, message-driven beans are closest in nature to stateless session beans. They are invoked indirectly and asynchronously, so they don't maintain conversational state on behalf of clients, and they can be pooled by the container to handle incoming messages.

From the client's perspective, a message-driven bean is seen as simply another JMS destination with which to exchange messages. Clients do not even know that an EJB container is involved; they simply perform the usual steps to establish a session with the JMS queue or topic and exchange messages with it. We only touch on these JMS-specific details here; for more details, see [Chapter 11](#).

Message-driven EJBs are the simplest to implement since they require only a bean implementation class. The only client interaction with them is through JMS messages, so there's no need for client interfaces or home interfaces. A message-driven bean must implement both the `MessageDrivenBean` interface from the EJB API as well as the `MessageListener` interface from JMS. The bean's lifecycle is managed by the EJB container, including its association with a JMS destination. While a simple, standalone JMS `MessageListener` needs to be programmatically associated with its message destination, the EJB container associates message-driven beans with their destinations automatically using the information provided in the bean's deployment descriptor. Once a message-driven bean is activated, it receives messages from its designated JMS destination and responds to them through its `onMessage()` method, just like any JMS `MessageListener`.

As an example, suppose we want to expose the profile management capabilities of our `Profile` EJB to messaging clients. We may decide to do this if we plan to support clients that can't utilize direct EJB calls or that have a need for asynchronous calls. We can use a message-driven EJB to do this. Our message-driven EJB can run within our EJB container along with our `Profile` EJB and act as a proxy between the `Profile` EJB and messaging clients. The full source for our message-driven `ProfileProxyBean` can be found in the source code bundle for the book, in the file `com/oreilly/jent/ejb/messageDriven/ProfileProxyBean.java`.

Message-driven beans require the following EJB container callbacks:

`ejbCreate()`

Message-driven beans must have a single, no-argument `ejbCreate()` method.

`setMessageDrivenContext()`

This method is analogous to the `setSessionContext()` and `setEntityContext()` methods on session and entity beans, respectively. The `MessageDrivenContext` is used to access the runtime environment of the EJB container. The container calls this method just before it calls

the bean's `ejbCreate()` method.

`ejbRemove()`

This method is called when the container decides that a message-driven bean is no longer needed in active memory and wants to destroy the bean to reclaim its resources.

Our `ProfileProxyBean` includes these methods, of course. The `ejbRemove()` and `setMessageDrivenContext()` methods are fairly simple: `ejbRemove()` doesn't need to do anything in our case because we don't have any resources to clean up, and `setMessageDrivenContext()` simply stores the context in an instance member variable. The `ejbCreate()` method is a little more interesting because it needs to initialize a connection to the home interface for the `Profile` entity EJB:

```
public void ejbCreate ( ) throws CreateException {
    System.out.println("Create called on ProfileProxyBean.");
    try {
        Context ctx = new InitialContext( );
        mProfileHome =
            (ProfileHome)PortableRemoteObject.narrow(
                ctx.lookup("ejb/ProfileHome"), ProfileHome.class);
    }
    catch (NamingException ne) {
        throw new CreateException("Failed to locate home interface: " +
            ne.getMessage( ));
    }
}
```

A message-driven bean also implements the `javax.jms.MessageListener` interface and therefore needs to have an `onMessage()` method for dealing with incoming messages. In our `ProfileProxyBean`, we expect to receive a `MapMessage` containing name/value pairs representing entries to be set on a particular profile.

```
public void onMessage(Message msg) {
    if (msg instanceof MapMessage) {
        MapMessage mMsg = (MapMessage)msg;
        try {
            String name = mMsg.getString("OWNER");
            if (name != null) {
                Profile prof = mProfileHome.findByPrimaryKey(name);
                Enumeration eNames = mMsg.getMapNames( );
                while (eNames.hasMoreElements( )) {
                    String eName = (String)eNames.nextElement( );
                    String eVal = mMsg.getString(eName);
                    if (!eName.equals("OWNER")) {
                        prof.setEntry(eName, eVal);
                    }
                }
            }
        }
    }
}
```

```

    }
}
catch (JMSEException je) {
    System.out.println("JMS error processing message to ProfileProxy: " +
        je.getMessage( ));
}
catch (RemoteException re) {
    System.out.println("Remote exception while accessing profile: " +
        re.getMessage( ));
}
catch (FinderException fe) {
    System.out.println("Failed to find Profile named in message: " +
        fe.getMessage( ));
}
}
else {
    System.out.println("Non-MapMessage received by ProfileProxy, type = " +
        msg.getClass( ).getName( ));
}
}

```

The name of the profile owner is expected in the `OWNER` property on the `MapMessage`. In the `onMessage()` method, we get the name of the target profile, then use `findByPrimaryKey()` on the `ProfileHome` to get a reference to the `Profile` to be updated. Next, we get a list of the property names present in the `MapMessage` and set the corresponding entry on the `Profile` according to the values found in the message.

6.8.1. Deploying Message-Driven EJBs

Deploying our message-driven bean is the same general procedure as for session and entity beans. We need to provide a `<message-driven>` element in an `ejb-jar.xml` file that describes our bean. In this case, we would use an entry such as the following:

```

<ejb-jar ...>
  <enterprise-beans>
    ...
    <!-- A message-driven proxy for the Profile beans -->
    <message-driven>
      <ejb-name>MessageDrivenProfileProxy</ejb-name>
      <ejb-class>
        com.oreilly.jent.ejb.messageDriven.ProfileProxyBean
      </ejb-class>
      <transaction-type>Container</transaction-type>
      <message-destination-type>
        javax.jms.Queue
      </message-destination-type>
    </message-driven>
  </enterprise-beans>
</ejb-jar>

```

```

        ...
    </enterprise-beans>
    ...
</ejb-jar>

```

The `<message-driven>` element lists the name used to refer to the bean (`<ejb-name>`), the class of the bean implementation (`<ejb-class>`), whether we want the transactions managed by the bean or the container (`<transaction-type>`), and what type of JMS destination the bean should be connected to (`<message-destination-type>`).

6.8.2. Message-Driven Clients

A client of a message-driven EJB simply connects to the JMS destination for the bean and exchanges messages with it: it doesn't need to use any EJB API calls at all. A client that wanted to use our message-driven `Profile` bean would use basic JMS code such as the following to update a user's profile:

```

Context context = new InitialContext(...);

// Look up the JMS queue connection factory, make a connection, start it
QueueConnectionFactory qFactory =
    (QueueConnectionFactory)context.lookup(
        "java:comp/env/jms/jent-EJB-connFactory");
QueueConnection qConn = qFactory.createQueueConnection( );
qConn.start( );

// Look up the JMS message queue for the message-driven bean, and
// create a session with it
Queue profQueue = (Queue)context.lookup("jms/ProfileProxyQueue");
QueueSession qSession =
    qConn.createQueueSession(false, Session.AUTO_ACKNOWLEDGE);

// Create a sender
QueueSender sender = qSession.createSender(profQueue);

// Create a MapMessage
MapMessage msg = qSession.createMapMessage( );
// Set the name for the target profile
msg.setString("OWNER", "Kaitlyn");
// Set an entry value
msg.setString("favoriteColor", "green");
// Send the message
sender.send(msg);

```

The first half of this code segment is simply establishing a session with the message destination. Here, we've assumed that our message-driven bean has been associated with a message queue stored in JNDI under the name `java:comp/env/jms/jent-EJB-connFactory`. In the last few lines of this

client code, we construct a `MapMessage`, set the `OWNER` field in the map to be the name of the owner of the target profile, and then set one or more profile entries to be used to update the profile.



6.9. Transaction Management

One of the advanced value-added features that Enterprise JavaBeans provides over regular remote objects is transaction management. The EJB container can broker transaction contexts on behalf of your EJBs, making sure to "do the right thing" in terms of handling transaction defers, commits, or rollbacks.

The Enterprise JavaBeans architecture relies on the Java Transaction API (JTA) for transaction support. [Chapter 16](#) provides a general overview of transaction management concepts and the JTA. In that chapter, you'll also find some extended topics related to bean- and container-managed transactions with EJBs. If you're looking for material specifically about transaction management within an EJB context, we recommend that you start with the material in this section and then turn to [Chapter 16](#) if you need to know more about the general concepts of transaction management and the JTA. If you need to delve into the broader context of transaction management in general (e.g., if you need to integrate distributed transactions across multiple resource managers into your application), you may want to start with the general overview provided in [Chapter 16](#), then return to the EJB-specific details provided here.

The EJB container is the principal player in the area of transaction management since it is responsible for either generating transactions around client interactions with the bean or detecting client-requested transactions and then notifying the EJB objects about transaction boundaries (start and commit/rollback). This is another reason all interactions with EJB instances are brokered by the container so it can properly manage transactions for EJBs that support/require it.

In the context of an Enterprise JavaBeans component, transactions can be defined by any of the three runtime roles that participate in the EJB architecture: the client of the EJB object, the EJB container, or the EJB object itself. In all cases, the EJB container decides how to handle the transaction context whenever a method is invoked on an EJB object. During a bean's lifetime, the container decides whether to execute the bean's business methods within the client's transaction, or within a transaction that the container defines, or to allow the bean to manage its own transaction boundaries. JTA supports what is called a flat transaction model, meaning that transactions can't be nested within each other. If a client defines its own transaction (as shown previously) and invokes an EJB method, which attempts to define its own, separate transaction context, the EJB container needs to decide how to mitigate these two transaction contexts so that there is only one (or else throw an exception if the situation can't be resolved). What the container does in a particular transactional situation is determined largely by deployment descriptor elements that indicate how an EJB transaction is to be managed.

6.9.1. Transaction Management: Bean-Managed Versus Container-Managed

First, at a high level, you need to specify to the container whether your EJB will manage its own transactions (by creating its own `UserTransaction` objects and starting/committing them in its business methods) or will rely on the container to manage its transactions. This is done in the

deployment descriptor using the `<transaction-type>` element:

```

...
<enterprise-beans>
  <session>
    ...
    <transaction-type>Container</transaction-type>
    ...
  </session>
  ...
</enterprise-beans>
...

```

A `Container` value indicates container-managed transactions; a `Bean` value indicates bean management. If you specify bean-managed transactions, the container will automatically suspend any client-defined transactions before calling methods on your EJB and resume them after any bean-defined transaction contexts have been committed/rolled back. If you use container-managed transactions, the container will use the method-level transaction support attributes discussed in the next section to determine how it deals with transactions. Normally you will want to use container-managed transactions since managing your own transaction boundaries within your bean code isn't usually necessary. Bean-managed transactions may be necessary when you need to define multiple transaction contexts in the span of a single method. But situations like this are rare, and where possible you should avoid the additional complexity in your code and leave the transaction management to the EJB container.

Entity beans aren't allowed to use bean-managed transaction management because the persistence management services of the EJB container require that entity beans never create their own `UserTransactions`. So you never specify a `<transaction-type>` for entity beans in their deployment descriptor; it's always assumed to be `Container`.

6.9.2. Transaction Support Attributes

If your EJB uses container-managed transaction management, you can further specify how each method on the EJB "supports" transactions, and therefore how the container should deal with different transaction contexts when it invokes these methods on your beans. The following `<transaction-support>` values are available when specifying them in your deployment descriptor:

NotSupported

The method can't support transactions, so it must be called without a transaction context. If the client has initiated a transaction, it is suspended by the container before the bean's method is invoked. After the method completes, the container resumes the client's transaction. The container doesn't define its own transaction context for the method call, so if the EJB calls other EJBs from the method, no transaction context is passed along to these methods.

Supports

The method supports transactions if requested. If the client calls a method on the bean while within a transaction, the method is invoked within the client's transaction context; this transaction will be passed on to any other EJB methods called within this method. If the client calls the method with no transaction context, the container doesn't create one; it runs the method with no transaction context.

Required

The method must be executed within a transaction context. If the client is already in a transaction of its own, the transaction context is used to invoke the EJB method. If not, the container creates a new transaction before calling the bean's method and commits/rolls back the transaction when the bean's method finishes, but before the method results are returned to the client.

RequiresNew

The method must be executed within a new transaction. The container automatically starts a new transaction before calling a remote method on the bean, and commits the transaction when the method finishes, but before the results are returned to the client. If the client calls a remote method while within a transaction, the client's transaction is suspended by the container before executing the bean's method within the new transaction and resumed after the new transaction is committed.

Mandatory

The method must be run within the context of a client-initiated transaction. If the client calls the method on the bean without starting a transaction first, the container throws a `javax.transaction.TransactionRequiredException` if the client is remote. It throws a `javax.ejb.TransactionRequiredLocalException` for local clients.

Never

The method must never be called with a transaction context. If a client calls this method from within its own transaction, the container will throw an exception back to the client (`java.rmi.RemoteException` if the client is remote, `javax.ejb.EJBException` if the client is local).

These attributes are associated with methods on the EJB in the `<assembly-descriptor>` section of the deployment descriptor. You provide `<container-transaction>` elements that specify one or more methods and a transaction support attribute to apply to those methods. Using our stateless session `ProfileManager` EJB as an example, we can specify that its `getProfile()` method "supports" transactions using this XML stanza:

```
...
<assembly-descriptor>
  ...
  <container-transaction>
    <method>
      <ejb-name>ProfileManagerBean</ejb-name>
```

```

        <method-name>getProfile</method-name>
    </method>
    <trans-attribute>Supports</trans-attribute>
</container-transaction>
    ...
</assembly-descriptor>
    ...

```

There are some restrictions on the use of these attribute values for some types of EJBs, as described next.

6.9.2.1. Message-driven beans

For message-driven beans, only the `NotSupported` and `Required` attributes are allowed since client transactions can't be propagated to the EJB through a message passed to a JMS service.

6.9.2.2. CMP entity beans

Entity beans using CMP use transactions by definition when the container interacts with the persistent store on their behalf. Therefore, entity EJBs using CMP should use either the `Required`, `RequiresNew`, or `Mandatory` attributes since the others allow for contexts with no transactions. Technically, you can use the other attribute values with CMP beans, but EJB containers are not required to support them in this situation, so it's wise to avoid them.

6.9.2.3. Stateful session beans

Stateful session beans that implement the session synchronization interface, as discussed at the end of "Session Bean Specifics" earlier in this chapter, depend on transaction notifications from the container to keep their state in sync. Without a transaction, the container is not able to invoke the session synchronization callbacks. Therefore, this flavor of stateful session bean can use only attribute values that result in a transaction: `Required`, `RequiresNew`, and `Mandatory`.

6.9.2.4. Stateless session beans

If a stateless session bean is published as a web service (as detailed in [Chapter 12](#)), you are not allowed to use the `Mandatory` attribute value. A transaction context initiated by a SOAP client cannot be propagated across a SOAP call to the session bean, so mandating that the client initiate a transaction for the bean request doesn't make sense.

6.10. EJB 3.0

The 3.0 version of the EJB specification is currently in draft form and should be released in final form shortly after this book hits the shelves. Early versions of EJB 3.0 implementations have started to appear already. In anticipation of the eventual wide availability of EJB 3.0 containers, we provide a short introduction to the new programming model introduced in the 3.0 specification.

The biggest complaint by developers about the current EJB programming model (Version 2.1) is that it is just too complex. Several Java interfaces and classes have to be written for each EJB, and configuration details have to be supplied in the form of an XML deployment descriptor. And that's after you've figured out the various rules and contracts for implementing container callbacks, using container services, and so on. Some projects find the benefits of the EJB container services to be worth it, and their developers work through the EJB complexities to get these benefits. But many developers have found other, simpler solutions to get these services. The most notable "EJB competitor" has been Hibernate, discussed in [Chapter 20](#). Hibernate presents a much simpler programming model for handling object/relational persistence in enterprise applications, and many developers have voted with their feet and gone down the path of least resistance.

It's not surprising, then, that the primary goal in writing the EJB 3.0 specification was *simplification*. The specification leads listened to the critics, borrowed design principles (and developers) from Hibernate and other projects, and put together a streamlined programming model in EJB 3.0. Some of the significant new features in the new model are:

- An EJB can be implemented with a single Java class. You aren't required to write home and client interfaces (though you still can if you really want to).
- Annotations (based on the J2SE annotations model discussed in [Chapter 21](#)) have been introduced. These annotations can be used to specify most if not all of the configuration details for the EJB, including deployment descriptor details, persistence mappings, web service methods, container callbacks, and the like.
- The bean doesn't have to follow the `EnterpriseBean` interface (or its subclasses). Annotations can be used to map your methods to the container callbacks, letting you write your beans the way you want to write them.

To demonstrate the new model, [Example 6-9](#) shows our stateful `ProfileBean` implemented under the EJB 3.0 programming model. This one, smaller class replaces all the code in the home, client, and implementation classes we had earlier. The EJB container implementation will generate the client and home interfaces based on the annotations we've placed in the code, such as the `@Remote` indicator on the class and the `@BusinessMethod` tag on our `getName()` method. In addition, we may be able to eliminate writing an EJB deployment descriptor since we've included the specification of dependent resources with an `@Resource` annotation and the other information we specified in our deployment descriptor can be inferred by other features and annotations in the code, like the `@Stateful` tag.

Example 6-9. EJB 3.0 version of Profile bean

```
// Indicate this is a stateful session bean
@Stateful(name="ProfileEJB3")
// Request a remote client interface for this bean
@Remote
// Specify deployed resources this bean depends on
@Resource(name="ProfileDB",
          resourceType="javax.sql.DataSource",
          jndiName="jdbc/ProfileDB")
// Note: No need to extend the SessionBean interface
public class ProfileBeanEJB3 {
    // Name of the person owning the profile
    private String mName = "";
    // Entries in the profile (name/value pairs)
    private Properties mEntries = new Properties();

    // No activate, passivate methods - we don't need them,
    // no need to write them.

    // Tag this as a business method, so it appears in the generated
    // client interface(s)
    @BusinessMethod
    // Also tag this as a web method, so it will be callable from
    // the SOAP interface. Note that this annotation is actually
    // defined in the new JAX-RPC 2.0 spec, not in the EJB 3.0 spec.
    @WebMethod
    public String getName() {
        return mName;
    }
}
```

Note that the example we show here may not be directly applicable to the final EJB 3.0 specification when it is released. This example is based on the latest draft version of the specification available at this time.

If you have invested development time and resources into EJB 2.1 (or earlier) components, don't fret. There are backward-compatibility measures built into the EJB 3.0 specification. But given the cleaner, more flexible, and simpler model provided in EJB 3.0, you likely won't want to run your "legacy" EJBs for very long before converting them to this approach.

Chapter 7. Java and XML

Extensible Markup Language (XML) has become an essential technology for enterprise applications. The XML specification^[*] allows users to define unique and structured document formats, allowing for easy and flexible data exchange between applications. Since the syntax of an XML document is bound by a public specification, XML documents can be read and manipulated by a wide variety of tools. Because XML documents are text-based, they can be easily transmitted between different systems using a number of transportation mechanisms, from JMS to HTTP.

[*] The complete specification, developed by the World Wide Web Consortium, is available from the W3C web site at <http://www.w3.org/XML>. The site also includes a variety of related specifications and other useful resources.

XML documents can be freely structured, although they must abide by a basic set of XML rules that define a well-formed document. More commonly, however, the document structure is further defined by a Document Type Definition (DTD). With a standardized DTD, enterprise applications can exchange data without knowledge of each other's native formats. Industry working groups have defined DTDs for everything from bank transactions to medical records to electronic books. DTDs are very common, although they have been partially supplanted by XML Schemas, a more sophisticated way of describing the structure of an XML document that, among other things, supports defining data types.

The advantages of combining Java and XML are obvious: a cross-platform language and a cross-platform data specification. We don't have space here to discuss XML itself in depth; for more information, try *Learning XML* by Erik T. Ray (O'Reilly). This book covers the XML specification itself, including topics such as XML namespaces, DTDs, the XLink and XPointer specifications for rich links, and the XSLT transformation specification (which we will touch on later in this chapter). We have tried to include enough information to give newcomers a taste of what can be done.

In this chapter, we're going to take a quick look at Sun's Java API for XML Processing (JAXP) Version 1.2, which provides a standardized approach to processing XML files in Java. JAXP is included in Version 1.4 of the J2EE specification. While originally an enterprise API, JAXP joined the standard J2SE distribution with J2SE 1.4. JAXP includes three other specifications by reference: the Simple API for XML parsing (SAX), Version 2; the W3C's Document Object Model (DOM), Level 2; and the XSLT (XML Stylesheet Transformation) specification. We also discuss using JAXP to access DOM and SAX parsers and XSLT processors and offer a quick introduction to using DOM and SAX. If you want to know more about using Java with XML, try Brett McLaughlin's *Java and XML* (O'Reilly).

7.1. Using XML Documents

XML allows developers to create tag-based markup structures that are bound by a set of rules defined in a public specification. The actual content of any particular XML file is left undefined by the specification. Here's an example, *orders.xml*, that represents two orders made to a fictional online shopping site. Each order includes identifying information (an order number and a customer number), a shipping address, and one or more items. The shipping address encapsulates both the shipping method and the shipping destination, and each item includes an identifying number and the quantity ordered, as well as an optional handling instruction.

Most elements include an opening and closing tag, with the element attributes set in the opening tag. Some elements are "empty," such as the first example of the `<item>` tag. Empty elements terminate with `/>` instead of simply `>`, and don't need a separate closing tag. The significance of an empty tag is in either its attributes or its mere presence. The data is simple enough that the structure should be clear to the reader. Here's the actual XML:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE orders SYSTEM "orders.dtd">
<orders>
  <order idnumber="3123" custno="121312">
    <shippingaddr method="Camel"><![CDATA[One Main St
                                     Boston, MA 02112]]></shippingaddr>
    <item idnumber="7231" quantity="13"/>
    <item idnumber="1296" quantity="2">
      <handling>Please embroider in a tasteful manner!</handling>
    </item>
  </order>
  <order idnumber="3124" custno="12">
    <shippingaddr method="FedEx"><![CDATA[285 York St.
                                     New Haven, CT 06510]]></shippingaddr>
    <item idnumber="12" quantity="8"/>
  </order>
</orders>
```

At the simplest level, an XML document must merely be well-formed, meaning that the document adheres to all of the syntax rules defined by the XML specification. These rules define the XML declaration on the first line and specify how tags may be formed and nested^[*]. The requirements for a well-formed document don't include any particular XML tags except for the XML version and specify structure in only the broadest possible terms. So for most applications, simply knowing that a document is well-formed is not particularly helpful. Of course, one can specify that only files of a particular format should be used as input for a particular program, but without a way to define what that format should be and whether documents conform to it, an extensible markup language doesn't make a great deal of sense.

[*] And quite a bit more, including entity escape sequences, namespaces, valid character sets, and so on. But we don't have room here for a full discussion, so again, we recommend O'Reilly's *Learning XML*.

However, there is a solution. The second line of *orders.xml* specifies that the file should conform to a DTD. The DTD goes a step beyond the well-formed requirement and specifies the allowable XML tags their formats, and the allowable structure. The DTD for the *orders.xml* file requires that all `<order>` tags be nested within an `<orders>` tag, all orders have at least one item and a shipping address, all items include identifier and quantity attributes, and so forth. Here's the DTD *orders.dtd*:

```
<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT orders (order+)>
<!ELEMENT order (shippingaddr, item+)>
<!ELEMENT shippingaddr (#PCDATA)>
<!ELEMENT item (handling?)>
<!ELEMENT handling (#PCDATA)>
<!ATTLIST order
  idnumber CDATA #REQUIRED
  custno CDATA #REQUIRED
>
<!ATTLIST shippingaddr
  method (FedEx | UPS | USPS | Camel) #REQUIRED
>
<!ATTLIST item
  idnumber CDATA #REQUIRED
  quantity CDATA #REQUIRED
>
```

The XML layer of an application generally consists of one or more DTDs and a set of documents. The DTDs are written ahead of time, by an individual developer, a working group, an application server vendor, or other provider. Some documents, particularly those related to configuration and profiling tasks (such as the J2EE deployment descriptors), are edited by hand and read by software.

The previous example is more transaction-oriented. Documents like *orders.xml* would likely be generated by a purchasing frontend (such as a web site) and transmitted over the network to a fulfillment system (such as a corporate order tracking database) via HTTP, JMS, or some other transport layer. The receiving software reads the document and processes it, often without any human intervention at all. Standardized DTDs mean that the two sides of the exchange can easily be provided by different vendors.

7.1.1. XML Schema

You may have noticed in *orders.dtd* that the DTD standard is missing a few things. For one, the specification of data types for elements and their attributes is extremely limited. Your application may require that the contents of the `idnumber` attribute be a valid integer, for example. The closest you can get to this with a DTD is to declare the attribute as an enumerated type. But this is useful only if you can enumerate all of the allowed integer values for `idnumber`, which isn't possible in most cases. So typically the only option you'd have is to declare the attribute as character data, as we've done in

the example, and then validate the attribute values in your application code. Things are even more limited for data within elements, where DTDs allow only character data, no data (empty), or completely unspecified content. Again, in most cases, you would need to do additional data validation within your application.

DTDs are also limiting in terms of entity namespaces and granularity. You can have only a single DTD associated with a given XML document, and that DTD applies to the entire document.

The XML Schema standard provides a much richer mechanism for describing the structure and content of an XML file. A schema defines a set of data types that is then used to define element and attribute types. One or more schemas can then be referenced in a given XML document and applied to specific elements in the document. This provides a much more flexible model than DTDs. The data type descriptions are more powerful, the namespace facility allows you to integrate multiple schemas into a single document, and schema types can be applied at the individual element or attribute level in your XML documents.

In addition, XML schema files are regular XML files, so they can be processed using standard XML-handling tools. This makes it much easier to write software that can respond to incoming XML dynamically.

An XML schema describing the *orders.xml* file might look something like this:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<xs:schema xmlns="jent:xml-orders"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  targetNamespace="jent:xml-orders">
  <!-- An item contains a sequence of handling elements, as well
    as idnumber and quantity attributes -->
  <xs:complexType name="itemType">
    <xs:sequence>
      <xs:element name="handling" type="xs:string"
        minOccurs="0"/>
    </xs:sequence>
    <xs:attribute name="idnumber"
      type="xs:positiveInteger" use="required"/>
    <xs:attribute name="quantity" type="xs:integer"
      use="required"/>
  </xs:complexType>
  <!-- A shippingaddr contains a method attribute, that must be
    one of the enumerated values listed below, and contains
    a single text element as a child. -->
  <xs:complexType name="shippingaddrType" mixed="true">
    <xs:attribute name="method" use="required">
      <xs:simpleType>
        <xs:restriction base="xs:NMTOKEN">
          <xs:enumeration value="USPS"/>
          <xs:enumeration value="Camel"/>
          <xs:enumeration value="FedEx"/>
          <xs:enumeration value="UPS"/>
        </xs:restriction>
      </xs:simpleType>
    </xs:complexType>
```

```

        </xs:attribute>
    </xs:complexType>
    <!-- An order is a sequence of items with shippingaddrs, with
         idnumber and custno attributes that are integer values -->
    <xs:complexType name="orderType">
        <xs:sequence>
            <xs:element name="shippingaddr" type="shippingaddrType"/>
            <xs:element name="item" type="itemType"
                maxOccurs="unbounded"/>
        </xs:sequence>
        <xs:attribute name="idnumber" type="xs:positiveInteger"
            use="required"/>
        <xs:attribute name="custno" type="xs:positiveInteger"
            use="required"/>
    </xs:complexType>
    <!-- An orders element contains a sequence of order elements -->
    <xs:complexType name="ordersType">
        <xs:sequence>
            <xs:element name="order" type="orderType"
                maxOccurs="unbounded"/>
        </xs:sequence>
    </xs:complexType>
    <!-- Now define our root element, referencing the appropriate type
         from the set defined above -->
    <xs:element name="orders" type="ordersType"/>
</xs:schema>

```

The schema representation is longer than the DTD representation, but it's also easier to read. The most important pieces are the `<xs:element>` and `<xs:complexType>` elements. Complex types in XML Schema are collections of element and attribute descriptions. These types are bound to XML elements using the `type` attribute on `element` declarations. In our example, the root element, `orders` is defined at the end of the schema, and it uses the type `ordersType`, which is defined by the `complexType` defined above it.^[*] The `type` attribute on the `element` specifies that the data included in the element or attribute value should conform to a particular data type.

[*] This is a two-step process because we might want to use the same complex type definition for two or more different elements without having to copy and paste.

The `complexType` structure is the top of the food chain in terms of the data-typing facilities in XML Schema. In `orders.xsd`, we also constrain various elements and attributes using the `<xs:string>`, `<xs:integer>`, and `<xs:positiveInteger>` data types, standard data types specified in the XML Schema specification. XML Schema also allows you to define new data types based on existing data types, using the `<xs:simpleType>` and `<xs:restriction>` tags. These are used in the `shippingaddrType` to define the type for the name attribute, specified as an enumerated type using the built-in `NMTOKEN` type. Other restrictions are available for other data types. The `xs:string` data type, in particular, supports restrictions for minimum and maximum length and for regular expression matching, allowing for effectively unlimited control of data formatting.

As mentioned earlier, XML schemas are referenced within XML documents as a namespace that defines entities (elements and attributes) to be used in the document. Typically, this is done by specifying the namespace (and its corresponding schema) for the root element in your XML

document. Adjusting our *orders.xml*/document to refer to our XML schema would look like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<orders xmlns="jent:xml-orders"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="jent:xml-orders
                            http://mycompany.com/orders.xsd">
  <order idnumber="3123" custno="121312">
    <shippingaddr method="Camel"><![CDATA[One Main St
                                          Boston, MA 02112]]></shippingaddr>
    <item idnumber="7231" quantity="13"/>
    <item idnumber="1296" quantity="2">
      <handling>Please embroider in a tasteful manner!</handling>
    </item>
  </order>
  <order idnumber="3124" custno="12">
    <shippingaddr method="FedEx"><![CDATA[285 York St.
                                          New Haven, CT 06510]]></shippingaddr>
    <item idnumber="12" quantity="8"/>
  </order>
</orders>
```

Note the new attributes on the `<orders>` element. `xmlns` is a standard attribute supported by all XML elements and is used to specify a namespace to be used by the element and its children. If you want or need to use a prefix with the elements and attributes referenced from the schema, you can specify the prefix by appending it (with a colon separator) to the `xmlns` attribute entry. We've done this with our *orders.xml*/example when we import the XML Schema "instance" schema, a standard schema that defines some attributes that are useful when dealing with schemas in XML documents. We've imported the schema as a namespace using the `xsi` prefix. When we want to reference elements or attributes from this namespace, we use this prefix on their names. The `schemaLocation` attribute shows this in action; this attribute comes from the "instance" schema, so we refer to it as `xsi:schemaLocation`.

The `schemaLocation` attribute we use on the `orders` element also demonstrates one way to specify the location of schema definition files. In the `xmlns` attribute, we specified our orders schema using a URN, `jent:xml-orders`. The `schemaLocation` attribute tells the XML parser how to resolve this URN to a physical schema file. In our case, we're telling the parser that the schema file is located at the URL <http://mycompany.com/orders.xsd>.

There's quite a lot more to XML Schema, but this should get you started. For a much more detailed introduction, we suggest *XML Schema* by Eric van der Vlist (O'Reilly).^[*]

[*] There is also an excellent short tutorial on O'Reilly's [XML.com](http://www.xml.com/pub/a/2000/11/29/schemas/part1.html) at <http://www.xml.com/pub/a/2000/11/29/schemas/part1.html>.

7.2. Java API for XML Processing

The JAXP API is bundled into the JDK as of 1.4 and is an optional package for earlier versions. It is also a standard component of the J2EE 1.3 and 1.4 platforms. XML Schema support is available only in JAXP 1.2, which is part of J2EE 1.4 and JDK 1.4. J2EE 1.3 includes the 1.1 release of JAXP, which is otherwise functionally identical. If you're working in a Java 5.0 environment, you're using JAXP 1.3 which adds XPath and XInclude support to the API, among other things. The full specification and a reference implementation are available from <http://java.sun.com/xml>.

The SAX and DOM APIs that are actually used for processing XML files don't include a standard method for creating a parser object; this is one of the voids JAXP fills. The API provides a set of `Factory` objects that will create parsers or XSLT processors. Additionally, JAXP defines a programmatic interface to XSLT processors.

The actual parser and processor implementations used by JAXP are pluggable. You can use the Crimson parser, the Apache Xerces parser (available from <http://xml.apache.org>), or any other JAXP-compatible parser. Version 1.1 of the reference implementation shipped with Sun's Crimson XML parser and the Xalan XSL engine from the Apache XML project (again, see <http://xml.apache.org>). In JAXP 1.2, the Crimson parser was replaced with the Xerces parser. There are still variations in support for different levels of functionality across parser implementations. The examples in this chapter have been tested with the Xerces parser that shipped with JAXP 1.2.

7.2.1. Getting a Parser or Processor

To retrieve a parser or processor from inside a Java program, call the `newInstance()` method of the appropriate factory class, either `SAXParserFactory`, `DocumentBuilderFactory`, or `TransformerFactory`. The actual factory implementation is provided by the parser vendor. For example, to retrieve the platform default SAX parser:

```
SAXParserFactory spf = SAXParserFactory.newInstance();
spf.setValidating(true); //request a validating parser
try {
    SAXParser saxParser = spf.newSAXParser();
    // Process XML here
} catch (SAXException e) {
    e.printStackTrace();
} catch (ParserConfigurationException pce) {
    pce.printStackTrace();
} catch (IOException ioe) {
    ioe.printStackTrace();
}
```

The next three sections will deal with what you can do once you've actually retrieved a parser. For the time being, let's treat it as an end in itself.

`SAXParserFactory` includes a static method called `newInstance()`. When this method is called, the JAXP implementation searches for an implementation of `javax.xml.parsers.SAXParserFactory`, instantiates it, and returns it. The implementation of `SAXParserFactory` is provided by the parser vendor; it's `org.apache.xerces.jaxp.SAXParserFactoryImpl` for the Xerces parser.

The system looks for the name of the class to instantiate in the following four locations, in order:

1. In one of these system properties:

- `javax.xml.parsers.SAXParserFactory`
- `javax.xml.parsers.DocumentBuilderFactory`
- `javax.xml.parsers.TransformerFactory`

2. In the `lib/jaxp.properties` file in the `JRE` directory. The configuration file is in key=value format, and the key is the name of the corresponding system property. Therefore, to set Crimson as the default parser, `jaxp.properties` would contain the following line:

```
javax.xml.parsers.SAXParserFactory=org.apache.crimson.jaxp.  
SAXParserFactoryImpl
```

3. In the application jar file, via the Services API. The API looks for the classname in a file called `META-INF/services/parserproperty` in which the filename (`parserproperty`) is the property name corresponding to the desired factory. The runtime environment checks every available jar file, so if you have multiple parsers available to your application, specify the desired factory using one of the previous methods to prevent nondeterministic behavior.

4. In a platform default factory instance.

Once you have a factory, various parser options can be set using the factory-specific set methods. `SAXParserFactory` and `DocumentBuilderFactory`, for instance, include `setNamespaceAware()` and `setValidating()` methods, which tell the factory whether to produce a parser that is aware of XML namespaces (and will fail if the document being parsed doesn't properly conform to the namespace specification) and whether to validate against any DTD and/or schema specified by the XML document itself. To enable schema validation in parsers, you'll need to set the following attribute on the `DocumentBuilderFactory`, or set the equivalent property on the `SAXParser` using `setProperty()`:

```
DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();  
dbf.setAttribute(  
    "http://java.sun.com/xml/jaxp/properties/schemaLanguage",  
    "http://www.w3.org/2001/XMLSchema");
```

This attribute is a standard property defined in the JAXP specification and it tells the parser which schema language to use for validation. In this example, we're using the 2001 version of the XML Schema language.

Factories are threadsafe, so a single instance can be shared by multiple threads. This allows parser factories to be instantiated in a Java Servlet `init()` method or other centralized location. Parsers and processors, however, aren't guaranteed to be threadsafe.



7.3. SAX

The SAX API provides a procedural approach to parsing an XML file. As a SAX parser iterates through an XML file, it performs callbacks to a user-specified object. These calls indicate the start or end of an element, the presence of character data, and other significant events during the life of the parser.

SAX doesn't provide random access to the structure of the XML file; each tag must be handled as it is encountered by the browser. This means that SAX provides a relatively fast and efficient method of parsing. Because the SAX parser deals with only one element at a time, implementations can be extremely memory-efficient, making it often the only reasonable choice for dealing with particularly large files.

7.3.1. SAX Handlers

The SAX API allows you to create objects that handle XML parsing events, by implementing the `org.xml.sax.ContentHandler`, `org.xml.sax.ErrorHandler`, and `org.xml.sax.DTDHandler` interfaces. ^[*] Processing a document with SAX involves passing a handler implementation to the parser and calling the `parse()` method of `SAXParser`. The parser will read the contents of the XML file, calling the appropriate method on the handler when significant events (such as the start of a tag) occur. All handler methods may throw a `SAXException` in the event of an error.

[*] We're not covering `DTDHandler` here, because it's rarely used. It is useful only if you need to know about unparsed entities and notations in the DTD associated with an XML document.

We'll take a look at the `ContentHandler`, the `ErrorHandler`, and the generic but useful `DefaultHandler` interfaces next.

7.3.1.1. ContentHandler

Most, if not all, SAX applications implement the `ContentHandler` interface. The SAX parser will call methods on a `ContentHandler` when it encounters basic XML elements: chiefly, the start or end of a document, the start or end of an element, and character data within an element.

The `startDocument()` and `endDocument()` methods are called at the beginning^[*] and end of the parsing process and take no parameters. Most applications use `startDocument()` to create any necessary internal data stores and use `endDocument()` to dispose of them (for example, by writing to the database).

[*] The first method called by the parser is actually `setDocumentLocator()`, which provides the handler with an implementation of `org.xml.sax.Locator`. This object can report the current position of the parser within the XML file via its `getColumnNumber()`, `getLineNumber()`, `getPublicId()`, and `getSystemId()` methods. However, while parser implementations are strongly encouraged to implement this method, they aren't required to.

When the parser encounters a new element, it calls the `startElement()` method of the `ContentHandler`, passing a namespace URI, the local name of the element, the fully qualified name of the element (the namespace and the local name), and an `org.xml.sax.Attributes` object containing the element attributes.

The `Attributes` interface allows the parser to inform the `ContentHandler` of attributes attached to an XML tag. For instance, the `<order>` tag in our earlier example contained two attributes, `idnumber` and `custno`, specified like this:

```
<order idlabel="321" custno="98173">
```

To retrieve attributes when processing an element, call the `getValue()` method of attributes:

```
public void startElement(String namespaceURI, String localName,
                        String qName, Attributes atts)
    throws SAXException {
    if(localName.equals("order"))
        System.out.println("New Order Number " + atts.getValue("idnumber") +
                            " for Customer Number " + atts.getValue("custno"));
}
```

Note that before we can safely run this line, we need to make sure that we are processing an `<order>` tag; otherwise, there is no guarantee that the particular attributes we are querying will be available.^[*]

[*] The parser returns all the attributes specified in the XML document, either explicitly or through a default value specified in the DTD. Attributes without defaults that aren't explicitly specified in the XML document itself aren't included.

When the parser encounters the closing tag of an element (`</order>`, in this case), the parser calls the `endElement()` method, passing the same namespace URI, local name, and qualified name that were passed to `startElement()`. Every `startElement()` call will have a corresponding `endElement()` call, even when the element is empty.

These four methods all deal with handling information about XML tags but not with the data within a tag (unless that data is another tag). Much XML content consists of textual data outside the confines of tags and attributes. For example, here's the handling instruction from *orders.xml*.

```
<handling>Please embroider in a tasteful manner!</handling>
```

When the SAX parser encounters the text between the tags, it calls the `characters()` method, passing a character array, a starting index within that array, and the length of the relevant character sequence within the array. This simple implementation of `characters()` prints the output to the screen:

```
public void characters(char[] ch, int start, int length)
    throws SAXException {
```

```

    System.out.print(new String(ch, start, length));
}

```

Note that there is no guarantee that all of the characters you want will be delivered in the same call. Also, since the `characters()` method doesn't include any references to the parent element, to perform more complicated tasks (such as treating the characters differently depending on the element that contains them), you will need to store the name of the current element within the handler class itself. [Example 7-1](#), later in this chapter, shows how to do this via the `startElement()` method.

The `characters()` method might also be called when the parser encounters ignorable whitespace, such as a carriage return separating nested elements that don't otherwise have nested character data. If the parser is validating the document against a DTD, it must instead call the `ignoreableWhitespace()` method to report these characters.

7.3.1.2. ErrorHandler

Since SAX is a language-independent specification, it doesn't handle parsing errors by throwing exceptions. Instead, a SAX parser reports errors by calling methods on a user-supplied object that implements the `ErrorHandler` interface. The `ErrorHandler` interface includes three methods: `error()`, `fatalError()`, and `warning()`. Each method takes an `org.xml.sax.SAXParseException` parameter. The programmer is free to handle the errors in whatever manner she deems appropriate; however, the specification doesn't require parsing to continue after a call to `fatalError()`.

7.3.1.3. DefaultHandler

The API also provides the `org.xml.sax.helpers.DefaultHandler` class that implements all three handler interfaces. Since most handlers don't need to override every handler method, or even most, the easiest way to write a custom handler is to extend this object and override methods as necessary.

7.3.2. Using a SAX Parser

Once you have a handler or set of handlers, you need a parser. JAXP generates SAX parsers via a `SAXParserFactory`, as we saw earlier. The `SAXParserFactory` has three methods for further specifying parser behavior: `setValidating()` (which instructs the parser to validate the incoming XML file against its DTD or schema), `setNamespaceAware()` (which requests support for XML namespaces), and `setFeature()` (which allows configuration of implementation-specific attributes for parsers from particular vendors).

It is possible to parse a document directly from a `SAXParser` object by passing an object that implements the `ContentHandler` interface to the `parse()` method, along with a path, URI, or `InputStream` containing the XML to be parsed. For more control, call the `getXMLReader()` method of `SAXParser`, which returns an `org.xml.sax.XMLReader` object. This is the underlying parser that actually processes the input XML and calls the three handler objects. Accessing the `XMLReader` directly allows

programs to set specific `ErrorHandler` and `DTDHandler` objects, rather than being able to set a `ContentHandler` only.

All events in the SAX parsing cycle are synchronous. The `parse()` method will not return until the entire document has been parsed, and the parser will wait for each handler method to return before calling the next one.

7.3.2.1. A SAX example: Processing orders

[Example 7-1](#) uses a SAX `DefaultHandler` to process an XML document containing a set of incoming orders for a small business. It uses the `startElement()` method of `ContentHandler` to process each element, displaying relevant information. Element attributes are processed via the `Attributes` object passed to the `startElement()` method. When the parser encounters text within a tag, it calls the `characters()` method of `ContentHandler`.

You can also call the `set Property()` method on the `SAXParser` to control its behavior. The standard JAXP property we saw in the previous section can be set using this method, for example.

Example 7-1. Parsing XML with SAX

```
import javax.xml.parsers.*;
import org.xml.sax.*;

public class OrderHandler extends org.xml.sax.helpers.DefaultHandler {

    public static void main(String[] args) {
        SAXParserFactory spf = SAXParserFactory.newInstance();
        spf.setValidating(true); //request a validating parser

        XMLReader xmlReader = null;
        try {
            SAXParser saxParser = spf.newSAXParser();
            /* We need an XMLReader to use an ErrorHandler
               We could just pass the DataHandler to the parser if we wanted
               to use the default error handler. */
            xmlReader = saxParser.getXMLReader();
            xmlReader.setContentHandler(new OrderHandler());
            xmlReader.setErrorHandler(new OrderErrorHandler());
            xmlReader.parse("orders.xml");
        }
        catch (Exception e) {
            e.printStackTrace();
        }
    }

    // The startDocument() method is called at the beginning of parsing
    public void startDocument() throws SAXException {
```

```

        System.out.println("Incoming Orders:");
    }

    // The startElement() method is called at the start of each element
    public void startElement(String namespaceURI, String localName,
        String rawName, Attributes attrs)
        throws SAXException {
        if(localName.equals("order")) {
            System.out.print("\nNew Order Number " + attrs.getValue("idnumber") +
                " for Customer Number " + attrs.getValue("custno"));
        }
        else if (localName.equals("item")) {
            System.out.print("\nLine Item: " + attrs.getValue("idnumber") +
                " (Qty " + attrs.getValue("quantity") + ")");
        }
        else if (localName.equals("shippingaddr")) {
            System.out.println("\nShip by " + attrs.getValue("method") + " to:");
        }
        else if (localName.equals("handling")) {
            System.out.print("\n\tHandling Instructions: ");
        }
    }

    // Print Characters within a tag
    // This will print the contents of the <shippingaddr> and <handling> tags
    // There is no guarantee that all characters will be delivered in a
    // single call

    public void characters(char[] ch, int start, int length)
        throws SAXException {
        System.out.print(new String(ch, start, length));
    }

    /* A custom error handling class, although DefaultHandler implements both
        interfaces. Here we just throw the exception back to the user.*/
    private static class OrderErrorHandler implements ErrorHandler {

        public void error(SAXParseException spe) throws SAXException {
            throw new SAXException(spe);
        }

        public void warning(SAXParseException spe) throws SAXException {
            System.out.println("\nParse Warning: " + spe.getMessage());
        }

        public void fatalError(SAXParseException spe) throws SAXException {
            throw new SAXException(spe);
        }
    }
}

```

In a real application, we would want to treat error handling in a more robust fashion, probably by reporting parse errors to a logging utility or EJB. An actual order management utility would populate a database table or an Enterprise JavaBean's object.



7.4. DOM

The DOM API, unlike the SAX API, allows programmers to construct an object model representing a document and then traverse and modify that representation. The DOM API is not Java-specific; it was developed by the W3C XML working group as a cross-platform API for manipulating XML files (see <http://www.w3c.org/XML>). As a result, it sometimes doesn't take the most direct Java-based path to a particular result. The JAXP 1.1 API incorporated DOM Level 2. In JAXP 1.3, this was updated to support DOM Level 3.

DOM is useful when programs need random access to a complex XML document or to a document whose format is not known ahead of time. This flexibility does come at a cost, however, as the parser must build a complete in-memory object representation of the document. For larger documents, the resource requirements mount quickly. Consequently, many applications use a combination of SAX and DOM, using SAX to parse longer documents (such as importing large amounts of transactional data from an enterprise reporting system) and using DOM to deal with smaller, more complex documents that may require alteration (such as processing configuration files or transforming existing XML documents).

7.4.1. Getting a DOM Parser

The DOM equivalent of a `SAXParser` is the `org.w3c.dom.DocumentBuilder`. Many `DocumentBuilder` implementations actually use SAX to parse the underlying document, so the `DocumentBuilder` implementation itself can be thought of as a layer that sits on top of SAX to provide a different view of the structure of an XML document. We use the JAXP API to get a `DocumentBuilder` interface in the first place, via the `DocumentBuilderFactory` class:

```
DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance( );
// Validation
dbf.setValidating(false);
// Ignore text elements that are completely empty
dbf.setIgnoringElementContentWhitespace(false);
// Expand XML entities according to the DTD
dbf.setExpandEntityReferences(true);
// Treat CDATA sections the same as text
dbf.setCoalescing(true);

DocumentBuilder db = null;
try {
    db = dbf.newDocumentBuilder( );
} catch (ParserConfigurationException pce) {
    pce.printStackTrace( );
}
```

The `set()` methods, as with the `SAXParserFactory`, provide a simple method for configuring parser options:

`setCoalescing()`

Joins XML CDATA nodes with adjoining text nodes. The default is `false`.

`setExpandEntityReferences()`

Expands XML entity reference nodes. The default is `true`.

`setIgnoringComments()`

Ignores XML comments. The default is `false`.

`setIgnoringElementContentWhitespace()`

Ignores whitespace in areas defined as element-only by the DTD. The default is `false`.

`setNamespaceAware()`

Requests a namespace-aware parser. The default is `false`.

`setValidating()`

Requests a validating parser. The default is `false`.

`setAttribute()`

Sets various standard and implementation-specific parser attributes.

Once the `DocumentBuilder` is instantiated, call the `parse(String URI)` method to return an `org.w3c.dom.Document` object.

7.4.2. Navigating the DOM Tree

The `Document` object provides the starting point for working with a DOM tree. Once the parser has produced a `Document`, your program can traverse the document structure and make changes. In addition, `Document` implements the `Node` interface, which is the core of DOM's tree structure, and provides methods for traversing the tree and retrieving information about the current node.

Each element, attribute, entity, and text string (indeed, every distinct component within an XML document) is represented in DOM as a node. To determine what kind of node you are working with, you can call the `getNodeTypes()` method. This returns one of the constants specified by the `Node` interface. All node objects have methods for dealing with child elements, although not all nodes may have children. The DOM API also provides a set of interfaces that map to each node type.

The most important DOM node types and their corresponding interfaces are listed in [Table 7-1](#). If you attempt to add child elements to a node that doesn't support children, a `DOMException` is thrown.

Table 7-1. Important DOM node types

Interface	Name property contains	Value	Children	Node constant
Attr	Name of attribute	Yes	No	ATTRIBUTE_NODE
CDATASection	#cdata-section	Yes	No	CDATA_SECTION_NODE
Comment	#comment	Yes	No	COMMENT_NODE
Document	#document	No	Yes	DOCUMENT_NODE
DocumentFragment	#document-fragment	No	Yes	DOCUMENT_FRAGMENT_NODE
DocumentType	Document type name	No	No	DOCUMENT_TYPE_NODE
Element	Tag name	No	Yes	ELEMENT_NODE
Entity	Entity name	No	No	ENTITY_NODE
EntityReferenced	Name of referenced entity	No	No	ENTITY_REFERENCE_NODE
ProcessingInstruction	PI target	Yes	No	PROCESSING_INSTRUCTION_NODE
Text	#text	No	No	TEXT_NODE

For most applications, element nodes (identified by `Node.ELEMENT_NODE`) and text nodes (identified by `Node.TEXT_NODE`) are the most important. An element node is created when the parser encounters an XML markup tag. A text node is created when the parser encounters text that is not included within a tag. For example, if the input XML (we're using XHTML in this example) looks like this:

```
<p>
  Here is some <b>boldface</b> text.
</p>
```

The parser creates a top-level node that is an element node with a local name of `p`. The top-level node contains three child nodes: a text node containing "Here is some," an element node named `b`, and another text node containing "text." The `b` element node contains a single child text node containing the word "boldface."

The `getNodeValue()` method returns the contents of a text node or the value of other node types. It

returns `null` for element nodes.

To iterate through a node's children, use the `getFirstChild()` method, which will return a `Node` reference. To retrieve subsequent child nodes, call the `getNextSibling()` method of the node that was returned by `getFirstChild()`. To print the names of all the children of a particular `Node` (assume that the node variable is a valid `Node`):

```
for (c = node.getFirstChild( ); c != null; c = c.getNextSibling( )) {
    System.out.println(c.getLocalName( ));
}
```

Note that there is no `getNextChild()` method, and you can't iterate through child nodes except via the `getNextSibling()` method. As a result, if you use the `removeChild()` method to remove one of a node's children, calls to the child node's `getNextSibling()` method immediately return `null`.

7.4.2.1. Element attributes

Element attributes are accessed via the `getAttributes()` method, which returns a `NamedNodeMap` object. The `NamedNodeMap` contains a set of `Node` objects of type `ATTRIBUTE_NODE`. The `getNodeValue()` method can read the value of a particular attribute.

```
NamedNodeMap atts = elementNode.getAttributes( );
if(atts != null) {
    Node sizeNode = atts.getNamedItem("size");
    String size = sizeNode.getValue( );
}
```

Alternately, you can cast a node to its true type (in this case, an `org.w3c.dom.Element`) and retrieve attributes or other data more directly:

```
if(myNode.getNodeType( ) == Node.ELEMENT_NODE) {
    Element myElement = (org.w3c.dom.Element)myNode;
    String attributeValue = myElement.getAttribute("attr");
    // attributeValue will be an empty string if "attr" does not exist
}
```

This is often easier than retrieving attribute nodes from a `NamedNodeMap`.

7.4.3. Manipulating DOM Trees

DOM is particularly useful when you need to manipulate the structure of an XML file. [Example 7-2](#) is

an HTML document "condenser." It loads an HTML file (which must be well-formed XML, although not necessarily XHTML), iterates through the tree, and preserves only the important content. In this case, it's text within ``, `<th>`, `<title>`, ``, and `<h1>` through `<h6>`. All text nodes that aren't contained within one of these tags are removed. A more sophisticated algorithm is no doubt possible, but this one is good enough to demonstrate the DOM principle.

Example 7-2. DocumentCondenser

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

public class DocumentCondenser {
    public static void main(String[] args) throws Exception {

        DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance( );

        // For HTML, we don't want to validate without a DTD
        dbf.setValidating(false);
        // Ignore text elements that are completely empty:
        dbf.setIgnoringElementContentWhitespace(false);
        dbf.setExpandEntityReferences(true);
        dbf.setCoalescing(true);

        // Ensure that getLocalName() returns the HTML element name
        dbf.setNamespaceAware(true);

        DocumentBuilder db = null;
        try {
            db = dbf.newDocumentBuilder( );
        }
        catch (ParserConfigurationException pce) {
            pce.printStackTrace();
            return;
        }

        Document html = null;
        try {
            html = db.parse("enterprisexml.html");
            process(html);

            // Use the XSLT Transformer to see the output
            TransformerFactory tf = TransformerFactory.newInstance();
            Transformer output = tf.newTransformer();
            output.transform(new DOMSource(html), new StreamResult(System.out));
        }
        catch (Exception ex) {
```

```

        ex.printStackTrace();
        return;
    }
}

/* We want to keep text if the parent is <em>, <title>, <b>, <li>, <th>
or <h1>..<h6>. We also want to keep text if it is in a <font> tag with
a size attribute set to a larger than normal size */
private static boolean keepText(Node parentNode) {
    if (parentNode == null) return true; // top level

    String parentName = parentNode.getLocalName();
    if ((parentName.equalsIgnoreCase("em")) ||
        (parentName.equalsIgnoreCase("title")) ||
        (parentName.equalsIgnoreCase("b")) ||
        (parentName.equalsIgnoreCase("li")) ||
        (parentName.equalsIgnoreCase("th")) ||
        ((parentName.toLowerCase().startsWith("h")) &&
         (parentName.length() == 2))) {
        return true;
    }

    if ((parentNode.getNodeType() == Node.ELEMENT_NODE) &&
        (parentName.equalsIgnoreCase("font"))) {
        NamedNodeMap atts = parentNode.getAttributes();
        if (atts != null) {
            Node sizeNode = atts.getNamedItem("size"); //get an attribute Node
            if (sizeNode != null) {
                if (sizeNode.getNodeValue().startsWith("+")) {
                    return true;
                }
            }
        }
    }

    return false;
}

private static void process(Node node) {

    Node c = null;
    Node delNode = null;

    for (c = node.getFirstChild(); c != null; c = c.getNextSibling()) {
        if (delNode != null) {
            delNode.getParentNode().removeChild(delNode);
        }
        delNode = null;
        if ((c.getNodeType() == Node.TEXT_NODE) &&
            (!keepText(c.getParentNode()))) {
            delNode = c;
        }
    }
}

```

```

        else if (c.getNodeType() != Node.TEXT_NODE) {
            process(c);
        }
    } // End For

    if (delNode != null) // Delete, if the last child was text
        delNode.getParentNode().removeChild(delNode);
}
}

```

After the DOM tree has been processed, use the JAXP XSLT API to output new HTML. We will discuss how to use XSL with JAXP in the next section.

If you want to replace the text with a condensed version, call the `setNodeValue()` method of `Node` when processing a text node.

7.4.4. Extending DOM Trees

Manipulating DOM trees falls, broadly, into three categories. We can add, remove, and modify nodes on existing trees; we can create new trees; and finally, we can merge trees together.

Back in the last example, we saw how to delete nodes from a DOM tree with the `removeChild()` method. If we want to add new nodes, we have two options. While there is no direct way to instantiate a new `Node` object, we can copy an existing `Node` using its `cloneNode()` method. The `cloneNode()` method takes a single `Boolean` parameter, which specifies whether the node's children will be cloned as well:

```

Node newNodeWithChildren = oldElementNode.cloneNode(true);
Node childlessNode = oldElementNode.cloneNode(false);

```

Regardless of whether children are cloned, clones of an element node will include all of the attributes of the parent node. The DOM specification leaves certain cloning behaviors, specifically `Document`, `DocumentType`, `Entity` and `Notation` nodes, up to the implementation.

New nodes can also be created via the `createXXX()` methods of the `Document` object. The `createElement()` method accepts a `String` containing the new element name and returns a new element `Node`. The `createElementNS()` method does the same thing, but accepts two parameters, a namespace and an element name. The `createAttribute()` method also has a version that is namespace-aware, `createAttributeNS()`.

Once a new `Node` is created, it can be inserted or appended into the tree using the `appendChild()`, `insertBefore()`, and `replaceChild()` methods. Attribute nodes can be inserted into the `NamedNodeMap` returned by the `getAttributes()` method of `Node`. You can also add attributes to an element by casting the `Node` to an `Element` and calling `setAttribute()`.

Creating new trees involves creating a new `Document` object. The easiest way to do this is via the `DOMImplementation` interface. An implementation of `DOMImplementation` can be retrieved from a

DocumentBuilder object via the `getDOMImplementation()` method. [Example 7-3](#) builds a version of the XML from a blank slate.

Example 7-3. TreeBuilder

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

public class TreeBuilder {

    public static void main(String[] args) {
        DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
        dbf.setValidating(false);

        DocumentBuilder db = null;
        try {
            db = dbf.newDocumentBuilder();
        }
        catch (ParserConfigurationException pce) {
            pce.printStackTrace();
            return;
        }

        Document doc =
            db.getDOMImplementation().createDocument(null, "orders", null);
        // create the initial document element
        Element orderNode = doc.createElement("order");
        orderNode.setAttribute("orderno", "123433");

        Node item = doc.createElement("item");
        Node subitem = doc.createElement("number");
        subitem.appendChild(doc.createTextNode("3AGM-5"));
        item.appendChild(subitem);

        subitem = doc.createElement("handling");
        subitem.appendChild(doc.createTextNode("With Care"));
        item.appendChild(subitem);

        orderNode.appendChild(item);
        doc.getDocumentElement().appendChild(orderNode);

        // View the output
        try {
            TransformerFactory tf = TransformerFactory.newInstance();
            Transformer output = tf.newTransformer();
            output.transform(new DOMSource(doc),
```

```

        new StreamResult(System.out));
    }
    catch (TransformerException e) {
        e.printStackTrace();
    }
}
}

```

The second parameter to `createDocument()` specifies the name of the base document element in this case, the `<orders>` tag. Subsequent tags can be appended to the base tag. If we were to look at the results of this program as regular XML, it would look like this (we've added some whitespace formatting to make it more readable):

```

<?xml version="1.0" encoding="UTF-8"?>
<orders>
  <order orderno="123433">
    <item>
      <number>3AGM-5</number>
      <handling>With Care</handling>
    </item>
  </order>
</orders>

```

You've probably noticed that each `Node` implementation we've created has been based on a particular instance of the `Document` object. Since each node is related to its parent document, we can't go around inserting one document's nodes into another document without triggering an exception. The solution is to use the `importNode()` method of `Document`, which creates a copy of a node from another document. The original node from the source document is left untouched. Here's an example that takes the `<orders>` tag from the first document and puts it into a new document under an `<ordersummary>` tag:

```

Document doc2 = db.getDOMImplementation( ).createDocument(
    null, "ordersummary", null);

DocumentFragment df = doc.createDocumentFragment( );
df.appendChild(doc.getDocumentElement( ).cloneNode(true));
doc2.getDocumentElement( ).appendChild(doc2.importNode(df, true));

```

We use a `DocumentFragment` object to hold the data we're moving. Document fragment nodes provide a lightweight structure for dealing with subsets of a document. Fragments must be valid XML, but don't need to be DTD-conformant and can have multiple top-level children. When appending a document fragment to a document tree, the `DocumentFragment` node itself is ignored, and its children are appended directly to the parent node. In this example, we cloned the source element when creating the document fragment, since assigning a node to a fragment releases the node's relationship with its previous parent. The XML in the second document object looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<ordersummary>
  <orders>
    <order orderno="123433">
      <item><number>3AGM-5</number>
      <handling>With Care</handling></item>
    </order>
  </orders>
</ordersummary>
```



7.5. XSLT

Another specification incorporated into the JAXP API is the XML Stylesheet Transformation (XSLT) system. An XSLT transformation takes an input XML document and transforms it into an output format (not necessarily XML) according to a set of rules specified in an XSL stylesheet. One common application of XSL stylesheets is transforming XML into HTML for presentation in a browser; this is often done in the browser directly or on the web server via a content management system such as the Apache project's Cocoon (<http://cocoon.apache.org>).

The following XSL document converts the *orders.xml* file from the SAX example into HTML:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/
Transform" xmlns:fo="http://www.w3.org/1999/XSL/Format">
  <xsl:template match="orders">
    <html>
      <head><title>Order Summary</title></head>
      <body>
        <xsl:apply-templates/>
      </body>
    </html>
  </xsl:template>
  <xsl:template match="order">
    <h1>Order Number <xsl:value-of select="@idnumber"/></h1>
    Ship To:
    <pre>
      <xsl:value-of select="shippingaddr"/>
    </pre>
    <ul>
      <xsl:apply-templates select="item"/>
    </ul>
  </xsl:template>
  <xsl:template match="item">
    <li><xsl:value-of select="@quantity"/> of item
      <xsl:value-of select="@idnumber"/><xsl:apply-templates/></li>
  </xsl:template>
  <xsl:template match="handling">
    <br/>Special Instructions: <xsl:value-of select="."/>
  </xsl:template>
</xsl:stylesheet>
```

The XSL file consists of a series of *templates*. The XSL processor matches each template to a particular XML tag and replaces the tag with the template contents. For example, when the processor encounters an `<orders>` and an `</orders>` tag, it replaces them with:

```
<html>
  <head><title>Order Summary</title></head>
  <body>
    <xsl:apply-templates/>
  </body>
</html>
```

The `<xsl:apply-templates/>` command tells the XSLT processor to recursively apply all of the available templates to the XML content contained within the `<orders>` tag pair. Other XSL commands allow the processor to display element attributes, limit recursion to specific templates, and so forth. For more examples, please check one of the books recommended at the beginning of this chapter. After running the `orders.xml` file through the stylesheet, we get this HTML:

```
<html xmlns:fo="http://www.w3.org/1999/XSL/Format">
<head>
<META http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Order Summary</title>
</head>
<body>

<h1>Order Number 3123</h1>
  Ship To:
  <pre>One Main St
Boston, MA 02112</pre>
<ul>
<li>13 of item 7231</li>
<li>2 of item 1296
  <br>Special Instructions: Please embroider in a tasteful manner!
  </li>
</ul>
<!-- Snipped for brevity -->
</body>
</html>
```

The classes in the JAXP `javax.xml.transform` package allow programs to use XSLT Transformer objects, such as Apache Xalan (which ships with the JAXP reference implementation) to transform XML documents.

At this point, we should point out that JAXP doesn't require XSLT at all: instead, it is a generic transformations API that theoretically can be used with a variety of transformation systems. However, XSLT is by far the most popular and widely implemented, so the remainder of this section (and most books on the subject) will assume that XSLT is used as the transformation system.

7.5.1. JAXP Data Sources

The DOM example earlier in this chapter briefly introduced the `transformerFactory` class. `transformerFactory` works just like the SAX and DOM parser factories, but returns a `transformer` object instead of a parser. The default behavior of a `transformer` is to pass the input XML through without modification, which is what will occur if no transformation stylesheet is given.

Input and output from a `transformer` are handled via the `javax.xml.transform.Source` and `javax.xml.transform.Result` interfaces. Each has three implementing classes, one each for DOM, SAX, and streams (`DOMSource`, `SAXSource`, `StreamSource`, `DOMResult`, `SAXResult`, and `StreamResult`). Note that the JAXP processors will not necessarily support all six. We handled the output in the DOM example by creating a `DOMSource` and outputting to a `StreamResult` that targeted `System.out` (in this example, the document variable contains a `DOMDocument` object):

```
TransformerFactory tf = TransformerFactory.newInstance( );
Transformer output = tf.newTransformer( );
output.transform(new DOMSource(document), new StreamResult(System.out));
```

The `Source` objects can also specify the source of the stylesheet used for the conversion. If we want to use a stylesheet located in `/home/will/orderdisplay.xsl`.

```
TransformerFactory tf = TransformerFactory.newInstance( );
Transformer output = tf.newTransformer(
    new StreamSource("file://home/will/orderdisplay.xsl"));
output.transform(new DOMSource(document), new StreamResult(System.out));
```

The different source and result types can streamline processing. Rather than load an XML file, transform it, write it to disk, reload it, and parse it with a `SAXContentHandler`, the transformation process can feed its results directly into a `ContentHandler` that can deal with the transformation results.

[Example 7-4](#) shows transformation of a document and its immediate processing by a `SAXContentHandler`. We'll use the `OrderHandler` program from earlier in the chapter as the content handler, and we'll use the same `orders.xml` file. To keep things simple, we'll do a one-to-one transformation, rather than use an XSLT file to actually alter the structure of the `orders.xml` file.

Example 7-4. Transforming a document into a SAXResult

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

public class SAXTransformTarget {

    public static void main(String[] args) {
        try {
```

```

        StreamSource ss = new StreamSource("orders.xml");
        SAXResult sr = new SAXResult(new OrderHandler( ));
        TransformerFactory tf = TransformerFactory.newInstance( );
        Transformer t = tf.newTransformer( );
        t.transform(ss, sr);
    } catch (TransformerConfigurationException e) {
        e.printStackTrace( );
    } catch (TransformerException e) {
        e.printStackTrace( );
    }
}

```

The output from this program should be identical to the output from [Example 7-1](#); we've simply replaced the `XMLReader` with a transformation stream.

7.5.1.1. Determining data source support

The `transformerFactory` class includes a method named `getFeature()`, which takes a `String` and returns `TRue` if the feature identified by the `String` is supported by the processor. Each of the `Source` and `Result` implementations includes a `String` constant named `FEATURE`. So to determine whether the XSL processor supports DOM source:

```

TransformerFactory tf = TransformerFactory.newInstance( );
boolean supportsDOMSource = tf.getFeature(DOMSource.FEATURE);

```

7.5.1.2. Custom URI resolution

When processing XSL files, it is sometimes necessary to resolve relative URIs. Ordinarily, the parser will do the best possible job, but sometimes it is necessary to override this behavior (for instance, in a web content management system in which the document tree apparent to the content creator and client might not match the system structure or when XML output from a servlet is being transformed within the servlet). In these cases, the `setURIResolver(URIResolver)` method of `transformer` allows you to specify resolution behavior by implementing the `URIResolver` interface. The `resolve()` method of `URIResolver` must return a `Source` or `null`:

```

class XSLResolver implements URIResolver {
    public Source resolve(String href, String base)
        throws TransformerException {
        // Check for a null Base URI, and provider it if so
        if((base == null) || (base.equals("/servlet/")))
            base = "http://www.oreilly.com/catalog/jentnut3/";
        if(href == null)
            return null;
        return new StreamSource(base + href);
    }
}

```

```
}  
}
```

Finally, JAXP supports XSLT transformations that can convert between DOM, SAX, and streams; transform an XML document based on an XSL stylesheet; or both.



Chapter 8. JDBC

The JDBC^[*] API provides Java applications with midlevel access to most database systems, via the Structured Query Language (SQL). JDBC is a key enterprise API, as it's hard to imagine an enterprise application that doesn't use a database in some way.

[*] According to Sun, "JDBC" is not an abbreviation for "Java Database Connectivity."

In the first two editions of this book, we focused on the original JDBC 1.0 and 2.0 APIs. As of J2SE 1.4, the JDBC 3.0 API became part of the standard Java deployment, so we've reworked this chapter to focus on the JDBC 3.0 API, which includes a modest yet valuable set of new features. Some JDBC 3.0 features aren't yet widely implemented (or aren't supported by some underlying databases), so we'll call special attention to them.

Since JDBC has been around for so long (it even predates the J2EE abbreviation), the different versions can get a little confusing. A number of JDBC 2.0 features were found in the JDBC 2.0 Optional Package and distributed as part of the `javax.sql` package rather than regular JDBC's `java.sql` package. JDBC 3.0 has merged everything back into a single specification, where it will hopefully remain.

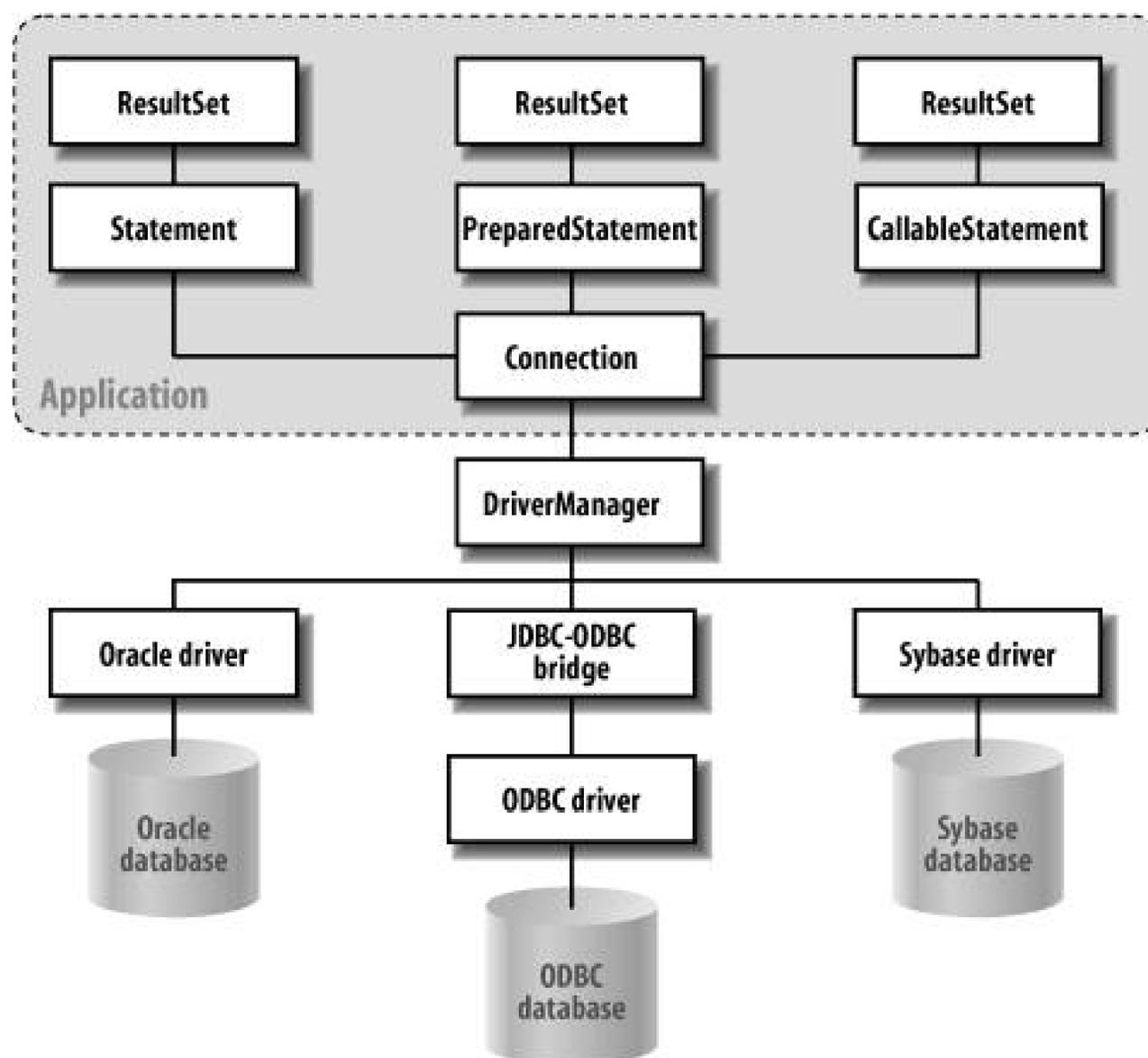
A word of caution: while the `java.sql` package is not tremendously complex, it does require grounding in general database concepts and the SQL language itself. This book includes a brief SQL reference (see [Appendix D](#)), but if you have never worked with a relational database system before, this chapter is not the place to start. For a more complete treatment of JDBC and general database concepts, we recommend *Database Programming with JDBC and Java* by George Reese (O'Reilly).

8.1. JDBC Architecture

Different database systems have surprisingly little in common: just a similar purpose and a mostly compatible query language. Beyond that, every database has its own API that you must learn in order to write programs that interact with the database. This has meant that writing code capable of interfacing with databases from more than one vendor has been a daunting challenge. Cross-database APIs exist, most notably Microsoft's ODBC API, but these tend to find themselves, at best, limited to a particular platform.

JDBC is Sun's attempt to create a platform-neutral interface between databases and Java. With JDBC, you can count on a standard set of database access features and (usually) a particular subset of SQL, SQL-92. The JDBC API defines a set of interfaces that encapsulates major database functionality, including running queries, processing results, and determining configuration information. A database vendor or third-party developer writes a JDBC *driver*, which is a set of classes that implements these interfaces for a particular database system. An application can use a number of drivers interchangeably. [Figure 8-1](#) shows how an application uses JDBC to interact with one or more databases without knowing about the underlying driver implementations.

Figure 8-1. JDBC-database interaction



8.1.1. JDBC Basics

Before we discuss all of the individual components of JDBC, let's look at a simple example that incorporates most of the major pieces of JDBC functionality. [Example 8-1](#) loads a driver, connects to the database, executes some SQL, and retrieves the results. It also keeps an eye out for any database-related errors.

Example 8-1. A simple JDBC example

```
import java.sql.*;

public class JDBCExample {

    public static void main(java.lang.String[] args) {
        try {
            // This is where we load the driver
            Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
        }
        catch (ClassNotFoundException e) {
```

```

        System.out.println("Unable to load Driver Class");
        return;
    }

    try {
        // All database access is within a try/catch block. Connect to database,
        // specifying particular database, username, and password.
        Connection con = DriverManager.getConnection("jdbc:odbc:companydb",
            "", "");

        // Create and execute a SQL statement
        Statement stmt = con.createStatement( );
        ResultSet rs = stmt.executeQuery("SELECT FIRST_NAME FROM EMPLOYEES");

        // Display the SQL results
        while(rs.next( )) {
            System.out.println(rs.getString("FIRST_NAME"));
        }

        // Make sure our database resources are released
        rs.close( );
        stmt.close( );
        con.close( );

    }
    catch (SQLException se) {
        // Inform user of any SQL errors
        System.out.println("SQL Exception: " + se.getMessage( ));
        se.printStackTrace(System.out);
    }
}

```

[Example 8-1](#) starts out by loading a JDBC driver class (in this case, Sun's JDBC-ODBC bridge). Then it creates a database connection, represented by a `Connection` object, using that driver. With the database connection, we can create a `Statement` object to represent a SQL statement. Executing a SQL statement produces a `ResultSet` that contains the results of a query. The program displays the results and then cleans up the resources it has used. If an error occurs, a `SQLException` is thrown, so our program traps that exception and displays some of the information it encapsulates.

Clearly, there is a lot going on in this simple program. Every Java application that uses JDBC follows these basic steps, so the following sections discuss each step in much more detail.

8.1.2. JDBC Drivers

Before you can use a driver, it must be registered with the `JDBC DriverManager`. This is typically done by loading the driver class using the `Class.forName()` method:

```

try {
    Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
    Class.forName("com.oracle.jdbc.OracleDriver");
}
catch (ClassNotFoundException e) {
    /* Handle Exception */
}

```

One reason most programs call `Class.forName()` is that this method accepts a `String` argument, meaning that the program can store driver selection information dynamically (e.g., in a properties file).

Drivers can be loaded by a J2EE server and connections made using these drivers can be provided to the application via JNDI. We'll see more about that toward the end of this chapter. For most applications, server-managed connections have replaced the use of `DriverManager`.

JDBC drivers are available for most database platforms, from a number of vendors and in a number of different flavors. There are four categories of drivers:

Type 1: JDBC-ODBC bridge drivers

Type 1 drivers use a bridge technology to connect a Java client to an ODBC database system. Type 1 drivers require some sort of non-Java software to be installed on the machine running your code, and they are implemented using native code.

Type 2: Native-API partly Java drivers

Type 2 drivers use a native code library to access a database, wrapping a thin layer of Java around the native library. For example, with Oracle databases, the native access might be through the Oracle Call Interface (OCI) libraries that were originally designed for C/C++ programmers. Type 2 drivers are implemented with native code, so they may perform better than all-Java drivers, but they also add an element of risk, as a defect in the native code can crash the Java virtual machine.

Type 3: Net-protocol all-Java drivers

Type 3 drivers define a generic network protocol that interfaces with a piece of custom middleware. The middleware component might use any other type of driver to provide the actual database access. BEA's WebLogic product line (formerly known as WebLogic Tengah and before that as `jdbckona/T3`) is an example. These drivers are especially useful for applet deployment since the actual JDBC classes can be written entirely in Java and downloaded by the client on the fly.

Type 4: Native-protocol all-Java drivers

Type 4 drivers are written entirely in Java. They understand database-specific networking protocols and can access the database directly without any additional software. These drivers

are also well-suited for applet programming, provided that the Java security manager allows TCP/IP connections to the database server.

When you are selecting a driver, you need to balance speed, reliability, and portability. Different applications have different needs. A standalone, GUI-intensive program that always runs on a Windows NT system will benefit from the additional speed of a Type 2, native-code driver. An applet might need to use a Type 3 driver to get around a firewall. A servlet that is deployed across multiple platforms might require the flexibility of a Type 4 driver.

A list of JDBC drivers is available at <http://developers.sun.com/product/jdbc/drivers>.

8.1.3. JDBC URLs

A JDBC driver uses a JDBC URL to identify and connect to a particular database. These URLs are generally of the form:

```
jdbc:driver:databasename
```

The actual standard is quite fluid, however, as different databases require different information to connect successfully. For example, the Oracle JDBC-Thin driver uses a URL of the form:

```
jdbc:oracle:thin:@site:port:database
```

while the JDBC-ODBC bridge uses:

```
jdbc:odbc:datasource;odbcoptions
```

The only requirement is that a driver be able to recognize its own URLs.

8.1.4. The JDBC-ODBC Bridge

The JDBC-ODBC bridge ships with Sun's JDK. The bridge provides an interface between JDBC and database drivers written using Microsoft's Open Database Connectivity (ODBC) API. The bridge was originally written to allow the developer community to get up and running quickly with JDBC. Since the bridge makes extensive use of native method calls, it is not recommended for long-term or high-volume deployment. The bridge also doesn't support JDBC 3.0 functionality (the driver bundled with JDK 1.4 and Java 5.0 supports JDBC 2.0).

The bridge is not a required component of the JDK, so it is not supported by most web browsers or other runtime environments. Using the bridge in an applet requires a browser with a JVM that supports the JDBC-ODBC bridge, as well as a properly configured ODBC driver and data source on

the client side.^[*]

[*] Finally, due to different implementations of the native methods interface, the bridge doesn't work with some development environments, most notably Microsoft Visual J++. This was a big deal when VJ++ was introduced, but since the product was discontinued years ago, you're likely to encounter this only when maintaining legacy code.

The JDBC URL subprotocol *odbc* has been reserved for the bridge. Like most JDBC URLs, it allows programs to encode extra information about the connection. ODBC URLs are of the form:

```
jdbc:odbc:datasourcename[;attribute-name=attribute-value]*
```

For instance, a JDBC URL pointing to an ODBC data source named *companydb* with the *CacheSize* attribute set to 10 looks like this:

```
jdbc:odbc:companydb;CacheSize=10
```



8.2. Connecting to the Database

The `java.sql.Connection` object, which encapsulates a single connection to a particular database, forms the basis of all JDBC data-handling code. An application can maintain multiple connections up to the limits imposed by the database system itself. A standard small office or web server Oracle installation can support 200 or so connections, while a major corporate database could host several thousand. The `DriverManager.getConnection()` method creates a connection:

```
Connection con = DriverManager.getConnection("url", "user", "password");
```

You pass three arguments to `getConnection()`: a JDBC URL, a database username, and a password. For databases that don't require explicit logins, the user and password strings should be left blank. When the method is called, the `DriverManager` queries each registered driver, asking if it understands the URL. If a driver recognizes the URL, it returns a `Connection` object. Because the `getConnection()` method checks each driver in turn, you should avoid loading more drivers than are necessary for your application.

The `getConnection()` method has two other variants that are less frequently used. One variant takes a single `String` argument and tries to create a connection to that JDBC URL without a username or password or with a username and password embedded in the URL itself. The other version takes a JDBC URL and a `java.util.Properties` object that contains a set of name/value pairs. You generally need to provide at least `username=value` and `password=value` pairs.

When a `Connection` has outlived its usefulness, you should be sure to explicitly close it by calling its `close()` method. This frees up any memory being used by the object, and, more importantly, it releases any other database resources the connection may be holding on to. These resources (cursors, handles, and so on) can be much more valuable than a few bytes of memory, as they are often quite limited. This is particularly important in applications such as servlets that might need to create and destroy thousands of JDBC connections between restarts. Because of the way some JDBC drivers are designed, it is not safe to rely on Java's garbage collection to remove unneeded JDBC connections.

8.2.1. DataSource Objects

The `DataSource` interface provides an alternative to the `DriverManager` class and conventional JDBC URLs. Instead, information about a database is stored within a naming service and retrieved via the JNDI API. Connection information (drivers, server locations, and so forth) is stored within the `DataSource` object, which uses it to create the actual `Connection` object used to execute JDBC commands. `DataSource` objects are also used to provide native driver-level support for connection pooling and distributed transactions.

Each `DataSource` is assigned a logical name, by convention beginning with `jdbc/`. The logical name

and associated connection metadata are configured in the J2EE setup process. This makes code more portable and allows for easy changes in drivers and connection information. Accessing a `DataSource` via JNDI is very simple:

```
Context ctx = new InitialContext( );
DataSource ds = (DataSource)ctx.lookup("jdbc/CamelDB");
Connection con = ds.getConnection("lawrence", "arabia");
```

The first two lines obtain the `DataSource` object from the naming service. The `getConnection()` method of `DataSource` then logs into the database and returns a `Connection` object. Unlike `DriverManager`, the only information required is a username and password.

8.2.2. Connection Pooling

Generally `DriverManager` provides the easiest way to retrieve a single database connection for a small application, but virtually every enterprise application, including J2EE and EJB systems, uses a `DataSource` instead for two main reasons. First, the `DataSource` interface is simpler to use. Second, `DataSource` objects also support *connection pooling*, which lets an application maintain several open database connections and spread the load among them. This is often necessary for enterprise-level applications, such as servlets, that may be called upon to perform tens of thousands of database transactions a day.

Connection pooling is implemented behind the scenes, either by an application server or a pooling utility such as Jakarta DBCP (<http://jakarta.apache.org/commons/dbcp>). This means you can write your application to simply rely on a `DataSource` object rather than worrying about the semantics of connection pooling.

When using a connection pool, the underlying driver creates a set of connections and distributes the connections to programs as needed (often on a per-thread basis). Once a connection has been used, it is returned to the pool to be reused later. This eliminates the substantial overhead of creating a new `Connection` for each request (a delay that can often be measured in seconds). For applications that make intensive use of database connections over extended periods, such as Java servlets (see [Chapter 3](#)), this added efficiency can be vital.

Like regular `DataSources`, connection pools are configured by the J2EE server administrator and must be supported by the database driver itself. Developers need only to remember to explicitly close all `Connection` objects after use, which is good programming practice anyway. The best way to handle this is with a `try/catch/finally` block:

```
Connection con = null;
try {
    ds = (DataSource)context.lookup("jdbc/oasisDB");
    con = ds.getConnection("larryl", "camel");
    // ... some worthwhile action
} catch (Exception e) {
} finally {
    if(con != null)
        con.close( );
}
```

}

← PREV

8.3. Statements

Once you have created a `Connection`, you can begin using it to execute SQL statements. This is usually done via `Statement` objects. There are actually three kinds of statements in JDBC:

`Statement`

Represents a basic SQL statement

`PreparedStatement`

Represents a precompiled SQL statement, which can offer improved performance

`CallableStatement`

Allows JDBC programs complete access to stored procedures within the database itself

We're going to discuss just the `Statement` object for now; `PreparedStatement` and `CallableStatement` are covered in detail later in this chapter.

To get a `Statement` object, call the `createStatement()` method of a `Connection`:

```
Statement stmt = con.createStatement( );
```

Once you have created a `Statement`, use it to execute SQL statements. A statement can be either a query that returns results or an operation that manipulates the database in some way. If you are performing a query, use the `executeQuery()` method of the `Statement` object:

```
ResultSet rs = stmt.executeQuery("SELECT * FROM CUSTOMERS");
```

Here we've used `executeQuery()` to run a `SELECT` statement. This call returns a `ResultSet` object that contains the results of the query (we'll take a closer look at `ResultSet` in the next section).

`Statement` also provides an `executeUpdate()` method for running SQL statements that don't return results, such as the `UPDATE` and `DELETE` statements. `executeUpdate()` returns an integer that indicates the number of rows in the database that were altered.

If you don't know whether a SQL statement is going to return results (such as when the user is entering the statement in a form field), you can use the `execute()` method of `Statement`. This

method returns `TRUE` if there is a result associated with the statement. In this case, the `ResultSet` can be retrieved using the `getResultSet()` method and the number of updated rows can be retrieved using `getUpdateCount()`:

```
Statement unknownSQL = con.createStatement( );
if(unknownSQL.execute(sqlString)) {
    ResultSet rs = unknownSQL.getResultSet( );
    // Display the results
}
else {
    System.out.println("Rows updated: " + unknownSQL.getUpdateCount( ));
}
```

It is important to remember that a `Statement` object represents a single SQL statement. A call to `executeQuery()`, `executeUpdate()`, or `execute()` implicitly closes any active `ResultSet` associated with the `Statement`. In other words, you need to be sure you are done with the results from a query before you execute another query with the same `Statement` object. If your application needs to execute more than one simultaneous query, you need to use multiple `Statement` objects. As a general rule, calling the `close()` method of any JDBC object also closes any dependent objects, such as a `Statement` generated by a `Connection` or a `ResultSet` generated by a `Statement`, but well-written JDBC code closes everything explicitly.

8.3.1. Multiple Result Sets

It is possible to write a SQL statement that returns more than one `ResultSet` or update count (exact methods of doing so vary depending on the database). The `Statement` object supports this functionality via the `getMoreResults()` method. Calling this method implicitly closes any existing `ResultSet` and moves to the next set of results for the statement. `getMoreResults()` returns `true` if another `ResultSet` is available to be retrieved by `getResultSet()`. However, the method returns `false` if the next statement is an update, even if there is another set of results waiting farther down the line. To be sure you've processed all the results for a `Statement`, you need to check that `getMoreResults()` returns `false` and that `getUpdateCount()` returns `-1`.

We can modify the previous `execute()` example to handle multiple results:

```
Statement unknownSQL = con.createStatement( );
unknownSQL.execute(sqlString);
while (true) {
    rs = unknownSQL.getResultSet( );
    if(rs != null)
        // Display the results
    else
        // Process the update data

    // Advance and quit if done
    if((unknownSQL.getMoreResults( ) == false) &&
        (unknownSQL.getUpdateCount( ) == -1))
```

```
        break;  
    }
```

Statements that return multiple results are actually quite rare. They generally arise from stored procedures or SQL implementations that allow multiple statements to be executed in a batch. Under SyBase, for instance, multiple `SELECT` statements may be separated by newline characters (`\n`).



8.4. Results

When a SQL query executes, the results form a pseudotable that contains all rows that fit the query criteria. For instance, here's a textual representation of the results of the query string `'SELECT NAME, CUSTOMER_ID, PHONE FROM CUSTOMERS'`:

NAME	CUSTOMER_ID	PHONE
Jane Markham	1	781 555-1212
Louis Smith	2	617 555-1213
Woodrow Lang	3	508 555-7171
Dr. John Smith	4	(011) 42 323-1239

This kind of textual representation is not very useful for Java programs. Instead, JDBC uses the `java.sql.ResultSet` interface to encapsulate the query results as Java primitive types and objects. You can think of a `ResultSet` as an object that represents an underlying table of query results, where you use method calls to navigate between rows and retrieve particular column values.

A Java program might handle the previous query as follows:

```
Statement stmt = con.createStatement( );
ResultSet rs = stmt.executeQuery(
    "SELECT NAME, CUSTOMER_ID, PHONE FROM CUSTOMERS");

while(rs.next( )) {
    System.out.print("Customer #" + rs.getString("CUSTOMER_ID"));
    System.out.print(", " + rs.getString("NAME"));
    System.out.println(", is at " + rs.getString("PHONE"));
}
rs.close( );
stmt.close( );
```

Here's the resulting output:

```
Customer #1, Jane Markham, is at 781 555-1212
Customer #2, Louis Smith, is at 617 555-1213
Customer #3, Woodrow Lang, is at 508 555-7171
Customer #4, Dr. John Smith, is at (011) 42 323-1239
```

The code loops through each row of the `ResultSet` using the `next()` method. When you start

working with a `ResultSet`, you are positioned before the first row of results. That means you have to call `next()` once just to access the first row. Each time you call `next()`, you move to the next row. If there are no more rows to read, `next()` returns `false`. Note that with the JDBC 1.0 `ResultSet`, you can only move forward through the results, and since there is no way to go back to the beginning, you can read them only once. Updateable `ResultSet`s were introduced in JDBC 2.0, and we'll discuss them later.

Individual column values are read in our example using the `getString()` method. `getString()` is one of a family of `getXXX()` methods, each of which returns data of a particular type. Each `getXXX()` method has two versions: one that takes the case-insensitive `String` name of the column to be read (e.g., `"PHONE"`, `"CUSTOMER_ID"`) and one that takes a SQL-style column index. Note that column indexes run from 1 to n , unlike Java array indexes, which run from 0 to $n-1$, where n is the number of columns.

The most important `getXXX()` method is `getObject()`, which can return any kind of data packaged in an object wrapper. For example, calling `getObject()` on an integer field returns an `Integer` object, while calling it on a date field yields a `java.sql.Date` object. [Table 8-1](#) lists the different `getXXX()` methods, along with the corresponding SQL data type and Java data type. Where the return type for a `getXXX()` method is different from the Java type, the return type is shown in parentheses. Note that the `java.sql.Types` class defines integer constants that represent the standard SQL data types.

Table 8-1. SQL data types, Java types, and default `getXXX()` methods

SQL data type	Java type	<code>getXXX()</code> method
CHAR	String	<code>getString()</code>
VARCHAR	String	<code>getString()</code>
LONGVARCHAR	String	<code>getString()</code>
NUMERIC	<code>java.math.BigDecimal</code>	<code>getBigDecimal()</code>
DECIMAL	<code>java.math.BigDecimal</code>	<code>getBigDecimal()</code>
BIT	Boolean (boolean)	<code>getBoolean()</code>
TINYINT	Integer (byte)	<code>getByte()</code>
SMALLINT	Integer (short)	<code>getShort()</code>
INTEGER	Integer (int)	<code>getInt()</code>
BIGINT	Long (long)	<code>getLong()</code>
REAL	Float (float)	<code>getFloat()</code>
FLOAT	Double (double)	<code>getDouble()</code>
DOUBLE	Double (double)	<code>getDouble()</code>
BINARY	<code>byte[]</code>	<code>getBytes()</code>
VARBINARY	<code>byte[]</code>	<code>getBytes()</code>
LONGVARBINARY	<code>byte[]</code>	<code>getBytes()</code>

SQL data type	Java type	getXXX() method
DATE	java.sql.Date	getDate()
TIME	java.sql.Time	getTime()
TIMESTAMP	java.sql.Timestamp	getTimestamp()
BLOB	java.sql.Blob	getBlob()
CLOB	java.sql.Clob	getClob()

Note that this table merely lists the default mappings according to the JDBC specification, and some drivers don't follow these mappings exactly. Also, a certain amount of casting is permitted. For instance, the `getString()` method returns a `String` representation of just about any data type.

8.4.1. Handling Nulls

Sometimes database columns contain `null`, or empty, values. However, because of the way certain database APIs are written, it is impossible for JDBC to provide a method to determine before the fact whether or not a column is `null`.^[*] Methods that don't return an object of some sort are especially vulnerable. `getInt()`, for instance, resorts to returning a value of `-1`. JDBC deals with this problem via the `wasNull()` method, which indicates whether or not the last column read was `null`:

[*] The driver can figure this out after reading the object, but since some driver implementations and database connection protocols allow you to reliably read a value from a column only once, implementing an `isNull()` method requires the `ResultSet` to cache the entire row in memory. While many programs do exactly this, it is not appropriate behavior for the lowest-level result handler.

```
int numberInStock = rs.getInt("STOCK");
if(rs.wasNull( ))
    System.out.println("Result was null");
else
    System.out.println("In Stock: " + numberInStock);
```

Alternately, you can call `getObject()` and test to see if the result is `null`:

```
Object numberInStock = rs.getObject("STOCK");
if(numberInStock == null)
    System.out.println("Result was null");
```

8.4.2. Large Data Types

You can retrieve large chunks of data from a `ResultSet` as a stream. This can be useful when reading images from a database or loading large documents from a data store, for example. The relevant `ResultSet` methods are `getAsciiStream()`, `getBinaryStream()`, and `getCharacterStream()`; each

method has column name and column index variants, just like the other `getXXX()` methods. Each of these methods returns an `InputStream`. Here's a code sample that retrieves an image from a `PICTURES` table and writes the image to an `OutputStream` of some kind (this might be a `ServletOutputStream` for a Java servlet that produces a GIF from a database):

```
ResultSet rs =
    stmt.executeQuery("SELECT IMAGE FROM PICTURES WHERE PID = " +
        req.getParameter("PID"));
if (rs.next( )) {
    BufferedInputStream gifData =
        new BufferedInputStream(rs.getBinaryStream("IMAGE"));
    byte[] buf = new byte[4 * 1024]; // 4K buffer
    int len;
    while ((len = gifData.read(buf, 0, buf.length)) != -1) {
        out.write(buf, 0, len);
    }
}
```

The JDBC 2.0 (and later) APIs include `Blob` and `Clob` objects to handle large data types; we discuss these objects later in this chapter.

8.4.3. Dates and Times

JDBC defines three classes devoted to storing date and time information: `java.sql.Date`, `java.sql.Time`, and `java.sql.Timestamp`. These correspond to the SQL `DATE`, `TIME`, and `TIMESTAMP` types. The `java.util.Date` class is not suitable for any of them, so JDBC defines a new set of wrapper classes that extends (or limits) the standard `Date` class to fit the JDBC mold.

The SQL `DATE` type contains only a date, so the `java.sql.Date` class contains only a day, month, and year. SQL `TIME` (`java.sql.Time`) includes only a time of day, without date information. SQL `TIMESTAMP` (`java.sql.Timestamp`) includes both, but at nanosecond precision (the standard `Date` class is incapable of handling more than milliseconds).

Since different DBMS packages have different methods of encoding date and time information, JDBC supports the ISO date escape sequences, and individual drivers must translate these sequences into whatever form the underlying DBMS requires. The syntax for dates, times, and timestamps is:

```
{d 'yyyy-mm-dd'}
{t 'hh:mm:ss'}
{ts 'yyyy-mm-dd hh:mm:ss.ms.microseconds.ns'}
```

A `TIMESTAMP` needs to be specified only up to seconds; the remaining values are optional. Here is an example that uses a date escape sequence (where `dateSQL` is a `Statement` of some sort):

```
dateSQL.execute("INSERT INTO FRIENDS(BIRTHDAY) VALUES ({d '1978-12-14'})");
```

8.4.4. Advanced Results Handling

With JDBC 1.0, the functionality provided by the `ResultSet` interface is rather limited. There is no support for updates of any kind, and access to rows is limited to a single, sequential read (i.e., first row, second row, third row, and so on with no going back). As of 2.0, JDBC supports scrollable and updateable result sets, which allows for advanced record navigation and in-place data manipulation.

With scrolling, you can move forward and backward through the results of a query rather than just using the `next()` method to move to the next row. In terms of scrolling, there are now three distinct types of `ResultSet` objects: forward-only (as in JDBC 1.0), scroll-insensitive, and scroll-sensitive. A scroll-insensitive result set generally doesn't reflect changes to the underlying data while scroll-sensitive ones do. In fact, the number of rows in a sensitive result set doesn't even need to be fixed.

As of JDBC 2.0, result sets are also updateable. From this perspective, there are two different kinds of result sets: read-only result sets that don't allow changes to the underlying data and updateable result sets that allow such changes, subject to transaction limitations and so on.

To create an updateable, scroll-sensitive result set, we pass two extra arguments to the `createStatement()` method;

```
Statement stmt = con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
    ResultSet.CONCUR_UPDATEABLE);
```

If you don't pass any arguments to `createStatement()`, you get a forward-only, read-only result set, just as you would using JDBC 1.0. Note that if you specify a scrollable result set (either sensitive or insensitive), you must also specify whether or not the result set is updateable. After you have created a scrollable `ResultSet`, use the methods listed in [Table 8-2](#) to navigate through it. As with JDBC 1.0, when you start working with a `ResultSet`, you are positioned before the first row of results.

Table 8-2. JDBC 2.0 (and later) record-scrolling functions

Method	Function
<code>first()</code>	Move to the first record.
<code>last()</code>	Move to the last record.
<code>next()</code>	Move to the next record.
<code>previous()</code>	Move to the previous record.
<code>beforeFirst()</code>	Move to immediately before the first record.
<code>afterLast()</code>	Move to immediately after the last record.
<code>absolute(int)</code>	Move to an absolute row number. Takes a positive or negative argument.

Method	Function
<code>relative(int)</code>	Move backward or forward a specified number of rows. Takes a positive or negative argument.

The JDBC 2.0 API also includes a number of methods that tell you where you are in a `ResultSet`. You can think of your position in a `ResultSet` as the location of a cursor in the results. The `isFirst()` and `isLast()` methods return `True` if the cursor is located on the first or last record, respectively. `isAfterLast()` returns `true` if the cursor is after the last row in the result set while `isBeforeFirst()` returns `true` if the cursor is before the first row.

With an updateable `ResultSet`, you can change data in an existing row, insert an entirely new row, or delete an existing row. To change data in an existing row, use the new `updateXXX()` methods of `ResultSet`. Let's assume we want to update the `CUSTOMER_ID` field of the first row we retrieve (OK, it's a contrived example, but bear with us):

```
Statement stmt = con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
                                     ResultSet.CONCUR_UPDATEABLE);
ResultSet rs = stmt.executeQuery(
    "SELECT NAME, CUSTOMER_ID FROM CUSTOMERS");

rs.first();
rs.updateInt(2, 35243);
rs.updateRow();
```

Here we use `first()` to navigate to the first row of the result set and then call `updateInt()` to change the value of the customer ID column in the result set. After making the change, call `updateRow()` to actually make the change in the database. If you forget to call `updateRow()` before moving to another row in the result set, any changes you made are lost. If you need to make a number of changes in a single row, do so with multiple calls to `updateXXX()` methods and then a single call to `updateRow()`. Just be sure you call `updateRow()` before moving on to another row.

The technique for inserting a row is similar to updating data in an existing row, with a few important differences. The first step is to move to what is called the insert row, using the `moveToInsertRow()` method. The *insert row* is a blank row associated with the `ResultSet` that contains all the fields, but no data; you can think of it as a pseudorow in which you can compose a new row. After you have moved to the insert row, use `updateXXX()` methods to load new data into the insert row and then call `insertRow()` to append the new row to the `ResultSet` and the underlying database. Here's an example that adds a new customer to the database:

```
ResultSet rs = stmt.executeQuery(
    "SELECT NAME, CUSTOMER_ID FROM CUSTOMERS");
rs.moveToInsertRow();
rs.updateString(1, "Tom Flynn");
rs.updateInt(2, 35244);
rs.insertRow();
```

Note that you don't have to supply a value for every column, as long as the columns you omit can accept `null` values. If you don't specify a value for a column that can't be `null`, you'll get a `SQLException`. After you call `insertRow()`, you can create another new row or you can move back to the `ResultSet` using the various navigation methods shown in [Table 8-2](#). One final navigation method that isn't listed in the table is `moveToCurrentRow()`. This method takes you back to where you were before you called `moveToInsertRow()`; it can be called only while you are in the insert row.

Deleting a row from an updateable result set is easy. Simply move to the row you want to delete and call the `deleteRow()` method. Here's how to delete the last record in a `ResultSet`:

```
rs.last( );
rs.deleteRow( );
```

Calling `deleteRow()` also deletes the row from the underlying database.

Note that not all `ResultSet` objects are updateable. In general, the query must reference only a single table without any joins. Due to differences in database implementations, there is no single set of requirements for what makes an updateable `ResultSet`.

As useful as scrollable and updateable result sets are, the JDBC specification doesn't require driver vendors to support them. If you are building middleware or some other kind of system that requires interaction with a wide range of database drivers, you should avoid this functionality for the time being. The extended `DatabaseMetaData` object, introduced in JDBC 2.0, can provide information about scrolling and concurrency support.

8.4.5. Java-Aware Databases

Java is object-oriented; relational databases aren't. As a result, it's decidedly difficult to shoehorn a Java object into a stubbornly primitive-oriented database table. Luckily, the wind is changing, and newer database systems, including object-oriented database management systems (OODBMSs) provide direct support for storing and manipulating objects, and in some cases Java code as well. While a regular relational database can store only a limited number of primitive types, a JDBC [§1](#) system can store entire, arbitrary Java objects.

[*] A number of these products also use Java as a trigger language, generally in a JDBC structure.

Say we want to store a customized Java `Account` object in the `ACCOUNTS` table in a database. With a standard DBMS and JDBC 1.0, we have to pull each piece of data (such as account number, account holder, and balance) out of the `Account` object and write it to a complicated database table. To get data out, we reverse the process. Short of serializing the `Account` object and writing it to a binary field (a rather complex operation), we're stuck with this clumsy approach.

JDBC 2.0 extended the `getObject()` method to support these new Java-aware databases. Provided that the database supports a Java-object type, we can read the `Account` object just like any primitive type:

```
ResultSet rs = stmt.executeQuery("SELECT ACCOUNT FROM ACCOUNTS");
```

```
rs.next( );  
Account a = (Account)rs.getObject(1);
```

To store an object, we use a `PreparedStatement` and the `setObject()` method:

```
Account a = new Account( );  
// Fill in appropriate fields in Account object  
  
PreparedStatement stmt = con.prepareStatement(  
    "INSERT INTO ACCOUNTS (ACCOUNT) VALUE (?)");  
stmt.setObject(1, a);  
stmt.executeUpdate( );
```

A column that stores a Java object has a type of `Types.JAVA_OBJECT`. The JDBC API doesn't take any special steps to locate the bytecodes associated with any particular class, so you should make sure that any necessary objects can be instantiated with a call to `Class.forName()`.

[← PREY](#)

8.5. Handling Errors

Any JDBC object that encounters an error serious enough to halt execution throws a `SQLException`. For example, database connection errors, malformed SQL statements, and insufficient database privileges all throw `SQLException` objects.

The `SQLException` class extends the normal `java.lang.Exception` class and defines an additional method called `getNextException()`. This allows JDBC classes to chain a series of `SQLException` objects together. `SQLException` also defines the `getSQLState()` and `getErrorCode()` methods to provide additional information about an error. The value returned by `getSQLState()` is one of the ANSI-92 SQL state codes; these codes are listed in [Appendix D](#). `getErrorCode()` returns a vendor-specific error code.

An extremely conscientious application might have a `catch` block that looks something like this:

```
try {
    // Actual database code
}
catch (SQLException e) {
    while(e != null) {
        System.out.println("\nSQL Exception:");
        System.out.println(e.getMessage( ));
        System.out.println("ANSI-92 SQL State: " + e.getSQLState( ));
        System.out.println("Vendor Error Code: " + e.getErrorCode( ));
        e = e.getNextException( );
    }
}
```

8.5.1. SQL Warnings

JDBC classes also have the option of generating (but not throwing) a `SQLWarning` exception when something is not quite right, but at the same time not sufficiently serious to warrant halting the entire program. For example, attempting to set a transaction isolation mode that is not supported by the underlying database might generate a warning rather than an exception. Remember, exactly what qualifies as a warning condition varies by database.

`SQLWarning` encapsulates the same information as `SQLException` and is used in a similar fashion. However, unlike `SQLException` objects, which are caught in `try/catch` blocks, warnings are retrieved using the `getWarnings()` methods of the `Connection`, `Statement`, `ResultSet`, `CallableStatement`, and `PreparedStatement` interfaces. `SQLWarning` implements the `getMessage()`, `getSQLState()`, and `getErrorCode()` methods in the same manner as `SQLException`.

If you are debugging an application and you want to be aware of every little thing that goes wrong within the database, you might use a `printWarnings()` method like this one:

```
void printWarnings(SQLWarning warn) {
    while (warn != null) {
        System.out.println("\nSQL Warning:");
        System.out.println(warn.getMessage( ));
        System.out.println("ANSI-92 SQL State: " + warn.getSQLState( ));
        System.out.println("Vendor Error Code: " + warn.getErrorCode( ));
        warn = warn.getNextWarning( );
    }
}
```

You can then use the `printWarnings()` method as follows:

```
// Database initialization code here
ResultSet rs = stmt.executeQuery("SELECT * FROM CUSTOMERS");
printWarnings(stmt.getWarnings( ));
printWarnings(rs.getWarnings( ));
// Rest of database code
```

8.6. Prepared Statements

The `PreparedStatement` object is a close relative of the `Statement` object. Both accomplish roughly the same thing: running SQL statements. `PreparedStatement`, however, allows you to precompile your SQL and run it repeatedly, adjusting specific parameters as necessary. Since processing SQL strings is a large part of a database's overhead, getting compilation out of the way at the start can significantly improve performance. With proper use, it can also simplify otherwise tedious database tasks.

As with `Statement`, you create a `PreparedStatement` object from a `Connection` object. In this case, though, the SQL is specified at creation instead of execution, using the `prepareStatement()` method of `Connection`:

```
PreparedStatement pstmt = con.prepareStatement(
    "INSERT INTO EMPLOYEES (NAME, PHONE) VALUES (?, ?)");
```

This SQL statement inserts a new row into the `EMPLOYEES` table, setting the `NAME` and `PHONE` columns to certain values. Since the whole point of a `PreparedStatement` is to be able to execute the statement repeatedly, we don't specify values in the call to `prepareStatement()`, but instead use question marks (?) to indicate parameters for the statement. To actually run the statement, we specify values for the parameters and then execute the statement:

```
pstmt.clearParameters( );
pstmt.setString(1, "Jimmy Adelphi");
pstmt.setString(2, "201 555-7823");
pstmt.executeUpdate( );
```

Before setting parameters, we clear out any previously specified parameters with the `clearParameters()` method. Then we can set the value for each parameter (indexed from 1 to the number of question marks) using the `setString()` method. `PreparedStatement` defines numerous `setXXX()` methods for specifying different types of parameters; see the `java.sql` reference material later in this book for a complete list. Finally, we use the `executeUpdate()` method to run the SQL.

The `setObject()` method can insert Java object types into the database, provided that those objects can be converted to standard SQL types. `setObject()` comes in three flavors:

```
setObject(int parameterIndex, Object x, int targetSqlType, int scale)
setObject(int parameterIndex, Object x, int targetSqlType)
setObject(int parameterIndex, Object x)
```

Calling `setObject()` with only a parameter index and an `Object` causes the method to try and automatically map the `Object` to a standard SQL type (see [Table 8-1](#)). Calling `setObject()` with a type specified allows you to control the mapping. The `setXXX()` methods work a little differently in that they attempt to map Java primitive types to JDBC types.

You can use `PreparedStatement` to insert `null` values into a database, either by calling the `setNull()` method or by passing a `null` value to one of the `setXXX()` methods that take an `Object`. In either case, you must specify the target SQL type.

Let's clarify with an example. We want to set the first parameter of a prepared statement to the value of an `Integer` object while the second parameter, which is a `VARCHAR`, should be `null`. Here's some code that does that:

```
Integer i = new Integer(32);
pstmt.setObject(1, i, Types.INTEGER);
pstmt.setObject(2, null, Types.VARCHAR);
// Or pstmt.setNull(2, Types.VARCHAR);
```

8.6.1. Batch Updates

The original JDBC standard was not very efficient at loading large amounts of information into a database. Even if you use a `PreparedStatement`, your program still executes a separate query for each piece of data inserted. If your software inserts 10,000 rows into the database, it can introduce a substantial performance bottleneck.

The `addBatch()` method of `Statement` allows you to lump multiple update statements into a unit and execute them all at once. Call `addBatch()` after you create the statement and before execution:

```
con.setAutoCommit(false); // If some fail, we want to roll back the rest
Statement stmt = con.createStatement( );

stmt.addBatch(
    "INSERT INTO CUSTOMERS VALUES (1, \"J Smith\", \"781 555-1323\");
stmt.addBatch(
    "INSERT INTO CUSTOMERS VALUES (2, \"A Smith\", \"617 555-1132\");
stmt.addBatch(
    "INSERT INTO CUSTOMERS VALUES (3, \"C Smith\", \"617 555-1238\");
stmt.addBatch(
    "INSERT INTO CUSTOMERS VALUES (4, \"K Smith\", \"617 555-7823\");

int[] upCounts = stmt.executeBatch( );
con.commit( );
```

Notice that we turn transaction autocommit off before creating the batch. This is because we want to roll back all the SQL statements if one or more of them fail to execute properly (for a more detailed discussion of transaction handling, see ["Transactions"](#) later in this chapter). After calling `addBatch()` multiple times to create our batch, we call `executeBatch()` to send the SQL statements off to the

database to be executed as a batch. Batch statements are executed in the order they are added to the batch. `executeBatch()` returns an array of update counts, in which each value in the array represents the number of rows affected by the corresponding batch statement. If you need to remove the statements from a pending batch job, you can call `clearBatch()`, as long as you call it before calling `executeBatch()`.

Note that you can use only SQL statements that return an update count (e.g., `CREATE`, `DROP`, `INSERT`, `UPDATE`, `DELETE`) as part of a batch. If you include a statement that returns a result set, such as `SELECT`, you get a `SQLException` when you execute the batch. If one of the statements in a batch can't be executed for some reason, `executeBatch()` throws a `BatchUpdateException`. This exception, derived from `SQLException`, contains an array of update counts for the batch statements that executed successfully before the exception was thrown. If we then call `rollback()`, the components of the batch transaction that did execute successfully will be rolled back.

The `addBatch()` method works slightly differently for `PreparedStatement` and `CallableStatement` objects. To use batch updating with a `PreparedStatement`, create the statement normally, set the input parameters, and then call the `addBatch()` method with no arguments. Repeat as necessary and then call `executeBatch()` when you're finished:

```
con.setAutoCommit(false); // If some fail, we want to roll back the rest
PreparedStatement stmt = con.prepareStatement(
    "INSERT INTO CUSTOMERS VALUES (?, ?, ?)");

stmt.setInt(1, 1);
stmt.setString(2, "J Smith");
stmt.setString(3, "781 555-1323");
stmt.addBatch( );

stmt.setInt(1, 2);
stmt.setString(2, "A Smith");
stmt.setString(3, "617 555-1132");
stmt.addBatch( );

int[] upCounts = stmt.executeBatch( );
con.commit( );
```

This batch functionality also works with `CallableStatement` objects for stored procedures. The catch is that each stored procedure must return an update count and may not take any `OUT` or `INOUT` parameters.

8.6.2. Auto-Generated Keys

Although it isn't part of the standard set of SQL capabilities, many databases, including the popular MySQL system, support autogenerated primary keys. This is a very useful capability, but in the past it's had a drawback: JDBC provided no way to retrieve the generated key values! So you were stuck running a second query to retrieve the new key or generating the key separately. But no more, as JDBC 3.0 provides methods for retrieving autogenerated key values, provided that the underlying database and driver support them.

The `execute()` and `executeUpdate()` methods of `Statement` now include additional versions that allow your program to register interest in generated key values by passing in the `Statement.RETURN_GENERATED_KEYS` constant. After executing the statement, you can call the `getGeneratedKeys()` method to return a `ResultSet` containing the generated keys. Keys are returned one to a row, so if your SQL statement generates multiple new rows, you'll get each key back, in order.

[Example 8-2](#) shows a simple program that inserts a single row into a table, retrieves the key generated by the table, and prints the result.

Example 8-2. KeyGenerator

```
import java.sql.*;

public class KeyGenerator {

    public static void main(String[] args) throws Exception {
        Class.forName("com.mysql.jdbc.Driver");
        Connection conn =
            DriverManager.getConnection("jdbc:mysql://localhost/test");

        int newKey = -1;
        Statement stmt = conn.createStatement( );
        stmt.executeUpdate(
            "insert into people (fullname) values ('John Morgan')",
            Statement.RETURN_GENERATED_KEYS);

        ResultSet keys = stmt.getGeneratedKeys( );
        if (keys.next( )) {
            newKey = keys.getInt(1);
        }
        keys.close( );

        System.out.println("New Key: " + newKey);

        stmt.close( );
        conn.close( );
    }
}
```

8.7. BLOBs and CLOBs

As users began to increase the volume of data stored in databases, vendors introduced support for large objects (LOBs). The two varieties of LOBs, binary large objects (BLOBs) and character large objects (CLOBs), store large amounts of binary or character data, respectively. The SQL99 specification formally defined these data types.

Support for LOB types across databases varies. Some don't support them at all, and most have unique type names (`BINARY`, `LONG RAW`, and so forth). JDBC 1.0 makes programs retrieve `BLOB` and `CLOB` data using the `getBinaryStream()` or `getAsciiStream()` methods. (A third method, `getUnicodeStream()`, has been deprecated in favor of the `getCharacterStream()` method, which returns a `Reader`.)

As of JDBC 2.0, the `ResultSet` interface includes `getBlob()` and `getClob()` methods, which return `Blob` and `Clob` objects, respectively. The `Blob` and `Clob` objects themselves allow access to their data via streams (the `getBinaryStream()` method of `Blob` and the `getCharacterStream()` method of `Clob`) or via direct-read methods (the `getBytes()` method of `Blob` and the `getSubString()` method of `Clob`).

To retrieve the data from a CLOB, simply retrieve the `Clob` object and call the `getCharacterStream()` method:

```
String s;
Clob clob = blobResultSet.getBlob("CLOBFIELD");
BufferedReader clobData = new BufferedReader(clob.getCharacterStream( ));
while((s = clobData.readLine( )) != null)
    System.out.println(s);
```

In addition, you can set `Blob` and `Clob` objects when you are working with a `PreparedStatement`, using the `setBlob()` and `setClob()` methods. Note that the life span of a `Blob` or `Clob` object is limited to the transaction that created it, so if you've retrieved one from a `ResultSet`, you need to keep that `ResultSet` open while you're working with it.

JDBC 3.0 extended the `java.sql.Blob` interface to include a `setBytes()` method to alter the BLOB's content, and the `java.sql.Clob` class includes a `setString()` method. When using these methods, it's up to the driver whether to update a local copy of the LOB or to directly update the copy in the database. The `locatorsUpdateCopy()` method of `DatabaseMetaData` will tell you which approach is used by your driver. If the method returns true, you'll need to issue a separate update statement to commit the changes to the LOB back to the database. To insert data in a new BLOB or CLOB field, first create a row with an empty LOB, select the row, retrieve the LOB, edit its contents, and write it back if necessary.

Modified `Blob` and `Clob` objects can be passed to the `setBlob()` and `setClob()` methods of `PreparedStatement` and to the new `updateBlob()` and `updateClob()` methods of `ResultSet` (for

updateable result sets).

JDBC driver support for BLOB and CLOB types varies wildly. Some vendors don't support any LOB functionality at all, and others support only JDBC 2.0-level (read-only) LOBs or provide their own extensions for manipulating them. Check your driver documentation for more details.





8.8. Metadata

Most JDBC programs are designed to work with a specific database and particular tables in that database; the program knows exactly what kind of data it is dealing with. Some applications, however, need to dynamically discover information about result set structures or underlying database configurations. This information is called *metadata*, and JDBC provides two classes for dealing with it: `DatabaseMetaData` and `ResultSetMetaData`. If you are developing a JDBC application that will be deployed outside a known environment, you need to be familiar with these interfaces.

8.8.1. DatabaseMetaData

You can retrieve general information about the structure of a database with the `java.sql.DatabaseMetaData` interface. By making thorough use of this class, a program can tailor its SQL and use of JDBC on the fly to accommodate different levels of database and JDBC driver support.

Database metadata is associated with a particular connection, so `DatabaseMetaData` objects are created with the `getMetaData()` method of `Connection`:

```
DatabaseMetaData dbmeta = con.getMetaData( );
```

`DatabaseMetaData` provides an overwhelming number of methods you can call to get actual configuration information about the database. Some of these return `String` objects (`getURL()`), some return `boolean` values (`nullsAreSortedHigh()`), and still others return integers (`getMaxConnections()`).

A number of other methods return `ResultSet` objects. These methods, such as `getColumns()`, `getTableTypes()`, and `getPrivileges()`, generally encapsulate complex or variable-length information. The `getTables()` method, for instance, returns a `ResultSet` that contains the name of every table in the database as well as a good deal of extra information.

Many of the `DatabaseMetaData` methods take string patterns as arguments, allowing for simple wildcard searching. A percent sign (%) substitutes for any number of characters, and an underscore (_) calls for a single character match. Thus, `%CUSTOMER%` matches `NEW_CUSTOMERS`, `CUSTOMER`, and `CUSTOMERS`, while `CUSTOMER%` matches only `CUSTOMER` and `CUSTOMERS`. All of these patterns are case-sensitive.

8.8.2. ResultSetMetaData

The `ResultSetMetaData` interface provides information about the structure of a particular `ResultSet`.

Data provided by `ResultSetMetaData` includes the number of available columns, the names of those columns, and the kind of data available in each. [Example 8-3](#) is a short program that displays the contents of a table and shows the data type for each column.

Example 8-3. TableViewer program

```
import java.sql.*;

public class TableViewer {

    final static String jdbcURL = "jdbc:oracle:customerdb";
    final static String jdbcDriver = "oracle.jdbc.driver.OracleDriver";
    final static String table = "CUSTOMERS";

    public static void main(java.lang.String[] args) {

        System.out.println("--- Table Viewer ---");

        try {
            Class.forName(jdbcDriver);
            Connection con = DriverManager.getConnection(jdbcURL, "", "");
            Statement stmt = con.createStatement( );
            ResultSet rs = stmt.executeQuery("SELECT * FROM "+ table);

            ResultSetMetaData rsmd = rs.getMetaData( );
            int columnCount = rsmd.getColumnCount( );
            for(int col = 1; col <= columnCount; col++) {
                System.out.print(rsmd.getColumnLabel(col));
                System.out.print(" (" + rsmd.getColumnTypeName(col)+")");
                if(col < columnCount)
                    System.out.print(", ");
            }
            System.out.println( );

            while(rs.next( )) {
                for(int col = 1; col <= columnCount; col++) {
                    System.out.print(rs.getString(col));
                    if(col < columnCount)
                        System.out.print(", ");
                }
                System.out.println( );
            }

            rs.close( );
            stmt.close( );
            con.close( );
        }
        catch (ClassNotFoundException e) {
            System.out.println("Unable to load database driver class");
        }
    }
}
```

```
    }  
    catch (SQLException e) {  
        System.out.println("SQL Exception: " + e.getMessage( ));  
    }  
}  
}
```

The key methods used here are `getColumnCount()`, `getColumnLabel()`, and `getColumnTypeName()`. Note that type names returned by `getColumnTypeName()` are database-specific (e.g., Oracle refers to a string value as a `VARCHAR`; Microsoft Access calls it `TEXT`). Here's some sample output for `TableViewer`:

```
--- Table Viewer ---  
CustNo (SHORT), CustName (VARCHAR), CustAddress (VARCHAR)  
1, Jane Markham, 12 Stevens St  
2, Louis Smith, 45 Morrison Lane  
3, Woodrow Lang, 4 Times Square
```

8.9. Transactions

A *transaction* is a group of several operations that must behave atomically (i.e., as if they are a single, indivisible operation). With regard to databases, transactions allow you to combine one or more database actions into a single atomic unit. If you have an application that needs to execute multiple SQL statements to fulfill one goal (say an inventory management system that needs to move items from an `INVENTORY` table to a `SHIPPING` table), you probably want to use JDBC's transaction services to accomplish the goal.

This section gives a brief overview of transactions with respect to relational databases and JDBC. For broader coverage of transactions in the general J2EE environment, see [Chapter 16](#).

Working with a transaction involves the following steps: start the transaction, perform its component operations, and then either commit the transaction if all the component operations succeed or roll it back if one of the operations fails. The ability to roll back a transaction is the key feature. This means that if any one SQL statement fails, the entire operation fails, and it is as though none of the component operations took place. Therefore, it is impossible to end up with a situation in which, for example, the `INVENTORY` table has been debited, but the `SHIPPING` table has not been credited.

Another issue with transactions and databases concerns changes to the database becoming visible to the rest of the system. Transactions can operate at varying levels of isolation from the rest of the database. At the most isolated level, the results of all the component SQL statements become visible to the rest of the system only when the transaction is committed. In other words, nobody sees the reduced inventory before the shipping data is updated.

The `Connection` object in JDBC is responsible for transaction management. With JDBC, you are always using transactions in some form. By default, a new connection starts out in transaction autocommit mode, which means that every SQL statement is executed as an individual transaction that is immediately committed to the database.

To perform a transaction that uses multiple statements, you have to call the `setAutoCommit()` method with a `false` argument. (You can check the status of autocommit with the `getAutoCommit()` method.) Now you can execute the SQL statements that comprise your transaction. When you are done, call the `commit()` method to commit the transaction or the `rollback()` method to undo it. Here's an example:

```
try {
    con.setAutoCommit(false);
    // Run some SQL
    stmt.executeUpdate("UPDATE INVENTORY SET ONHAND = 10 WHERE ID = 5");
    stmt.executeUpdate(
        "INSERT INTO SHIPPING (QTY) VALUES (5)");
    con.commit( );
}
catch (SQLException e) {
```

```

    con.rollback( ); //Undo the results of the transaction
}

```

When `autocommit` is set to `false`, you must remember to call `commit()` (or `rollback()`) at the end of each transaction, or your changes will be lost.

JDBC supports a number of transaction isolation modes that allow you to control how the database deals with transaction conflicts in other words, who sees what when. JDBC defines five modes, some of which may not be supported by all databases. The default mode varies depending on the underlying database and driver. Higher isolation levels generally yield poorer performance. Here are the five standard options, which are defined as integer constants in the `Connection` interface:

TRANSACTION_NONE

Transactions are either disabled or not supported.

TRANSACTION_READ_UNCOMMITTED

Minimal transaction support that allows dirty reads. In other words, other transactions can see the results of a transaction's SQL statements before the transaction commits itself. If you roll back your transaction, other transactions may be left with invalid data.

TRANSACTION_READ_COMMITTED

Transactions are prevented from reading rows with uncommitted changes; in other words, dirty reads aren't allowed.

TRANSACTION_REPEATABLE_READ

Protects against repeatable reads as well as dirty reads. Say one transaction reads a row that is subsequently altered (and committed) by another transaction. If the first transaction reads the row again, the first transaction doesn't get a different value the second time around. The new data is visible to the first transaction only after it calls `commit()` and performs another read.

TRANSACTION_SERIALIZABLE

Provides all the support of `TRANSACTION_REPEATABLE_READ` and guards against row insertions as well. Say one transaction reads a set of rows and then another transaction adds a row to the set. If the first transaction reads the set again, it doesn't see the newly added row. Put another way, this level of isolation forces the database to treat transactions as if they occurred one at a time.

Transaction isolation modes are set by the `setTransactionIsolation()` method. For example:

```

con.setTransactionIsolation(TRANSACTION_READ_COMMITTED);

```

You can use the `DatabaseMetaData` class to determine the transaction support of the underlying database. The most useful methods are `getDefaultTransactionIsolation()`, `supportsTransactions()`, `supportsTransactionIsolationLevel()`, and `supportsDataDefinitionAndDataManipulationTransactions()` (which may well be the longest method name in the Java API).

An application that uses transactions is a prime candidate for also using a connection pool (available as of JDBC 2.0). Since each database transaction requires its own `Connection` object, an application that performs multiple simultaneous transactions (for instance, spawning threads that perform database updates) needs multiple connections available. Maintaining a pool of connections is much more efficient than creating a new one whenever you need a new transaction.

8.9.1. Distributed Transactions

With appropriate driver support, the `XADataSource` interface can also be used to create connections supporting distributed transactions. A distributed transaction is spread across more than one database and is coordinated by an external transaction monitor. Like pooled connections, the `DataSource` must be configured by the administrator of the J2EE environment.

While connections supporting distributed transactions are nearly indistinguishable from regular connections, there is a functional difference: autocommit mode defaults to off, and when a connection is used within a distributed transaction, the `rollback()`, `commit()`, and `setAutoCommit()` methods should not be called.

Connections received from an `XADataSource` may be used for nondistributed transactions as well. All of the usual transaction management commands may be used in a nondistributed transaction.

8.9.2. Savepoints

Sometimes it's helpful to be able to roll back part of a transaction but not the whole thing. JDBC 3.0 comes to the rescue with support for an advanced database feature called a *savepoint*. A savepoint is a named point within a transaction. If the underlying database and driver support the functionality, you can use the `setSavepoint(String name)` method of `Connection` to create a named savepoint in the current transaction and get an object implementing the `Savepoint` interface. This object can be passed to the `rollback()` method of `Connection` to roll back all components of the current transaction that took place after the `setSavepoint()` method was called:

```
Statement stmt = con.createStatement();
stmt.executeUpdate("delete from clients");
stmt.executeUpdate(
    "insert into clients (NAME, ID) values ('Charles Babbage', 1)");
Savepoint save = con.setSavepoint("INSERT_POINT");
stmt.executeUpdate(
    "update clients set NAME = 'Ada Lovelace' where ID = 1");
```

```
con.rollback(save);  
con.commit( );
```

This example will leave the `clients` table with a single row, with a value of one in the ID column and `Charles Babbage` in the `NAME` column.

Savepoints can't be used in distributed transactions, at least with any of the databases we've tried.



8.10. Stored Procedures

Most RDBMS systems include some sort of internal programming language (e.g., Oracle's PL/SQL). These languages allow database developers to embed procedural application code directly within the database and then call that code from other applications. The advantage of this approach is that the code can be written just once and then used in multiple different applications (even with different platforms and languages). It also allows application code to be divorced from the underlying table structure. If stored procedures handle all of the SQL and applications just call the procedures, only the stored procedures need to be modified if the table structure is changed later on.

Here is an Oracle PL/SQL stored procedure:^[*]

[*] If it looks familiar, that's because it is from George Reese's *Database Programming with JDBC* (O'Reilly).

```
CREATE OR REPLACE PROCEDURE sp_interest
(id IN INTEGER
bal IN OUT FLOAT) IS
BEGIN
SELECT balance
INTO bal
FROM accounts
WHERE account_id = id;

bal := bal + bal * 0.03;

UPDATE accounts
SET balance = bal
WHERE account_id = id;

END;
```

This PL/SQL procedure takes two input values, an account ID and a balance, and returns an updated balance.

The `CallableStatement` interface is the JDBC object that supports stored procedures. The `Connection` class has a `prepareCall()` method that is very similar to the `prepareStatement()` method we used to create a `PreparedStatement`. Because each database has its own syntax for accessing stored procedures, JDBC defines a standardized escape syntax for accessing stored procedures with `CallableStatement`. The syntax for a stored procedure that doesn't return a result set is:

```
{call procedure_name[(?[,?...])]}
```

The syntax for a stored procedure that returns a result is:

```
{? = call procedure_name[(?[,?...])]}
```

In this syntax, each question mark (?) represents a placeholder for a procedure parameter or a return value. Note that the parameters are optional. The JDBC driver is responsible for translating the escape syntax into the database's own stored procedure syntax.

Here's a code fragment that uses `CallableStatement` to run the `sp_interest` stored procedure:

```
CallableStatement cstmt = con.prepareCall("{call sp_interest(?,?)}");
cstmt.registerOutParameter(2, Types.FLOAT);
cstmt.setInt(1, accountID);
cstmt.setFloat(2, 2343.23);
cstmt.execute( );
out.println("New Balance:" + cstmt.getFloat(2));
```

In this example, we first create a `CallableStatement` using the `prepareCall()` method and passing in the appropriate escape syntax for the stored procedure. Since this stored procedure has an output parameter (actually, in this case, an `IN OUT` parameter, which means it also serves as an input parameter), we use the `registerOutParameter()` method to identify that parameter as an output of type `FLOAT`. Note that, just as with prepared statements, substituted parameters are numbered from 1 to *n*, left to right. Any time you have an output parameter in a stored procedure, you need to register its type using `registerOutParameter()` before you execute the stored procedure.

Next we set the two input parameters, the account ID and the balance, using the appropriate `setXXX()` methods. Finally, we execute the stored procedure and then use the `getFloat()` method to display the new balance. The `getXXX()` methods of `CallableStatement` are similar to those of the `ResultSet`.

You need to use `CallableStatement` only with stored procedures that have output values, such as the one we just saw. You can use either of the other statement objects to execute stored procedures that take parameters but don't return anything.

8.11. Escape Sequences

Escape sequences allow JDBC programs to package certain database commands in a database-independent manner. Since different databases implement different features (especially scalar SQL functions) in different ways, in order to be truly portable, JDBC needs to provide a standard way to access at least a subset of that functionality. We've already seen escape sequences twice: with the various SQL date and time functions and with the `CallableStatement` object.

A JDBC escape sequence consists of a pair of curly braces, a keyword, and a set of parameters. Thus `{call}` is the keyword for stored procedures, while `{d}`, `{t}`, and `{ts}` are keywords for dates and times. One keyword we haven't seen yet is `{escape}`. This keyword specifies the character that is used to escape wildcard characters in a `LIKE` statement:

```
stmt.executeQuery(
    "SELECT * FROM ApiDocs WHERE Field_Name like 'TRANS\_%' {escape '\\'}");
```

Normally, the underscore character (`_`) is treated as a single-character wildcard, while the percent sign (`%`) is the multiple-character wildcard. By specifying the backslash (`\`) as the escape character, we can match on the underscore character itself. Note that the `{escape}` keyword can also be used outside wildcard searches. For example, SQL string termination characters (such as the single quote) need to be escaped when appearing within strings.

The `{fn}` keyword allows the use of internal scalar database functions. Scalar functions are a fairly standard component of most database architectures, even though the actual implementations vary. For instance, many databases support the `SOUNDEX(string)` function, which translates a character string into a numerical representation of its sound. Another function, `DIFFERENCE(string1, string2)`, computes the difference between the soundex values for two strings. If the values are close enough, you can assume the two words sound the same ("Beacon" and "Bacon"). If your database supports `DIFFERENCE`, you can use it by executing a SQL statement that looks like this:

```
{fn DIFFERENCE("Beacon", "Bacon")}
```

Available scalar functions differ depending on the database being used. Also, some drivers, such as Oracle's, don't support the `{fn}` escape mechanism at all.

The last escape keyword is `{oj}`, which is used for outer joins. The syntax is simply:

```
{oj outer-join}
```

Outer joins aren't supported by some databases and are sufficiently complex (and unrelated to the

JDBC API per se) as to be beyond the scope of this chapter. For more information, consult the SQL documentation for your database.

Note that when performance is an issue, you can use the `setEscapeProcessing()` method of `Statement` to turn off escape sequence processing.



8.12. RowSets

JDBC 2.0 introduced the `RowSet` interface. But unlike `ResultSet`, `RowSet` encapsulates data using a JavaBean interface, allowing it to participate in the JavaBean event model. A `RowSet`, as the name implies, encapsulates a set of rows produced by a query. Since a `RowSet` is a Java bean, it can be used easily in a graphical development environment. In addition to the `RowSet` interface itself, the JDBC 3.0 API introduced a set of standard interfaces for particular kinds of row sets.

Sun's JDK 1.4 environment doesn't ship with any `RowSet` implementations. However, J2SE 5.0 does, and for earlier releases, Sun has made a number of implementations available at <http://java.sun.com/products/jdbc/download.html>. The Sun implementations are found in the `com.sun.rowset` package and have class names that echo the interface names, followed by `Impl`. Here's how to use the simplest of these, `JdbcRowSet`, which simply encapsulates a `ResultSet`:

```
JdbcRowSet jdbcRowSet
    = new JdbcRowSetImpl( );
jdbcRowSet.setCommand("SELECT * FROM CUSTOMERS WHERE CUSTNO = ?");
jdbcRowSet.setUrl("jdbc:oracle:thin:@dbhost.co.com:1521:ORCL");
jdbcRowSet.setUsername("SAMSON");
jdbcRowSet.setPassword("DELILAH");
jdbcRowSet.setType(ResultSet.TYPE_SCROLL_INSENSITIVE);
jdbcRowSet.setConcurrency(ResultSet.CONCUR_UPDATABLE);
jdbcRowSet.setInt(1, 10);
jdbcRowSet.execute( );

jdbcRowSet.first( );
System.out.println(jdbcRowSet.getString(1));
jdbcRowSet.last( );
System.out.println(jdbcRowSet.getString(1));

jdbcRowSet.close( );
```

The `getXXX` and `setXXX` methods are the same as in the `ResultSet` interface. Support for scrollable and updateable row sets depends on the underlying driver and database implementation.

The `addRowSetListener()` method of `RowSet` can be used to register other components as listeners. They must implement the `RowSetListener` interface to be implemented.

The `com.sun.rowset` package also contains a `CachedRowSet` object that will hold a `ResultSet` independently of the originating JDBC connection:

```
Connection con = DriverManager.getConnection(dbURL, dbUser, dbPassword);
Statement stmt = con.createStatement( );
```

```
ResultSet rs = stmt.executeQuery("SELECT * FROM CUSTOMERS");
CachedRowSet crs = new CachedRowSetImpl( );
crs.populate(rs);
rs.close( );
stmt.close( );
crs.next( );
System.out.println(crs.getString("CUSTNAME"));
```

In order to update a `CachedRowSet`, you must set the JDBC connection information and specify the underlying SQL via the `setCommand()` method. Now, let's continue our code sample:

```
crs.setUrl(dbURL);
crs.setUsername(dbUser);
crs.setPassword(dbPassword);
crs.setCommand("SELECT * FROM CUSTOMERS");
//...
crs.setString("CUSTNAME", "John Smith");
crs.updateRow( );
crs.moveToCurrentRow( );
crs.acceptChanges( );
```

Note that we call the `acceptChanges()` method after performing our update. This is necessary to propagate the changes back to the original `DataSource`.

The final `RowSet` available from Sun is the `WebRowSet`, which is identical to the `CachedRowSet` but has a persistence engine based on XML.

While a `RowSet` object would generally be used with JDBC, there is no actual requirement that this be so. `RowSet` implementations could be written to act against tabular data, text files, and more esoteric storage mechanisms.

Chapter 9. JNDI

The Java Naming and Directory Interface (JNDI) is an API that supports accessing naming and directory services in Java programs. The purpose of a naming service is to associate names with objects and to provide a way to access objects based on their names. You should be familiar with naming systems; you use them every day when you browse the filesystem on your computer or surf the Web by typing in a URL. Objects in a naming system can range from files in a filesystem and names located in Domain Name System (DNS) records to Enterprise JavaBeans (EJB) components in an application server and user profiles in an LDAP (Lightweight Directory Access Protocol) directory. If you want to use Java to write an application such as a search utility, a network-enabled desktop, an application launcher, an address book, a network management utility, or a class browser in short, anything that accesses objects in a naming system JNDI is a good candidate for writing that application.

As its name implies, JNDI doesn't just deal with naming services. JNDI also encompasses directory services, which are a natural extension of naming services. The primary difference between the two is that a directory service allows the association of attributes with objects, such as an email address attribute for a user object, while a naming service does not. Thus, with a directory service, you can access the attributes of objects and search for objects based on their attributes. You can use JNDI to access directory services like LDAP and Novell Directory Services (NDS) directories.

JNDI is an integral part of the J2EE standard framework; it is used by J2EE components to access deployment information and various runtime resources, such as Enterprise JavaBean references, JDBC data sources, and so on. Details on how JNDI plays a role specifically in accessing EJBs can be found in [Chapter 6](#). Full details on the use of JNDI for application deployment can be found in [Chapter 2](#).

9.1. JNDI Architecture

The architecture of JNDI is somewhat like the JDBC architecture in that both provide a standard protocol-independent API built on top of protocol-specific driver or provider implementations. This layer insulates an application from the actual data source it is using, so, for example, it doesn't matter whether the application is accessing an NDS or LDAP directory service.

The JNDI architecture includes both an application programming interface (API) and a service provider interface (SPI), as shown in [Figure 9-1](#). A Java application uses the JNDI API to access naming and directory services, primarily through the `Context` and `DirContext` interfaces. The JNDI API is defined in the `javax.naming` and `javax.naming.directory` packages. Note that JNDI is a standard extension to the Java 2 platform; it is included in the core APIs starting with JDK 1.3. If you are using an earlier version of the Java environment, the JNDI API classes and various providers are also available separately at <http://java.sun.com/products/jndi>. This chapter covers JNDI Version 1.2, which is the version included in JDK 1.3, 1.4, and 1.5, and is a standard component in the J2EE 1.3 and 1.4 specifications.

Figure 9-1. The JNDI architecture

In order for an application to actually interact with a particular naming or directory service, there

must be a JNDI service provider for that service. This is where the JNDI SPI comes in. A service provider is a set of classes that implements various JNDI interfaces for a specific naming or directory service, much as a JDBC driver implements various JDBC interfaces for a particular database system. The provider can also implement other interfaces that aren't part of JNDI, such as Novell's `NdsObject` interface.

The classes and interfaces in the `javax.naming.spi` package are of interest only to developers who are creating service providers. For instance, the `NamingManager` class defines methods for creating `Context` objects and otherwise controlling the operation of the underlying service provider. As an application programmer, you don't have to worry about the JNDI SPI. All you have to do is make sure that you have a service provider for each naming or directory service you want to use. Sun maintains a list of available service providers on the JNDI web page.



9.2. A Simple Example

Before going any further, let's take a look at a simple JNDI example. To access an object in a naming system, we need to create an initial context for the naming system, to give us an entry point into it. Once we have an initial context, we can look up an object by name.

[Example 9-1](#) demonstrates the basic JNDI tasks of getting an initial context to a naming system and looking up an object in that naming system. With slight modification, this code can be used to look up objects with any JNDI provider. So, for example, you could use this `Lookup` class to look up Enterprise JavaBeans or remote objects in an RMI registry and handle them however you like. All you have to change is the properties that control the naming system being accessed.

Example 9-1. Looking up an object in a naming system

```
import java.util.Properties;
import javax.naming.*;

public class Lookup {
    public static void main(String[] args) {
        String name = "";
        if (args.length > 0)
            name = args[0];

        try {
            // Create a Properties object and set default properties
            Properties props = new Properties( );
            props.put(Context.INITIAL_CONTEXT_FACTORY,
                    "com.sun.jndi.fscontext.RefFSContextFactory");
            props.put(Context.PROVIDER_URL, "file:///");

            // Optional command-line args to specify alternate factory and URL
            if (args.length > 1) {
                props.put(Context.INITIAL_CONTEXT_FACTORY, args[1]);
            }
            if (args.length > 2) {
                props.put(Context.PROVIDER_URL, args[2]);
            }

            // Create the initial context from the properties we just created
            Context initialContext = new InitialContext(props);

            // Look up the named object
```

```

    Object obj = initialContext.lookup(name);
    if (name.equals(""))
        System.out.println("Looked up the initial context");
    else
        System.out.println(name + " is bound to: " + obj);
}
catch (NamingException ne) {
    System.out.println("Encountered a naming exception");
    ne.printStackTrace( );
}
}
}

```

The first thing the `Lookup` application does is create a `java.util.Properties` object and use it to store some `String` values. The keys used for these values are constants defined in the `javax.naming.Context` class. Each constant corresponds to an underlying standard JNDI property name that is meant to communicate specific information about the JNDI service the application is using. `Context.INITIAL_CONTEXT_FACTORY` specifies the factory class that creates an initial context for the service we want to use. The class `com.sun.jndi.fscontext.RefFSContextFactory` is a factory class from the filesystem service provider from Sun. It isn't provided by default in the Sun JDK downloads, but it can be downloaded from <http://java.sun.com/products/jndi>. `Context.PROVIDER_URL` tells the factory class the protocol, server name, and path to use in creating an initial context. We specify the URL `file:///` to indicate the root of the local filesystem. This works on any Unix or Windows filesystem.

Once we have created the `Properties` object and set the default values for the context factory and the provider URL, we check the command-line arguments for specific values that the user may have provided. Finally, we pass the `Properties` to the `javax.naming.InitialContext` constructor, which returns the initial context object that is our entry point into this particular naming system. Next, we call the `lookup()` method on `initialContext`, specifying the name we want to look up. This call returns an object from the naming system, which, in this case, is a file or directory.

You can run `Lookup` from the command line and specify an optional name to look up. For example, on a typical Windows machine:

```

% java Lookup boot.ini
boot.ini is bound to: C:\boot.ini

```

If the name is actually a directory, the output looks a bit different:

```

% java Lookup System
System is bound to: com.sun.jndi.fscontext.RefFSContext@803adec0

```

Note that if we wanted to make `Lookup` more general, we might change it so that it reads its property values from a properties file. Luckily, JNDI takes care of this for you; it automatically loads any of its standard properties from a `jndi.properties` file if it finds the file in your `CLASSPATH`. Changing the naming system is a simple matter of editing this properties file to specify an appropriate factory object and URL. You can always override the properties loaded from `jndi.properties` in your code, as

we've done in the `Lookup` example. In the next section, we'll discuss the properties that can be set to control how JNDI operates.

JNDI throws naming exceptions when naming operations can't be completed. The root naming exception, `javax.naming.NamingException`, is a catchall for any JNDI exception. The `javax.naming` package defines numerous subclasses of `NamingException`. A common naming exception, `NameNotFoundException`, is thrown when a name can't be found, because it either doesn't exist or is spelled incorrectly. JNDI throws a `NoPermissionException` when a program doesn't have sufficient rights or permissions and throws an `OperationNotSupportedException` when an application uses a JNDI method on an object that doesn't support that specific naming operation.



9.3. Introducing the Context

A naming service associates names with objects. An association between a name and an object is called a *binding*, and a set of such bindings is called a *context*. A name in a context can be bound to another context that uses the same naming conventions; the bound context is called a *subcontext*. Using a filesystem as an analogy, a directory (such as */temp*) is a context that contains bindings between filenames and objects that the system can use to manipulate the files (often called *file handles*). If a directory contains a binding for another directory (e.g., */temp/javax*), the subdirectory is a subcontext.

JNDI represents a context in a naming system using the `javax.naming.Context` interface. This is the key interface for interacting with naming services. A `Context` knows about its set of bindings in the naming system, but little else. While you might be tempted to think of a `Context` as an exotic `java.io.File` object, you should resist making that analogy, as it will just confuse you. Unlike a `File` object, which can tell you its absolute and relative names as well as return a reference to its parent, a `Context` object can tell you only about its bindings. A `Context` can't go up a level, tell you its absolute pathname or even tell you its own name. When you think of a `Context`, think of an object that encapsulates its children as data and has methods that perform operations on that data, not on the `Context` itself.

9.3.1. Using the InitialContext Class

The `javax.naming.InitialContext` class implements the `Context` interface and serves as our entry point to a naming system. To use JNDI to access objects in a naming system, you must first create an `InitialContext` object. The `InitialContext` constructors take a set of properties in the form of a `java.util.Hashtable` or one of its subclasses, such as a `Properties` object. Here is how we created an `InitialContext` in the `Lookup` example:

```
// Create a Properties object and set default properties
Properties props = new Properties( );
props.put(Context.INITIAL_CONTEXT_FACTORY,
          "com.sun.jndi.fscontext.RefFSContextFactory");
props.put(Context.PROVIDER_URL, "file:///");

...

// Create the initial context from the properties we just created
Context initialContext = new InitialContext(props);
```

The most fundamental property key is `java.naming.factory.initial`, which corresponds to the `Context.INITIAL_CONTEXT_FACTORY` constant. The value for this property specifies the name of a factory class in a JNDI service provider. It is the job of this factory class to create an `InitialContext`

that is appropriate for its service and then hand the object back to us. We have to give the factory class all the information it needs to create an `InitialContext` in the form of other property values. For example, the factory class learns the protocol, server name, and path to use from the `java.naming.provider.url` property (`Context.PROVIDER_URL`).

The filesystem factory class (`com.sun.jndi.fscontext.RefFSContextFactory`) requires a minimal set of properties; it needs only to have its context factory and provider URL specified. Other JNDI providers can be more demanding. For example, the factory class in Sun's LDAP service provider requires the URL of the LDAP server and directory entry you want to access, a username and password, and an authentication type. Here are some properties (shown in the file format used by the `Properties` class) you might use to create an `InitialContext` with the LDAP factory class:

```
java.naming.factory.initial=com.sun.jndi.ldap.LdapCtxFactory
java.naming.provider.url=ldap://192.168.1.20/o=Planetary,c=US
java.naming.security.authentication=simple
java.naming.security.principal=cn=kris
java.naming.security.credentials=secret
```

These properties create an `InitialContext` for an organization called Planetary in the global X.500 namespace.

The standard JNDI properties can be specified in several ways. Our `Lookup` example set them explicitly in the source, using some hardwired default values and (optionally) reading alternate values from the command line. Since these are Java properties, they can also be specified using `-D` arguments to the Java runtime, for example:

```
% java -Djava.naming.factory.initial=org.my.jndi.factory
-Djava.naming.provider.url=file:/// MyJNDIClient
```

JNDI can also read its properties from a `jndi.properties` file located in your `CLASSPATH`. If we put the following into a `jndi.properties` file in the root of one of the directories on our `CLASSPATH`:

```
java.naming.factory.initial=org.my.jndi.factory
java.naming.provider.url=file:///
```

it will have the same effect as the Java runtime arguments just shown.

Note that some JNDI services, such as nonpublic LDAP servers, require authentication in order to access them. In these cases, you would use the `Context.SECURITY_PRINCIPAL` and `Context.SECURITY_CREDENTIALS` properties to indicate the username and password for the account you are using to access the underlying service. The corresponding property names for entering these in your `jndi.properties` file are `java.naming.security.principal` and `java.naming.security.credentials`, respectively.

9.3.2. Other Naming Systems

Since many companies support JNDI, there are many naming system service providers. You can find a reasonably comprehensive list of public JNDI providers from the JNDI page on the Sun web site (currently at <http://java.sun.com/products/jndi/serviceproviders.html>). You should contact the vendor of your enterprise naming system or directory for more details regarding its specialized providers. [Table 9-1](#) lists the factory classes for some common JNDI providers. The first three are provided out-of-the-box with Sun's JDK 1.3, 1.4, and 1.5 implementations. JDK 1.4 and 1.5 also include the DNS provider by default. The others can be downloaded from the URL just specified.

Table 9-1. JNDI factory classes

Service	Factory class
LDAPv3	<code>com.sun.jndi.ldap.LdapCtxFactory</code>
COSNaming	<code>com.sun.jndi.cosnaming.CNCtxFactory</code>
RMI registry	<code>com.sun.jndi.rmi.registry.RegistryContextFactory</code>
DNS	<code>com.sun.jndi.dns.DnsContextFactory</code>
	<code>com.sun.jndi.fscontext.FSContextFactory</code>
Filesystem	or
	<code>com.sun.jndi.fscontext.RefFSContextFactory</code>
NDS	<code>com.novell.naming.service.nds.NdsInitialContextFactory</code>
NIS	<code>com.sun.jndi.nis.NISCtxFactory</code>

9.4. Looking Up Objects in a Context

Retrieving an object by name from a naming system or directory is called *looking up the object*. This is the job of the `lookup()` method of `Context`. Performing a lookup is analogous to getting the number of a friend from a telephone book by looking up his name. You can use JNDI to look up and retrieve an EJB home interface from an application server or a remote object from a remote RMI registry.

When you call `lookup()`, you specify the name of the child of the `Context` you want to find. `lookup()` returns a `java.lang.Object` that represents the child. Here's how we did it in the `Lookup` example:

```
Object obj = initialContext.lookup(name);
```

Calling `lookup()` retrieves an object from the underlying naming system. The JNDI service provider determines the Java representation of these objects, and we have no way of affecting the provider's decision. Depending on the naming system and the design of the provider, the object you retrieve may or may not implement `Context`. For example, if you use the Sun filesystem provider and your current context is a directory, looking up a child that is a file returns an instance of `java.io.File`. Looking up a directory, however, returns an instance of `FSContext` or `RefFSContext`, both of which implement `Context`. As another example, say you use Novell's NDS provider and the current context is an NDS tree. If you look up an organization, you get back an `OrganizationDirContext` that implements both `Context` and Novell's `NdsObject` interface. The bottom line is that the class you get back from `lookup()` depends on how the service provider is implemented.

JNDI leaves it up to the service provider to choose whether objects should implement `Context`. There are no strict rules about when an object should implement it, but there are some general guidelines. An object with children is a container, and the guideline for containers is that they should implement `Context`. This is because we generally perform naming operations upon these objects. In the filesystem provider, for example, directories can contain other objects (files and other directories), so the object that represents a directory should implement `Context` (which is how Sun's filesystem provider behaves). If directories don't support `Context` methods, we can't use JNDI to look up any children of the directory.

Objects without children are *leaves*, and leaves may or may not implement `Context`, depending on how they are used. For example, because files have no children to look up, the methods we perform on them lie outside the naming system. The author of the Sun filesystem provider made a design choice that once we have looked up a file, we're done with JNDI. So, we can read the input stream of a file or write an output stream to it, but we can't use JNDI to perform naming operations on the file.

9.5. The NamingShell Application

The rest of the examples in this chapter are going to be based on the `NamingShell` code shown in [Example 9-2](#). `NamingShell` is an extensible JNDI shell that enables us to perform naming operations in any JNDI-accessible naming system. The shell provides methods for getting and setting the current object and other shell-related details, and it also keeps track of the name of the current object, something a `Context` can't do for itself.

Example 9-2. The NamingShell class

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.util.Hashtable;
import java.util.StringTokenizer;
import java.util.Vector;

import javax.naming.Context;

class NamingShell {

    // Private variables
    private static Hashtable COMMAND_TABLE = new Hashtable();
    private static String PROMPT = "[no initial context]";
    private static String VERSION = "1.0";
    private static Context CURRENT_CONTEXT, INITIAL_CONTEXT;
    private static String CURRENT_NAME, INITIAL_NAME;
    private static boolean RUNNING = true;
    private static String CMD_PKG = "com.oreilly.jent.jndi";

    // Shell operations
    private static void exit(int status) { System.exit(status); }

    // Accessor methods
    public static Hashtable getCommands() { return COMMAND_TABLE; }
    public static Context getCurrentContext() { return CURRENT_CONTEXT; }
    public static String getCurrentName() { return CURRENT_NAME; }
    public static Context getInitialContext() { return INITIAL_CONTEXT; }
    public static String getInitialName() { return INITIAL_NAME; }
    public static String getPrompt() { return PROMPT; }
    public static void setCurrentContext(Context ctx) {
        CURRENT_CONTEXT = ctx; }
    public static void setInitialContext(Context ctx) {
        INITIAL_CONTEXT = ctx; }
```

```

public static void setInitialName(String name) {
    INITIAL_NAME = name; }
public static void setPrompt(String prompt) { PROMPT = prompt; }
public static void setCurrentName(String name) {
    CURRENT_NAME = name;
    setPrompt(name);
}

// Executes a preinstantiated command we are sure is already
// present in the table
private static void execute(Command c, Vector v) {
    if (c == null) {
        System.out.println("No command was loaded; cannot execute the command.");
        return;
    }
    try {
        c.execute(CURRENT_CONTEXT, v);
    }
    catch (CommandException ce) {
        System.out.println(ce.getMessage());
    }
}

// Another private method that enables us to specify a command
// by its string name and that loads the command first
private static void execute(String s, Vector v) {
    execute(loadCommand(s), v);
}

// Loads the command specified in commandName; the help command
// relies on this method
public static Command loadCommand(String commandName) {
    // The method returns a null command unless some of its
    // internal logic assigns a new reference to it
    Command theCommand = null;

    // First see if the command is already present in the hashtable
    if (COMMAND_TABLE.containsKey(commandName)) {
        theCommand = (Command)COMMAND_TABLE.get(commandName);
        return theCommand;
    }

    try {
        // Here we use a little introspection to see if a class
        // implements Command before we instantiate it
        Class commandInterface = Class.forName(CMD_PKG + ".Command");
        Class commandClass = Class.forName(CMD_PKG + "." + commandName);

        // Check to see if the class is assignable from Command
        // and if so, put the instance in the command table
        if (!(commandInterface.isAssignableFrom(commandClass)))
            System.out.println "[" + commandName + "]: Not a command");
    }
}

```

```

        else {
            theCommand = (Command)commandClass.newInstance();
            COMMAND_TABLE.put(commandName, theCommand);
        }
    }
}
catch (ClassNotFoundException cnfe) {
    System.out.println "[" + commandName + "]: command not found";
}
catch (IllegalAccessException iae) {
    System.out.println "[" + commandName + "]: illegal access";
}
catch (InstantiationException ie) {
    System.out.println "[" + commandName + "]: command couldn't be instantiated";
}

return theCommand;
}

// This method reads a line of input, gets the command and arguments
// within the line of input, and then dynamically loads the command
// from the current directory of the running shell
private static void readInput() {
    // Get the input from System.in
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

    // Begin reading input
    try {
        while (RUNNING) {
            System.out.print(PROMPT + "% ");

            // Tokenize the line, read each token, and pass the token
            // into a convenient remaining arguments Vector that we
            // pass into the Command
            StringTokenizer tokenizer = new StringTokenizer(br.readLine());
            Vector remainingArgs = new Vector();
            String commandToken = "";
            if (tokenizer.hasMoreTokens()) {
                commandToken = tokenizer.nextToken();
                while (tokenizer.hasMoreTokens())
                    remainingArgs.addElement(tokenizer.nextToken());
            }

            // Dynamically load the class for the appropriate command
            // based upon the case-sensitive name of the first token,
            // which is the command token
            if (!(commandToken.equals("")))
                execute(commandToken, remainingArgs);
        }
    }
    catch (java.io.IOException ioe) {
        System.out.println("Caught an IO exception reading a line of input");
    }
}

```

```

}
// Constructor
NamingShell(String[] args) {
}

// Main method that reads input until the user exits
public static void main(String[] args) {
    NamingShell shell = new NamingShell(args);
    NamingShell.readInput();
    System.out.println("Exiting");
}
}

```

Once you've loaded `NamingShell`, you can use the shell to execute JNDI-related commands, just as you would use a regular shell to execute operating system commands. We encourage you to download the code for `NamingShell` right now, so that you can experiment with it as we proceed through the rest of the chapter. `NamingShell` uses the name you type to locate a command dynamically from the classpath. The shell has no interpreter; however, `NamingShell` expects a command to implement the `Command` interface and its `execute()` method. This means a command really interprets itself. A command throws a `CommandException` when execution fails.

As you can see, `NamingShell` itself contains very little real JNDI code. All the JNDI functionality is implemented in the various `Command` classes we create to handle particular JNDI operations. The shell simply supports the loading of commands and keeps track of various shell-related details.

9.5.1. The Command Interface

The `Command` interface (shown in [Example 9-3](#)) describes a standard interface for a shell command. It has an `execute()` method that contains the command logic and a `help()` method for displaying online help for the command. If `execute()` encounters a naming exception (or some other exception), it throws a `CommandException` (shown in [Example 9-4](#)), which stores the first exception as an instance variable so that the shell can display the exception appropriately.

Example 9-3. The Command interface

```

import java.util.Vector;
import javax.naming.Context;

public interface Command {
    public void execute(Context c, Vector v)
        throws CommandException;
    public void help( );
}

```

Example 9-4. The CommandException class

```
public class CommandException extends Exception {
    Exception e; // Root exception
    CommandException(Exception e, String message) {
        super(message);
        this.e = e;
    }
    public Exception getRootException( ) {
        return e;
    }
}
```

9.5.2. A Command for Loading an Initial Context

As we said earlier, to use JNDI to look up an object in a naming system (or, in fact, to do anything with the naming system), you first have to create an `InitialContext` for that naming system. So, the first command we need to implement is *initctx*, for loading an initial context into `NamingShell`. [Example 9-5](#) shows an implementation of this command.

Example 9-5. The initctx command

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

public class initctx implements Command {

    public void execute(Context c, Vector v) {
        String jndiPropsFilename = null;

        // Check for a properties filename
        if (!v.isEmpty( ))
            jndiPropsFilename = (String)v.firstElement( );

        System.out.println("file = " + jndiPropsFilename);

        try {
            // If no properties file is specified, let JNDI get its properties from
            // the default jndi.properties file on the CLASSPATH. Otherwise, use
            // the specified properties file.
            if (jndiPropsFilename != null) {
                Properties props = new Properties( );
                File jndiProps = new File(jndiPropsFilename);
                props.load(new FileInputStream(jndiProps));
            }
        }
    }
}
```

```

        NamingShell.setInitialContext(new InitialContext(props));
    }
    else {
        NamingShell.setInitialContext(new InitialContext( ));
    }
    NamingShell.setInitialName("/");
    NamingShell.setCurrentContext(NamingShell.getInitialContext( ));
    NamingShell.setCurrentName(NamingShell.getInitialName( ));
    System.out.print("Created initial context using ");
    if (jndiPropsFilename != null) {
        System.out.println(jndiPropsFilename);
    }
    else {
        System.out.println("jndi.properties.");
    }
}
catch (NamingException ne) {
    System.out.println("Couldn't create the initial context");
}
catch (FileNotFoundException fnfe) {
    System.out.print("Couldn't find properties file: ");
    System.out.println(jndiPropsFilename);
}
catch (IOException ioe) {
    System.out.print("Problem loading the properties file: ");
    System.out.println(jndiPropsFilename);
}
catch (Exception e) {
    System.out.println("There was a problem starting the shell");
}
}

public void help( ) { System.out.println("Usage: initctx [filename]"); }
}

```

The *initctx* command accepts an optional argument that specifies the name of a properties file to use in creating the `Properties` object that is passed to the `InitialContext` constructor. If no filename is specified, *initctx* creates the `InitialContext` using the no-argument constructor, which in turn uses the default *jndi.properties* file in the `CLASSPATH`. So, with `NamingShell`, all you have to do to use a particular naming service is create an appropriate properties file for that service and either specify it explicitly when you invoke the *initctx* command or put it in your `CLASSPATH` as *jndi.properties*.

9.5.3. Running the Shell

With `NamingShell` and *initctx*, we have enough functionality to actually run the shell. Before you try this, make sure that any specialized providers are specified in your `CLASSPATH`. Here's how we might start `NamingShell` and establish an initial context, once the `CLASSPATH` is set appropriately:

```
% java NamingShell  
[no initial context]% initctx  
Created initial context using jndi.properties  
/%
```

In this case, since we didn't specify a properties file on the command line, the no-argument `InitialContext` constructor is called, and it looks for a `jndi.properties` file in the `CLASSPATH`. For the purpose of our next few examples, let's assume that this file contains property settings that allow us to use the filesystem provider from Sun. You can change initial contexts at any time during the shell session by running `initctx` with a new filename. After you create an initial context, you can begin performing naming operations by typing in commands. To exit the shell, simply use the `exit` command.^[*] If you aren't sure how a command works, you can get help for that command by typing:

[*] The `help` and `exit` commands are implemented as separate classes, just like the JNDI-related commands. We don't examine the code for these commands, because they don't use JNDI. However, the code is provided in the examples available online at <http://www.oreilly.com/catalog/javaentnut3>.

```
/% help command
```

9.6. Listing the Children of a Context

A common JNDI operation is retrieving the list of names of an object's children. For example, an application might get the names of Enterprise JavaBeans in a Java application server or list the names of user profile information in an LDAP server in order to populate a Swing `JTree` component in an address book application. You list the names of an object's children using the `list()` method of `Context`:

```
NamingEnumeration children = initialContext.list("");
```

The `list()` method returns a `javax.naming.NamingEnumeration` of `javax.naming.NameClassPair` objects; each `NameClassPair` contains the name and class of a single child of the `Context`. Note that the `NameClassPair` is not the child itself. Its `getName()` method, however, enables us to learn the name of the child while `getClassName()` lets us access the child's class name. `NamingEnumeration` implements the `java.util.Enumeration` interface, so it allows us to loop through the results of calling `list()` using the familiar enumeration methods. JNDI actually uses `NamingEnumeration` as the return type of a number of naming operations; the actual objects in the enumeration vary depending on the operation.

[Example 9-6](#) shows the implementation of a `list` command for our `NamingShell`. Because executing `list()` requires a current `Context`, the `execute()` method queries the shell to determine whether one exists. If there is no current `Context`, the method throws an exception.

Example 9-6. The list command

```
import java.util.Vector;
import javax.naming.*;

public class list implements Command {
    public void execute(Context c, Vector v) throws CommandException {

        String name = "";

        // An empty string is OK for a list operation as it means
        // list children of the current context
        if (!(v.isEmpty()))
            name = (String)v.firstElement();

        // Check for current context; throw an exception if there isn't one
        if (NamingShell.getCurrentContext() == null)
            throw new CommandException(new Exception(),
                "Error: no current context.");
    }
}
```

```

// Call list( ) and then loop through the results, printing the names
// and class names of the children
try {
    NamingEnumeration enum = c.list(name);
    while (enum.hasMore( )) {
        NameClassPair ncPair = (NameClassPair)enum.next( );
        System.out.print(ncPair.getName( ) + " (type ");
        System.out.println(ncPair.getClassName( ) + ")");
    }
}
catch (NamingException e) {
    throw new CommandException(e, "Couldn't list " + name);
}
}

public void help( ) { System.out.println("Usage: list [name]"); }
}

```

Let's continue with our example of using `NamingShell` with the filesystem provider. Say that we are accessing a filesystem in which we have unpacked a jar file that contains, among others, a `javax` directory and a `naming` subdirectory. If the current `Context` is the `naming` directory (ignoring for a moment how we set the current `Context`; we'll see how to do that shortly), we can use the `list` command with the following results:

```

naming% list
AuthenticationException.class (type java.io.File)
AuthenticationNotSupportedException.class (type java.io.File)
BinaryRefAddr.class (type java.io.File)
Binding.class (type java.io.File)
CannotProceedException.class (type java.io.File)
CommunicationException.class (type java.io.File)
CompositeName.class (type java.io.File)
CompoundName.class (type java.io.File)
ConfigurationException.class (type java.io.File)
Context.class (type java.io.File)
ContextNotEmptyException.class (type java.io.File)
directory (type javax.naming.Context)
...

```

9.6.1. How Names Work

The `list()` method on `Context` allows us to list the names of the children of any arbitrary child of a `Context`. We just saw that we can list the names of the children of a `Context` itself (in this case, the `naming` directory) by calling its `list()` method using an empty string as a parameter. Again, let's assume we have a `Context` object for the `naming` subdirectory under `javax`. Here's how a call to get the names of the children of this `Context` might look:

```
NamingEnumeration childrenOfNaming = namingContext.list("");
```

The result is a `NamingEnumeration` that contains `NameClassPair` objects representing all the children of *naming* (i.e., the classes and subpackages of `javax.naming`), including the *directory* directory (i.e., the `javax.naming.directory` subpackage).

To list the names of the children of an arbitrary child of a `Context`, we have to pass a name to `list()`. For example, we can list the children of *directory* by specifying the `String "directory"` as a parameter to `list()`:

```
NamingEnumeration childrenOfDirectory = namingContext.list("directory");
```

The result here is a `NamingEnumeration` that contains `NameClassPair` objects representing all the children of *directory* (i.e., the classes of `javax.naming.directory`, such as `DirContext`).

You can also specify a name using something called a compound name. A *compound name* is composed of atomic names, like "naming" and "directory," that are separated by separator characters. In the case of the filesystem provider, these can be either a Unix-style forward slash (`/`) or a Windows-style backslash (`\`). Any JNDI method that takes a name as a parameter can accept a compound name.

Say we have a `Context` object for the *javax* directory. We can get a list of the children of that directory as follows:

```
NamingEnumeration childrenOfDirectory = javaxContext.list(
    "naming/directory");
```

This call returns the same `NamingEnumeration` we got earlier. Now consider the following call:

```
NamingEnumeration childrenOfContext = javaxContext.list(
    "naming/Context");
```

The compound name here specifies an object that is not a `Context`, so it has no children. In this case, the call to `list()` throws a `NamingException`.

The separator character used in JNDI compound names varies across naming and directory services; the separator is analogous to the separator used in `java.io.File`. Although the Sun filesystem provider allows us to use the Unix-style forward slash and the Windows-style backslash interchangeably, most service providers are very picky about the separator character used for that service. Unfortunately, the JNDI API doesn't provide a way to get the separator character programmatically the way `java.io.File` does. Although the `javax.naming.CompoundName` class reads a property called `jndi.syntax.separator` that contains the separator character, this property can't be accessed outside the service provider. So, to find out the separator character for a particular service provider, you have to consult the documentation or some sample code for that provider.

9.6.2. Browsing a Naming System

So far, we know how to look up an object in a `Context` using `lookup()` and list the children of that `Context` with `list()`. Browsing is a composite operation that involves repeated calls to `list()` and `lookup()` to see what objects are available in the naming system and to move around in those objects.

`Context` objects are the key to browsing. You start with a current `Context` and list the children of that `Context` to see which child you (or, more likely, the user) are interested in. Once you have selected an interesting child, look up that child to get the actual child object. If the object implements `Context`, you can use this new `Context` object to continue browsing, by calling `list()` again, selecting a child, and looking up its object. If the object doesn't implement `Context`, however, you obviously can't continue browsing down that branch of the naming system. Once you have a `Context` object, it is always possible to list its children and look up objects within it. So, for example, you can always use the `InitialContext` for a naming system to go back and start browsing at the entry point to the naming system.

[Example 9-7](#) shows an implementation of a `cd` command for `NamingShell`. The `cd` command changes the current context of `NamingShell` to the specified context; you use it in conjunction with the `list` command to browse the naming system. The name of this command comes from the Unix `cd` command for changing directories, because changing the directory on a Unix system is an analogous operation to changing the current context when `NamingShell` is used with the filesystem provider. To change the current context back to the initial context, use either `cd /` or `cd .`. Note, however, that you can't use `cd ..` because `Context` objects don't know about their parents, and so you can't go up the `Context` hierarchy.

Example 9-7. The cd command

```
import java.util.Vector;
import javax.naming.*;

class cd implements Command {
    public void execute(Context ctx, Vector v) throws CommandException {
        if (NamingShell.getCurrentContext( ) == null)
            throw new CommandException(new Exception( ), "No current context");
        else if (v.isEmpty( ))
            throw new CommandException(new Exception( ), "No name specified");

        // Get args[0] and throw away the other args
        else {
            String name = (String)v.firstElement( );
            try {
                if (name.equals("..")) {
                    throw new CommandException(new Exception( ),
                        "Contexts don't know about their parents.");
                }
                else if (((name.equals("/") || (name.equals("\\\\")))) {
```

```

        NamingShell.setCurrentContext(NamingShell.getInitialContext( ));
        NamingShell.setCurrentName(NamingShell.getInitialName( ));
        System.out.println("Current context now " + name);
    }
    else {
        Context c =
            (Context) (NamingShell.getCurrentContext( )).lookup(name);
        NamingShell.setCurrentContext(c);
        NamingShell.setCurrentName(name);
        System.out.println("Current context now " + name);
    }
}
catch (NamingException ne) {
    throw new CommandException(ne, "Couldn't change to context " + name);
}
catch (ClassCastException cce) {
    throw new CommandException(cce, name + " not a Context");
}
}
}

public void help( ) { System.out.println("Usage: cd [name]"); }
}

```

Earlier, when demonstrating the *list* command, we asked you to assume that the current `Context` for `NamingShell` was the *naming* subdirectory. Now you can see just how to change the current `Context` to that directory:

```

initctx% cd temp
Current context now temp
temp% cd javax
Current context now javax
javax% cd naming
Current context now naming

```

Of course, these commands assume you are starting from the initial context and that the *naming* directory is available in the filesystem at */temp/javax/naming*.

9.6.3. Listing the Bindings of a Context

The `listBindings()` method of `Context` provides an alternative way to access the children of a `Context`. We've seen that `list()` returns a `NamingEnumeration` of `NameValuePair` objects, where each `NameValuePair` provides access to the name and class name of a single child of the `Context`. `listBindings()` also returns a `NamingEnumeration`, but, in this case, the enumeration contains `Binding` objects. `Binding` is a subclass of `NameValuePair` that contains the actual child object in addition to its name and class. You can use the `getObject()` method of `Binding` to get the child

object.

Just as with `list()`, we can pass an empty string to `listBindings()` to return the bindings for a `Context`:

```
NamingEnumeration bindings = initialContext.listBindings("");
```

`listBindings()` is designed for situations in which you need to perform some sort of operation on all the children of a `Context`, and you want to save yourself the time and trouble of looking up each child individually. Be aware, however, that `listBindings()` is potentially a very expensive operation, as it has to get each child object from the underlying naming system. If you don't need all the objects, you are better off using `list()` to get the names of the children and then just looking up the objects you need.

[← PREV](#)

9.7. Creating and Destroying Contexts

With JNDI, you can create a context in a naming system using the `createSubcontext()` method of an existing `Context`. All you have to specify in this call is the name of the new subcontext. Note that `Context` doesn't provide a public constructor; creating a new context requires a parent `Context` (such as an `InitialContext`) whose `createSubcontext()` method we can call.

When you call `createSubcontext()`, the JNDI service provider you are using looks at the class of the `Context` whose method you are calling. Based on this class and the provider's own internal logic, the provider creates a new object of a particular class. You don't get to pick the class of this object; the provider has all the control over the class of the object it creates (you do, however, have control over the class of object that is created when using directory services, as we'll see shortly). The documentation for a service provider should tell you what kinds of objects `createSubcontext()` can create. Note that whatever object the provider creates, it always implements `Context`; there is no way to use JNDI to create an object that doesn't implement `Context`.

For example, if we are using the Sun filesystem provider and our current `Context` is a directory, calling `createSubcontext()` causes the provider to create a directory, not a file. This makes sense, as a directory can have subordinates and thus implements `Context`. There is actually no way to create a file using the JNDI API and the filesystem provider; you have to drop out of JNDI to do this, as we'll see in the next section.

[Example 9-8](#) shows the implementation of a `create` command for `NamingShell` that demonstrates how to use `createSubcontext()`.

Example 9-8. The create command

```
import java.util.Vector;
import javax.naming.*;

public class create implements Command {

    public void execute(Context c, Vector v) throws CommandException {

        // Check to see if we have the name we need to create a context
        if (v.isEmpty( ))
            throw new CommandException(new Exception( ), "No name specified");

        String name = (String)v.firstElement( );
        try {
            c.createSubcontext(name);
            System.out.println("Created " + name);
        }
        catch (NoPermissionException npe) {
```

```

        throw new CommandException(npe,
            "You don't have permission to create " + name + " at this context");
    }
    catch (NamingException ne) {
        throw new CommandException(ne, "Couldn't create " + name);
    }
}

public void help( ) { System.out.println("Usage: create [name]"); }
}

```

Here is the *create* command, in conjunction with the *cd* and *list* commands we've already seen:

```

/% create test
Created test
/% cd test
Current context now test
test% create another
Created another
test% list
another (type javax.naming.Context)

```

The `destroySubcontext()` method of `Context` destroys a context, as you might expect from its name. Again, you have to specify the name of the context to be destroyed; you can't destroy the current object by specifying an empty name. Calling the `destroySubcontext()` method on a `Context` from the Sun filesystem provider is analogous to removing a directory in the filesystem.

[Example 9-9](#) shows the implementation of a *destroy* command for `NamingShell`. Note that it contains several `catch` statements, to handle such exceptions as insufficient permission to destroy a context, trying to destroy an object that doesn't implement the `Context` interface, and trying to destroy an object that has children.

Example 9-9. The destroy command

```

import java.util.Vector;
import javax.naming.*;

public class destroy implements Command {
    public void execute(Context c, Vector v) throws CommandException {

        // Check to see if we have the name we need
        if (v.isEmpty( ))
            throw new CommandException(new Exception( ), "No name specified");

        String name = (String)v.firstElement( );

        try {

```

```
        c.destroySubcontext(name);
        System.out.println("Destroyed " + name);
    }
    catch (NameNotFoundException nnfe) {
        throw new CommandException(nnfe, "Couldn't find " + name);
    }
    catch (NotContextException nce) {
        throw new CommandException(nce,
            name + " is not a Context and couldn't be destroyed");
    }
    catch (ContextNotEmptyException cnee) {
        throw new CommandException(cnee,
            name + " is not empty and couldn't be destroyed");
    }
    catch (NamingException ne) {
        throw new CommandException(ne, name + " couldn't be destroyed");
    }
}

public void help( ) { System.out.println("Usage: destroy [name]"); }
```



9.8. Binding Objects

A `Context` stores its subordinates as a set of `Binding` objects. A binding is an association between an object and its name. Thus, as we've already seen, a `Binding` object contains an object, its name, and its class. We can add a new `Binding` to a `Context` with the `bind()` method. For example, here's how to add a binding for a new file object to an existing `Context`:

```
java.io.File newfile = java.io.File("c:\temp\newfile");
tempContext.bind("newfile", newfile);
```

Now, if we call `list()` on this `Context`, you see a new child named `newfile`. Recall in the previous section we said that you have to drop out of JNDI to create a new file when using the Sun filesystem provider. The previous example shows what we meant. To create a file, use the `java.io.File` constructor, which isn't part of JNDI. To bind the file into the naming system, use the `bind()` method of `Context`.

If you try to bind a name to an object, and the name has already been used, the method throws a `NameAlreadyBoundException`. If you want to bind a new object to an existing name, use the `rebind()` method instead. `Context` also has an `unbind()` method you can use to remove a binding.

9.9. Accessing Directory Services

So far, we've discussed JNDI only in the context of naming services. Now it's time to turn to directory services. At its root, a directory is merely a naming service whose objects have attributes as well as names. Programming for a directory service, such as an LDAP directory, is roughly as hard as programming for a relational database.

As we've seen, a binding in JNDI is an association between a name and an object. While this association is sufficient for some naming services, a directory service needs to be able to associate more than just a name with an object. Attributes associate specialized data with an object. In JNDI, an object with attributes as well as a name is called a *directory entry*.

We've been talking about the filesystem as though it were a naming system because that is how Sun's filesystem provider implements it. But if you think about it, a filesystem is really a directory system; files and directories have attributes like permissions, user IDs, and group IDs (we just can't get at these attributes using Sun's filesystem provider).

9.9.1. The DirContext Interface

`javax.naming.directory.DirContext` is JNDI's directory services interface. It extends `Context` and provides modified methods that support operations involving attributes. Like a `Context`, a `DirContext` encapsulates a set of name-to-object bindings. In addition, a `DirContext` contains a `javax.naming.directory.Attributes` object for each bound object that holds the attributes and values for that object.

The names of objects in X.500-based directories (like LDAP) look a little different from the names we've seen so far for filesystems. If you've worked with an LDAP directory, you've probably seen names like `"cn=Billy Roberts, o=Acme Products"`. This name is actually a compound name while something like `"o=Acme Products"` is an atomic name. By convention, in an LDAP directory, the part of the name before the equals sign (e.g., `"cn"`, `"o"`) is stored as an attribute of the directory entry, and the rest of the name (e.g., `"Billy Roberts"`, `"Acme Products"`) is stored as its value. This attribute is called the *key attribute*. [Table 9-2](#) lists some commonly used key attributes. Note that when a `DirContext` is used with an LDAP directory, it knows its name, unlike a `Context`.

Table 9-2. Common key X.500 attributes

Attribute	Meaning
"c"	A country, such as the United States or Lithuania
"o"	An organization or corporation, such as the Humane Society or Omni Consumer Products

Attribute Meaning

"ou"	A division of an organization, such as the Public Relations Department or the Robotic Peace Officer Division
"cn"	The common name of an entity (often a user, in which case it can be a first name or a full name)
"sn"	The surname (last name) of a user

The key attribute is closely tied to the directory entry's *object class definition*, otherwise known as its type. For example, in an LDAP directory, an entry that has an object class of "user" has a key attribute of "cn" while an object with a class of "organization" has a key attribute of "o".

The schema for a directory defines the object classes that can be used in the directory, analogous to the way that the schema for a relational database defines the data tables that are available in the database. The object class of a directory entry is stored as an attribute of the entry. Note that the values used for object classes are directory-dependent, so a user entry from one directory might have a different object class than a user entry from another directory.

9.9.2. The Attributes Interface

The `Attributes` interface represents the set of attributes for a directory entry. It has accessor methods that enable access to the entire set, as well as to specific attributes. In X.500-based directories, the name of an attribute (also called an attribute ID), such as "name", "address", or "telephonenumber", determines the type of the attribute and is called the *attribute type definition*. An attribute type definition is part of a directory's schema; the corresponding *attribute syntax definition* specifies the syntax for the attribute's value and whether it can have multiple values, among other things.

You can retrieve all attributes of a directory entry by calling the `getAttributes()` method of `DirContext`, followed by the `getAll()` method of `Attributes`. `getAttributes()` returns an `Attributes` object. Calling the `getAll()` method of this object returns a `NamingEnumeration` of `javax.naming.directory.Attribute` objects, one for each attribute of the directory entry.

[Example 9-10](#) shows the implementation of a `listattrs` command for `NamingShell`. This command prints the attributes of a directory entry, as well as string representations of the attribute values.

Example 9-10. The listattrs command

```
import java.util.Vector;
import javax.naming.*;
import javax.naming.directory.*;

class listattrs implements Command {
    public void execute(Context c, Vector v) throws CommandException {

        String name = "";
```

```

// An empty string is OK for a listattrs operation
// as it means list attributes of the current context
if (!(v.isEmpty( )))
    name = (String)v.firstElement( );

if (NamingShell.getCurrentContext( ) == null)
    throw new CommandException(new Exception( ), "No current context");

try {
    // Get the Attributes and then get enumeration of Attribute objects
    Attributes attrs = ((DirContext)c).getAttributes(name);
    NamingEnumeration allAttr = attrs.getAll( );
    while (allAttr.hasMore( )) {
        Attribute attr = (Attribute)allAttr.next( );
        System.out.println("Attribute: " + attr.getID( ));

        // Note that this can return human-unreadable garbage
        NamingEnumeration values = attr.getAll( );
        while (values.hasMore( ))
            System.out.println("Value: " + values.next( ));
    }
}
catch (NamingException e) {
    throw new CommandException(e, "Couldn't list attributes of " + name);
}
catch (ClassCastException cce) {
    throw new CommandException(cce, "Not a directory context");
}
}

public void help( ) { System.out.println("Usage: listattrs [name]"); }
}

```

To use the *listattrs* command, you need access to a live directory server. To experiment with a live LDAP directory server, you might try the University of Michigan's server at <ldap://ldap.itd.umich.edu> or Novell's test server at <ldap://nldap.com>. Another option is to download and compile the OpenLDAP source code from <http://www.openldap.org> and get an LDAP server running on your local network. To use the University of Michigan's LDAP server with `NamingShell`, you need to create a properties file that contains the following properties:

```

java.naming.factory.initial=com.sun.jndi.ldap.LdapCtxFactory
java.naming.provider.url=ldap://ldap.itd.umich.edu/

```

Make sure that the jar file for the LDAP service provider is in the `CLASSPATH` of `NamingShell` when you use this initial context information. If you are using JDK 1.3 or later, this LDAP provider is built into the runtime.

Once you have `NamingShell` set up to use a directory server, here's how you might use the *listattrs*

command:

```
o=NOVELL% listattrs cn=admin
Attribute: groupMembership
Value: cn=DEVNET SYSOP,ou=Groups,o=NOVELL
Attribute: revision
Value: 235
Attribute: uid
Value: admin
Attribute: objectClass
Value: top
Value: person
Value: organizationalPerson
Value: inetOrgPerson
Attribute: sn
Value: admin
Attribute: cn
Value: admin
```

The following code in *listattrs* retrieves the `Attributes` object of the named directory context and enumerates the individual `Attribute` objects:

```
Attributes attrs = ((DirContext)c).getAttributes(name);
NamingEnumeration allAttr = attrs.getAll( );
```

Calling `getAttributes()` with the name of a directory entry returns an `Attributes` object that contains all the attributes for that entry. Another variation of `getAttributes()` allows you to pass the name of a directory entry and an array of attribute names (as `String` objects). This method returns an `Attributes` object that contains only the specified attributes. For example:

```
String[] attrIDs = {"name", "telephonenumber"};
Attributes partialAttrs = dirContext.getAttributes(name, attrIDs);
```

In *listattrs*, we used the `getAll()` method of `Attributes` to return an enumeration of `Attribute` objects. The `Attributes` interface also provides a `getIDs()` method that returns an enumeration of just the attribute names (or IDs) for the directory entry. If you know the attribute you want, you can specify the attribute name in a call to the `get()` method, which returns a single `Attribute` object. For example:

```
Attribute addr = attrs.get("address");
```

9.9.3. The Attribute Interface

The `Attribute` interface represents a single directory attribute. We've already seen this interface in the `listattrs` command, where we used it to print the names and values of all the attributes of a directory context.

An attribute can have a single value or multiple values, as specified in the schema for the directory. For example, a `"name"` attribute might have a single value (e.g., `"Billy"`) while a `"telephonenumber"` attribute might have multiple values (e.g., `"800 555 1212"` and `"303 444 6633"`).

JNDI provides several methods for working with values in an attribute. For instance, we can get one or more values, add or remove a single value, remove all values, or determine if a particular value is present.

The `get()` method of `Attribute` returns a single attribute value as a `java.lang.Object`. If the attribute has only a single value, `get()` returns that value. If the attribute has multiple values, the service provider determines the value that is returned. The following code shows how to get a single value from an attribute:

```
DirContext user ... ; // Created somewhere else in the program
Attributes attrs = user.getAttributes("");
Attribute attr = attrs.get("telephonenumber");
Object onePhoneNumber = attr.get( );
```

The `getAll()` method returns multiple attribute values as a `NamingEnumeration` of objects, as we saw in `listattrs`. Here's how to print all values stored in an attribute:

```
Attribute attr = attrs.get("telephonenumber");
NamingEnumeration phoneNumbers = attr.getAll( );
while (phoneNumbers.hasMore( ))
    System.out.println(phoneNumbers.next( ));
```

The `add()` method of `Attribute` enables us to add another value to an attribute:

```
Attribute attr = attrs.get("telephonenumber");
attr.add("520 765 4321"); // Add a new number
```

If we try to add a value to an attribute that doesn't support multiple values, the method doesn't throw an exception. The attribute simply doesn't accept the new value. By the same token, you can use the `remove()` method to remove a value from an attribute.

```
Attribute attr = attrs.get("telephonenumber");
attr.remove("303 444 6633"); // Remove the old number
```

To remove all the values from an attribute, you can call the `clear()` method. Note that none of

these method calls actually affect the directory entry; they simply modify the local `Attribute` object. To make a permanent change, you have to call the `modifyAttributes()` method of `DirContext` and provide it with a modified `Attribute` object, as discussed in the next section.

The `contains()` method lets you determine whether an attribute has a certain value, while `size()` returns the number of values the attribute has:

```
Attribute attr = attrs.get("telephonenumber");  
// Check for certain value  
boolean itsThere = attr.contains("800 555 1212");  
int valuesItHas = attr.size( ); // Check how many values it has
```



9.10. Modifying Directory Entries

Modifying the attribute values of a directory entry involves using the `modifyAttributes()` method of `DirContext`. One variant of this method takes the name of a directory entry, a modification type, and an `Attributes` object that contains modified `Attribute` objects; another variant takes a name and an array of `javax.naming.directory.ModificationItem` objects. A `ModificationItem` encapsulates a modified `Attribute` object and a modification type.

The only part of this operation that warrants much explanation is the creation of modified `Attribute` objects. The `javax.naming.directory.BasicAttributes` and `javax.naming.directory.BasicAttribute` classes implement the `Attributes` and `Attribute` interfaces, respectively. These are the classes you'll typically use to create modified attribute values.

For example, let's say we want to remove the phone number "303 444 6633" from a user entry's "telephonenumber" attribute and replace it with the new number "520 765 4321". In the following code, we create two `BasicAttributes` objects, `newNumber` and `oldNumber`, and use them in calls to `modifyAttributes()`:

```
DirContext user ... ; // Created somewhere else in the program

BasicAttribute newAttr = new BasicAttribute( );
newAttr.add("telephonenumber", "520 765 4321");
BasicAttributes newNumber = new BasicAttributes( );
newNumber.put(newAttr);
BasicAttributes oldNumber =
    new BasicAttributes("telephonenumber", "303 444 6633");

user.modifyAttributes("", DirContext.REMOVE_ATTRIBUTE, oldNumber);
user.modifyAttributes("", DirContext.ADD_ATTRIBUTE, newNumber);
```

In this code, we use two different techniques to create `BasicAttributes` objects. For `newNumber`, we first create a new `BasicAttribute` and add a "telephonenumber" attribute to it. Then we create a new `BasicAttributes` object and put the `BasicAttribute` in it. With `oldNumber`, we use the convenience constructor of `BasicAttributes` to accomplish the same task in one line of code.

Now we use the two `BasicAttributes` objects in two calls to `modifyAttributes()`, one to remove the old number and one to add the new. `DirContext` defines three constants we can use to specify the type of modification we are doing: `ADD_ATTRIBUTES`, `REMOVE_ATTRIBUTES`, and `REPLACE_ATTRIBUTES`. With any of these types, `modifyAttributes()` uses the ID of each `Attribute` object to determine which attribute to modify by adding, removing, or replacing attribute values. The net result of our two calls is the old number is replaced with the new number. Of course, we could have done this with one call to `modifyAttributes()` if we had used the `REPLACE_ATTRIBUTES` modification type.

The following code shows how to make the same change using a variant of `modifyAttributes()` that

takes an array of `ModificationItem` objects:

```
ModificationItem[] mods = new ModificationItem[2];
mods[0] = new ModificationItem(DirContext.REMOVE_ATTRIBUTE,
    new BasicAttribute("telephonenumber", "303 444 6633"));
mods[1] = new ModificationItem(DirContext.ADD_ATTRIBUTE,
    new BasicAttribute("telephonenumber", "520 765 4321"));
user.modifyAttributes("", mods);
```

Again, this change could have been done with a single `ModificationItem`, using `REPLACE_ATTRIBUTES`. You should be cautious when using `REPLACE_ATTRIBUTES`, however, since it replaces *all* values of the attribute with the value or values you provide.

Note that the examples here don't reflect any particular directory. In order to change a directory's `"telephonenumber"` attribute value, you need to consult the schema of that directory for the appropriate attribute type and syntax definitions.

Note also that we have discussed only modifying existing attribute values, not adding new attributes altogether. The reason is that adding new attribute IDs requires modifying the schema, or type system, of a directory. JNDI supports schema access and modification, but the details on how to do so are beyond the scope of this chapter.

9.11. Creating Directory Entries

So far, we have been accessing directory entries that are already present in the directory. Now it's time to learn how to create directory entries of our own, using the `createSubcontext()` method of `DirContext`. As we discussed earlier, when you create a subcontext of a `Context` object, the service provider controls the type of object that is created. With a `DirContext`, this is not the case; you actually have complete control over the type of object you create with `createSubcontext()` (within the constraints of the directory schema, of course).

As noted earlier, the object class definition determines the type of a directory entry, and the entry stores its object class as an attribute. So, in order to create a directory entry, we must pass the object class attribute and some other attributes into the parent entry's `createSubcontext()` method.

Most directories require that you specify attributes for at least the object class definition (e.g., `"objectclass="`) and key attribute (e.g., common name, `"cn="`) of a directory entry. Often directories require that you specify more attributes than just these. The minimum set of attributes necessary for creating a directory entry is called the *mandatory attributes*. They are mandatory because, if you don't specify them, `createSubcontext()` throws an `InvalidAttributesException`. Other attributes that aren't required, but that add more useful data to the entry, are called *extended attributes*.

Say we have a reference to a `DirContext` called `orgUnit` (where this directory entry lives in an LDAP v3 directory) and we want to create a user entry that is a child of `orgUnit` to represent the network user Billy Roberts. Here's how we can create a user entry for Billy:^[*]

[*] Note that we didn't implement a "create directory entry" command for `NamingShell` because most public access LDAP servers don't allow you to create new entries.

```
DirContext orgUnit = ... ; // Created somewhere else in the program
BasicAttributes mandatory = new BasicAttributes("cn", "Billy");
BasicAttribute objectclass = new BasicAttribute("objectclass", "user");
BasicAttribute surname = new BasicAttribute("surname", "Roberts");
mandatory.put(objectclass);
mandatory.put(surname);

orgUnit.createSubcontext("cn=Billy", mandatory);
```

Note that the `createSubcontext()` method of `DirContext` resembles the `createSubcontext()` method of `Context`; the only difference is the addition of an `Attributes` parameter. In this example, we create a `BasicAttributes` object and put three attributes in it. While all the attribute values here are `String` objects (because that's what an LDAP directory requires), the JNDI API allows you to specify any kind of object as an attribute value.

In this example, `orgUnit` represents an organizational unit, under which Billy Roberts's newly created user entry resides. In an LDAP directory, an organizational unit is an object class definition that represents a division of a company, and a user is an object class definition that represents a person

who uses network resources. It is natural that a division of a company can contain a person, but it doesn't necessarily work in the opposite direction; it doesn't make sense that a user can contain an organizational unit. The LDAP schema dictates these rules and also specifies the values we can use for the `objectclass` attribute (which is where `"user"` came from in the example code).

When you are creating your own directory entries, be sure to consult the schema for the directory you are using. If you attempt to create a type of entry that can't reside under a particular `DirContext` or you specify an incorrect value for the `objectclass` attribute, `createSubcontext()` throws an exception.

[← PREV](#)

9.12. Searching a Directory

One of the most useful features a directory service can offer is the ability to search its entries for attribute values that meet certain criteria. JNDI supports this kind of searching in directory systems, which means you can implement search functionality in your JNDI applications. `DirContext` provides different `search()` methods that allow you to specify what you are searching for and control how the search operates.

9.12.1. Search Criteria

You can specify what you are searching for in two ways. The simpler technique is to create a set of attributes that serves as the search criteria. In this case, you can set an attribute value, meaning that an entry must have that attribute value to match, or leave the value empty so that all entries that have the attribute match no matter what the value.

The more flexible way to specify search criteria is with a search filter string. A search filter allows you to express search criteria using LDAP search syntax, specified in RFC 2254. Note that this syntax works with all JNDI providers, not just LDAP; it's the JNDI standard for searching all kinds of directories. The search filter is a `String` that takes the following general form:

```
(attribute operator value)
```

You can use an asterisk (*) to represent a wildcard. For example, here's how to search for all entries in an LDAP directory:

```
(objectclass=*)
```

A search for all users takes the form of:

```
(objectclass=user)
```

You can also use the wildcard character to represent completion, just as in a Unix shell or a DOS prompt. For example, here's a filter for searching for all users whose first names start with "k":

```
(cn=k*)
```

You can use operators other than equals (=), as in:

```
(revision<24)
```

You can also combine search filters with operators such as AND (&) and OR (|). The way to do this is to wrap the entire expression in parentheses:

```
(&(objectclass=computer)(cn=Billy))
```

Finally, you can nest search expressions:

```
(&( |(objectclass=computer)(objectclass=user))(cn=Billy))
```

Obviously, the attributes you specify in a search depend on the directory service you are searching.

9.12.2. Search Results

Regardless of how you specify the search criteria, the `search()` method you call returns a `NamingEnumeration` of `SearchResult` objects. There is a `SearchResult` for each directory entry that matches the search criteria. `SearchResult` is a direct subclass of `Binding` that stores a set of `Attributes` along with the usual name, class name, and object. (As we'll see shortly, the object in a `SearchResult` may be `null`, depending on the `SearchControls` you set.) Since a search operation returns a `NamingEnumeration`, you must cast the object that the enumeration returns from the `next()` method to a `SearchResult` object. Once you've done this, you can retrieve attributes with the `getAttributes()` method and use methods inherited from `Binding` (and `NameClassPair`) to get other information about the matching entry.

9.12.3. Search Controls

The `search()` methods that take a `SearchControls` object allow you to control how a search operates. You can set the scope of a search, whether the search should return objects, and the maximum amount of time the search should take, among other things. The easiest way to create a `SearchControls` object is to use the default constructor and then call various `set()` methods to set particular search properties.

For example, the `setSearchScope()` method controls where the search should look for matching directory entries. Most of the time, you set the scope of a `SearchControls` object to search an entire subtree, but you can also limit the search to an object or its children. [Table 9-3](#) lists the available search scopes.

Table 9-3. SearchControls search scopes

Scope	Meaning
<code>OBJECT_SCOPE</code>	Searches only the object itself
<code>ONELEVEL_SCOPE</code>	Searches only the children of the search target
<code>SUBTREE_SCOPE</code>	Searches the entire subtree

The `setReturningObjFlag()` method determines whether the results of a search contain references to the actual directory entries or only the names and class names of the entries. The default behavior is to not return the actual entries, meaning that calling `getObject()` on a `SearchResult` returns `null`.

The `SearchControls` object also allows you to specify other aspects of the behavior of a search:

- The number of milliseconds to wait for the directory to return the search results (by default, a search can take as long as it takes)
- The number of entries that can be returned from the search (by default, as many as are present or as many as the underlying directory allows you to retrieve)
- Whether to follow links to finish the search (no by default)
- What attributes if any to return (all by default)

In general, the default behavior is typically what you want for these parameters.

9.12.4. A Search Command

Now that we've discussed how the various `search()` methods work, let's look at a real example. [Example 9-11](#) shows the implementation of a `search` command for `NamingShell`. This example uses the `search()` method that takes the name of the context to be searched, a search filter that describes the search criteria, and a `SearchControls` object.

Example 9-11. The search command

```
import java.util.Vector;
import javax.naming.*;
import javax.naming.directory.*;

class search implements Command {
    public void execute(Context c, Vector v) throws CommandException {
```

```

if (NamingShell.getCurrentContext( ) == null)
    throw new CommandException(new Exception( ), "No current context");
else if (v.isEmpty( ))
    throw new CommandException(new Exception( ), "No filter specified");
String filter = (String)v.firstElement( );
try {
    SearchControls cons = new SearchControls( );
    cons.setSearchScope(SearchControls.SUBTREE_SCOPE);
    NamingEnumeration results = ((DirContext)c).search("", filter, cons);
    while (results.hasMore( )) {
        SearchResult result = (SearchResult)results.next( );
        System.out.println(result.getName( ));
    }
}
catch (InvalidSearchFilterException isfe) {
    throw new CommandException(isfe,
        "The filter [" + filter + "] is invalid");
}
catch (NamingException e) {
    throw new CommandException(e, "The search for " + filter + " failed");
}
catch (ClassCastException cce) {
    throw new CommandException(cce, "Not a directory context");
}
}

public void help( ) { System.out.println("Usage: search filter"); }
}

```

The *search* command always starts searching in the current context, so you need to move to the appropriate location in the directory service using *cd* before you use *search*. *search* requires you to specify a search filter as its first argument. Note that you can't use any spaces in the filter, or the filter will be parsed as multiple arguments and therefore will not work. Here's how to use the *search* command:

```

o=Novell% search (&(objectclass=person)(cn=a*))
cn=admin
cn=admin,ou=cook1,ou=user
cn=admin,ou=fj,ou=user
cn=admin,ou=Stanford,ou=user
cn=admin,ou=Ed Reed,ou=user
cn=admin,ou=antimony,ou=user
cn=admin,ou=keaves,ou=user
cn=admin,ou=acme,ou=user
cn=admin,ou=nld,ou=user
cn=admin,ou=wibble,ou=user
cn=admin,ou=xxx,ou=user
cn=admin,ou=piet,ou=user
cn=admin,ou=adamtest1,ou=user
cn=admin,ou=novell,ou=user

```

...

← PREV

9.13. Event Notification

In addition to finding, browsing, searching, and altering objects stored in a naming or directory service, the JNDI API also supports the concept of event notification. This involves registering listeners for various types of events that may occur in a naming/directory server. This functionality is provided through the `javax.naming.event` package.

Essentially, JNDI event notification is similar to other event models in the Java platform: you obtain a reference to an event source and then register an event listener with the source. When the specified events fire in the event source, notifications are made by invoking callback methods on the event listeners.

9.13.1. Event Sources

Event sources in JNDI are represented by special subclasses of `Context` and `DirContext`: `EventContext` and `EventDirContext`, both from the `javax.naming.event` package. These interfaces provide the additional functionality required to register event listeners with a naming/directory service. Acquiring a reference to an event source is simply a matter of casting a returned `Context` or `DirContext`. For example, if we wanted to get an event source for the "person" branch of a directory service:

```
Context ctx = new InitialContext(props).lookup("ou=person");
EventDirContext evCtx = (EventDirContext)ctx;
```

In order to take advantage of event notification, the JNDI provider (and the underlying naming or directory service) needs to support event notification, and its `Context` implementations need to implement the `EventContext` or `EventDirContext` interfaces. If the provider doesn't support event notification, the cast in the second line of the previous example fails with a `ClassCastException`. There is a more fault-tolerant way to check for event notification support:

```
Context ctx = new InitialContext(props).lookup("ou=People");
if (ctx instanceof EventDirContext) {
    // Register listeners with directory event source
    ...
}
else if (ctx instanceof EventContext) {
    // Register listeners with naming event source
    ...
}
else {
    // No event notification support
```

```

    System.out.println("Sorry, provider doesn't support event notification");
}

```

The LDAP provider supplied with JDK 1.3 and later does support event notification, so the remainder of our examples will assume that we're accessing LDAP services.

9.13.2. Writing Event Listeners

Once you have an event source from your naming/directory service, you can register listeners for various events. Event listeners in JNDI are simply classes that implement the `NamingListener` interface. This interface acts as the base interface for the other types of listeners supported by JNDI, which we'll discuss next. It also supports handling errors that occur in the naming service. If you register a listener with the naming service and the service then encounters some error that will prevent it from notifying you of events, it makes you aware of the error by calling the `namingExceptionThrown()` callback on the `NamingListener` interface.

Two types of events can occur in a naming or directory service. The first involves changes relative to the namespace of the service (i.e., when objects are added or removed from a naming service or an object's name in the service is changed). The second type of event involves changes relative to the actual objects stored in the naming service (i.e., when an object's binding in the naming service has been changed or its attributes (in a directory service) have been altered).

There are two subclasses of `NamingListener` provided in JNDI to match these two types of events: `NamespaceChangeListener` and `ObjectChangeListener`. To be notified of changes in the namespace, create an implementation of the `NamespaceChangeListener` interface and register it with an event source. There are three event notification callbacks on this interface that you can implement. The `objectAdded()` method is called when a new object is added to the namespace, `objectRemoved()` is called when any object is removed, and `objectRenamed()` is called when an object's binding in the namespace is changed. A very simple `NamespaceChangeListener` implementation is shown in [Example 9-12](#). This listener simply logs events to standard output as they occur.

Example 9-12. Listener for namespace changes

```

import javax.naming.*;
import javax.naming.event.*;

public class NamespaceChangeLogger implements NamespaceChangeListener {
    // Default constructor
    public NamespaceChangeLogger( ) {}

    // Callback for object addition events
    public void objectAdded(NamingEvent ev) {
        Binding b = ev.getNewBinding( );
        System.out.println("--> ADD: Object of type " + b.getClassName( ) +
            " added at binding \"" + b.toString( ) + "\"");
    }
}

```

```

// Callback for object removal events
public void objectRemoved(NamingEvent ev) {
    Binding b = ev.getOldBinding( );
    System.out.println("--> REMOVE: Object of type " + b.getClassName( ) +
        " removed from binding \"" + b.toString( ) + "\"");
}

// Callback for object addition events
public void objectRenamed(NamingEvent ev) {
    Binding bNew = ev.getNewBinding( );
    Binding bOld = ev.getOldBinding( );
    System.out.println("--> RENAME: Object of type " + bNew.getClassName( ) +
        " renamed from binding \"" + bOld.toString( ) +
        "\" to binding \"" + bNew.toString( ) + "\"");
}

// Callback for errors in the naming service
public void namingExceptionThrown(NamingExceptionEvent ev) {
    System.out.println(
        "--> ERROR: An error occurred in the naming service:");
    ev.getException( ).printStackTrace( );
}
}

```

Each event notification callback takes a single `NamingEvent` argument. `NamingEvent` is a subclass of the generic `java.util.EventObject` class that is the base class for events in the AWT event model and the JavaBeans event model. A `NamingEvent`, as the name implies, contains information describing an event that occurred in the naming service. The `getOldBinding()` and `getNewBinding()` methods return `Binding` objects that indicate the state of the affected object's binding before and after the event. In the case of add events, the old binding is `null`; in the case of removal events, the new binding is `null`. The `NamingEvent` interface also allows you to access the `EventContext` that fired the event. In our example listener, we simply log the old and new binding when we are notified of add, remove, or change events.

In the case of errors encountered by the naming service, the `namingContext-Thrown()` callback will be called on our listener and given a `NamingExceptionEvent`. This is another subclass of `EventObject`; `NamingExceptionEvent` wraps a `NamingException` that has been thrown within the naming service. In our example, when a naming error occurs, we simply log it to standard output and print out the stack trace for the `NamingException` enclosed in the `NamingExceptionEvent`.

Changes to objects in the naming service are tracked using an implementation of the `ObjectChangeListener` interface. Creating an `ObjectChangeListener` is similar to the `NamespaceChangeListener`: create a subclass that includes concrete implementations of the notification callbacks defined in the interface. For `ObjectChangeListener`, there is only one event callback to implement: `objectChanged()`. You also need to implement `namingExceptionThrown()` to handle any errors. [Example 9-13](#) shows an example of an `ObjectChangeListener` implementation.

Example 9-13. Listener for object change events

```

import javax.naming.*;
import javax.naming.event.*;

public class ObjectChangeLogger implements ObjectChangeListener {
    // Default constructor
    public ObjectChangeLogger( ) {}

    // Callback for object change events
    public void objectChanged(NamingEvent ev) {
        Binding bNew = ev.getNewBinding( );
        Binding bOld = ev.getOldBinding( );
        System.out.println("--> CHANGE: Object of type " + bNew.getClassName( ) +
            " changed, previous binding = \"" + bOld.toString( ) +
            "\" post-change binding = \"" + bNew.toString( ) + "\"");
    }

    // Callback for errors in the naming service
    public void namingExceptionThrown(NamingExceptionEvent ev) {
        System.out.println(
            "--> ERROR: An error occurred in the naming service:");
        ev.getException( ).printStackTrace( );
    }
}

```

Of course, you could also create a single listener that receives both namespace change events and object change events by simply implementing both listener interfaces with a single class.

9.13.3. Registering Event Listeners

Event listeners are registered with an event source by specifying a target (the subset of objects in the naming service for which you want notifications), along with the listener that should receive the notifications. This is done using the `addNamingListener()` methods available on the `EventContext` and `Eventdir-Context` interfaces.

The target objects for a listener can be specified in a few different ways. To define a subset of objects based strictly on their bindings and parent-child relationships in the naming service, you can use the `addNamingListener()` methods on the `EventContext` interface. With these registration methods, the target of a listener is specified by providing the name of an object (in the form of either a string or a `Name` object) and a scope. The name is simply a valid binding within the naming service being used in an LDAP directory it is an X.500-style name such as "ou=people,cn=Smith, John". The scope is indicated by one of three static constants defined on the `EventContext` interface, with the following meanings:

`OBJECT_SCOPE`

Limit the target to just the named object or context.

ONELEVEL_SCOPE

Limit the target to the children of the named context, excluding the context itself. The name must refer to a context when this scope is used.

SUBTREE_SCOPE

Extend the target to include the entire subtree of the naming hierarchy, starting at (and including) the named object or context.

So, for example, if we wanted to listen for object-change events on the entry for "John Smith" in our LDAP directory, we would register one of our `ObjectChangeListener` listeners like so:

```
EventContext ctx = (EventContext)(new InitialContext( ));
ObjectChangeListener ocLogger = new ObjectChangeListener( );
ctx.addNamingListener("ou=people,cn=Smith, John",
    EventContext.OBJECT_SCOPE, ocLogger);
```

If we wanted to be notified for any changes in the namespace in the entire context starting with "John Smith", we might do this:

```
NamespaceChangeListener ncLogger = new NamespaceChangeListener( );
ctx.addNamingListener("ou=people,cn=Smith, John",
    EventContext.SUBTREE_SCOPE, ncLogger);
```

It's also possible to use more elaborate filters to specify the target of your listener, using the `EventDirContext` interface and its overloaded versions of `addNamingListener()`. In these versions, specify the name of an object or context as before, but instead of a simple scope value, specify a search filter that defines the subset of the naming service that you want to target. Specifying your notification target this way is very similar to performing a search on a directory service using the `DirContext.search()` method. The search filter takes the form of a filter string, an optional set of filter arguments (in the form of an `Object` array), and a set of search controls (in the form of a `SearchControls` object). So if we wanted to listen for object change events on any objects in the "people" branch with a surname of "Jim", we could do the following:

```
EventDirContext dirCtx = (EventDirContext)(new InitialContext( ));
ObjectChangeListener ocLogger = new ObjectChangeListener( );
SearchControls sctl = new SearchControls( );
sctl.setSearchScope(SearchControl.SUBTREE_SCOPE);
dirCtx.addNamingListener("ou=people", "(sn=Jim)", sctl, ocLogger);
```

Notice that in this case, we specify the scope of the target using the `setSearchScope()` method on the `SearchControls` that we provide with the filter.

9.13.3.1. A listen command

Returning to our `NamingShell` example, we can use the event notification features of JNDI to add a `listen` command to our shell. We can use this command to register listeners for events in the naming service. First, we'll create a single listener that can be registered for both namespace events and object events, by implementing both the `NamespaceChangeListener` and `ObjectChangeListener` interfaces. This listener, `AnyChangeListener`, is just a merging of the two listeners we created in the previous section and is shown in [Example 9-14](#).

Example 9-14. Listener for any naming events

```
import javax.naming.*;
import javax.naming.event.*;

public class AnyChangeListener
    implements NamespaceChangeListener, ObjectChangeListener {
    // Default constructor
    public AnyChangeListener( ) {}

    // Callback for object addition events
    public void objectAdded(NamingEvent ev) {
        Binding b = ev.getNewBinding( );
        System.out.println("--> ADD: Object of type " + b.getClassName( ) +
            " added at binding \"" + b.toString( ) + "\"");
    }

    // Callback for object removal events
    public void objectRemoved(NamingEvent ev) {
        Binding b = ev.getOldBinding( );
        System.out.println("--> REMOVE: Object of type " + b.getClassName( ) +
            " removed from binding \"" + b.toString( ) + "\"");
    }

    // Callback for object rename events
    public void objectRenamed(NamingEvent ev) {
        Binding bNew = ev.getNewBinding( );
        Binding bOld = ev.getOldBinding( );
        System.out.println("--> RENAME: Object of type " + bNew.getClassName( ) +
            " renamed from binding \"" + bOld.toString( ) +
            "\" to binding \"" + bNew.toString( ) + "\"");
    }

    // Callback for object change events
    public void objectChanged(NamingEvent ev) {
        Binding bNew = ev.getNewBinding( );
        Binding bOld = ev.getOldBinding( );
        System.out.println("--> CHANGE: Object of type " + bNew.getClassName( ) +
```

```

        " changed, previous binding = \"" + bOld.toString( ) +
        "\" post-change binding = \"" + bNew.toString( ) + "\"");
    }

    // Callback for errors in the naming service
    public void namingExceptionThrown(NamingExceptionEvent ev) {
        System.out.println(
            "--> ERROR: An error occurred in the naming service:");
        ev.getException( ).printStackTrace( );
    }
}

```

Our *listen* command takes one required argument, the name of the target object or context, and an optional search filter argument that can further define the target of the listener. The command class is shown in [Example 9-15](#).

Example 9-15. The listen command

```

import java.util.Vector;
import javax.naming.*;
import javax.naming.event.*;
import javax.naming.directory.*;

class listen implements Command {
    public void execute(Context c, Vector v) throws CommandException {

        if (NamingShell.getCurrentContext( ) == null)
            throw new CommandException(new Exception( ), "No current context");
        else if (v.isEmpty( ))
            throw new CommandException(new Exception( ), "No target specified");
        String name = (String)v.firstElement( );
        String filter = null;
        if (v.size( ) > 1) {
            filter = (String)v.elementAt(1);
        }

        try {
            // Cast context to an event context
            EventContext evCtx = (EventContext)c;
            // Create our listener
            NamingListener listener = new AnyChangeLogger( );
            // If no filter specified, just register a listener using EventContext
            if (filter == null) {
                evCtx.addNamingListener(name, EventContext.OBJECT_SCOPE, listener);
            }
            // If we have a filter, use the EventDirContext to specify the target
            else {
                EventDirContext evDirCtx = (EventDirContext)c;

```

```

        evDirCtx.addNamingListener(name, filter, null, listener);
    }
    System.out.println("Registered listener for " + name +
        (filter != null ? (" and filter " + filter) : ""));
}
catch (ClassCastException cce) {
    cce.printStackTrace( );
    throw new CommandException(cce,
        "The current context does not support event notification.");
}
catch (NamingException e) {
    throw new CommandException(e, "The search for " + filter + " failed");
}
}

public void help( ) { System.out.println("Usage: listen name [filter]"); }
}

```

The `execute()` method checks the incoming command arguments to see what kind of target is being specified. If just a single argument is given, it's assumed to be the name of the object or context target, and we use the `EventContext` interface to register one of our listeners. If a second argument is provided, it's assumed to be a search filter string, and we use the `EventdirContext` interface to register the listener using both the name and filter to specify the target. If we are unable to cast the current `Context` to an `EventContext` or `EventdirContext`, then we assume that the naming service doesn't support event notification, and report that to the user by throwing a `CommandException`.

With this command, we can use the `NamingShell` to register listeners in our naming service. For example, if we're connected to an LDAP-based person directory and want to listen for any events associated with a person with a UID of `jsmith`:

```

> java com.oreilly.jent.jndi.NamingShell
Created initial context using jndi.properties
/% cd ou=People
Current context now ou=People
ou=People% listen uid=jsmith
Registered listener for uid=jsmith
ou=People%

```

Chapter 10. J2EE Security

Security is paramount in any application, especially in large-scale, distributed J2EE applications. Although J2EE security is critical, it is only one piece of a much larger picture. Security requires an enterprise strategy that addresses physical security, application security, and network security. J2EE security is but one link in that larger chain.

When developing J2EE applications, security is a critical concern. You want to make sure that only authorized users access the application and that hackers can't steal sensitive data. This chapter describes how you can secure your J2EE application.

The chapter begins by providing an overview of transport and application security as well as defining the important concepts of authentication and authorization. It then explains security in the web and application tiers. The chapter explains the concepts of programmatic security as well as declarative security and then goes on to address various specific enterprise security concerns. Finally, we conclude this chapter by highlighting some of the limitations of the current J2EE security model.

Before we get started, here's a caveat. In order to cover J2EE security in some detail in this chapter, we assume that you are familiar with numerous security concepts and techniques, including authentication, authorization, encryption, message digests, and digital signatures, to name a few. Depending on your background, you may want to do some further reading in this area as well as begin monitoring some important online resources.

10.1. Basic Security Concepts

Applications need multiple layers of security. Roughly speaking, these can be described as two high-level layers: *transport layer security* and *application layer security*. Transport layer security refers to protection of data while it's in transit across the network. Application layer security ensures that only authorized users gain access to the application and its data.

Before we discuss the issues related to transport and application layer security, let's have a quick primer on public-key cryptography, which plays an important role in common solutions to both areas of security concerns.

10.1.1. Public-Key Cryptography

Public-key cryptography is a powerful approach for secure operations that can be used for data encryption, digital signatures, and authentication and in various other security contexts. In the public-key cryptography model, each party participating in a secure exchange has two keys: a public key and a private key that fit together mathematically. The public key can be widely and openly published. Private keys, on the other hand, must be carefully guarded, since they can be used to create digital signatures. Public and private keys are used in conjunction with each other to perform highly secure operations. An encrypted data exchange, for example, is performed by encoding a message with the recipient's public key. It can then be decrypted only with the recipient's private key. In this way, only the intended recipient can decrypt the message.^[*]

[*] Where digital signatures are not used, a shared key can be used to encrypt data instead.

Public-key methods are also a critical element of digital signatures. A *digital signature* is an encoded piece of information included with a message that ensures the recipient that the message came from the sender, not from some malicious third party impersonating the sender. The sender of a message takes the message contents and uses an algorithm called a one-way hash to generate a hash value for the message. This hash value is then encrypted with the sender's private key to generate the signature for the message. Like any public-key-encrypted message, the signature can be decrypted only with the sender's public key, which, as mentioned earlier, can be freely shared. Both the original message and the signature for the message are sent to the recipient together. Once received, the recipient can decrypt the signature using the sender's public key. Running the same one-way hash on the message should produce an identical hash value. If the hashes match, the integrity of the message is intact; the message was not modified in transit. If the message contents had been altered at some point between the sending and receiving of the message, the hash values would not match. In addition, since the hash was proven to be encrypted by the sender's private key, we know that the message came from the owner of the private key.

In the context of J2EE applications, the senders and recipients in this transaction are often components or systems rather than people, but the basic idea is the same.^[*] However, it is important to point out that, while public-key cryptography is highly secure for end user authentication, creating a public-key infrastructure (PKI) that issues, expires, and revokes user certificates is daunting, and installing them on the client side, whether in hardware or software, also takes work. Only in some industries in which privacy is paramount can the business case for full-

blown PKI be made. However, certificates are commonly used on a server-to-server basis to provide SSL encryption between applications, for example.

[*] It is easier to explain the concepts in this section using people as illustrations, but bear in mind that the user might be a component or class rather than a person.

10.1.2. Transport Layer Security: Integrity and Confidentiality

Security at the transport layer ensures the *integrity* as well as the *confidentiality* of the data as applications exchange the data over the wire.

Integrity protection ensures that data is not modified in transit, guaranteeing that the data sent is the data received. One technique for ensuring data integrity over the wire is the use of digital signatures in conjunction with message digests.

Confidentiality protection ensures that unauthorized parties cannot read data if they intercept it. The primary means to ensure this is to scramble, or encrypt, the data. One technique for ensuring the confidentiality of data in transit is Secure Sockets Layer (SSL).^[*]

[*] SSL has become standardized as Transport Layer Security (TLS), defined in RFC 2246. SSL remains the more common term, and so we use it in this chapter.

10.1.3. Application Layer Security: Authentication and Authorization

Two fundamental concepts in application security are *authentication* and *authorization*.

Authentication ensures that a user is who she says she is. Authorization assures that the user can access only the portions of the program and the data to which she is entitled.

Authentication refers to the process that a user or program uses to identify itself to the application. Authentication typically involves something you have, something you know, or something you are. Passwords, the most common authentication method, involve something you know. Digital signatures involve something you have a private key, whether stored on a hard drive or on a smartcard and something you know, the passphrase that guards the private key. *Biometric authentication* involves technology such as fingerprint readers, facial or voice recognition systems, and retinal scans. This approach is based on something you are (the whorls on your fingerprints or the modulation patterns in your voice). Biometric authentication is more rare, even in enterprise applications, but it exists and is becoming more broadly available. Some laptops even offer fingerprint-based authentication to access the computer system.

Authorization ensures that users have access only to the resources that they are allowed to access and may regulate what they can do (read but not modify data, initiate process X but not Y, etc.). The application server allows only authorized users access to given resources; otherwise, access to the resource is restricted. In a web application, these resources can be URLs or URL patterns. In the EJB tier, these resources can be components or a component's methods.

The problem with authorization is that specifying what resources each individual user can access is cumbersome. As the application changes, as people come and go, and as user roles change over time, all these settings must be kept up-to-date. In recognition of this security management issue, J2EE security is role-based. Rather than specifying authorizations on a per-user basis, you define

them for a set of well-defined roles, such as customer, administrator, salesperson, student, and professor. Users are given permissions in the system by granting them one or more of these roles. Creating categories of users greatly streamlines authorization because you've defined the allowed combinations of permissions in one place (the role definitions), and a given user's authorizations are represented as a simple list of roles that he's been assigned.

Role-based authentication involves users, roles, permissions, and resources. A *user* provides a username and authentication credentials of some kind to the application. At that point, the authenticated user is considered by the application to have an identity, which is the username. Users have one or more roles. This classification of users makes it relatively easy to create fine-grained authorization policies. Instead of specifying authorizations for each individual user, you specify authorizations for a role. Therefore, a *role* is a grouping of *permissions* that a given type of user possesses. You typically specify authorizations as policies. You specify these policies, in a declarative programming model that we will discuss later, in special policy files, deployment descriptors, or special-purpose policy stores. An Access Control List (ACL) maps roles to resources. We will provide more details about how to do this for various kinds of applications later in this chapter.

Application server vendors may provide additional security services beyond the ones discussed earlier, such as auditing and nonrepudiation. Auditing services log security information for later review. These logs can be used to detect malicious attempts to break into applications, among other things. Nonrepudiation facilities provide proof of user actions in such a way that the user cannot later deny it (obviously a particularly good feature for business transactions).

Now that we've outlined these terms and concepts, the next section describes how the Java platform implements these basic security features. The J2EE security model leverages Java platform security.

10.2. A Look at Java and J2EE Security Standards

J2EE security builds on the foundation of the Java security APIs. These APIs include application-level interfaces such as the Java Authentication and Authorization Service (JAAS) and the Java Authorization Contract for Containers (JACC) as well as lower-level APIs such as the Java Cryptography Extension (JCE), the Java Secure Socket Extension (JSSE), and the Java Generic Security Services (GSS) API. While the nuts and bolts of these standards are often hidden from the developer, it helps to understand what security pieces the standards provide, especially as you evaluate application servers or security providers, for example.

10.2.1. Authentication and Authorization in Java Security

The core Java security model is based on *.policy* files that govern the allowed operations for specific pieces of code. These pieces of code are defined based on their packages and classes, where the code originated, and who (if anyone) signed the code. This level of security is often called *code-level* security, since you are authorizing chunks of code to do specific things rather than the users who are interacting with the code.

JAAS enhances the Java security model by providing user-level security. JAAS deals with authenticating users (referred to in JAAS as *subjects*) and authorizing them to run certain programs, applets, or jars based on a variety of criteria, all specified in the same *.policy* file used for code-level security. JAAS is a required element in both the 1.3 and 1.4 versions of the J2EE specification, so any compliant application server will offer support for it. The authentication aspects of JAAS are highly applicable to J2EE applications since they allow for pluggable runtime authentication, which is useful for developing and deploying applications in various environments using different security infrastructures. Standalone Java clients can also use JAAS for authentication, provided that their runtime environment has a JAAS implementation available.

The authorization features of JAAS allow you to authorize users to perform specific actions within an application. A set of low-level permissions are available out of the box with JAAS, but you can implement your own context-specific (or even application-specific) permissions by subclassing the `java.security.Permission` class. Permissions can be specified to restrict users to specific URLs in a web application, for example, or to perform certain administrative actions on a financial system.

The Java Authorization Contract for Containers (JACC) is a newer security API, introduced as a standard part of J2EE as of the 1.4 version of the specification. JACC builds on JAAS in three fundamental ways. First, it provides a set of standard JAAS/Java security permission types that are directly relevant to enterprise applications (e.g., standard permissions related to web resources and EJB business methods). Next, it provides a standard model for role-based authorization, grouping sets of permissions into named roles, such as customer or salesperson, that can be applied to users within policy definitions. Last, JACC defines a standard service provider API (SPI) that allows J2EE containers and servers to plug in their own authorization providers into the standard JACC model. This pluggability layer also allows you to integrate third-party authorization providers into your application server. [\[*\]](#)

[*] J2EE 1.4-compliant application servers such as Apache Geronimo, BEA WebLogic 9, and IBM WebSphere 6 support JACC; Tomcat, for example, does not.

10.2.2. Encryption in Java Security

JCE is a lower-level security API aimed at data protection needs in applications. JCE is a pluggable framework for encryption, key generation and agreement, and Message Authentication Code (MAC) algorithms. Sun provides a default implementation of JCE, but several third-party implementations of the framework and of specific cryptographic algorithms are available in both commercial and open source products. Application server vendors may implement underlying security mechanisms by leveraging JCE, but in general JCE has a limited role in the security aspects of the J2EE specification. However, JCE can be a very useful tool in your application code. JCE can be used to encrypt sensitive information such as passwords or credit card numbers, in transport or in storage.

JSSE is also related to data protection through encryption. It is a Java implementation of the SSL protocol. Like JCE, application servers and containers may or may not use JSSE in their internal implementations, in this case for SSL network communications. Some versions of Tomcat, for example, use JSSE internally to implement SSL-enabled HTTP communications. JSSE can also be used by both server- and client-side application code to execute secure socket communications directly.

The Java GSS-API defines and implements Java mappings for GSS-API features including support for Kerberos authentication.

10.3. Declarative Security Versus Programmatic Security

Two basic approaches to implementing security are programmatic security, in which you embed explicit security calls directly into your code, and declarative security, in which you specify security policies using deployment descriptors and configuration files.

When making use of programmatic security, you embed security enforcement within the application with programmatic checks on policies, to determine what a user is allowed to do or see in the application, and then react to this information appropriately. Programmatic security can be useful, even required in some situations, especially in application contexts in which the expressive power of declarative security is insufficient. But programmatic security should be used with care. If the design and implementation are not clever and the developers are not careful, such checks become scattered and strewn throughout the application code. This situation can become extremely difficult to maintain because of duplicated code and nonuniform implementation styles. For example, a change in a particular piece of business logic might require changing several pieces of code. This is expensive, because there is more code to update and test, and also error-prone, because it's difficult to ensure that you've covered all the bases when the code is implemented and distributed in an ad hoc fashion. So with programmatic security, the onus is on you to architect your security code in such a way that it's contained and well encapsulated.

Declarative security involves the use of external deployment descriptors and configuration files to describe the authentication and authorization requirements for your application code. The application server or container is responsible for applying these security constraints to your code at runtime. Implementing security in this manner can result in more straightforward and maintainable code, because there's less of it. However, in many situations, declarative security is not sufficient and you still need to add some carefully designed programmatic security in order to achieve all of your application's security requirements.

At this point, having laid this groundwork, let's look at how J2EE security works in both web-based and EJB applications.

10.4. Web Component Security

J2EE provides comprehensive support for security in the web tier including transport-level security with SSL as well as authentication and authorization for application-level security. These security services are provided as both declarative security features (configured through *web.xml*/deployment descriptor elements) and programmatic security calls integrated with the Servlet and JSP APIs.

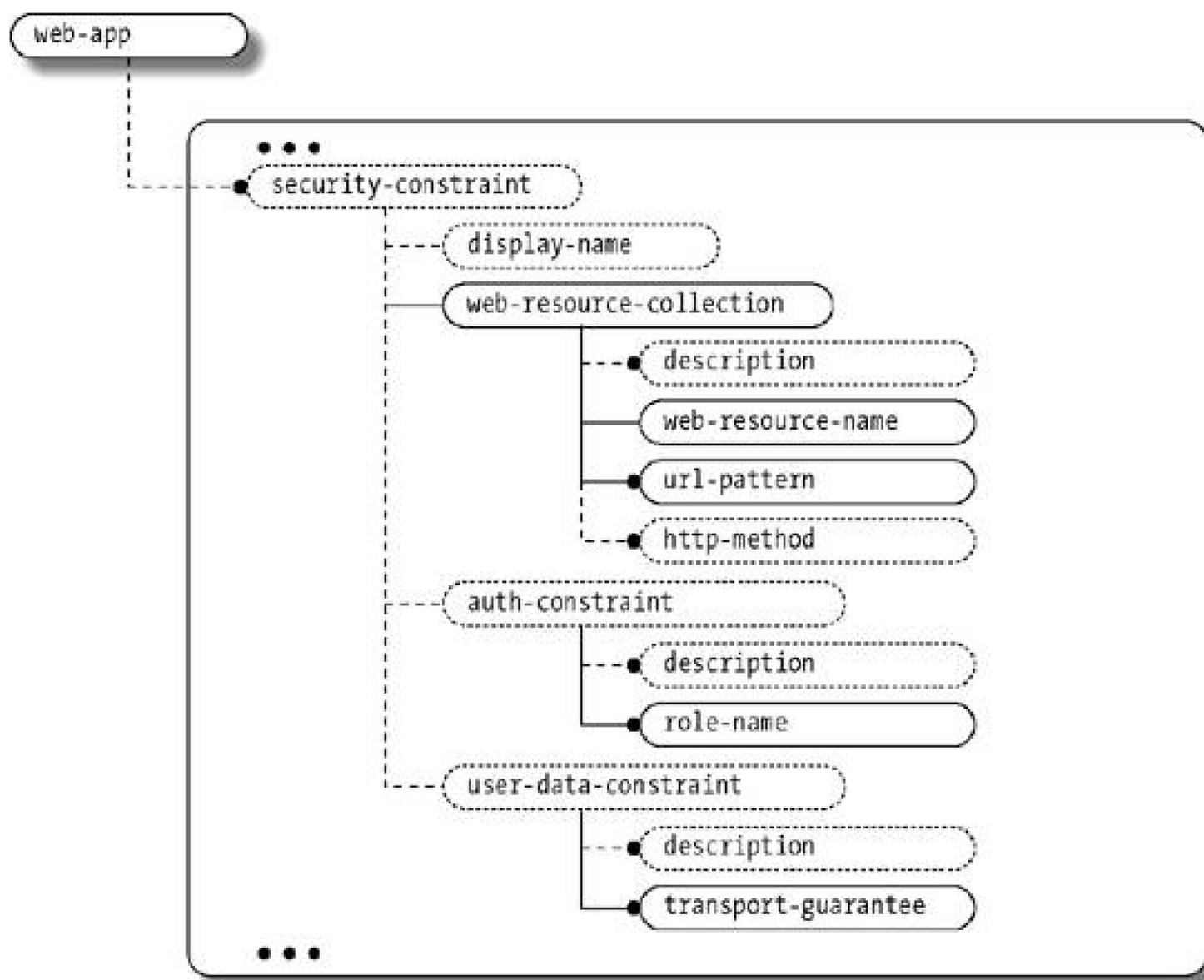
10.4.1. Web-Tier Transport-Level Security

The web tier provides fine-grained control over transport-level security. There are three modes of transport-level security: **NONE**, **INTEGRAL**, and **CONFIDENTIAL**. **NONE**, as the name implies, means no transport-level security; all information is sent in the clear. This is the default mode. **INTEGRAL** means that the client and the server cooperate to ensure that content is not changed in transit. **CONFIDENTIAL** means that the client and server keep the content secret as it is transported over the wire. **INTEGRAL** and **CONFIDENTIAL** imply using SSL to communicate between the clients (typically browsers) and the server. Such transport-level security is also known as *one-way SSL*. You specify the transport-level security constraints in the *web.xml*/deployment descriptor as part of the `<security-constraint>` elements as depicted in [Figure 10-1](#).

The `<security-constraint>` element uses the `<web-resource-collection>` element to specify the subset of the application that is the target of the constraint. The `<web-resource-collection>` element also supports specification of specific HTTP request types, such as GET and POST. You can specify the URL collection for the constraint using a fully specified URL (e.g., `/admin/edit.jsp`) or you can use the `*` wildcard to specify groups of URLs, such as all URLs that fall within a particular path (`/admin/*`) or all URLs that end with a particular suffix (`/admin/*.jsp`). The rules for matching URLs to `url-pattern` values within `<web-resource-collection>` entries are the same as those used for matching `<servlet-mapping>` entries and `<filter-mapping>` entries in web deployment descriptors. Refer to [Chapter 3](#) for full details on these mapping preferences.

What happens if you access an **INTEGRAL** or **CONFIDENTIAL** resource using HTTP instead of HTTPS? There are at least two alternatives. The first alternative is to display an error informing the user that access to the resource is restricted. The second alternative is to redirect the request using the HTTPS protocol. The second approach is seamless and elegant in terms of the user experience. Recent servlet specifications mandate such redirects. [Example 10-1](#) shows a security constraint. The constraint specifies a web resource collection that encapsulates all URLs that start with `/admin`. Therefore, the constraint will match a URL such as `/admin/manageRegions.jsp`. The `<user-data-constraint>` entry specifies that this collection of URLs must be protected to the **CONFIDENTIAL** level. This means that the URLs must be accessed over SSL. If a user attempts to access `/admin/manageRegions.jsp` over HTTP, for example, the web container triggers a redirect to the same URL over HTTPS in order to enforce this constraint.

Figure 10-1. Security constraint schema diagram



Example 10-1. Example security constraint configuration

```

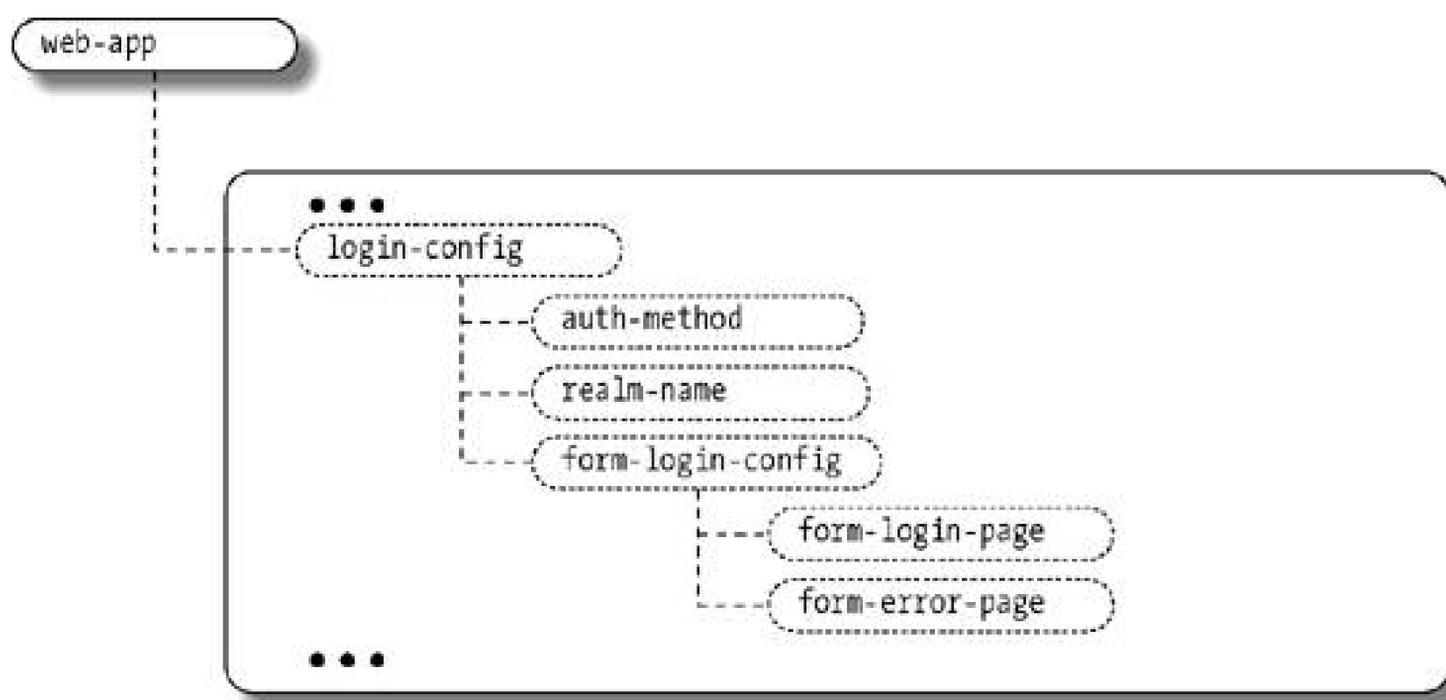
<web-app>
  ...
  <security-constraint>
    <display-name>The user must access these over SSL</display-name>
    <web-resource-collection>
      <web-resource-name>AdminURLs</web-resource-name>
      <description>Collection of admin urls</description>
      <url-pattern>/admin/*</url-pattern>
      <http-method>GET</http-method>
      <http-method>POST</http-method>
    </web-resource-collection>
    <user-data-constraint>
      <description>Only confidential is allowed.</description>
      <transport-guarantee>CONFIDENTIAL</transport-guarantee>
    </user-data-constraint>
  </security-constraint>
  ...
</web-app>

```

10.4.2. Web-Tier Authentication

J2EE web containers also support the declaration of authentication schemes for applications. The standard HTTP authentication mechanisms available in the web tier include **BASIC**, **DIGEST**, **FORM**, and **CLIENT**. The authentication mechanism is specified in the *web.xml* file using the `<login-config>` element shown in [Figure 10-2](#). The authentication method applies to all web components in the application.

Figure 10-2. Login configuration schema diagram



10.4.2.1. BASIC authentication

[Example 10-2](#) shows a *web.xml* fragment that specifies **BASIC** authentication for a web application. With **BASIC** authentication, when a user interacts with a page that requires authentication, the underlying HTTP server prompts the user with a dialog box requesting a username and password. The browser controls the look and feel of the dialog, which is one of the limitations of **BASIC** authentication. The username and password are base64 encoded when sent to the server. Base64 encoding simply obscures the text so that it does not pass in the clear; it does not encrypt it. Transport-level security such as SSL should be used to ensure that the usernames and passwords are sent securely.

Example 10-2. BASIC authentication

```
<web-app>
```

```

...
<login-config>
  <auth-method>BASIC</auth-method>
</login-config>
...
</web-app>

```

10.4.2.2. DIGEST authentication

In an application configured with **DIGEST** authentication, the user experience is very similar to **BASIC** authentication. A request to a protected page triggers a dialog box that asks the user for a username and password. Again, the browser controls the look-and-feel of the dialog. The client base64 encodes the username but encrypts the password. This mechanism is more secure than **BASIC** authentication, but compatibility problems between different HTTP server implementations and web browsers have limited its adoption in web applications.

10.4.2.3. FORM authentication

[Example 10-3](#) shows a declaration of **FORM** authentication for a web application. In an application configured with **FORM** authentication, a developer creates a login screen, *login.jsp* in this example, that displays an HTML form to prompt the user for a username and password. When the user requests a page that requires authentication, the web container redirects her to this login page. The user fills out the login form and submits the request to the server. This mechanism is not secure from a transport level, unless used in conjunction with SSL or other network encryption protocols. One of the neat features of **FORM** authentication is that the server remembers the requested page, including all request parameters. Upon successful authentication, the application server directly takes the user to the authenticated page originally requested. If authentication fails, the application server takes the user to a customized login failure page, *loginError.jsp* in this example.

Example 10-3. FORM authentication configuration

```

<web-app>
...
<login-config>
  <auth-method>FORM</auth-method>
  <form-login-config>
    <form-login-page>/login.jsp</form-login-page>
    <form-error-page>/loginError.jsp</form-error-page>
  </form-login-config>
</login-config>
...
</web-app>

```

What action should you specify for the login form, and who implements that action? As shown in the HTML fragment in [Example 10-4](#), the application server implements a special internal action called

`j_security_check`, and you specify this special action as the target of your login form. How does the `j_security_check` action know about the username and password fields? It expects you to use the standard name of `j_username` for the user's identifier and the standard name of `j_password` for the password field.

Example 10-4. Login form for FORM-based authentication

```
<form action="j_security_check" method="POST">
  <table>
    <tr>
      <td>Username:</td>
      <td><input type="text" name="j_username"></td>
    </tr>
    <tr>
      <td>Password:</td>
      <td><input type="password" name="j_password"></td>
    </tr>
    <tr>
      <td><input type="submit" value="Login"></td>
      <td><input type="reset" name="Reset"></td>
    </tr>
  </table>
</form>
```

10.4.2.4. CLIENT authentication

When a user interacts with a page that requires authentication, when using `CLIENT` authentication, the client browser sends the client's digital certificate to the server. This method of authentication works in conjunction with the transport-level security. The browser never prompts the user for a username and password. The `CLIENT` authentication approach is also known as *two-way SSL*. `CLIENT` authentication is the most secure method among the standard methods offered by web containers, but it is not widely implemented since it mandates issuing digital certificates to all clients that access the application.

10.4.2.5. Custom authentication

Another option for implementing authentication in your web application is to implement your own scheme within the general web component model. In other words, rather than use the declarative authentication configuration offered by the web deployment model, you can write your own login scheme. One common approach to custom web authentication is to write a servlet filter that checks for credentials on the request (cookies or session data defined by the filter, for example). If none are found, the filter logs in the user, perhaps by forwarding him to a custom login page or component.

10.4.3. Web-Tier Authorization

Web containers support both declarative and programmatic authorization, as described in the following sections.

10.4.3.1. Declarative authorization

Declarative authorization in the web tier allows you to declare authorization policies for collections of URLs in your web application. As with the other declarative security features in J2EE, these policies are defined in a deployment descriptor. In this case, since we're discussing web components, the declarative authorization is done in the *web.xml*/deployment descriptor.

You specify declarative authorization in *web.xml* using the `<security-constraint>` element, the same element that is used to specify transport-level security settings. Refer back to [Figure 10-1](#) for the schema diagram of the `<security-constraint>` element. The `<security-constraint>` element specifies which roles are allowed to access specific sets of web resources; the application server enforces these constraints. As shown in [Example 10-5](#), the resources are specified using the same `<web-resource-collection>` element used for transport constraints. The roles that are allowed to access the collection of URLs are specified using the `<auth-constraint>` element. You can specify roles by name, or you can use an asterisk (*) to match all the roles in the deployment descriptor.

Example 10-5. Security constraint example

```
<web-app>
  ...
  <security-constraint>
    <display-name>Seller constraints</display-name>
    <web-resource-collection>
      <web-resource-name>SellerURLs</web-resource-name>
      <description>Collection of seller urls</description>
      <url-pattern>/seller/*</url-pattern>
      <http-method>GET</http-method>
      <http-method>POST</http-method>
    </web-resource-collection>
    <auth-constraint>
      <description>Only individuals with seller role.</description>
      <role-name>seller</role-name>
    </auth-constraint>
  </security-constraint>
  ...
</web-app>
```

When an unauthenticated user accesses a restricted resource, the application server rejects the request by returning an HTTP 401 status code. The 401 status code (`SC_UNAUTHORIZED`) indicates that the server must authenticate the user in order to process this request. The return of a 401 status code causes the browser to prompt the user for authentication (typically a username and password). If the user is not authorized to access the resource (because her role does not give her permission to access that resource), the application server rejects the request by returning an HTTP 403

(`SC_FORBIDDEN`) status code.

10.4.3.2. Programmatic authorization

The primary methods that support programmatic authorization in the web tier are `getRemoteUser()`, `getUserPrincipal()`, and `isUserInRole()` on the `javax.servlet.http.HttpServletRequest` interface. The `getRemoteUser()` method returns the username of the currently authenticated user as a string. The `getUserPrincipal()` method returns similar information to `getRemoteUser()`, but instead of returning a string, it returns a `java.security.Principal` object. What if there is no currently authenticated user? The `getRemoteUser()` and `getUserPrincipal()` methods return `null`, and the `isUserInRole()` method returns `false`.

The `isUserInRole()` method takes a role in a string form. You define the role in the `web.xml` file using a `<security-role>` element and then reference it within a web component declaration using a `<security-role-ref>` element as depicted in [Figure 10-3](#).

Figure 10-3. Security role reference schema elements

Let's look at an example that illustrates programmatic authorization. Your company sells widgets online. You have regular buyers and special VIP buyers. You would like to offer a significant discount, say 20%, to the VIP buyers. One way to do this is to assign VIP buyers a special `vip` role. This role is specified in the `web.xml` file in a `<security-role>` element as shown in [Example 10-6](#). [Example 10-7](#) shows how the application gives the discount to VIP buyers in the `DiscountServlet` class. As you can see, role-based authorization can do more than simply admit or bar access to a given resource; it has business applications as well, especially when combined with programmatic security.

Example 10-6. Security role reference

```

<web-app>
  ...
  <servlet>
    <description>The discount servlet.</description>
    <servlet-name>DiscountServlet</servlet-name>
    <servlet-class>com.oreilly.jent.securitydemo.servlets.DiscountServlet
    </servlet-class>
    <security-role-ref>
      <role-name>VIPRole</role-name>
      <role-link>vip</role-link>
    </security-role-ref>
  </servlet>
  ...
  <security-role>
    <description>The VIP role.</description>
    <role-name>vip</role-name>
  </security-role>

```

Example 10-7. Programmatic authorization code

```

public static String VIP_ROLE = "VIPRole";
String aDiscount = "10";
if (aRequest.isUserInRole(VIP_ROLE)) {
  aDiscount = "20";
}
...

```

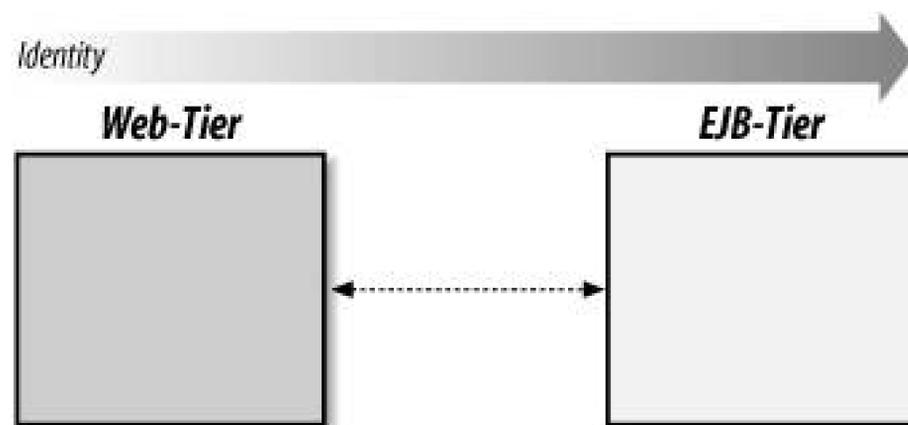
You specify the roles in the deployed application using `<security-role>` elements. These elements declare roles that must be resolved to actual user principals and groups by the underlying application server or container. However, you may want or need to use these roles under different names in the application code itself, and one security role can be referenced multiple times under different names by various web components. In our example, we've linked the role name used in the code, `VIPRole`, to the deployed role name, `vip`. `VIPRole` is specified in the `<role-name>` element of the `<security-role-ref>`. You map the role name to the actual deployed role, using the `<role-link>` element. Using a `<security-role-ref>` is optional if the role name in the code is the same as the role name in the deployed environment. However, it's a good idea to specify the `<security-role-ref>` element anyway, using identical values for the `<role-name>` element and the `<role-link>` element. This makes it easier to adjust deployed role names in the deployment descriptor when needed, and it also makes it clear in the deployment descriptor which roles are used by which web components.

10.4.4. Identity Propagation from the Web Tier to the EJB Tier

We've discussed programmatic and declarative authentication as well as authorization in the web tier. Frequently, though, the web tier is not completely independent of the EJB tier. One common scenario is for the web tier to act as a client of one or more EJB components, delegating business behavior to the EJB tier. In order for security to work seamlessly across tiers, we need to pass along the user's credentials from the web tier to the EJB tier. This technique is called *identity propagation*.

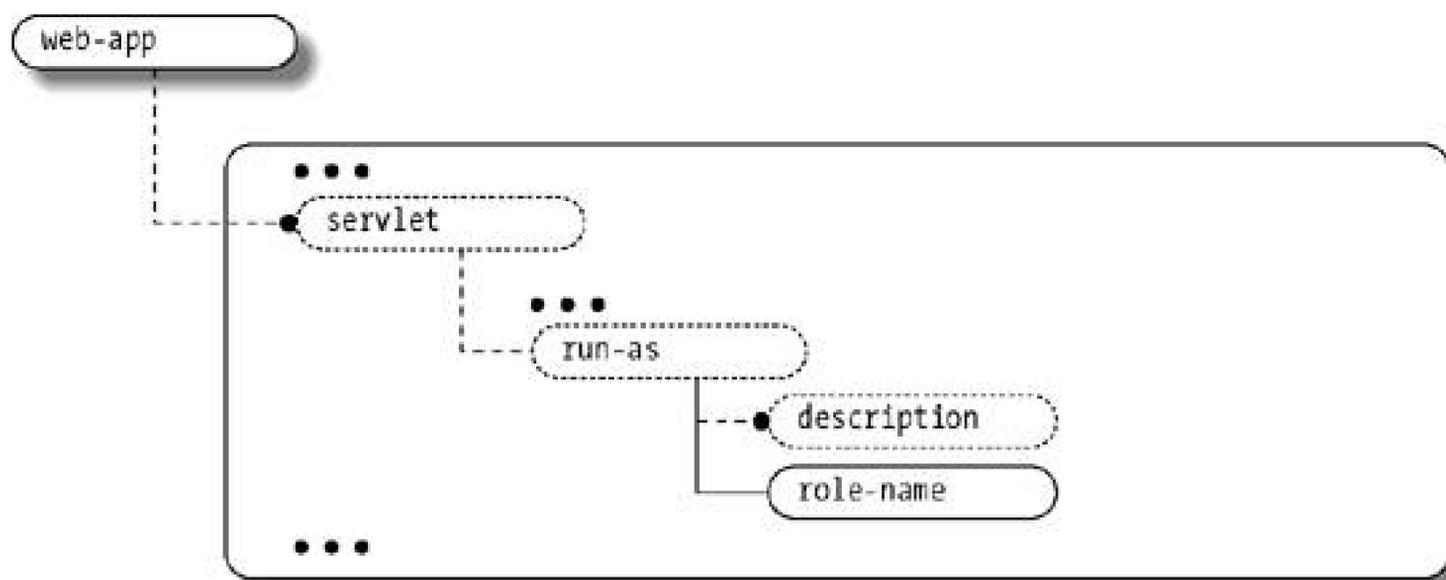
In the EJB tier, clients are responsible for authenticating users. For example, if the client is a web application, the web tier specifies the mechanisms for authentication. Once the application server authenticates a user in the web tier, his identity is established. The application server passes along this identity to the EJB components as depicted in [Figure 10-4](#).

Figure 10-4. Propagation from the web tier to the EJB tier



Identity propagation, passing along the credentials from the web tier to the EJB tier, is the default and works well in most applications. However, in some circumstances, the components in the EJB container do not care about the client's identity but only the client's role. In other situations, the web application may allow unauthenticated users to access certain web components, but the EJB components require a role to be associated with all incoming requests. In such scenarios, you can use the `<run-as>` element in the declaration of a web component within the deployment descriptor. The position of the `<run-as>` element in the `web.xml` structure is depicted in [Figure 10-5](#). When a role is specified in a `<run-as>` element, all user requests of the component are run under the specified role, and this role is propagated to any EJBs accessed by the web component.

Figure 10-5. run-as element in web.xml schema



[Example 10-8](#) illustrates the use of `<run-as>` within a deployment descriptor. In this example, we declare that the `TaxServlet` does not pass along the caller identity. Instead, the `TaxServlet` passes the role name `TaxServiceUserRole` to any EJB components that it invokes.

Example 10-8. run-as

```

<web-app>
  ...
  <servlet>
    <description>The tax servlet.</description>
    <servlet-name>TaxServlet</servlet-name>
    <servlet-class>com.oreilly.jent.securitydemo.servlets.TaxServlet
    </servlet-class>
    <run-as>
      <description>Specify the role to invoke tax service</description>
      <role-name>TaxServiceUserRole</role-name>
    </run-as>
  </servlet>
  ...
</web-app>
  
```

10.4.5. Web Tier Best Practices

We recommend the following best practices for the web tier:

- Use declarative security whenever possible, and resort to programmatic security only when necessary.
- Be very cautious about the paths in your web application that cross the boundary from SSL to non-SSL mode and back again. These mode switches can lead to inadvertent leakage of secure and sensitive data, if you're not careful. It's possible, for example, to request secure information

from the user over SSL, put this information into session data while the user is within SSL portions of the application, and then display the information to the user in the clear within a non-SSL page.

- Be aware that if you use `RequestDispatcher` forward or include operations, the forward or include happens on the server side and the container has no opportunity to enforce security policies. As a result, users are not authenticated and authorization checks are not performed.
- Stay informed about new web security vulnerabilities related to specific web servers and browsers, and be proactive about dealing with them, if possible, in your application. One good source for this kind of information is the Open Web Application Security Project (OWASP) Top Ten Most Critical Web Application Vulnerabilities list (see <http://www.owasp.org/documentation/topten.html>). There are also many alert services to which you can subscribe to receive email about new vulnerabilities as they are found.

 PREVIOUS

10.5. EJB Component Security

EJB containers provide fairly rich security capabilities for EJB components that they manage. The following sections discuss transport-level security, authentication, declarative and programmatic authorization, identity propagation, and security best practices for the EJB tier. The material presented here complements the coverage provided in [Chapter 6](#). We'd recommend reviewing the relevant security-related material in that chapter first, then returning to this section for further details.

10.5.1. EJB-Tier Transport-Level Security

As discussed in [Chapter 6](#), remote EJB clients can use either RMI/JRMP or CORBA/IIOP protocols to communicate over the network with EJB components. If you need to secure the communications between clients and the EJB container, IIOP traffic can be tunneled over SSL using various IIOP/SSL tunneling protocols. But support for SSL-enabled IIOP is server-specific, so you'll have to ensure that both the EJB container and the client environments support the same secure IIOP protocols.

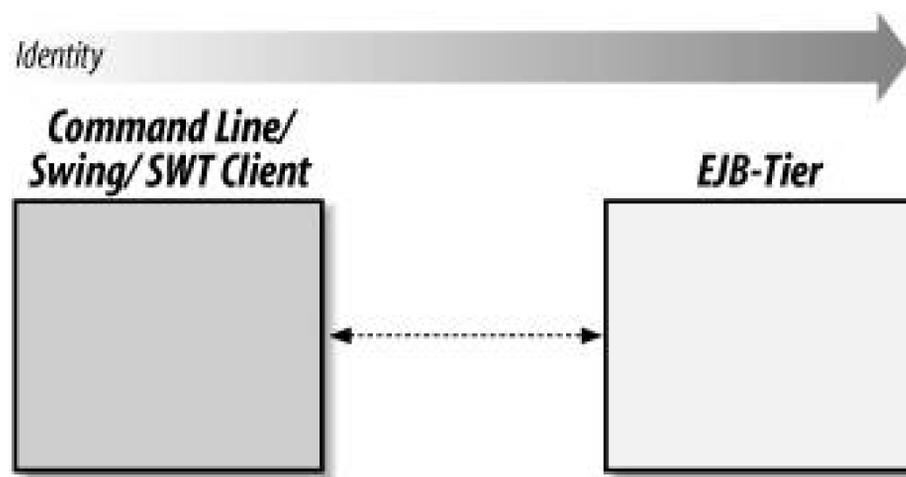
Using secure JRMP connections has been problematic in the past because of a lack of standard implementations of secure JRMP sockets in the Java platform. JDK 1.5 introduced a new standard SSL-enabled RMI socket implementation, but it will likely take some time for these to make their way into EJB container implementations as an option for deployed components.

In either case, configuring EJB containers to use secure transport communications is a vendor-specific detail. Consult the documentation for your EJB container for details.

10.5.2. EJB-Tier Authentication

As we discussed earlier, clients of EJB components are responsible for authentication of users and for propagating the user's identity to the EJB tier. We already described what happens with a web client, which passes along or propagates credentials from the web application to the EJB. Now, how is authentication handled for a standalone Swing or Standard Widget Toolkit (SWT) application or a command-line application? One option is to use JAAS for authentication. Once the client container authenticates a user in the client tier, the client container passes along the user's identity to the EJB components as shown in [Figure 10-6](#). You leverage JAAS to authenticate the user with various mechanisms such as Kerberos, LDAP, or even the operating system.

Figure 10-6. Propagation from client to EJB tier



10.5.3. EJB-Tier Authorization

J2EE security supports declarative as well as programmatic authorization for the EJB tier as detailed in the following sections.

10.5.3.1. EJB-tier declarative authorization

Declarative authorization in the EJB tier involves defining authorization policies in your EJB deployment descriptor. These policies specify which roles may access the methods on your EJB components and how these roles are propagated (or not) from the client. Role-based authorizations for accessing EJB methods are set using `<method-permission>` elements. You can also specify EJB methods that should not be accessed by clients by including them in `<exclude-list>` elements.

The propagation of user roles from the client to the EJB container is controlled using the `<security-identity>` element with each component's entry in the deployment descriptor. Using a `<use-caller-identity>` element specifies that any client requests to the EJB component should inherit the identity and roles of the client making the request. If you want all calls to the EJB component to assume a singular role, you can specify a `<run-as>` element within the `<security-identity>` element.

For details and examples of using these EJB declarative authorization settings, see the section "[Deploying EJBs](#)" in [Chapter 6](#).

10.5.3.2. EJB-tier programmatic authorization

The primary methods that support programmatic authorization in the EJB tier are `getCallerPrincipal()` and `isCallerInRole()` of the `javax.ejb.EJBContext` interface. `getCallerPrincipal()` returns an instance of `java.security.Principal` object. What if there is no current authenticated user? The `getCallerPrincipal()` method does not return `null`; it returns a `Principal` that represents an unauthenticated identity, i.e., an anonymous user.

The `isCallerInRole()` method takes a role in string form as shown in [Example 10-9](#).

Example 10-9. `isCallerInRole`

```

public static final String VIP_ROLE = "vip_role";

public BigDecimal getPercentageCost( ) {
    sLogger.info(
        " in shipping manager bean. get percentage cost : "
        + mSessionContext.getCallerPrincipal( ).getName( )
    );
    if (mSessionContext.isCallerInRole(VIP_ROLE)) {
        return new BigDecimal("3.00");
    }
    return new BigDecimal("7.00");
}

```

Before you pass the role name as an argument to the `isCallerInRole()` method, the role name must be declared in the `<security-role-ref>` element of the `ejb-jar.xml`/deployment descriptor as shown in [Example 10-10](#). The scope of `<security-role-ref>` is at the entity bean or session bean level. There is no client caller in case of a message driven bean.

Example 10-10. Security role reference

```

<ejb-jar>
    ...
    <session>
        ...
        <ejb-name>ShippingManager</ejb-name>
        ...
        <security-role-ref>
            <role-name>vip_role</role-name>
            <role-link>vip</role-link>
        </security-role-ref>
        ...
    </session>
    ...
</ejb-jar>

```

You map the role in the `<security-role-ref>` element to the `<security-role>` element using the `<role-link>` element. The scope of the `<security-role>` is the entire `ejb-jar.xml`/file and applies to all the enterprise beans in that file. [Example 10-11](#) shows how to specify a `<security-role>` as part of the `<assembly-descriptor>`.

Example 10-11. Security role in assembly descriptor

```

<ejb-jar>

```

```
...  
<assembly-descriptor>  
  ...  
  <security-role>  
    <description>The VIP role.</description>  
    <role-name>vip</role-name>  
  </security-role>  
  ...  
</assembly-descriptor>  
...  
</ejb-jar>
```

10.5.4. EJB Tier Best Practices

We recommend the following best practices for managing security at the EJB tier:

- As with the web tier, use declarative security whenever possible and resort to programmatic security only when necessary.
- Keep your security policies as coarse-grained as possible to avoid complex, highly granular authorization settings that are difficult to maintain and troubleshoot. Session bean façades in front of your entity beans can be useful in this regard since they can be used to channel users through a few well-defined entry points, thus reducing the number of required roles.

10.6. Other J2EE Security Topics

In this section, we discuss a few other topics relevant to J2EE security, including securing enterprise resource managers, web service security, and single sign-on.

10.6.1. Accessing Enterprise Resource Managers

An enterprise may have a wide range of enterprise resource managers such as relational database systems and message-oriented middleware products. To establish a connection to these resources, authentication is often required. You can manage the authentication details for these resource managers in one of two ways. You can manage it yourself in your application code, or you can use the application server or container to manage the connection details for you.

Resource manager references are declared in web and EJB deployment descriptors using `<resource-ref>` elements. Full details on declaring resource references in deployment descriptors can be found in [Chapter 2](#). In terms of the authentication mode to be used with a given resource, this is specified using the `<res-auth>` element within the `<resource-ref>` element. To indicate that the container should handle authentication details, set the `<res-auth>` element to `Container`. You then need to configure the authentication details for the resource in the container, using its management tools or container-specific configuration files. If you'd prefer to manage the authentication details in your application code, set the `<res-auth>` value to `Application`. It's usually preferable to use container-managed authentication since it facilitates the portability of your application across development and production environments. It also reduces the amount of code you need to maintain.

10.6.2. Web Services Security

[Chapter 12](#) provides general details on implementing SOAP-based web services using JAX-RPC and SAAJ. Securing SOAP services is a topic that is still in flux in the market in general. A number of competing standards exist for sending client credentials across a SOAP request, for encrypting SOAP communications, and for managing access to SOAP services. In part because of this uncertainty around web service security protocols, the J2EE 1.4 and JAX-RPC 1.1 specifications do not make any provisions for standardized web service security models.

Here, we simply introduce a few of the current techniques in practical use for securing web services. Future versions of the Java standards around web services, and broader agreement in the industry at large about web service security protocols, will make the job of securing web services much easier than it is today.

One common way to authenticate a web services client is to use `BASIC` authentication. In this approach, the web service client is responsible for providing authentication credentials through HTTP headers to the web server that fronts the web services being invoked. Of course, this approach works only in SOAP messages exchanged over HTTP, and in order to be truly secure, the underlying communication transport should actually be HTTPS. It is also possible to use `CLIENT` authentication

(i.e., the client provides its credentials in the form of a certificate) with web services. In fact, this form of web-based authentication is more suited for web clients since web clients are often other applications, which means the certificate exchange is between two servers and much easier to manage than end user certificate distribution.

We discussed the concept of digital signatures at the beginning of this chapter. XML documents can also take advantage of this technique, using standards like XML Signature. XML Signature is a joint IETF and W3C standards effort, and, among other things, it provides well-defined schema elements that can be used to attach digital signatures to XML documents, such as SOAP messages. The Java XML Digital Signature API (JSR 105) provides a standard Java API for both producing and consuming XML documents signed using XML Signature.

XML Encryption is a complementary standard, also maintained jointly by the IETF and W3C, that provides a standard approach for encrypting and decrypting the contents of XML documents. A corresponding Java API, the Java XML Digital Encryption API (JSR 106), is currently in development.

From the standpoint of SOAP-based web services, however, both XML Signature and XML Encryption are low-level security protocols. They deal directly with the XML contents of SOAP messages, so for a Java developer to use the corresponding Java APIs in either a SOAP client or service, she would have to delve into the ugly details of the XML data being sent back and forth. In an ideal world, there would be a standard way to integrate these XML security protocols into a SOAP message context and extract the relevant application-level information (the sender's identity, roles, etc.) to drive authorization and personalization features. In the world of general XML and web service protocols, the WS-Security standard (managed by the Organization for the Advancement of Structured Information Standards (OASIS)) is a higher-level security protocol for web services. It integrates XML Signature and XML Encryption and specifies details on how these and other security measures are to be applied within SOAP messages. Unfortunately, a standard Java API for interacting with WS-Security-protected SOAP messages, and mapping them into standard Java security entities like Principals, Subjects and Permissions, is still pending. Some open source and commercial implementations of WS-Security, such as the Apache Axis WSSE project, do exist, but broad support for WS-Security in the J2EE environment is still in development.

10.6.3. Single Sign-On (SSO)

An enterprise has a multitude of applications, each requiring security services including, but not limited to, authentication. Single Sign-On (SSO) is a system for authenticating a particular user just once to permit access to all the different applications in an enterprise.

SSO has usability benefits because the user does not have to remember different usernames and passwords for all the different applications. SSO has benefits for the organization as well because the organization provisions an individual just once. When an individual joins the organization, he is given access to the appropriate applications all at once, and when the individual leaves the organization, it discontinues the user's access to all applications. In addition to the convenience features of SSO, organizations can save money; password resets are a frequent request and occur largely because of having to remember a multitude of passwords. Administrative costs can potentially drop when SSO systems are implemented.

SSO is principally concerned with authentication. What about other security features such as authorization? The industry is moving toward a standardization of security information exchange with Security Assertion Markup Language (SAML). SAML supports not only authentication but also

authorization. You can even use SAML to create federated identities that integrate different SSO providers. The Liberty Alliance Project is embracing SAML with support for federated identities. JSR 155 provides the standard Java specification as well as a reference implementation for SAML.

What is involved in adopting SSO? The following sections discuss some of the issues related to implementing SSO in the context of J2EE security.

10.6.3.1. SSO products

Many different SSO products are on the market today, including Computer Associates' eTrust Identity and Access Management (which includes Netegrity SiteMinder), Entrust GetAccess, IBM Tivoli Access Manager, Oblix COREid, and RSA ClearTrust. When evaluating the products, be sure to check out their integration with your favorite application server(s) and client desktop applications.

A common temptation is to do point-to-point SSO integration between systems and applications without a clearly defined SSO strategy. The danger with such integration is that the implementation may turn out to be more time-consuming to develop, maintain, test, and deploy. Additionally, such an implementation may potentially be insecure and fragile.

10.6.3.2. Security stores

A single master location for user identities, roles, and permissions, sometimes called a *security store*, is ideal for implementing SSO. One of the most common security store solutions utilized in enterprises is an LDAP-based service. LDAP is ideally suited to store security information that is read many times and updated few times. Some other common solutions for security stores are custom or product-defined relational database schemas or operating system domains such as Windows Active Directory. Accessing security stores from your J2EE applications can be done using low-level APIs like JNDI or JDBC, or you can integrate the security store behind a JAAS provider. Alternatively, at the highest level, you can integrate the security store directly into your application server, allowing you to use declarative security measures in your deployment descriptors using identities and roles managed by the SSO environment. Your choice of these options will depend on the particular security store and application server(s) in use in your organization.

The ideal SSO situation of a single enterprise-wide security store is relatively rare, sadly. A typical organization will more likely have multiple security stores for various reasons such as legacy, performance issues, license limitations, and organizational and political factors. In these situations, the enterprise needs to keep these stores in sync with a variety of techniques, such as live data links, data replication, batch updates, and others.

10.7. Limitations of J2EE Security

One important aspect of security is preventing unauthorized access, either inadvertent or intentional. For example, the application prevents a regular user from accessing a set of URLs that only an admin can access. As another example, the application must prevent a regular user from invoking a set of methods that only an admin can invoke. Declarative security helps achieve this goal.

Sometimes, prevention happens too late from a usability perspective. Ideally, the application should present users only the functionality that they can access. The functionality that they do not have access to should be hidden or disabled. Despite the advantages of declarative security, it can't do checks at this level of granularity. Programmatic security can definitely help in this regard, especially with web contexts where specific visual or functional elements on web pages need to be enabled or disabled based on the identity and roles of the user.

For example, a common security feature is to allow users to edit their own data. The application allows the customer to edit his own order, and not somebody else's order. Granular security checks like this need to occur programmatically in the codedeclarative authorization measures in J2EE aren't expressive enough to represent business logic at this level.

Other low-level gaps in the J2EE security landscape can also lead to custom programmatic measures. J2EE web components, for example, do not have a standard way to explicitly log out a user from a web application. The most common way programmers implement this feature is by invalidating the HTTP session in their code. However, this technique works consistently only for form-based logins since other authentication schemes can leave a persistent cookie on the client web browser. This limitation may be addressed in future versions of the servlet specification, but at the moment, it's a practical issue that forces developers into custom programming.

Sadly, once developers move down the path of programmatic security measures, they are inclined to abandon J2EE security altogether in favor of custom development. Rather than provision the application server to integrate with the organization's security stores, the pressures of deadlines and short-term product or service goals can lead to application-specific security solutions that can be expensive to maintain, confusing to users (in terms of login mechanics, authorization models, etc.), and potentially risky in terms of security holes created in distributed security implementation code.

Naturally, these inclinations should be resisted as much as possible. You should strive to create a consistent security model across your organization's applications and systems, rather than a patchwork of local, custom implementations. And where you have J2EE application servers in place, you should leverage their security services as much as is practical in your environment. If they can tie directly into your security stores (LDAP servers, Active Directory, and so on), you should attempt to establish this level of integration and stick to business logic in your application code.

Chapter 11. Java Message Service

The standard Java networking APIs, as well as remote object systems such as RMI and CORBA, operate by default under the assumption of synchronous communications. In other words, if a client makes a request of a remote server (e.g., opens a socket and attempts to read some data from it or makes a remote method call on a remote object), the thread that made the request will block until the response comes back from the server. In some situations, it might be necessary or useful to engage in asynchronous communications (e.g., a client sends a request to the server and then continues doing other work, while the server possibly invokes some kind of callback on the client when the request is complete). This is where the Java Message Service (JMS) comes in.

JMS is an API for performing asynchronous messaging. JMS is principally a client-focused API, in that it provides a standard, portable interface for Java/J2EE clients to interact with native message-oriented middleware (MOM) systems like IBM WebSphere MQ, SonicMQ, and so on. JMS isn't intended to be a platform for implementing a full messaging system, since it doesn't provide a service provider interface for all of the internals of a message service implementation. In a sense, JMS plays a role with native messaging systems that is analogous to the role that JDBC plays with relational database systems or the role JNDI plays with naming and directory services. Java clients using JMS to interact with messaging systems can (ostensibly) be more easily ported from one native messaging system to another, because they are insulated from the proprietary particulars of the underlying vendor's message system.

The JMS API is provided in the `javax.jms` package. The material in this chapter is based on Version 1.1 of the JMS specification, released in April 2002, which is the most current version at this writing and the version that is required in J2EE 1.4 environments. J2EE 1.3 servers use JMS 1.0. We highlight relevant issues between these JMS versions in this chapter. The examples in the chapter have been tested against the JMS services embedded in the JBoss 4.0 server.

[← PREV](#)

11.1. JMS in the J2EE Environment

As of J2EE 1.3, compliant servers are required to provide a full JMS provider, including support for both point-to-point and publish-subscribe message destinations (these are described in detail next). So any compliant J2EE 1.3 (or later) server will have its own JMS server capable of hosting its own message destinations.

[← PREV](#)

11.2. Elements of Messaging with JMS

The principal players in a JMS system are *messaging clients*, *message destinations*, and a JMS-compatible *messaging provider*.

Messaging clients produce and consume messages. Typically, messaging takes place asynchronously: a client produces a message and sends it to a message destination, and some time later another client receives the message. Message clients can be implemented using JMS, or they can use a native messaging API to participate in the messaging system. If a native message client (e.g., a client using the native IBM MQ Series APIs) produces a message to a message destination, a JMS connection to the native message system is responsible for retrieving the message, converting it into the appropriate JMS message representation, and delivering it to any relevant JMS-based clients.

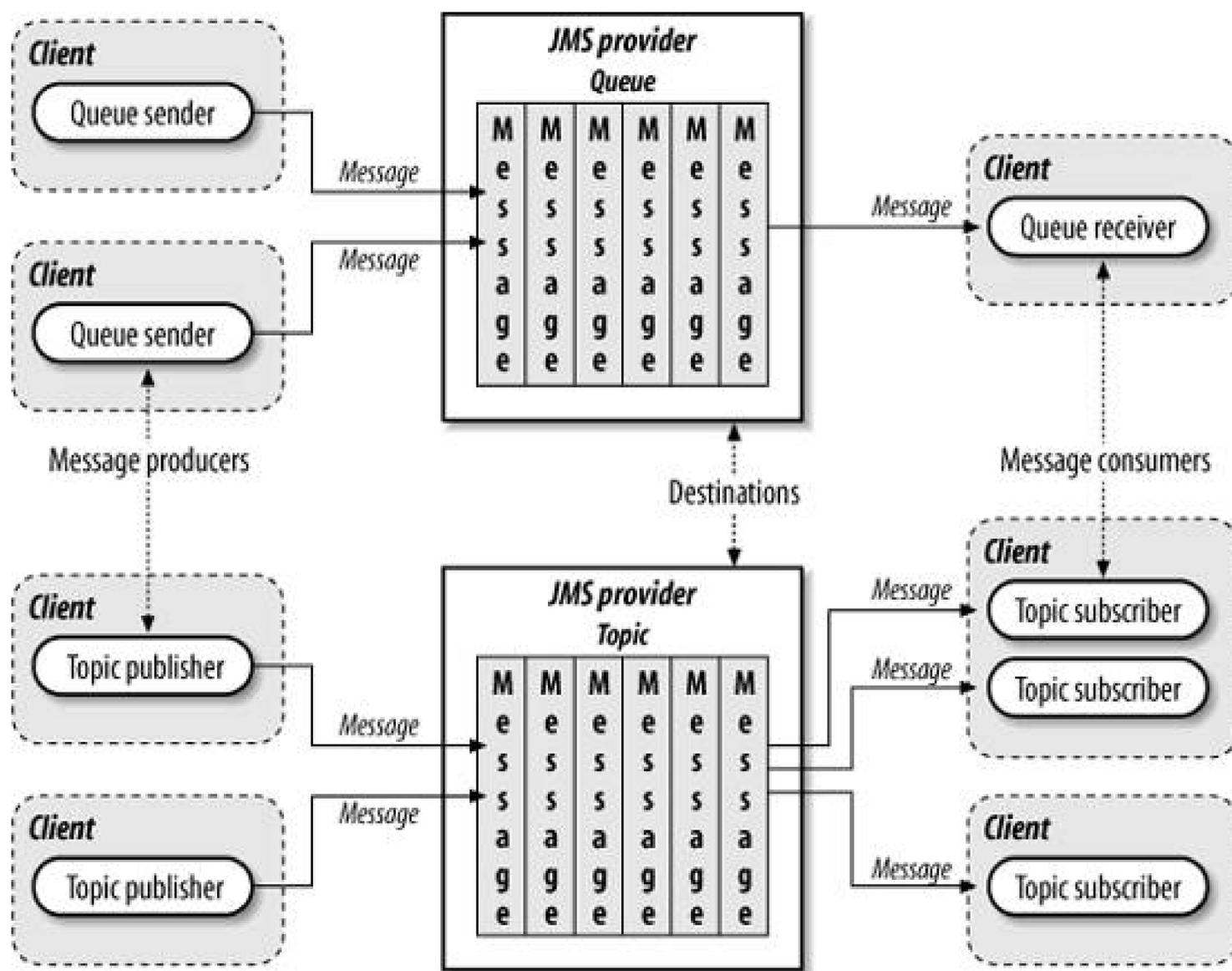
Message destinations are places to and from which JMS clients send and receive messages. Message destinations are created within a JMS provider that manages all of the administrative and runtime functions of the messaging system. At a minimum, a JMS provider allows you to specify a network address for a destination, allowing clients to find the destination on the network. But providers may also support other administrative options on destinations, such as persistence options, resource limits, and the like.

11.2.1. Messaging Styles: Point-to-Point and Publish-Subscribe

Generally speaking, asynchronous messaging usually comes in two flavors: a message can be addressed and sent to a single receiver (*point-to-point*), or a message can be published to a particular channel or topic and any receiver that subscribes to that channel will receive the message (*publish-subscribe*). These two messaging styles have analogies at several levels in the distributed computing "stack," all the way from the network level (standard TCP packet delivery versus multicast networking) to the application level (email versus newsgroups) [Figure 11-1](#) depicts the two message models supported by JMS, as well as the key interfaces that come into play in a JMS context. We discuss the specifics of these interfaces later in the chapter.

Most messaging providers support one or both of these messaging styles, so JMS provides support for them both in its API. JMS includes a set of generic messaging interfaces, described next. Each style of messaging is supported by specialized subclasses of these generic interfaces.

Figure 11-1. JMS message models



11.2.2. Key JMS Interfaces

The following key interfaces represent the concepts that come into play in any JMS client application, whether it is using point-to-point or publish-subscribe messaging:

Message

Messages are at the heart of JMS, naturally. **Messages** have accessor methods for their header fields, properties, and body contents. Subtypes of this interface provide implementations for different types of content.

MessageListener

A **MessageListener** is attached to a **MessageConsumer** by a client and receives a callback for each **Message** received by that consumer. **MessageListeners** are the key to asynchronous message delivery to clients, since the client attaches a listener to a consumer and then carries on with its thread(s) of control. **MessageListeners** must implement an `onMessage()` method, which is the callback used to notify the listener that a message has arrived.

ConnectionFactory

A `ConnectionFactory` creates connections to a JMS provider. `ConnectionFactory` references are obtained from a JMS provider through a JNDI lookup. A `QueueConnectionFactory` creates connections in a point-to-point context; `TopicConnectionFactory` creates connections in publish-subscribe contexts. The `createConnection()` method (added in JMS 1.1) creates a `Connection` appropriate for the type of messaging supported by the factory, either a `QueueConnection` or a `TopicConnection`. You can also cast a `ConnectionFactory` to its specific subtype (`QueueConnectionFactory` or `TopicConnectionFactory`) and use the `createQueueConnection()` or `createTopicConnection()` methods (respectively).

Destination

A `Destination` represents a network location, managed by a JMS provider, that can be used to exchange messages. A JMS client sends messages to `Destinations` and attaches `MessageListeners` to `Destinations` to receive messages from other clients. A client obtains references to `Destinations` using JNDI lookups. `Queues` and `Topics` are the `Destinations` in point-to-point and publish-subscribe contexts, respectively.

Connection

A `Connection` is a live connection to a JMS provider used for the receipt and delivery of messages. Before a client can exchange messages with a JMS destination, it must have a live connection that was started by the client. A `Connection` is obtained from a `ConnectionFactory` using its `createXXXConnection()` methods. A `QueueConnectionFactory.createQueueConnection()` returns a `QueueConnection`; a `TopicConnectionFactory.createTopicConnection()` returns a `TopicConnection`. The `createConnection()` method on `ConnectionFactory` (added in JMS 1.1) returns a `Connection`, which can be cast to its subtype if needed.

Session

A `Session` can be thought of as a single, serialized flow of messages between a client and a JMS provider. A `Session` is used to create message consumers and producers and to create `Messages` that a client wishes to send. A `Session` is used within a single thread of control on a client. Since a `Session` is accessed only from within a single thread, the messages sent or received through its consumers and producers are serialized with respect to the client. `Sessions` also provide a context for defining transactions around message operations; further details can be found in "[Transactional Messaging](#)" later in this chapter. `Sessions` are created from `Connections` using their `createXXXSession()` methods. The `QueueConnection.createQueueSession()` method returns a `QueueSession`, and the `TopicConnection.createTopicSession()` method returns a `TopicSession`. The `Connection.createSession()` method (added in JMS 1.1) returns a `Session`, which can be cast to its subtype if needed.

MessageProducer/MessageConsumer

`MessageProducers` and `MessageConsumers` are used to send and receive messages to and from a

destination, respectively. Producers and consumers are created using various `createXXX()` methods on `Sessions`, using the target `Destination` as the argument. In a point-to-point context, `QueueSenders` are created using the `QueueSession.createSender()` method, and `QueueReceivers` are created using the `QueueSession.createReceiver()` methods. In a publish-subscribe context, `TopicPublishers` are created using `TopicSession.createPublisher()`, and `TopicSubscribers` are created using `TopicSession.createSubscriber()` and `TopicSession.createDurableSubscriber()` methods. You can also use the `Session.createProducer()` and `Session.createConsumer()` methods (added in JMS 1.1) to create `MessageProducers` and `MessageConsumers` that can be cast to their subtypes.

11.2.3. A Generic JMS Client

A JMS client follows the same general sequence of operations, regardless of whether it's using point-to-point or publish-subscribe messaging, or both. We'll walk through these steps here, using the point-to-point JMS interfaces to demonstrate. For the most part, the same pseudocode can be used with the publish-subscribe interfaces by just substituting `Topic` for `Queue` in the code samples in this section.

11.2.3.1. General setup

The very first step for a JMS client is to get a reference to an `InitialContext` for the JNDI service of the JMS provider. Full details on the various options for obtaining a `JNDIContext` can be found in [Chapter 9](#), but in general, the client will create an `InitialContext` using a set of `Properties` that specify the location and type of the JNDI service associated with the JMS provider:

```
Properties props = ...;
Context ctx = new InitialContext(props);
```

Next, the JMS client needs to acquire a `ConnectionFactory` from the JMS provider using a JNDI lookup. The client would have to know what name the JMS provider used to publish the `ConnectionFactory` in JNDI space. Here, we look up a `QueueConnectionFactory` registered in JNDI under the name `jms/someQFactory`.

```
QueueConnectionFactory qFactory =
    (QueueConnectionFactory)ctx.lookup("jms/someQFactory");
```

An administrator would have to set up this `ConnectionFactory` on the JMS provider and associate it with this JNDI name on the server.

The client also uses JNDI to find `Destinations` published by the JMS provider. Here, we look up a `Queue` published under the JNDI name `jms/someQ`.

```
Queue queue = (Queue)ctx.lookup("jms/someQ");
```

Once we have a `ConnectionFactory` and one or more `Destinations` to talk to, we need to create a `Connection` with the JMS provider. This `Connection` is the conduit through which messages will be physically sent and received. A `Connection` has to be started before messages can be received through it, but a `Connection` can always be used to send messages, regardless of whether it's started or stopped. Normally, a client won't `start()` a `Connection` until it's ready to receive and process messages. Here, we use our `QueueConnectionFactory` to create a `QueueConnection` and defer starting it until we create a `MessageConsumer` to receive messages:

```
QueueConnection qConn = qFactory.createQueueConnection(...);
```

11.2.3.2. Client identifiers

When a client makes a connection to a JMS provider, a client identifier is associated with the client. The client identifier is used to maintain state on the JMS provider on behalf of the client, and the state data can persist beyond the lifetime of a client connection. The server-side state can be retrieved for the client when it reconnects using the same client ID. The only client state information defined by the JMS specification is durable topic subscriptions (described in ['Durable Subscriptions'](#) later in this chapter), but a JMS provider may support its own state information on behalf of clients as well. Only one client is allowed to be associated with a client ID (and its state information) on the JMS provider, so only a single connection with a given client ID can be made to a JMS provider at any given time.

The JMS client identifier can be set in two ways. A client can set a client ID on any `Connections` that it makes with the JMS provider, using the `Connection.setClientID()` method:

```
qConn.setClientID("client-1");
```

Again, only a single connection with a given client ID is allowed at any given time. If a client with this same client ID (even this one) already has a connection with the client ID, then an `InvalidClientIDException` will be thrown when `setClientID()` is called. Alternatively, a `ConnectionFactory` can be configured on the JMS provider with a client ID that is applied to any `Connections` that are created through it. The `ConnectionFactory` interface doesn't provide a facility for the client to set the factory's client ID; this is a function that would have to be provided in the JMS provider's administrative interface. A `ConnectionFactory` with a preset client ID is, by definition, intended to be used by a single client.

11.2.3.3. Authenticated connections

When a client creates a connection, it has the option to provide a username and password that will be

authenticated by the JMS provider. This is done using overloaded versions of the `createXXXConnection()` methods on a `ConnectionFactory`. We can create an authenticated `QueueConnection`, for example, with a call like this:

```
QueueConnection authQConn =
    qFactory.createQueueSession("JimFarley", "myJMSPassword");
```

If this is successful, the client will operate under the given principal name and be given the appropriate rights. JMS providers aren't required to support authentication of connections. If a JMS provider does support authenticated connections, the principals and access rights will be administered on the JMS server.

11.2.3.4. Sessions

Once a connection to the JMS provider is established, we need to create one or more `Sessions` to be used to send and receive messages. Again, `Sessions` are a single-threaded context for handling messages, so we need a separate `Session` for each concurrent thread that we plan to use for messaging. `Sessions` are created from `Connections`. Here, we create a `QueueSession` from our `QueueConnection`:

```
QueueSession qSess =
    qConn.createQueueSession(false, Session.AUTO_ACKNOWLEDGE);
```

When creating either `QueueSessions` or `TopicSessions`, two arguments are used to create the `Session`. The first is a `boolean` flag indicating whether we want the `Session` to be transacted. (See "[Transactional Messaging](#)" later in this chapter for details on transactional sessions.) The second argument indicates how we want the `Session` to acknowledge received messages with the JMS provider. The three options for the acknowledge mode of a `Session` are specified using static final values on the `Session` class:

`Session.AUTO_ACKNOWLEDGE`

This instructs the `Session` to acknowledge messages automatically for the client. A message is acknowledged when received by the client. If a `MessageListener` handles the message, then the acknowledgment is not sent until the listener's `onMessage()` method returns. If the message is received because of a call to `receive()` on a `MessageConsumer`, then the acknowledgment is sent immediately after the call to `receive()` returns.

`Session.DUPS_OK_ACKNOWLEDGE`

This option instructs the `Session` to do "lazy acknowledgment": acknowledgments can be delayed if the `Session` decides to do so. This could lead to a message being delivered to a client

more than once, if the delay between delivery and acknowledgment is longer than the JMS provider's timeout and it assumed the message was never received.

`Session.CLIENT_ACKNOWLEDGE`

This option is used when the client wants to manually acknowledge messages, by calling the `acknowledge()` method on the `Message`.

11.2.3.5. Sending messages

`Messages` are sent to `Destinations` using `MessageProducers`, which are created from `Sessions`. When they are created, producers are associated with a `Destination`, and any `Messages` sent using the producer are delivered to that `Destination` using the `Connection` from which the `Session` was generated.

In a point-to-point context, the message producers are `QueueSenders`, generated from `QueueSessions` using the `Queue` the sender should point to:

```
QueueSender qSender = qSess.createSender(queue);
```

Once a producer has been created, the client needs to create and initialize `Messages` to be sent. `Messages` are also created from a `Session`. Here, we create a `TextMessage` from our `QueueSession` and set its text body to be some interesting text we want to send to the `Queue`:

```
TextMessage tMsg = qSess.createTextMessage( );
tMsg.setText("The sky is blue.");
```

To actually send the message, we simply invoke the appropriate method on our `MessageProducer`. Here, we call `send()` on our `QueueSender`:

```
qSender.send(tMsg);
```

Note that it's not necessary to ensure that the underlying `Connection` (from which we generated our `Session`) is started in order to send messages. Starting the `Connection` is required only to commence delivery of messages from the `Destination` to the client.

11.2.3.6. Receiving messages

Receiving messages involves creating a `MessageConsumer` that is associated with a particular `Destination`. This establishes a consumer with the JMS provider, and the provider is responsible for delivering any appropriate messages that arrive at the `Destination` to the new consumer. `MessageConsumers` are also generated from `Sessions`, in order to associate them with a serialized flow of messages. In a point-to-point context, a `QueueReceiver` is generated from a `QueueSession` using its `createReceiver()` methods. Here, we simply create a new receiver tied to our `Queue`. Other options for creating `QueueReceivers` are discussed in "[Point-to-Point Messaging](#)" later in this chapter.

```
QueueReceiver qReceiver = qSess.createReceiver(queue);
```

By creating a `MessageConsumer`, all we've done is tell the JMS provider that we want to receive messages from a particular `Destination`. We haven't specified what to do with the `Messages` on the client side. Since JMS is an asynchronous message delivery system, it uses the same listener pattern that is used in Swing GUI programming or JavaBeans event handling (two other asynchronous event contexts). `Messages` in JMS are processed using `MessageListeners`. A client needs to implement a `MessageListener` with an `onMessage()` method that does something useful with the `Messages` coming from the `Destination`. [Example 11-1](#) shows a basic `MessageListener` a `TextLogger` that simply prints the contents of any `TextMessages` it encounters.

Example 11-1. Simple MessageListener implementation

```
import javax.jms.*;

public class TextLogger implements MessageListener {
    // Default constructor
    public TextLogger( ) {}

    // Message handler
    public void onMessage(Message msg) {
        // If it's a text message, print it to stdout
        if (msg instanceof TextMessage) {
            TextMessage tMsg = (TextMessage)msg;
            try {
                System.out.println("Received message: " + tMsg.getText( ));
            }
            catch (JMSEException je) {
                System.out.println("Error retrieving message text: " +
                    je.getMessage( ));
            }
        }
        // For other types of messages, print an error
        else {
            System.out.println("Unsupported message type encountered.");
        }
    }
}
```

Once a `MessageListener` has been defined, the client needs to create one and register it with a `MessageConsumer`. In our running example, we create one of our `TextLoggers` and associate it with our `QueueReceiver` using its `setMessageListener()` method:

```
MessageListener listener = new TextLogger( );
qReceiver.setMessageListener(listener);
```

It's important to remember that no messages will be delivered over our underlying `Connection` until it's been started. In our running example, we created our `QueueConnection` but never started it, so we do that now to start delivery of `Messages` to our `QueueReceiver`, and from there to our `TextLogger` listener:

```
qConn.start( );
```

11.2.3.7. Temporary destinations

A client can create its own temporary destinations, which are `Destinations` that are visible only to the `Connections` that created them, and that live only for the duration of the `Connection` used to create them. Although a temporary destination lives only for the life of the `Connection` it was created from, it is created using methods on the `Session`. For example, to create a `TemporaryQueue`:

```
Queue tempQueue = qSession.createTemporaryQueue( );
```

Temporary destinations can be used, for example, to receive responses to messages that are sent with a `JMSReplyTo` header:

```
TextMessage request = qSession.createTextMessage( );
request.setJMSReplyTo(tempQueue);
```

They can also be used to exchanging asynchronous messages between threads in the same client.

11.2.3.8. Cleaning up

`Connections` and `Sessions` require resources to be allocated by the JMS provider (similar to how JDBC connections use up resources on an RDBMS), so it's a good idea to free them up explicitly when you are done with them.

Sessions are closed by simply calling `close()` on them:

```
qSess.close( );
```

When a `Session` is closed, all `MessageConsumers` and `MessageProducers` associated with it are rendered unusable. If you try to use them to communicate with the JMS provider, they will throw an `IllegalStateException`. A call to `Session.close()` will block until any pending processing of incoming `Messages` (e.g., a `MessageListener`'s `onMessage()` method) is complete.

Closing a `Session` doesn't close the underlying `Connection` from which it came. You can close one `Session` and open up another one as long as the `Connection` is active. To close a `Connection` and free up its server-side resources, call its `close()` method:

```
qConn.close( );
```

All `Sessions` (and, subsequently, all of their consumers and producers) generated from a `Connection` become unusable once it is closed. The call to `Connection.close()` will block until incoming `Message` processing has completed on all of the `Sessions` associated with it.

11.3. The Anatomy of Messages

Creating a messaging-based application involves more than establishing communication channels between participants. Players in a message-driven system need to understand the content of the messages and know what to do with them.

Native messaging systems, such as WebSphere MQ or Microsoft MQ (MSMQ), define their own proprietary formats for messages. JMS attempts to bridge these native messaging systems by defining its own standard message format. All JMS clients can interact with any messaging system that supports JMS. "Supports" in this case can mean one of two things. The messaging system can be implemented in a native, proprietary architecture, with a JMS bridge that maps the JMS message formats (and other aspects of the JMS specification) to the native scheme and back again. Or the messaging system can be written to use the JMS message format as its native format.

JMS messages are made up of a set of standard header fields, optional client-defined properties, and a body. JMS also provides a set of subclasses of `Message` that support various types of message bodies.

11.3.1. Message Header Fields and Properties

[Table 11-1](#) lists the standard header fields that any JMS message can have. The table indicates the name and type of the field, when the field is set in the message delivery process, and a short description of the semantics of the field.

Table 11-1. Standard JMS message headers

Field name	Data type	When set	Description
JMSCorrelationID	String	Before send	Correlates multiple messages. This field can be used in addition to the <code>JMSMessageID</code> header as an application-defined message identifier (<code>JMSMessageIDs</code> are assigned by the provider).
JMSDeliveryMode	Destination	During send	Indicates to the message receiver which <code>Destination</code> the <code>Message</code> was sent to.
JMSDestination	int	During send	Indicates which delivery mode to use to deliver this message, <code>DeliveryMode.PERSISTENT</code> or <code>DeliveryMode.NON_PERSISTENT</code> . <code>PERSISTENT</code> delivery indicates that the messaging provider should take measures to ensure that the message is delivered despite

Field name	Data type	When set	Description
			failures on the JMS server. <code>NON_PERSISTENT</code> delivery doesn't require the provider to deliver the message if a failure occurs on the JMS server.
JMSExpiration	long	During send	The time the message will expire on the provider. If no client receives the message by this time, the provider drops the message. It is calculated as the sum of the current time plus the time-to-live of the <code>MessageProducer</code> that sent the message. The value is given in milliseconds since the epoch (January 1, 1970, 00:00:00 GMT). A value of 0 indicates no expiration time.
JMSMessageID	String	During send	A unique message ID assigned by the provider. Message IDs always start with the prefix "ID:". These IDs are unique for a given JMS provider. Applications can set their own message identifier using the <code>JMSCorrelationID</code> header.
JMSPriority	int	During send	A provider-assigned value indicating the priority with which the message will be delivered. JMS providers aren't required to implement strict priority ordering of messages. This field is simply a "hint" from the server about how the message will be handled. Message priorities and how they are assigned are determined by the configuration of the JMS provider.
JMSRedelivered	boolean	Before delivery	A provider-provided value that indicates to the receiver that the message may have been delivered in the past with no acknowledgment from the client. On the sender, this header value is always unassigned.
JMSReplyTo	Destination	Before send	A <code>Destination</code> , set by the sending client, indicating where a reply message should be sent.
JMSTimestamp	long	During send	The time at which the message was handed off to the JMS provider to be sent. This value is given in milliseconds since the epoch (January 1, 1970, 00:00:00 GMT).
JMSType	String	Before send	A message type, set by the sending client. Some JMS providers require that this header be set, so it's a good idea to set it even if your application isn't using it. Some JMS providers also allow an administrator to configure a set of message types that will be matched against this header and used to selectively set handling of the message based on its type.

These standard message headers are read and written using corresponding accessors on the `Message` interface. The `JMSTimestamp` field, for example, is set using `setJMSTimestamp()` and read using `getJMSTimestamp()`.

A client can also create its own custom properties on a `Message`, using a set of generic property accessors on the `Message` interface. Custom message properties can be `boolean`, `byte`, `short`, `int`, `long`, `float`, `double`, or `String` values, and they are accessed using corresponding

`get/setXXXProperty()` methods on `Message`. A `boolean` header can be set using the `setBooleanProperty()` method, for example. Each custom property has to be given a unique name, specified when the value is set. For example, we could set a custom `boolean` property with the name "reviewed" on a message, like so:

```
TextMessage tMsg = ...;
tMsg.setBooleanProperty("reviewed", false);
```

Custom property names have certain restrictions on them. They have to be valid Java identifiers, they can't begin with "JMSX" or "JMS_" (these are reserved for JMS-defined and vendor-defined properties, respectively), and they can't be one of the following reserved words: `NULL`, `true`, `FALSE`, `NOT`, `AND`, `OR`, `BETWEEN`, `LIKE`, `IN`, `IS`, or `ESCAPE`. Custom property names are also case-sensitive, so "reviewed" isn't the same property as "Reviewed".

11.3.2. JMS Message Types

To support various application scenarios, JMS provides the following subclasses of `Message`, each providing a different type of message body:

TextMessage

Arguably the most popular type of `Message`, this has a body that is a simple `String`. The format of the `String` contents is left to the application to interpret. The `String` may contain a simple informational phrase, it may contain conversational text input by a user in a collaboration application, or it may contain formatted text such as XML.

BytesMessage

This type of message contains an array of bytes as its body. `BytesMessages` can be used to send binary data in a message, and it can be used to wrap a native message format with a JMS message.

ObjectMessage

The body of this message is a serialized Java object.

MapMessage

The body of this message is a set of name/value pairs. The names are `Strings`, and the values are Java primitive types, like `double`, `int`, and `String`. The values of the entries are accessed using `get/setXXX()` methods on `MapMessage`. Note that these entries are stored in the body of the message: they aren't message header properties and can't be used for message selection

(see "[Filtering Messages](#)" later in this chapter).

StreamMessage

A `StreamMessage` contains a stream of Java primitive data types (`double`, `int`, `String`, etc.). Data elements are written sequentially to the body of the message using various `writeXXX()` methods, and they are read sequentially on the receiving end using corresponding `readXXX()` methods. If the receiver doesn't know the types of data in the message, they can be introspected using the `readObject()` method:

```
StreamMessage sMsg = ...;
Object item = sMsg.readObject( );
if (item instanceof Float) {
    float fData = ((Float)item).floatValue( );
    ...
}
```

11.3.3. Accessing Message Content

When a client receives a `Message`, its body is read-only. Attempting to change the body of a received `Message` will cause a `MessageNotWriteableException` to be thrown.

When a message sender first creates a `Message` object, the body of the `Message` is unset. For `TextMessages` and `ObjectMessages`, this means that their body starts with a `null` value, and for `MapMessages`, this means there are no entries in the message body. For `BytesMessages` and `StreamMessages`, their bodies start in write-only mode, and they can't be read by the sending client until it calls their `reset()` method. If you create a `BytesMessage` or `StreamMessage` and attempt to read its body content before calling `reset()` on it, a `MessageNotReadableException` will be thrown.

A `Message`'s body can be emptied by the sender at any point by calling its `clearBody()` method. This reverts its body back to its initial state, but doesn't affect any of the header or property values on the `Message`. Calling `clearBody()` on a `BytesMessage` or `StreamMessage` puts its body back into write-only mode, until a subsequent call to `BytesMessage.reset()` or `StreamMessage.reset()` is made. The sender can clear any custom properties on a `Message` by calling its `clearProperties()` method. The standard JMS header fields have to be updated using their specific accessors on the `Message` interface. Receivers of messages can't call `clearBody()`, `clearProperties()`, or `reset()` on a received `Message` since the `Message` is read-only at this point.

11.3.4. Filtering Messages

JMS allows messaging participants to selectively filter what `Messages` it receives from a JMS provider. This is done using *message selectors*, which are expressions that filter messages based on the values found in their headers and custom properties. The syntax of message selectors is based on SQL92 conditional expressions. A complete description of the JMS message selector syntax is provided in [Appendix E](#), but let's take a quick look at the concept of message filters.

A message selector is associated with a `MessageConsumer` when it is created from a `Session`. Each type of `Session` (`QueueSession` and `TopicSession`) has overloaded versions of their consumer create methods that take a message selector as a `String` argument. For example, the following creates a `QueueReceiver` that receives only messages that have a custom property named `transaction-type` and whose `JMSType` header field is `acknowledge`:

```
String selector =  
    "JMSType = 'acknowledge' AND transaction-type IS NOT NULL";  
QueueReceiver receiver = qSession.createReceiver(queue, selector);
```

Message filtering is performed by the JMS provider. When the provider determines that a message should be delivered to a particular `MessageConsumer`, based on the rules of the particular message context (point-to-point or publish-subscribe), it first checks that consumer's message selector, if one exists. If the selector evaluates to `true` when the message's headers and properties are applied to it, then the message is delivered; otherwise, it isn't. Undelivered messages are handled differently, depending on the message context, as described in the following sections.

[← PREV](#)

11.4. Point-to-Point Messaging

Point-to-point messaging involves the sending of messages from one or more senders to a single receiver through a message queue. Point-to-point messaging is analogous to email messaging: a client delivers messages to a named mailbox (queue), and the owner of the mailbox (queue) reads them in the order they were received. Queues attempt to maintain the send order of messages generated by the sender(s) attached to them. In other words, if sender A sends messages A1, A2, and A3, in that order, to a queue, then the receiver attached to the queue will receive message 2 after message 1, and message 3 after message 2 (assuming that no message selectors filter out any of these messages). If there are multiple senders attached to a queue, then the relative order of each individual sender is preserved by the queue when it delivers the messages, but the queue doesn't attempt to impose a predefined absolute order on the messages across all senders. So if there is another sender, B, attached to the same queue as A, and it sends messages B1, B2, and B3, in that order, then the receiver will receive B2 after B1, and B3 after B2, but the messages from sender A may be interleaved with the messages from sender B. The receiver may receive the messages in order A1, A2, B1, A3, B2, B3; the messages may be delivered in order B1, B2, B3, A1, A2, A3; or the messages may arrive some other order altogether. There is nothing in the JMS specification that dictates how a JMS provider should queue messages from multiple senders.

Point-to-point messaging is performed in JMS using the queue-related interfaces and classes in the `javax.jms` package. Queues are represented by `Queue` objects, which are looked up in JNDI from the JMS provider. `QueueConnectionFactory` objects are looked up in JNDI as well and used to create `QueueConnections`. `QueueConnections` and `Queues` are used to create `QueueSessions`, which are in turn used to create `QueueSenders` and `QueueReceivers`.

11.4.1. Sample Client

[Example 11-2](#) shows a full point-to-point messaging client. The `PTPMessagingClient` is capable of sending and receiving a message from a given queue, as well as browsing the current contents of the queue.

Example 11-2. Point-to-point messaging client

```
import java.util.*;
import javax.naming.*;
import javax.jms.*;

public class PTPMessagingClient implements Runnable {

    // Our connection to the JMS provider. Only one is needed for this client.
    private QueueConnection mQueueConn = null;
```

```
// The queue used for message passing
private Queue mQueue = null;

// Our message receiver -- need only one
private QueueReceiver mReceiver = null;

// A single session for sending and receiving from all remote peers
private QueueSession mSession = null;

// The message type we tag all our messages with
private static String MSG_TYPE = "JavaEntMessage";

// Constructor, with client name, and the JNDI locations of the JMS
// connection factory and queue that we want to use
public PTPMessagingClient(String cFactoryJNDIName, String queueJNDIName) {
    init(cFactoryJNDIName, queueJNDIName);
}

// Do all the JMS setup for this client. Assumes that the JVM is
// configured (perhaps using jndi.properties) so that the default JNDI
// InitialContext points to the JMS provider's JNDI service.
protected boolean init(String cFactoryJNDIName, String queueJNDIName) {
    boolean success = true;

    Context ctx = null;
    // Attempt to make connection to JNDI service
    try {
        ctx = new InitialContext( );
    }
    catch (NamingException ne) {
        System.out.println("Failed to connect to JNDI provider:");
        ne.printStackTrace( );
        success = false;
    }

    // If no JNDI context, bail out here
    if (ctx == null) {
        return success;
    }

    // Attempt to look up JMS connection factory from JNDI service
    QueueConnectionFactory connFactory = null;
    try {
        connFactory = (QueueConnectionFactory)ctx.lookup(cFactoryJNDIName);
        System.out.println("Got JMS connection factory.");
    }
    catch (NamingException ne2) {
        System.out.println("Failed to get JMS connection factory: ");
        ne2.printStackTrace( );
        success = false;
    }
}
```

```

try {
    // Make a connection to the JMS provider and keep it.
    // At this point, the connection is not started, so we aren't
    // receiving any messages.
    mQueueConn = connFactory.createQueueConnection( );
    // Try to find our designated queue
    mQueue = (Queue)ctx.lookup(queueJNDIName);
    // Make a session for queueing messages: no transactions,
    // auto-acknowledge
    mSession =
        mQueueConn.createQueueSession(false,
                                       javax.jms.Session.AUTO_ACKNOWLEDGE);
}
catch (JMSEException e) {
    System.out.println("Failed to establish connection/queue:");
    e.printStackTrace( );
    success = false;
}
catch (NamingException ne) {
    System.out.println("JNDI Error looking up factory or queue:");
    ne.printStackTrace( );
    success = false;
}

try {
    // Make our receiver, for incoming messages.
    // Set the message selector to receive only our type of messages,
    // in case the same queue is being used for other purposes.
    mReceiver = mSession.createReceiver(mQueue,
                                       "JMSType = '" + MSG_TYPE + "'");
}
catch (JMSEException je) {
    System.out.println("Error establishing message receiver:");
    je.printStackTrace( );
}

return success;
}

// Send a message to the queue
public void sendMessage(String msg) {
    try {
        // Create a JMS msg sender connected to the destination queue
        QueueSender sender = mSession.createSender(mQueue);
        // Use the session to create a text message
        TextMessage tMsg = mSession.createTextMessage( );
        tMsg.setJMSType(MSG_TYPE);
        // Set the body of the message
        tMsg.setText(msg);
        // Send the message using the sender
        sender.send(tMsg);
        System.out.println("Sent the message");
    }
}

```

```

    }
    catch (JMSEException je) {
        System.out.println("Error sending message " + msg + " to queue");
        je.printStackTrace( );
    }
}

// Register a MessageListener with the queue to receive
// messages asynchronously
public void registerListener(MessageListener listener) {
    try {
        // Set the listener on the receiver
        mReceiver.setMessageListener(listener);
        // Start the connection, in case it's still stopped
        mQueueConn.start( );
    }
    catch (JMSEException je) {
        System.out.println("Error registering listener: ");
        je.printStackTrace( );
    }
}

// Perform a synchronous receive of a message from the queue.  If it's a
// TextMessage, print the contents.
public String receiveMessage( ) {
    String msg = "-- No message --";
    try {
        Message m = mReceiver.receive( );
        if (m instanceof TextMessage) {
            msg = ((TextMessage)m).getText( );
        }
        else {
            msg = "-- Unsupported message type received --";
        }
    }
    catch (JMSEException je) {
    }
    return msg;
}

// Print the current contents of the message queue, using a QueueBrowser
// so that we don't remove any messages from the queue
public void printQueue( ) {
    try {
        QueueBrowser browser = mSession.createBrowser(mQueue);
        Enumeration msgEnum = browser.getEnumeration( );
        System.out.println("Queue contents:");
        while (msgEnum.hasMoreElements( )) {
            System.out.println("\t" + (Message)msgEnum.nextElement( ));
        }
    }
    catch (JMSEException je) {
    }
}

```

```
        System.out.println("Error browsing queue: " + je.getMessage( ));
    }
}

// When run within a thread, just wait for messages to be delivered to us
public void run( ) {
    while (true) {
        try { this.wait( ); } catch (Exception we) {}
    }
}

// Take command-line arguments and send or receive messages from the
// named queue
public static void main(String args[]) {
    if (args.length < 3) {
        System.out.println("Usage: PTPMessagingClient" +
            " connFactoryName queueName" +
            " [send|listen|recv_synch] <messageToSend>");
        System.exit(1);
    }

    // Get the JNDI names of the connection factory and
    // queue, from the commandline
    String factoryName = args[0];
    String queueName = args[1];

    // Get the command to execute (send, recv, recv_synch)
    String cmd = args[2];

    // Create and initialize the messaging participant
    PTPMessagingClient msger =
        new PTPMessagingClient(factoryName, queueName);

    // Run the participant in its own thread, so that it can react to
    // incoming messages
    Thread listen = new Thread(msger);
    listen.start( );

    // Send a message to the queue
    if (cmd.equals("send")) {
        String msg = args[3];
        msger.sendMessage(msg);
        System.exit(0);
    }
    // Register a listener
    else if (cmd.equals("listen")) {
        MessageListener listener = new TextLogger( );
        msger.registerListener(listener);
        System.out.println("Client listening to queue " + queueName
            + "...");
        System.out.flush( );
        try { listen.wait( ); } catch (Exception we) {}
    }
}
```

```

    }
    // Synchronously receive a message from the queue
    else if (cmd.equals("recv_synch")) {
        String msg = msger.receiveMessage( );
        System.out.println("Received message: " + msg);
        System.exit(0);
    }
    else if (cmd.equals("browse")) {
        msger.printQueue( );
        System.exit(0);
    }
}
}
}

```

The `main()` method takes a minimum of three command-line arguments. The first two are the JNDI names of a target JMS connection factory and queue, in that order. The third argument is a command indicating what to do:

`send`

Sends a message, using the next command-line argument as the text of a `TextMessage`

`recv`

Registers a listener with the queue and waits for messages to come in

`recv_synch`

Synchronously polls the queue for the next message that's sent

`browse`

Sends a request to print the current contents of the queue without emptying it, using a `QueueBrowser`

The `main()` method creates a `PTPMessagingClient` using the two JNDI names. The constructor passes these to the `init()` method, where all of the JMS initialization we've discussed takes place. The client attempts to connect to its JNDI provider and get its `InitialContext` first. There are no properties provided to the `InitialContext` constructor, so the environment would have to have these properties specified in a `jndi.properties` file or on the command line using `-D` options to the JVM (see [Chapter 9](#) for details on these JNDI-related system properties). Once the `Context` is acquired, the client looks up the `QueueConnectionFactory` and `Queue` from JNDI. It also creates a `QueueConnection` and a `QueueSession`, so that it can later create senders and receivers as needed. Finally, the `init()` method creates a `QueueReceiver` from the session, in case it's needed later. The connection hasn't been started yet, so the receiver is not receiving messages from the JMS provider yet.

Back in the `main()` method, once the client is created, it's put into a `Thread` and run. This is useful for the case in which we're going to wait for messages sent to a listener. Finally, the requested command is checked. If the command is `send`, we call the client's `sendMessage()` method, which creates a `QueueSender` and a `TextMessage` (using the last command-line argument, passed in from the `main()` method). Then the message is sent by passing it to the `send()` method on the `QueueSender`. If a `recv` command is given, we create a `TextLogger` (see [Example 11-1](#)) and attach it as a `MessageListener` to our `QueueReceiver`, by calling the client's `registerListener()` method where the call to the receiver's `setMessageListener()` method is made. If a `recv_synch` command is given, then we call the client's `receiveMessage()` method, where the `receive()` method on the `QueueReceiver` is called. This will block until the next message is sent to the queue. Finally, `abrowse` command causes a call to the client's `printQueue()` method, where a `QueueBrowser` is created from our session, then asked for an `Enumeration` of the current messages in the queue. Each message is printed to the console, in the order they would be received.

11.4.2. Browsing Queues

In addition to the conventional use of queues for sending and receiving of messages, a client can also browse the contents of a queue without actually pulling the messages from the queue. This is done using a `QueueBrowser`, which is generated from a client's `QueueSession` using its `createQueueBrowser()` methods:

```
QueueBrowser browser = qSession.createQueueBrowser(queue);
```

Like `QueueReceiverS`, `QueueBrowserS` can use message selectors to filter what messages they see in the queue:

```
QueueBrowser filterBrowser =
    qSession.createQueueBrowser(queue, "transaction-type = 'update'");
```

This `QueueBrowser` "sees" only messages in the queue that have a `transaction-type` property set to `update`.

To iterate over the messages in the queue, a client asks the browser for an `Enumeration` of the messages in the queue that match the browser's message selector, if it has one:

```
Enumeration msgEnum = browser.getEnumeration( );
while (msgEnum.hasMoreElements( )) {
    Message msg = (Message)msgEnum.nextElement( );
    System.out.println("Found message, ID = " + msg.getJMSMessageID( ));
}
```

The `Enumeration` returns messages in the order that they would be delivered to the client, using the message selector set on the `QueueBrowser`. So if you had an existing `QueueReceiver` and wanted to look ahead in the queue to see what messages would be delivered based on the current contents of the queue, you could create a browser using the same message selector as the receiver:

```
QueueReceiver recvr = ...;  
QueueBrowser recvrBrowser =  
    qSession.createQueueBrowser(queue, recvr.getMessageSelector( ));
```



11.5. Publish-Subscribe Messaging

Publish-subscribe messaging involves one or more `MessageProducers` "publishing" messages to a particular topic and one or more `MessageConsumers` "subscribing" to the topic and receiving any messages published to it. The JMS provider is responsible for delivering a copy of any message sent to a topic to all subscribers of the topic at the time that the message is received. Unlike point-to-point messaging, where messages are kept on the queue until a receiver reads them, any messages received at a topic while a subscriber is not active (e.g., hasn't subscribed to the topic yet or subscribed and then went out of scope or exited) are lost with respect to that subscriber.

Publish-subscribe messaging is performed in JMS using the topic-related interfaces and classes in the `javax.jms` package. Topics are represented by `Topic` objects, which are looked up in JNDI from the JMS provider. `TopicConnectionFactory` objects are looked up in JNDI as well and used to create `TopicConnections`. `TopicConnections` and `Topics` are used to create `TopicSessions`, which are in turn used to create `TopicPublishers` and `TopicSubscribers`.

11.5.1. Sample Client

The downloadable source code for this book includes a publish-subscribe client, `PubSubMessagingClient`, that mirrors the `PTPMessagingClient` in [Example 11-2](#). The structure and function of the client is virtually identical to that described for the `PTPMessagingClient`, except that topics, subscribers, and publishers are used instead of queues, receivers, and senders. The only significant difference with this client is it doesn't have a "browse" option, since browsing a topic is not possible. As they arrive, topics deliver their messages to any subscribers currently attached to the topic, otherwise they are dropped, so browsing a topic's contents doesn't make much sense.

11.5.2. Durable Subscriptions

If a client needs to guarantee delivery of messages from a `Topic` beyond the lifetime of a single subscriber, it can register a durable subscription with the JMS provider for the target `Topic`. A durable subscription to a `Topic` is made using the `createDurableSubscriber()` methods on a `TopicSession`. In its simplest form, a durable subscriber is created by specifying a `Topic` and a subscriber name:

```
TopicConnection tConn = ...;
tConn.setClientID("client-1");
TopicSession tSession =
    tConn.createTopicSession(false, Session.AUTO_ACKNOWLEDGE);
TopicSubscriber durableSub =
    tSession.createDurableSubscriber(topic, "subscriber-1");
```

This registers a durable subscription to the `Topic` under the name "subscriber-1." Durable subscriptions and their names are associated by the JMS provider with the client ID of the client that created them (see "[Client identifiers](#)" earlier in this chapter for details on client IDs). Here, we're setting our client ID to "client-1" by calling `setClientID()` on the `TopicConnection`.

As long as this `TopicSubscriber` is live, it will receive any messages published to the `Topic`, as it would if it were a nondurable subscriber. But if the subscriber dies (goes out of scope or the client dies), the JMS provider will retain messages on behalf of the named subscriber (based on its client identifier), until another durable subscriber attaches to the topic from the same connection using the same client ID and specifying the same subscriber name. Any pending messages will be delivered to the new subscriber when it attaches.

It's important to remember that durable subscriptions can be a costly resource on the JMS provider. The provider will have to create database records, or otherwise allocate server resources, in order to preserve the subscription information and any pending messages for the client. If there are many durable subscriptions or if the number of pending messages being held on the server for subscribers becomes large, this can eventually have a significant impact on the performance of the JMS provider. So durable subscriptions should be used with discretion.

[← PREVIOUS](#)

11.6. Unified Messaging

JMS 1.1 introduced new elements to the API that allow messaging clients to use the same interfaces and methods to do either point-to-point messaging or publish-subscribe messaging. In JMS 1.0 (used in J2EE 1.3 and earlier environments), a client needed to use the specific subclasses discussed in the previous sections.

As an example, our previous clients had to explicitly cast the `Destination` and `ConnectionFactory` objects retrieved from the JNDI context in order to create a `Session`. Our `PTPMessagingClient`, for example, had to perform the following sequence of steps in order to create a `Session` (a `QueueSession`, to be more precise):

```
Context ctx = new InitialContext( );
QueueConnectionFactory connFactory =
    (QueueConnectionFactory)ctx.lookup(cFactoryJNDIName);
QueueConnection mQueueConn =
    connFactory.createQueueConnection( );
Queue mQueue = (Queue)ctx.lookup(queueJNDIName);
QueueSession mSession =
    mQueueConn.createQueueSession(false,
        javax.jms.Session.AUTO_ACKNOWLEDGE);
```

Note that we've simplified this code from the original code in [Example 11-2](#) by eliminating a lot of exception handling.

With the new unified API elements added in JMS 1.1, a client can use the common base classes (`Destination` instead of `Queue` or `Topic`, `Connection` instead of `QueueConnection` or `TopicConnection`, etc.). With JMS 1.1, we can shorten that code example considerably:

```
Context ctx = new InitialContext( );
ConnectionFactory connFactory =
    (ConnectionFactory)ctx.lookup(cFactoryJNDIName);
Connection mConn = connFactory.createConnection( );
Destination mDestination =
    (Destination)ctx.lookup(destJNDIName);
Session mSession =
    mConn.createSession(false,
        javax.jms.Session.AUTO_ACKNOWLEDGE);
```

This generic version of the code to create a `Session` is made possible by the addition of the `ConnectionFactory.createConnection()` method and the `Connection.createSession()` methods to

the JMS API in Version 1.1. Other new elements in the API, not shown here, allow you to create `MessageConsumerS`, `MessageProducerS`, and other messaging entities in a generic way.

These new unified messaging aspects in JMS 1.1 allow you to simplify your messaging code quite a bit. You can create connections and exchange messages the same way no matter which messaging model is in use. The unified API elements also make it much easier to mix point-to-point and publish-subscribe messaging operations within the same session/transaction context. The downloadable source code for this book includes an example called *MixedMessagingClient.java* that shows how to merge our point-to-point and publish-subscribe client using the unified API elements in JMS 1.1.

Note that any earlier code that used the explicit `Queue` and `Topic` subclasses of the various JMS interfaces (such as our `PTPMessagingClient` and `PubSubMessagingClient` examples) will still work just fine in a JMS 1.1 environment. But any code that uses the new unified API elements, such as the preceding example, will work only in JMS 1.1 and later.



11.7. Transactional Messaging

JMS supports transactional messaging in two ways. In its simplest form, a `Session` is created with the transactional option (the first argument to `QueueConnection.createQueueSession()` and `TopicConnection.createTopicSession()`):

```
QueueSession xactSession =
    qConn.createQueueSession(TRUE, Session.AUTO_ACKNOWLEDGE);
```

When using a transactional `Session`, the client performs a series of "transactions" with the `Session` (sends or receives messages from consumers and producers associated with the `Session`). These sends and receives are either committed by calling the `Session`'s `commit()` method or cancelled by calling `rollback()`. If a `Session` is committed, all of the sends and receives are committed to the JMS provider, which causes the new state of the `Destination(s)` affected to be committed. If a `Session` is rolled back, all changes to the resources on the JMS provider are rolled back. In either case, the transaction is closed and a new one is started automatically, for any subsequent messaging actions.

JMS providers can also support transactional messaging through the Java Transaction API (JTA), which allows messaging transactions to be integrated with other resources, like databases. For broader coverage of JTA in a variety of contexts, see [Chapter 16](#). The following material highlights some transactional details specific to JMS messaging.

JTA-based transactions are distributed: the underlying transactional resources can be distributed across the enterprise. The JMS API supports transactional messaging with a set of interfaces that provide access to JTA-aware `Connections` and `Sessions`. If a JMS provider supports JTA, it can export an `XAConnectionFactory` in its JNDI space. An `XAConnectionFactory` is used to create `XAConnections`, and `XAConnections` are used to create `XASessions`. An `XASession` is a specialization of `Session` that overloads the `commit()` and `rollback()` methods to implement them within a JTA context. There are subclasses of these XA interfaces for point-to-point and publish-subscribe messaging. For example, to create a JTA-aware `TopicSession`:

```
XATopicConnectionFactory xFactory =
    (XATopicConnectionFactory)ctx.lookup("xact-factory");
XATopicConnection xConn = xFactory.createXATopicConnection( );
XATopicSession xSession = xConn.createXATopicSession( );
```

When a client performs a series of sends/receives with a JTA-aware `Session`, these actions are performed in the context of the surrounding `UserTransaction`, if one exists. For example, if we use our `XATopicSession` to create a `TopicPublisher` (`xPublisher`) and a `TopicSubscriber` (`xSubscriber`),

we can create our own JTA transaction and use it to commit or roll back a series of message operations:

```
javax.transaction.UserTransaction xaction = ...;
xaction.start( );
try {
    Message request = ...;
    Message response = ...;
    xPublisher.publish(request);
    response = xSubscriber.receive( );
    // Made it here, so commit the topic changes
    xaction.commit( );
}
catch (JMSEException je) {
    // Something bad happened, so cancel the topic changes caused
    // by our message sends/receives
    xaction.rollback( );
}
```

[← PREV](#)

Chapter 12. Web Services with JAX-RPC and SAAJ

Web services are a very resilient and powerful tool in the enterprise architecture toolbox. They were born from the marriage of XML, a very flexible data formatting language, with HTTP and other similarly ubiquitous communication protocols.

[← PREV](#)

12.1. What's Covered Here?

There have been a few generations of "web services" over the years. XML-RPC was arguably the first XML-based services scheme that was used widely, and it is still available today in various programming environments. More recently, a standard set of protocols for XML-based services has been defined and adopted by the software community. The primary protocol in this family is the Simple Object Access Protocol (SOAP).

In this chapter, we cover the standard APIs used in Java for implementing and using SOAP-based web services. These include the Java API for XML-based RPC (JAX-RPC) and the SOAP with Attachments API for Java (SAAJ). We will cover the versions included in the J2EE 1.4 specification, and that translates to JAX-RPC Version 1.1 and SAAJ Version 1.2.

We will also be discussing the Web Services for J2EE specification, Version 1.1. This API is also required in J2EE 1.4 servers and defines a standard deployment model for web services within a J2EE environment.

This chapter begins with a general overview of web services and the SOAP and Web Service Descriptor Language (WSDL) protocols that modern-day web services use predominantly. The bulk of the chapter, though, discusses how to build web services and web service clients in the Java/J2EE framework. The examples cover both the standard J2EE approaches to web services (in terms of APIs, tools, and deployment procedures) as well as the somewhat nonstandard but very popular Axis web service engine. We've tested the examples discussed in this chapter using both Axis 1.2, the Apache open source implementation of the JAX-RPC and SAAJ APIs, and the JBossWS engine bundled in the JBoss 4.0 application server. Both of these web service engines support the JAX-RPC and SAAJ APIs, but Axis implements a legacy, nonstandard deployment process for web services while JBossWS implements the newer, standardized deployment process specified in the Web Services for J2EE specification. In addition, the two engines use slightly different tools for managing the web service development process. We demonstrate both here, rather than picking one, because while Axis remains a popular web service engine, the standard web service deployment scheme is the presumed direction forward and makes for more portable Java-based web services. We also discuss both engines because contrasting the two deployment schemes highlights the basic issues with web service deployment in a Java environment and will help you to more easily navigate some other Java web service environments if the need arises.

This chapter focuses primarily on developing, deploying, and using web services in your enterprise applications. For additional relevant details, you may want to refer to [Appendix A](#), where we provide details on the deployment descriptors for JAX-RPC web services, as well as [Chapter 21](#), where we discuss code annotation facilities that can be used to automatically generate these deployment descriptors, as well as actual source code, for web services.

12.2. Brief Introduction to Web Services

Let's start with a brief overview of web services and the basic protocols used in web services: SOAP and WSDL. If you're already familiar with web services (perhaps from experience developing them or other platforms) and you want to dive into the details of using web services in Java, you can skip directly to the next main section, "Java Web Services."

A web service is a software component that communicates using XML-formatted messages, transmitted over a standard network protocol such as HTTP (the core protocol behind the Web) or SMTP (the core protocol used by most email servers). These XML messages are formatted using SOAP. A web service can also (optionally) export a description of the service it provides, which includes the operations that it can perform and the specific types and formats of the messages it expects. These service descriptors are typically in WSDL. The WSDL descriptor for a web service describes the types of data, messages, and operations that a web service supports and where the service can be located on the network. A client can use this WSDL service description to automatically generate client-side interfaces to interact with web services.

Web services can also be (optionally) registered with a web service registry. Without a web service registry, a client needs to acquire the WSDL description of the service through some direct, ad hoc channel, such as picking up the WSDL from a local file or loading it from a predetermined web site. A registry based on an industry standard, such as UDDI or ebXML, allows a client to acquire the WSDL for a service using a well-defined protocol and a protocol that supports more functionality, such as searching for web services based on their capabilities. Discussion of web service registries and UDDI/ebXML is outside the scope of this chapter, but if you're interested in using or creating directories of web services, please consult *Java Web Services* or *Java Web Services in a Nutshell* by Kim Topley, both from O'Reilly.

12.2.1. Simple Object Access Protocol (SOAP)

The SOAP acronym is the butt of more than a few jokes in the industry, since SOAP is not really that simple, and there are no true objects in the specification. Still, it's easy to remember, which is at least half of the purpose of most acronyms. SOAP is an industry standard that is managed and published by the World Wide Web Consortium (W3C). The current version of the SOAP specification can always be found on the W3C web site at the URL <http://www.w3.org/TR/SOAP>.

It's important to keep in mind that SOAP defines only the format for the messages that web services send and receive. It doesn't define an API at all. SOAP defines only how to package a message into a standard format that can then be delivered using a standard transfer protocol like HTTP or SMTP. APIs come into the picture when you need to generate and consume SOAP messages within a particular programming environment.

So what does a SOAP message look like? Well, every SOAP message must include an envelope and a body. The envelope contains information about how the message is encoded and definitions for various XML schemas that are used in the message contents. These XML schemas provide

descriptions of various XML elements that are used in the SOAP message and can be used to both construct and validate SOAP messages and XML documents in general. More details on XML schemas can be found in [Chapter 7](#). The body of the message contains XML-encoded information. The information can represent a specific request made by a client to a web service, it can be information returned as the result of a request, or it can be details on why a web service request failed.

[Figure 12-1](#) shows a sample SOAP message demonstrating a remote procedure call being made by a client to a web service via HTTP. This sample SOAP message is being sent over HTTP, so HTTP headers specify the HTTP request type (`POST`, in this case), the actual contents of the HTTP request, and the HTTP location to which the request is being sent (<http://myservices.com/myapp/services/peopleFinder> in this example). There is also a SOAP-specific custom header, `SOAPAction`, that specifies the entry point of the web service that should receive the enclosed SOAP message if the HTTP URL is not sufficient. In this example, no additional information is needed to locate the service on the server, so the `SOAPAction` header is blank. The body of the HTTP request contains the entirety of the SOAP message being sent to the web service. In this example, the envelope contains references to a number of XML Schema namespaces being invoked in the message and an indication that the message is using the default SOAP encoding style for the message. The header contains a single element, which is using a custom application schema to specify some kind of access control parameter for the message. Finally, the body of the SOAP message contains a request to invoke the `findPeople` operation on the target web service, with a single argument, an array of `SearchArg` XML elements.

Figure 12-1. Sample SOAP message



A SOAP web service, therefore, is a runtime entity that can accept SOAP messages such as the one shown in [Figure 12-1](#), do what's requested in the message, and generate an appropriate response in the same general SOAP format.

Luckily, it's unlikely that you'll need to directly parse or generate the SOAP content in these messages. Instead, you'll rely on a SOAP web service engine of some kind to convert your native programming entities (objects, data structures, and the like) into SOAP and back again. The process of converting programming entities to SOAP XML is called *serialization* (the same term used in RMI and `java.io` to refer to converting a Java object to a binary, streamable format). The process of converting the SOAP XML back into programming entities is called *deserialization*. In the Java/J2EE environment, SOAP serialization and deserialization are handled by a Java web service engine, such as Axis or JBossWS.

12.2.2. Web Service Description Language (WSDL)

Suppose you've written a web service, and now you want to advertise your service to the world. You need some way to tell potential users about the service to describe things like where the service is available on the network, what operations it supports, what message formats it expects, and what message formats your clients can expect to receive in response to their requests. There are a number of ad hoc ways to do this: you could post example messages to and from your web service to demonstrate how a client should interact with the service, you could write up the relevant details as a help document and post it on a web page somewhere, you could write sample client code for your web service in various languages and distribute it to potential users.

The problem with all of these approaches is that they're ad hoc and therefore not easily reproducible for either the web service publisher or the clients. If you write another web service, you need to create a similar set of sample messages, hand-written documentation, or client libraries, and developers of clients for the service will need to manually incorporate the new service based on this ad hoc information. And meanwhile, people offering their own web services are going to describe them in their own ad hoc way, making the whole situation pretty confusing and error-prone.

Ideally, there would be a standard format for describing web services that could be used by the publishers and consumers of web services. Publishers of web services would generate the description of each of their web services in the standard format, and consumers would take these descriptions and use them to generate calls to the services from their application environments. And since the web service descriptions are standardized, tools can be written to autogenerate descriptors for web services and to autogenerate client code from existing service descriptors.

Thankfully, smart people have already thought of this, and the result is WSDL. WSDL is an XML format that describes web services in terms of the operations that they support and the messages and data types that these operations expect as arguments and return as responses. WSDL uses a set of XML entities to describe web services. [Example 12-1](#) shows the general contents of a WSDL descriptor.

Example 12-1. Skeleton WSDL descriptor

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions
  targetNamespace="http://myserver.com/services/peopleFinder"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
  <wsdl:types>
    <schema . . .>
      <complexType name="type1">
        . . .
      </complexType>
      . . .
    </schema>
    . . .
  </wsdl:types>
  <wsdl:message name="message1">
```

```

        <wsdl:part name="messageRequest" type="ns:type1"/>
</wsdl:message>
. . .
<wsdl:portType name="PeopleFinder">
    <wsdl:operation name="findPeople" parameterOrder="args">
        <wsdl:input message="impl:messageRequest"
            name="findPeopleRequest"/>
        <wsdl:output . . ./>
        <wsdl:fault . . ./>
    </wsdl:operation>
    . . .
</wsdl:portType>
<wsdl:binding name="peopleFinderSoapBinding"
    type="impl:PeopleFinderImpl">
    <wsdlsoap:binding style="rpc"
        transport="http://schemas.xmlsoap.org/soap/http"/>
    <wsdl:operation name="findPeople">
        . . .
    </wsdl:operation>
</wsdl:binding>
. . .
<wsdl:service name="PeopleFinderImplService">
    <wsdl:port binding="impl:peopleFinderSoapBinding"
        name="peopleFinder">
        <wsdlsoap:address
            location="http://myserver.com:8080/services/peopleFinder"/>
    </wsdl:port>
</wsdl:service>
. . .
</wsdl:definitions>

```

In order to make sense of the WSDL format, it helps to think of it in terms of the entities that go into a WSDL description of a web service. These entities and the bottom-up approach that WSDL uses to define services are shown in [Figure 12-2](#).

At the beginning of the WSDL food chain are *data types*, which are the building blocks for SOAP messages. A data type can be as simple as a string, or it can be a complicated XML structure with internal hierarchy. Here is the WSDL data type containing data about a person, consisting of three strings (first name, last name, and ID):

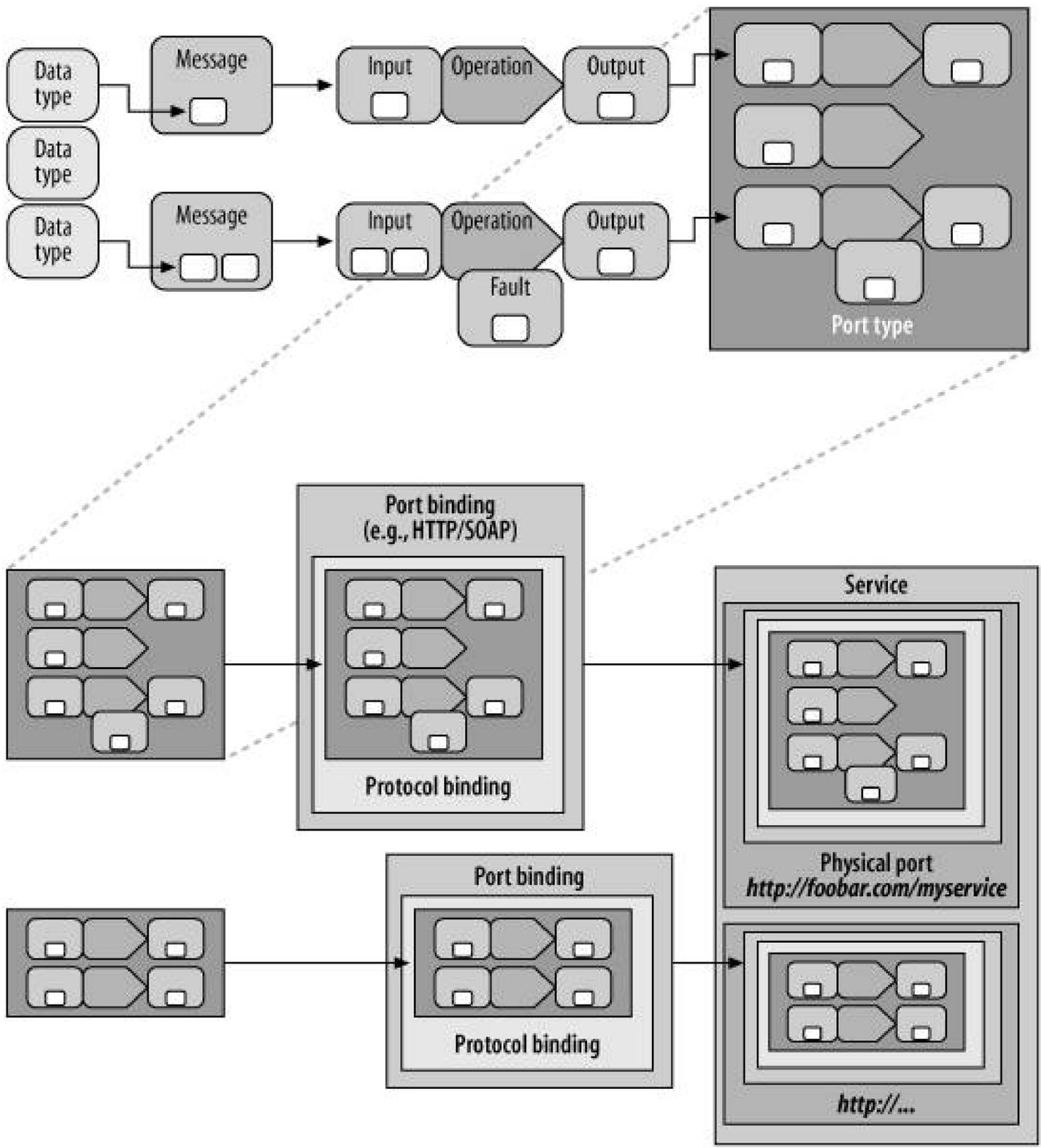
```

<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions . . .>
    <wsdl:types>
        <schema . . .>
            <complexType name="Person">
                <sequence>
                    <element name="firstName" nillable="true"
                        type="soapenc:string"/>
                    <element name="lastName" nillable="true"

```

```
        type="soapenc:string"/>  
    <element name="id" nillable="true"  
        type="soapenc:string"/>  
  </sequence>  
</complexType>  
</schema>  
.  
.  
.  
</wsdl:definitions>
```

Figure 12-2. WSDL entity model



A *message* is one or more data types grouped together. Messages are used as the inputs, outputs, and faults used by operations. Here is the WSDL definition of a message containing a *Person* data entity:

```
<wsdl:definitions . . .>
. . .
<wsdl:message name="findPersonResponse">
  <wsdl:part name="findPersonReturn" type="impl:Person"/>
</wsdl:message>
. . .
</wsdl:definitions>
```

An *operation* represents the atomic unit of work for a web service clients invoke operations on the service to get work done. *Port types* are groupings of operations into sensible interfaces. The following example defines a port type that contains a single operation. The operation has input and output messages as well as fault messages that can be thrown if an error occurs:

```
. . .
<wsdl:portType name="PeopleFinderImpl">
  <wsdl:operation name="findPerson" parameterOrder="args">
    <wsdl:input message="impl:findPersonRequest"
      name="findPersonRequest"/>
    <wsdl:output message="impl:findPersonResponse"
      name="findPersonResponse"/>
    <wsdl:fault message="impl:BadArgumentFault"
      name="BadArgumentFault"/>
    <wsdl:fault message="impl:NoSuchPersonFault"
      name="NoSuchPersonFault"/>
  </wsdl:operation>
</wsdl:portType>
. . .
```

A port type represents a conceptual description of a web service entry point. But a potential client of the service entry point still needs to know how this port type has been "instantiated." In other words how and where is this port callable? Over HTTP or SMTP or some other communication scheme? Using SOAP or some other encoding protocol? At what physical location(s) on the network? WSDL provides this information in the form of bindings and physical service descriptions. A binding describes how a particular port type has been bound to a communication and encoding protocol. The following shows the *PeopleFinderImpl* port type bound to SOAP over HTTP, for example:

```
. . .
<wsdl:binding name="peopleFinderSoapBinding" type="impl:PeopleFinderImpl">
  <wsdlsoap:binding style="rpc"
```

```

        transport="http://schemas.xmlsoap.org/soap/http"/>
<wsdl:operation name="findPerson">
  <wsdlsoap:operation soapAction="urn:findPerson"/>
  <wsdl:input name="findPeopleRequest">
    <wsdlsoap:body
      encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
      namespace="http://myservices" use="encoded"/>
  </wsdl:input>
  <wsdl:output name="findPeopleResponse">
    <wsdlsoap:body
      encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
      namespace="http://myservices" use="encoded"/>
  </wsdl:output>
  <wsdl:fault name="BadArgumentFault">
    <wsdlsoap:fault
      encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
      name="BadArgumentFault"
      namespace="http://myservices" use="encoded"/>
  </wsdl:fault>
  . . .
</wsdl:operation>
</wsdl:binding>
. . .

```

Finally, the following shows a WSDL service, which ties the HTTP/SOAP binding of the port type to a physical address that can be used by clients:

```

. . .
<wsdl:service name="PeopleFinderImplService">
  <wsdl:port binding="impl:peopleFinderSoapBinding" name="peopleFinder">
    <wsdlsoap:address
      location="http://myserver.com:8080//services/peopleFinder"/>
  </wsdl:port>
</wsdl:service>
. . .

```

This hierarchical definition of a service provides a client with all the information it needs to determine whether it can talk to this service (i.e., does it use protocols and encodings that the client can handle?) and to actually make a connection to a physical endpoint (in this case, a URL).

12.2.3. Web Service Styles and Encoding

Before we jump into the Java implementations of these web service protocols, this is a good point to briefly discuss an often confusing aspect of web services: *styles* and *encodings*.

As we saw in the previous discussion of SOAP and WSDL, web services are made up of ports. Each port has one or more operations; each operation has input, output, and fault messages associated with it; and each of these messages is composed of XML data types. SOAP messages sent to a service and SOAP messages generated from a service need to be composed in such a way that it is clear which operation in which port is in effect and how the message types are expressed in the XML of the message. SOAP messages to and from a service can be formatted in several ways, and these formatting options boil down to picking a style and an encoding for the body of the SOAP message.

The *style* of the message refers to how the XML structure of the SOAP body is structured. The two options for the style are:

RPC

In this style, the XML of the SOAP body is structured to match the operation and message definitions. The root element of the body is named after the operation name in effect, for example, and there is one child element for each message (input or output, depending on whether the operation is receiving or sending the SOAP message). In the WSDL for the service each message definition can contain multiple parts, and each part references an XML data type (one defined in the WSDL, in the standard XML Schema data types, or in an XML data type imported in the WSDL).

Document

In this style, the root element of the SOAP body is matched to the operation name, but the contents of the root element are not reflective of the signature of the operation. Instead, a single XML data type specifies the entire structure of the SOAP message body, and that structure can be in whatever form you like. In the WSDL for the service, each message definition contains a single part, and the part references the XML data type that defines the structure of the message body.

The names of these styles might lead you to assume that RPC style can be used only for bidirectional procedure calls, while document style can be used only for one-way messaging. But that's not the case. These styles do lend themselves well to these respective communication styles (hence their names), but they can actually be used in any context.

The other SOAP formatting option is the *encoding* of the message. The encoding determines how data is encoded into XML within the message. The two options for the encoding of service messages are:

Literal

This option specifies that the XML structure of the data within the message will be totally specified by XML Schema elements referenced in the WSDL description of the service. The XML data types in the WSDL (or imported by the WSDL) will be used literally to encode the XML data in the message.

SOAP-encoded

This option requests that the message data follow the default encoding rules (and XML structures) specified in the SOAP specification. These rules specify things like "each data member on a particular message element will be represented as a child XML element," whereas the literal encoding style lets you determine whether a particular piece of data should be an element or an attribute.

The combination of a style and an encoding option results in an overall format for a web service. The two combinations that are most commonly used are document/literal and RPC/encoded. The details on when and why these two service formats are preferred (or not) are beyond the scope of this short tutorial. We mention these details so that you are aware that there are different options in formatting the SOAP messages for your service, in case interoperability issues arise with your web services, for example. The web service examples in this chapter all have been implemented and deployed using the RPC/encoded format.



12.3. Java Web Services

There are several standard APIs for implementing and using web services in Java. The principal ones we'll discuss here are the Java API for XML Remote Procedure Calls (JAX-RPC), the SOAP with Attachments API for Java (SAAJ), and the Web Services for J2EE specification. JAX-RPC and SAAJ are APIs aimed at helping you develop both web services and web service clients. These web service clients might be standalone Java applications or code running within a J2EE application (e.g., JSP or servlet web components, EJBs, etc). The Web Services for J2EE specification describes a standard approach for the packaging and deployment of web services.

Relative to the standard web service protocols discussed earlier (SOAP, WSDL, etc.), JAX-RPC and SAAJ together provide a standard Java framework for consuming SOAP messages from web services and for generating SOAP messages and delivering them to web services. In a sense, you can think of JAX-RPC and SAAJ as being in the same category as other remote object APIs in Java, such as RMI and the Java bindings of CORBA. The web service APIs in Java use a different set of protocols to accomplish their remote object and messaging services.

12.3.1. Mapping SOAP and WSDL to Java

JAX-RPC provides more than just an API for consuming and producing web services. It also defines standard mappings from SOAP and WSDL entities to Java entities and back again. This is a very important asset, since it makes it clear how various web service data types will be represented in Java, and it allows your Java web service clients and implementations to be much more portable across different JAX-RPC and SAAJ implementations.

[Table 12-1](#) shows the JAX-RPC mapping from standard XML Schema data types to Java types. In all of these mappings, the `xsd:` prefix on an XML type refers to an entity defined in the XML Schema basic data types. Each XML data type has two Java mappings. The first mapping is used by default for that XML type. The second mapping is used if the XML entity is used in a context in which its presence is optional. Examples of these "optional" situations are XML elements that are marked as `nillable` in their schema, elements whose `minOccurs` attribute is zero and `maxOccurs` attribute is nonzero, and attributes whose `use` attribute is set to `optional` without specifying a default value.

Table 12-1. JAX-RPC mappings from XML types to Java types

XML type	Java type (default)	Java type if optional in XML
<code>xsd:anySimpleType</code>	<code>java.net.URI</code> (J2SE 1.4 and later) or <code>java.lang.String</code>	Same
<code>xsd:anyURI</code>	<code>java.lang.String</code>	Same

XML type	Java type (default)	Java type if optional in XML
xsd:base64Binary	byte[]	Same
xsd:boolean	boolean	java.lang.Boolean
xsd:byte	byte	java.lang.Byte
xsd:date	java.util.Calendar	Same
xsd:dateTime	java.util.Calendar	Same
xsd:decimal	java.math.BigDecimal	Same
xsd:double	double	java.lang.Double
xsd:float	float	java.lang.Float
xsd:hexBinary	byte[]	Same
xsd:int	int	java.lang.Integer
xsd:integer	java.math.BigInteger	Same
xsd:long	long	java.lang.Long
xsd:QName	javax.xml.namespace.QName	Same
xsd:short	short	java.lang.Short
xsd:string	java.lang.String	Same
xsd:time	java.util.Calendar	Same
xsd:unsignedByte	short	java.lang.Short
xsd:unsignedInt	long	java.lang.Long
xsd:unsignedShort	int	java.lang.Integer

When mapping Java entities into equivalent XML/SOAP entities (e.g., when a Java-based web service is being published), JAX-RPC specifies the mappings shown in [Table 12-2](#). Notice that Java basic data types, like `int`, and their wrapper classes, like `java.lang.Integer`, are mapped to the same XML data type, but the wrapper class is mapped with the `nillable` attribute enabled. A Java `int`, for example, will be mapped to an XML Schema like this:

```
<xsd:element name="myintvar" type="xsd:int"/>
```

while a `java.lang.Integer` will be mapped like this:

```
<xsd:element name="myIntegerVar" type="xsd:int" nillable="true"/>
```

The `nillable` attribute for the `Object` data wrappers allows for the situation in which a `null` value needs to be mapped from Java to XML.

Table 12-2. JAX-RPC mappings from Java types to XML types

Java type	XML type
java.math.BigDecimal	xsd:decimal
java.math.BigInteger	xsd:integer
boolean	xsd:boolean
java.lang.Boolean	xsd:boolean ^a
byte	xsd:byte
java.lang.Byte	xsd:byte ^a
java.util.Calendar	xsd:dateTime
java.util.Date	xsd:dateTime
double	xsd:double
java.lang.Double	xsd:double ^a
float	xsd:float
java.lang.Float	xsd:float ^a
int	xsd:int
java.lang.Integer	xsd:int ^a
long	xsd:long
java.lang.Long	xsd:long ^a
short	xsd:short
java.lang.Short	xsd:short ^a
java.lang.String	xsd:string
javax.xml.namespace.QName	xsd:QName
java.net.URI	xsd:anyURI

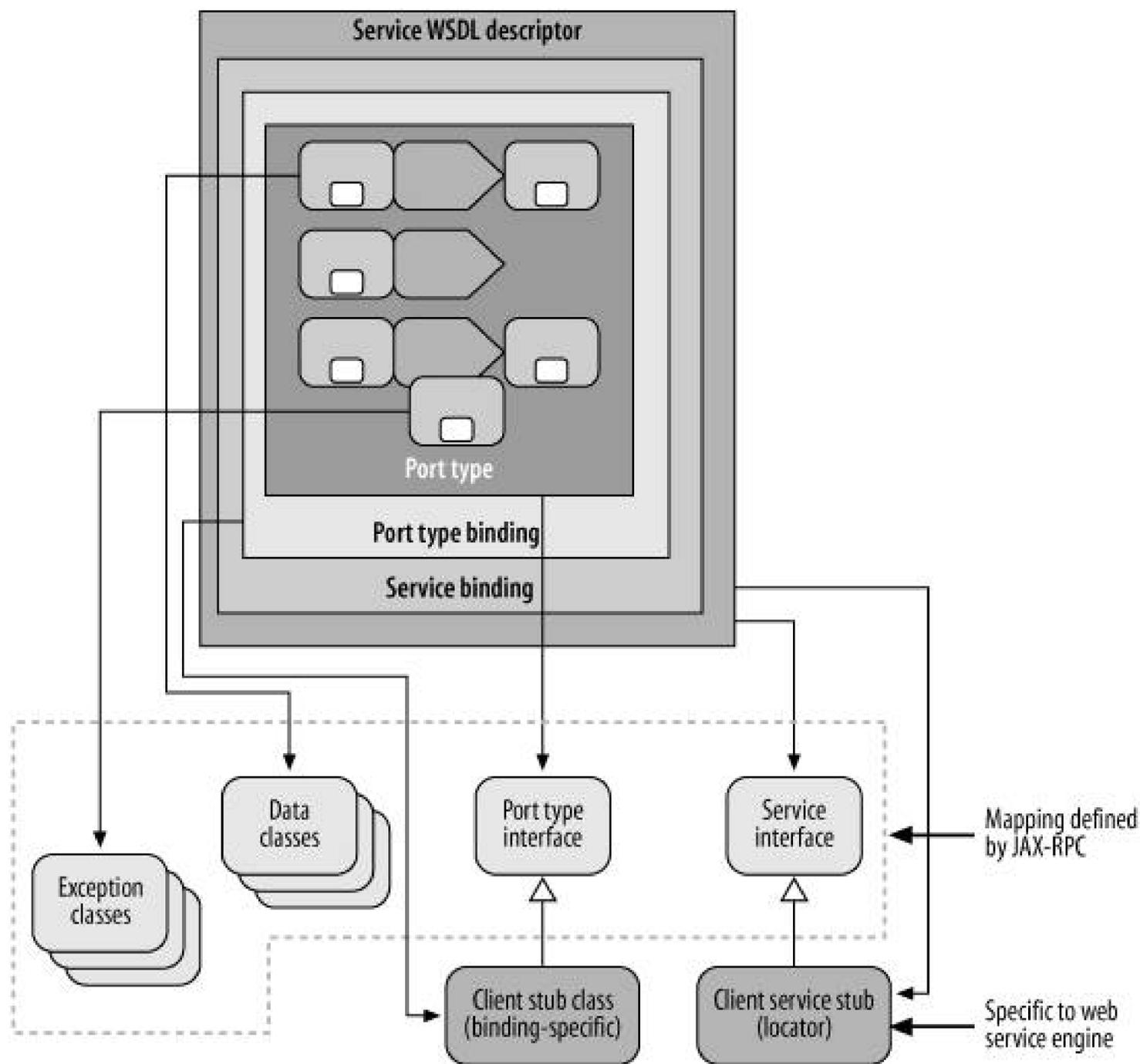
^a This type has the `nillable` attribute.

On top of these basic data type mappings, JAX-RPC also specifies how various complex XML types and structures are mapped to Java. Both the SOAP encoding and the WSDL schemas define an XML `Array` type, and these are mapped to equivalent Java arrays. An `<xsd:complexType>` structure is mapped to an equivalent JavaBean class whose name is mapped from the name attribute of the `complexType`, and whose properties are mapped from the elements defined for the `complexType` using these mappings.

Finally, in addition to these type mappings, JAX-RPC defines a set of standard mappings from WSDL entities to Java entities, as depicted in [Figure 12-3](#). This mapping is used in both directions by a Java web service engine. WSDL entities describing an external web service are mapped into Java entities for use by a Java client of the web service. A Java web service will also have its WSDL description

mapped from its interface and implementation classes using this same mapping. A web service engine will typically provide a tool that will implement this mapping. For clients, the tool will read in a WSDL file (either from the network or from a local file) and generate a set of equivalent Java code that can be used by a client to interact with the web service. This generated code serves as a client-side stub, or proxy, for the web service, performing the runtime mapping of standard Java objects, data, and method calls into their equivalent SOAP messages and back again. For service implementations, the tool will parse a set of Java classes that implement a service and generate a WSDL description for the service.

Figure 12-3. JAX-RPC entity mappings from WSDL to Java



12.3.2. Web Service Deployment

The Web Services for J2EE specification sits next to JAX-RPC and SAAJ as a complementary standard

aimed at defining standard approaches for describing web services to Java application servers and for packaging web services for deployment to these application servers. It plays a role for Java-based web services analogous to the deployment descriptors and archive formats defined for web components and EJB components. Describing web services is slightly more complicated, however, since you not only need to tell the application server about the Java elements that make up your web service, but you also need to tell it how to handle the mapping from Java objects to SOAP elements (and vice versa) in cases in which the standard mappings described earlier don't cover everything. Also, web services in Java can be implemented using a handful of different models (an EJB can serve as a web service, and so can a regular Java object running in a web container). Supporting these aspects of Java-based web services makes the deployment of web services in a J2EE environment a bit more involved in some cases.

In the remainder of this chapter, we're going to look at how to write Java clients for web services, how to implement web services in Java, and how to deploy Java-based web services. We're starting with clients because they are the easiest someone else has done the hard work of implementing a web service, and you "only" want to interact with it as a client. Writing Java-based web services isn't that much more difficult than writing a client, but deploying a web service in a Java application server involves some extra steps, and various web service engines and application servers introduce their own spin on this process.

[← PREV](#)

12.4. Writing Web Service Clients

A client of a service wants to use the service to get something done. When a service is implemented using SOAP, this means that the client needs to send SOAP messages to the service, targeted at specific operations and ports provided by the service, and if these ports and operations generate response messages, the client needs to receive these messages and interpret them.

In order to do this, the client needs a way to map the Java objects and data it has into the ports and operations and messages exposed by the SOAP service, and back again. You can accomplish this using JAX-RPC and SAAJ in several ways, which can be roughly grouped into the following three categories:

Static proxy

This approach involves mapping the service's WSDL descriptor to a set of client-side Java code. All classes needed by the client are pregenerated by this mapping, and the client uses them to interact with the service.

Dynamic proxy

This approach is similar to the static proxy, except that the stub used by the client is generated dynamically by the JAX-RPC runtime instead of using a class file being generated offline by mapping the WSDL. The client only needs to have a Java interface available that is compatible with the WSDL description of the service.

Dynamic invocation interface (DII)

In this approach, the client dynamically constructs a call to the target service, specifying all the particulars of the service explicitly when configuring the service call.

In terms of the mapping we mentioned (from Java to the service's ports, operations, and messages and back), these three options essentially boil down to doing the mapping once and encoding it "statically" as a set of concrete Java classes that the client uses (the static proxy approach); doing the same mapping dynamically at runtime, with the generated classes existing only at runtime (the dynamic proxy approach); or having the client do the mapping programmatically in its code, with no service interface classes being generated (the dynamic invocation interface (DII) approach).

Let's see each approach in detail, by creating clients that interact with a very simple web service that simply echoes a string back to the client. In its WSDL description, this service is named `EchoService` and has a single port named `echo`. This port has a single operation, also called `echo`, that accepts a message containing a single string and returns a message containing a single string. It is about as simple as a web service can get.

12.4.1. Static Proxy Approach

The static proxy approach for interacting with web services from a Java client is by far the simplest. It involves generating, prior to runtime, a set of Java artifacts mapped from the WSDL descriptor for the web service. The Java artifacts generated from the WSDL can be bundled with the client code, and the client code is very clean, with all the SOAP communication details hidden in generated `Service` and `Stub` implementations. In addition, since you have the actual mapped Java interfaces and classes on the client side, as a developer you can import these into your IDE and view/search/exercise these interfaces just like other Java code.

This approach is called the static proxy approach because the mapping performed by the web service engine results in a set of actual concrete Java class files. The client will use these classes as-is to interact with the service, and these classes implement a hardcoded mapping of service ports and operations to Java interfaces and methods, so it's in this sense that the mapping is static.

In Axis, the tool that generates these static proxy classes is called WSDL2Java. It's provided in two forms: a Java class, `org.apache.axis.wsdl.WSDL2Java`, that can be invoked from the command line to perform the mapping, and an Ant task, `axis-wsdl2java`, that can be used in your Ant build file to do the same.

If the WSDL for the Echo service were published at the URL <http://mywebservices.com/services/echo?wsdl>, we would then use WSDL2Java to generate client stubs from the command line:

```
> java org.apache.axis.wsdl.WSDL2Java
   -o /home/dev/generated-src
   http://mywebservices.com/services/echo?wsdl
```

This command generates a set of client-side Java classes mapped from the WSDL file and writes the Java source files into the `/home/dev/generated-src` directory that was specified using the `-o` option. In the case of our `echo` service, the following interfaces will be generated, according to the JAX-RPC mapping specifications:

Echo

The `echo` port in the WSDL is mapped to this interface, which extends `java.rmi.Remote`. JAX-RPC specifies that mapped class names should always be mixed-case, so `echo` becomes the interface name `Echo`. Each operation defined in the WSDL for this port type is mapped to a corresponding method on this interface, and the input and output messages for the operation are mapped to corresponding Java method arguments and return values. In our case, there is a single operation defined for the port, also named `echo`, that has an input message consisting of a single string and an output message consisting of a single string. This operation is mapped to an `echo()` method with a single `String` argument and a `String` return value. No SOAP faults are associated with this operation in the WSDL, so there are no service-specific exceptions in the `echo()` method declaration, only the `RemoteException` required for all RMI remote methods.

EchoService

In the WSDL, our service is named `EchoService`, so this Java interface is mapped from it. It extends the `javax.xml.rpc.Service` interface from the JAX-RPC API. For each port defined in the service, there is a `get<portname>()` method that returns an instance of the Java class mapped from that port. In our case, there is a `getechoPort()` method that returns an `Echo` stub. because the bound port is named `echoPort` in the service binding and in our WSDL. Note that the accessor methods are named after the name used for the port binding, not for the port definition itself. A single port can be bound to multiple protocols or physical endpoint URLs in the WSDL file, so the binding name is used to distinguish between them.

The web service engine also needs to provide concrete implementations of these mapped interfaces for the client to use. The names and implementation details of these concrete implementations are not specified by the JAX-RPC standard, so using these generated classes directly may tie your client code to a particular web service engine. In the case of Axis, the following concrete classes are generated from the WSDL:

EchoBindingStub

This is a concrete implementation of the `Echo` interface mapped from the port type. This class serves as a client-side stub: all of the methods mapped from the WSDL operations are implemented here as calls to the service endpoint.

EchoServiceLocator

This is an implementation of the `EchoService` interface mapped from the WSDL service. The `get<portname>()` methods declared in the interface are implemented here to construct an appropriate implementation of the interface mapped from the WSDL port type. In this case, the `getecho()` method from `EchoService` is implemented to return an instance of `EchoBindingStub`, the Axis concrete implementation of the `Echo` interface.

These interfaces and classes are mapped into packages according to the namespaces used in the WSDL for their corresponding XML entities. JAX-RPC does not specify this mapping it's left up to the web service engine to decide how to deal with this. If you don't tell it otherwise, Axis converts the namespace of each WSDL entity into a valid package name. For example, if the WSDL file used for the mapping specifies a namespace for its entities like so:

```
<wsdl:definitions targetNamespace="http://myservices/services/echo">
. . .
```

then Axis places all of the Java classes into the package `myservices.services.echo`. Often, this isn't the desired behavior a client will probably want to have the Java code fit into its own package structure. Axis allows you to configure the package mapping by specifying a properties file when invoking the WSDL2Java utility. You can specify a namespace-to-package mapping file using the `-N` option on the command line:

```
> java org.apache.axis.wsdl.WSDL2Java
  -o /home/dev/generated-src -N nsmap.properties
  http://mywebservices.com/services/echo?wsdl
```

The namespace-to-package mapping file is a Java properties file, with namespaces as the "property name" and the corresponding package name as the property value. A mapping file for our previous example might look like the following:

```
http\://myservices/services/echo=com.myapp.soap.echoclient
```

Here, we're saying that any entities found in the WSDL namespace <http://myservices/services/echo> should be mapped to the Java package `com.myapp.soap.echoclient`. Note the backslash in front of the colon in the namespace value: Java properties files use colons as an optional separator between names and values, so we have to escape the colon in the URL to prevent it from being interpreted this way. When setting up these package mapping files yourself, don't forget to escape the colon after the URL protocol and colons in front of any port numbers (e.g., <http://myservices.com:8080>).

It's important to remember that the only part of this mapping process that is specified in detail by the JAX-RPC standard is the mapped port type interface (`Echo`) and the name of the mapped service interface (`EchoService`), both of which are taken from the names and attributes of their corresponding WSDL entities. The concrete implementation classes of these interfaces can be named whatever the mapping tool wants to name them. In addition, the mapping tool can choose to map namespaces to packages in a totally different way if it chooses.

Given all this, the static proxy approach for building a client to a web service simply involves using the concrete stub class generated by the web service engine's tools. In the case of Axis, a client uses the `EchoSoapBindingStub` class to interact with the `echo` web service. Instances of the stub class are obtained using the concrete `Service` implementation as a factory, calling the `getecho()` method to generate a stub instance:

```
try {
    // Get a stub to the remote service
    Echo echo = new EchoServiceLocator( ).getechoPort( );

    // Invoke the operation and collect the result
    String resp = echo.echo(msg);
    System.out.println("Sent: \"\" + msg +
        "\"\", got response \"\" + resp + "\"");
}
catch (ServiceException se) {
    System.err.println("Failed to invoke service: \"\" +
        se.getMessage( ));
}
catch (RemoteException re) {
```

```

        System.err.println("Error invoking service: " +
            re.getMessage( ));
    }

```

The call to `getechoPort()` on the `Service` implementation generated by Axis returns an instance of `EchoBindingStub`, the proxy for the echo service. When we call the `echo()` method on the proxy, it internally constructs the appropriate SOAP message to the web service (based on the information gleaned from the WSDL), sends the message to the service endpoint, receives the SOAP response, extracts the string data, and returns it as the result of the method call. In the client code, though, this is simply a normal-looking method call on a Java object.

You may have noticed that nowhere in the client do we specify the URL location of the web service (the service endpoint) that is the target of our request. The `getechoPort()` method on the `Service` uses, by default, the endpoint URL found in the WSDL that was used to generate the client-side Java classes. The endpoint will be found in the service element of the WSDL description, for example:

```

. . .
<wsdl:service name="EchoService">
    <wsdl:port binding="impl:echoSoapBinding" name="echoPort">
        <wsdlsoap:address location="http://myservices.com/services/echo"/>
    </wsdl:port>
</wsdl:service>
. . .

```

If the client needs to contact an alternate instance of the web service running at a different location, Axis generates an alternate `getecho()` method on the mapped `Service` class. This version accepts a URL to indicate the service endpoint:

```

Echo echo =
    new EchoServiceLocator( ).getechoPort(new URL(
                                                "http://altserver.com/echo" ));
String resp = echo2.echo(msg);

```

12.4.2. Dynamic Proxy Approach

The dynamic proxy approach to interacting with a web service does all of the WSDL-to-Java mapping dynamically. The client asks the web service engine to generate a client stub class on the fly, at runtime. The client does this by creating a `javax.xml.rpc.Service` instance that references the WSDL descriptor for the target service. The client then uses the generic `getPort()` methods on the `Service` to dynamically generate a stub class that can be used to communicate with the target web service. In the dynamic proxy approach, since no mapped Java interfaces are generated from the WSDL, the client has to provide its own Java interface for the stub class, and this interface needs to be

compatible with the service.

The dynamic proxy approach is useful in cases in which a WSDL descriptor for a service is available but generating mapped Java code beforehand isn't desired or possible for some reason. One reason for this is portability across Java web service engines. By using the dynamic proxy approach described next, you avoid using the engine-specific concrete class implementations of the mapped WSDL interfaces (for Axis, these are the `EchoServiceLocator` and `EchoBindingStub` classes described earlier). So the client code can be more easily run using different web service engines or even different versions of the same web service engine.

Returning to our `echo` web service, if we wanted to invoke it using the dynamic proxy approach, we would do the following:

```
import javax.xml.rpc.ServiceFactory;
import javax.xml.rpc.Service;

. . .

// Define an inner class that serves as our stub interface to
// the echo service
interface MyEcho extends Remote {
    public String echo(String msg) throws RemoteException;
}

. . .

public void invokeService( ) {
    String msg = "Hi there";
    try {
        // The WSDL that describes the service we want to contact
        String wsdlLoc =
            "http://myservices.com/services/echo?wsdl";
        // The qualified name of the service within the WSDL doc
        QName serviceName =
            new QName("http://myservices.com/services/echo",
                "EchoService");

        ServiceFactory sFactory = ServiceFactory.newInstance( );
        Service service =
            sFactory.createService(new URL(wsdlLoc), serviceName);
        MyEcho echo = (MyEcho)service.getPort(MyEcho.class);
        // Invoke the operation and collect the result
        String resp = echo.echo(msg);
        System.out.println("Sent: \" + msg +
            "\", got response \" + resp + "\"");
    }
    catch (ServiceException se) {
        System.err.println("Service generated error: " + se.getMessage( ));
    }
    catch (RemoteException re) {
        System.err.println("Error invoking service: " + re.getMessage( ));
    }
}
```

```

    }
    catch (MalformedURLException mue) {
        System.err.println("Bad service endpoint");
    }
}
. . .

```

Here, our client code creates a `ServiceFactory` instance by invoking the static `ServiceFactory.newInstance()` method that creates a `ServiceFactory` instance provided by the resident web service engine in use. We use the `ServiceFactory` to create a `Service` that represents the particular web service we're targeting. The `createService()` method is called to do this, passing in the URL for a WSDL document that contains the service description, plus the qualified name of the target service within the document. In our case, the WSDL defines the service within the namespace <http://myservices.com/services/echo>, and the service is named `EchoService` in the WSDL, so we pass in a `QName`, `serviceName`, that represents this qualified name within the document. Finally, we ask the `Service` to dynamically map the target port of the service to our Java interface for the port by calling the `getPort()` method on the `Service`. In the call, we pass in the class we want the generated stub instance to implement, `MyEcho`. `Service` will create a concrete stub class on the fly, this stub class will extend the class provided in the `getPort()` call, and it will be connected to the requested port within the target web service.

The stub interface that the client provides must extend (directly or indirectly) `java.rmi.Remote`, and its interface has to be compatible with the WSDL description of the target port. In other words, the methods on the Java interface have to correspond with the operations defined for the port in the WSDL, according to the JAX-RPC mapping rules. If the web service engine has any problems doing the dynamic proxy generation, the `getPort()` method throws a `ServiceException`.

You may notice that we did not specify the qualified name of the port when we called `getPort()` to generate the stub. We just passed in the stub class that we wanted the generated stub instance to implement. When this version of `getPort()` is invoked, the web service engine is responsible for determining the "best match" between the ports defined in the WSDL document and the stub interface passed into the `getPort()` call. In the case of Axis, it first looks for a port type whose name matches the name of the Java interface. If that fails, it tries to use the first port defined in the WSDL. In our example, the Java interface name (`MyEcho`) does not match the target port name in the WSDL (`echo`). But only one port is defined in the WSDL, so that one will be mapped by the Axis runtime and our client will work fine. A more robust approach (and the one you really should use when more than one port is present in the WSDL) is to use the alternate version of `Service.getPort()`, which accepts the qualified name of the target port as well as the desired stub interface. In our case, we would invoke this version of `getPort()` like so:

```

QName portName =
    new QName("http://localhost:8080/JEnt-examples/services/echo",
        "echo");
MyEcho echo2 = (MyEcho)service.getPort(portName, MyEcho.class);

```

12.4.3. Dynamic Invocation Interface (DII) Approach

The DII approach puts the most burden on the client, because all of the service mapping details are provided by the client. But it also offers the most flexibility, since your client can construct a call to any web service without needing to define a stub interface or generate any proxy code beforehand. There is no stub class involved at all in the DII approach; the client programmatically constructs the call to the service itself and is responsible for ensuring that the arguments included in the call correspond to the messages expected by the web service. The client is also responsible for knowing what kind of response to expect from the web service.

The DII approach is best suited to situations in which the client can't assume much about the service to be contacted or simply doesn't have the information about the service prior to runtime. One typical scenario suited for DII is a service that does not publish a WSDL descriptor. With DII, you can still talk to the service by programmatically specifying the service configuration. DII is also useful in contexts in which the web service configuration needs to be externalized from the client application: either in configuration files that are read at runtime or in the user's head. A web service IDE, for example, might provide a dynamic web service test wizard, allowing the user to specify all the web service details. The IDE can use DII to generate the actual web service call based on the user's runtime inputs.

The DII approach involves creating a `Service` instance, as we saw in the dynamic proxy case. But instead of asking the `Service` to map the service to a proxy class, we ask the `Service` to give us a raw `javax.xml.rpc.Call` object that we can configure with all of our service call specifics. Here's our client of the echo service rewritten to use DII to make the service call:

```
// The endpoint, service, port, and operation we want to invoke
String endpoint = "http://localhost:8080/JEnt-examples/services/echo";
String serviceName = "EchoService";
String portName = "echo";
String operationName = "echo";

// Some namespaces we'll be using in our client call
String serviceNS = "http://localhost:8080/JEnt-examples/services/echo";
String xsdNS = "http://www.w3.org/2001/XMLSchema";
String jaxrpcNS = "http://java.sun.com/jax-rpc-ri/internal";

// Create qualified names for the service, port, and operation
QName serviceQName = new QName(serviceNS, serviceName);
QName portQName = new QName(serviceNS, portName);
QName operationQName = new QName(serviceNS, operationName);

try {
    // Make a service factory
    ServiceFactory factory = ServiceFactory.newInstance();
    // Create a reference to the service
    Service service = factory.createService(serviceQName);
    // Create a call to a particular port on the service
    Call call = service.createCall(portQName);
    // Set the endpoint URL for the call
    call.setTargetEndpointAddress(endpoint);

    // Set the return type of the operation we want to invoke
```

```

    call.setReturnType(new QName(xsdNS, "string"));
    // Set the operation we want to invoke
    call.setOperationName(operationQName);
    // Set the type of the single parameter to the operation
    call.addParameter("message", new QName(xsdNS, "string"),
        ParameterMode.IN);
    // Create the actual operation arguments and put them in an
    // object array
    String msg = "Hi there";
    Object[] opParams = {msg};
    // Invoke the operation and collect the result
    String resp = (String)call.invoke(opParams);
    System.out.println("Sent: \"" + msg +
        "\", got response \"" + resp + "\"");
}
catch (ServiceException se) {
    System.err.println("Service generated an error: " + se.getMessage());
}
catch (RemoteException re) {
    System.err.println("Communication error: " + re.getMessage());
}
}

```

We start by declaring a number of string parameters that indicate the names of various namespaces as well as the qualified names of the service, port, and operation we intend to invoke. Then we create a `ServiceFactory` as before. But, when we create the `Service`, we simply provide the qualified name of the service, with no WSDL description to go along with it. We're constructing the service call ourselves, and there's no mapping to be done by the web service engine: `Service` needs to know only its qualified name. We'll supply all the other details previously extracted from the WSDL when we configure the call to the service.

Once we have the `Service`, we ask it to create a `Call` for us, providing the qualified name of the port that we want to use. Once the `Call` is created, we set all of its parameters: the qualified name of the operation we want to invoke, the return type of that operation, and the type of arguments that the operation expects. This is all done using XML namespaces and qualified names within these namespaces to match the configuration of the service. For example, we specify the operation we want to invoke by setting the operation name on the `Call` to the qualified name of the target operation:

```

String serviceName = "EchoService";
String operationName = "echo";
. . .
QName operationQName = new QName(serviceNS, operationName);
. . .

call.setOperationName(operationQName);

```

We set the return type of the operation in a similar fashion, but since the operation we want to

invoke simply returns a string, the qualified name of the return type comes from a standard XML Schema namespace:

```
String xsdNS = "http://www.w3.org/2001/XMLSchema";  
.  
.  
.  
  
call.setReturnType(new QName(xsdNS, "string"));
```

Once the `Call` is configured, we invoke the target operation by passing our arguments into the `invoke()` method. The arguments are provided in the form of an `Object` array, which allows for any kind of arguments that any web service operation might require:

```
String msg = "Hi there";  
Object[] opParams = {msg};  
  
String resp = (String)call.invoke(opParams);
```

The return value of the `invoke()` call is the return value of the web service operation, in the form of an `Object`, that we then cast to the expected `String`.

12.5. Writing Web Services

As you might expect, implementing a SOAP web service is moderately more difficult than just using one as a client. Thankfully, JAX-RPC and the Java web service engines make the writing of the web service implementations themselves very simple (even simpler than writing a client, in some cases). The added complexity of implementing web services comes in the deployment step, which we'll discuss in the next major section.

12.5.1. Simple Java Web Services

If you look at the static proxy approach for implementing SOAP clients, you may notice a lot of similarities with the client stubs used in RMI for talking to remote RMI objects. That's not a coincidence. JAX-RPC uses the RMI programming model to provide a simple way to write web services in Java. The published interface for the web service is defined as a `Remote` interface. If we were implementing our simple `echo` web service in Java, we could define the interface for the service as shown in [Example 12-2](#).

Example 12-2. A JAX-RPC service interface

```
public interface IEcho extends java.rmi.Remote {
    public String echo(String message) throws java.rmi.RemoteException;
}
```

The implementation for the service is equally simple to define. We just write a concrete implementation of this `Remote` interface, as shown in [Example 12-3](#).

Example 12-3. A JAX-RPC service implementation

```
public class Echo implements IEcho {
    public Echo( ) {
        super( );
    }

    public String echo(String message) throws RemoteException {
        return message;
    }
}
```

That's it. The `IEcho` interface and the `Echo` concrete implementation together serve as an implementation of a web service in Java. As we mentioned earlier, the hard part is deploying a web service to an application server, which we'll cover shortly. But essentially, a web service engine will manage this web service, converting SOAP messages targeted for the service into equivalent method calls on the implementation class and converting the method return values or exceptions into SOAP messages or faults that are delivered back to the client making the request.

You need to keep a few concerns in mind when writing web services. These concerns are similar to those you face when writing RMI server objects, but SOAP web services introduce the additional concern of portability, because clients of your service could be implemented in many different development environments and languages.

Arguably, the most important issue to keep in mind is the mapping of your Java service implementation into WSDL and/or SOAP entities. As described earlier, JAX-RPC defines a set of standard mappings for a specific set of Java types into their corresponding XML Schema entities. These supported Java types include the basic data types (`int`, `boolean`, etc.), their `Object` wrapper classes (`Integer`, `Boolean`, etc.), Java interfaces that use the JavaBeans pattern, and arrays of these supported data types. You should make every effort to restrict your service implementations so that their interfaces expose only these supported Java types. Otherwise, it's likely that you'll run into interoperability issues with some clients, especially those implemented in other platforms.

As a concrete example, let's take a look at our example "white pages" `PeopleFinder` application (seen in [Chapters 2](#), [17](#), and [18](#)). Suppose we wanted to create a web service, using the RMI programming model described earlier, which exposes the search capabilities of the `PeopleFinder` system. If we looked at this as "just" a Java developer, ignoring any web service mapping or interoperability issues we might throw together this `Remote` interface for our web service:

```
public interface PeopleFinder extends java.rmi.Remote {

    public Collection findPeople(Map searchParams)
        throws InvalidSearchException, PersistenceException,
            RemoteException;

}
```

This interface makes the implementation of the service trivial. The `PersonDAO` that is the core of the system has a `search()` method that accepts a `Map` of the search arguments and returns a `Collection` of `Person` objects. By using this same signature for our web service method, our implementation of the `findPeople()` operation on the service interface would simply be:

```
public class PeopleFinderImpl implements PeopleFinder {

    . . .
    public Collection findPeople(HashMap searchArgs)
        throws InvalidSearchException, PersistenceException {
        return dao.findPeople(searchParamsMap);
    }
    . . .
}
```

```
}
```

The problem we will likely face with the web service is that no standard mappings are defined for Java collection objects into XML entities. So we don't know how the web service engine will deal with the `Map` method argument or the `Collection` return value. In fact, the web service engine would be within its rights to just throw this interface back at us and refuse to deploy it. How should it map a `Collection`, which by definition is simply a set of `Objects`? How can the web service engine possibly map this into WSDL entities in such a way that a client will know what to expect as a response from the operation? The `Collection` could contain just about anything `Strings`, `Integers`, complex JavaBeans structures, and so on. The same goes for the `Map` argument to the method: what should a web service client pass into this operation in order to get the desired behavior? Strings mapped to strings, something else? A much more clear, portable, and interoperable interface for our Java web service would be something like the following, which replaces the collections with arrays of specific types:

```
public interface PeopleFinder extends Remote {

    public Person[] findPeople(SearchArg[] args)
        throws InvalidSearchException, PersistenceException,
            RemoteException;

}
```

We've converted the `Collection` to an explicit array of `Person` beans, and we've introduced a new JavaBeans object, `SearchArg`, that holds a name/value pair representing a search argument:

```
public class SearchArg implements Serializable {
    private String mArgName = null;
    private String mArgValue = null;

    public SearchArg( ) { super( ); }
    public SearchArg(String name, String value) {
        setName(name);
        setValue(value);
    }

    public String getName( ) { return mArgName; }
    public void setName(String argName) { mArgName = argName; }
    public String getValue( ) { return mArgValue; }
    public void setValue(String argValue) { mArgValue = argValue; }
}
```

This makes our service implementation slightly more complicated, of course we'll have to convert the incoming array of `SearchArg` objects into a `Map` to pass into the `PersonDAO.findPeople()` method, and we'll have to convert the `Collection` returned from that method call into an array of `Person`

objects to return to the client. But we'll have a lot fewer interoperability issues with this version and fewer issues diagnosing mapping issues with our web service engine as well.

12.5.2. EJB Web Services

In addition to the "straight" Java implementation of a web service described in the previous section, JAX-RPC and the EJB specifications support the use of session EJBs as web services as well. Put another way, the previous section showed how to use the RMI programming model to implement a web service in Java, but you can also use the EJB programming model to implement a web service. Put yet another way, JAX-RPC and EJB provide a standard way to expose a session EJB component as a SOAP web service.

Only stateless session beans can support web service clients. Beyond that, the restrictions on the interface to an EJB that will be exposed as a web service are roughly the same as for web services implemented using the RMI programming model described in the previous section. You need to be mindful of the arguments and return values on the methods to be mapped to service operations to ensure that they can be cleanly mapped using a standard JAX-RPC mapping.

The following is the interface for an EJB implementation of our PeopleFinder service:

```
public interface PeopleFinder extends javax.ejb.EJBObject {
    public Person[] findPeople(SearchArg[] args)
        throws InvalidSearchException, PersistenceException,
            RemoteException;
}
```

This interface looks very similar to the one we saw earlier using the RMI model—the only difference is that we're extending `EJBObject` instead of `Remote`, in keeping with the EJB programming model for components.

Our EJB implementation class, `PeopleFinderBean`, has an implementation of the `findPeople()` method that is virtually identical to our earlier RMI version. The difference here, again, is the use of the EJB component model, which requires that a session bean have certain callback methods implemented as well, such as `ejbCreate()` and `setSessionContext()`. The purpose and implementations of these EJB-related methods are covered in [Chapter 6](#).

```
public class PeopleFinderBean implements javax.ejb.SessionBean {
    public PeopleFinderBean( ) {
        super( );
    }

    // EJB-related callback methods
    public void ejbActivate( ) throws EJBException, RemoteException {}
    public void ejbPassivate( ) throws EJBException, RemoteException {}
    public void ejbRemove( ) throws EJBException, RemoteException {}
    public void setSessionContext(SessionContext arg0)
```

```

        throws EJBException, RemoteException {}
    public void ejbCreate( ) throws CreateException {}

    // Perform the actual search, using the PersonDAO
    public Person[] findPeople(SearchArg[] args)
        throws InvalidSearchException, PersistenceException {
        Map searchArgsMap = new HashMap( );
        for (int i = 0; i < args.length; i++) {
            searchArgsMap.put(args[i].getName( ), args[i].getValue( ));
        }
        Collection people =
            PersonDAO.getInstance( ).findPeople(searchArgsMap);
        Person[] peopleArray = new Person[people.size( )];
        Iterator pIter = people.iterator( );
        int i = 0;
        while (pIter.hasNext( )) {
            peopleArray[i++] = (Person)pIter.next( );
        }
        return peopleArray;
    }
}

```

From the standpoint of implementing a web service, this session bean represents a web service implementation that will be managed by an EJB container, in terms of the lifecycle of the component, its security services, and so forth. These runtime management services are the real reason that having this EJB option for implementing a web service is interesting. If, for example, our web service needs to make heavy use of transactions (with databases, messaging services, ERP systems, etc.), having the transaction management services of an EJB container for our web service might be very useful. Similarly, if our web service is expected to receive high load levels from clients, the runtime instantiation and instance-pooling services of EJB could be valuable as well.

12.5.3. Nonstandard Implementation Schemes

There are several nonstandard, proprietary models for implementing Java-based web services in addition to the standard approaches specified in the JAX-RPC and related specifications. Typically, these implementation schemes are meant to provide a faster way to get a web service up and running but work only with a specific Java web service engine and therefore won't port to alternative engines.

12.5.3.1. Axis JWS approach

Apache Axis, for example, provides a very simple scheme called JWS for implementing a web service. You simply take the source file for a concrete Java class, rename it with a *.jws* extension, and place it somewhere in your application's web context. Assuming that your web service has the Axis engine servlet installed to handle all *.jws* files, the Axis engine automatically processes the *.jws* file when a client requests its URL and maps the code found in the JWS file to a web service, complete with a WSDL descriptor. For example, we could implement our earlier **Echo** web service using JWS with the

following Java class:

```
public class EchoJWS {  
    public String echo(String message) {  
        return message;  
    }  
}
```

If we put this Java code into an *EchoJWS.jws* file in the root of our web application (either in a war file or in the root of the expanded web application directory, depending on how we're deploying our application to the application server), then Axis will deploy this as a web service. If our application is deployed with a context root of *myApp* on the server myserver.com, the endpoint URL of the web service will be:

```
http://myserver.com/myApp/EchoJWS.jws
```

and the WSDL for the service will be available at the URL:

```
http://myserver.com/myApp/EchoJWS.jws?wsdl
```

The Axis JWS scheme for implementing web services has a number of limitations. The most important one is that the mapping process (from Java entities to WSDL and back) is not configurable. As we'll see in the upcoming section on deploying web services, the ability to configure the mapping of Java entities (interfaces, JavaBeans, exceptions, etc.) into XML entities for the SOAP service can be very useful, sometimes critical. If you need to use a complex Java structure that requires a custom mapping, for example, you cannot use JWS.

In addition to the lack of configurability, JWS-based web services cannot make use of Java packages. The JWS file must have no package declaration, and the file has to sit in the root of the web application context, not in a subdirectory. This limits your ability to organize and manage your web service implementation code and its corresponding endpoint URLs.

So while the JWS scheme implemented in Axis is useful for quick prototypes of simple web services, it's not likely that it will be sufficient for all of your needs for implementing web services in enterprise systems.

12.6. Deploying Web Services

As with the other J2EE components (EJB, web) and resources (JMS `Destinations`, JavaMail `Sessions`, JDBC `DataSources`, and so on) discussed in this book, web service implementations need to be deployed to an application server. And, as with these other things, deploying a web service isn't simply putting your code up on the server. You need to tell the web service engine how to manage your web services at runtime.

The web service engine is responsible for accepting SOAP requests from clients, converting them into appropriate method calls on your Java implementation code, and taking the results of those method calls and converting them into the appropriate SOAP responses to the clients. In order to do this, the web service engine needs to know the following information for each web service that you want to deploy:

Java implementation code

What Java code implements the web service, and what aspects of the Java code should be part of the web service?

Java-to-XML mapping

How should the Java entities in your service implementation (interfaces, data objects, exceptions, etc.) be converted into SOAP/XML entities and vice versa? This covers both the description of your web service in the form of a WSDL document and conversions between SOAP entities and Java entities at runtime when operations are invoked.

This information is provided in the form of deployment descriptors and/or configuration files. These typically include:

Web service deployment file

This file defines the web services that you want to run, with pointers to the Java classes and components that serve as the implementation for each service.

Mapping file

This file is used in those cases in which default mappings aren't sufficient and specifies how Java entities are to be mapped to XML and back.

To deploy a web service, you need to provide this information, package up your Java code (and any other necessary runtime resources) into an appropriate archive format, and deliver the code and the configuration information to the application server that will run the web service engine.

The actual deployment and configuration file formats that you will use, and the physical steps you take to deploy a web service, depend on the particular Java web service engine and application server in use. The Web Services for J2EE specification (JSR 109) provides a standard model for deploying web services, along with standard deployment file formats. Web service engines that are compliant with these specifications will support these deployment descriptors. Other web service engines still use their own deployment and configuration schemes, but the same basic information will be needed.

In the following section, we'll examine the standard J2EE model for deploying web services. Following that, we'll look at the nonstandard deployment model used by Axis.

12.6.1. J2EE Standard Model

As we saw in earlier sections, JAX-RPC defines two approaches for implementing web services: as a simple Java object using the RMI programming model or as an EJB. In the case of a simple Java object, the web service is run within a web container, and in the case of an EJB, the web service is run within an EJB container. The Web Services for J2EE specification spells out how web services are deployed in each of these cases. It lays out the information that must be provided within the existing component deployment descriptors (*web.xml* and *ejb-jar.xml*) and defines new deployment descriptors that associate specific components with their corresponding web services, as well as the mapping information for the various Java-to-XML conversions.

In general, the J2EE deployment model for web services involves the following steps. We'll go into these in more detail in the sections that follow.

1. *Deploy the implementation component*

The web service implementation needs to have a standard component entry in its component archive. For a Java object, this will be a `<servlet>` entry in a *web.xml*/deployment descriptor. For an EJB component, this will be a `<session-bean>` entry in an *ejb-jar.xml*/deployment descriptor.

2. *Create a web service deployment descriptor*

This takes the form of a *webservices.xml*/file that is included in the component archive (web or EJB). It lists each web service being deployed within that component archive, linking the web service to a particular component and specifying the WSDL file for the service and the Java-to-XML mapping file for the service.

3. *Generate a WSDL descriptor for the web service*

Each web service must come bundled with a WSDL descriptor within the component archive, and the *webservices.xml*/deployment descriptor references the WSDL file for each web service. This WSDL file can be created from scratch (if you're feeling ambitious), or you can use a tool to automatically generate a WSDL file from the Java classes that implement your web service (such as the `Java2WSDL` tool provided with Axis or the `wscompile` tool provided with Sun's Java Web Services Developer Pack (JWS DP)). The *webservices.xml*/file requires a WSDL file for each web service at deploy time because the Web Services for J2EE specification could not make the

assumption that every Java web service engine would be capable of autogenerating a WSDL descriptor at runtime and because in some cases you will want to specify your own WSDL file rather than relying on the default one mapped by a particular web service engine.

4. *Generate a JAX-RPC mapping file for each web service*

Each web service must also have a mapping file that specifies how various Java entities in the implementation are to be mapped into XML entities in the WSDL descriptor. This includes mapping Java packages to XML namespaces, Java methods to WSDL operations, Java objects to WSDL data types and messages, and so forth. This mapping file is also in XML format, using a schema specified in the Web Services for J2EE specification. Again, the specification requires a mapping file be provided at deployment time because it couldn't assume that all web service engines would be able to generate the mapping at runtime and because in some cases you will want to explicitly specify the mapping for your web services (e.g., when custom XML serializers/deserializers are in use).

As shown in [Figure 12-4](#), the *webservices.xml*/deployment descriptor augments the component deployment descriptor (*web.xml*/or *ejb-jar.xml*) in the component module. It enumerates the web services in the module and references the components (servlets or EJBs) that implement each service. *webservices.xml*/also references a Java/XML mapping file and a WSDL descriptor for each web service. These files are also included in the component module archive.

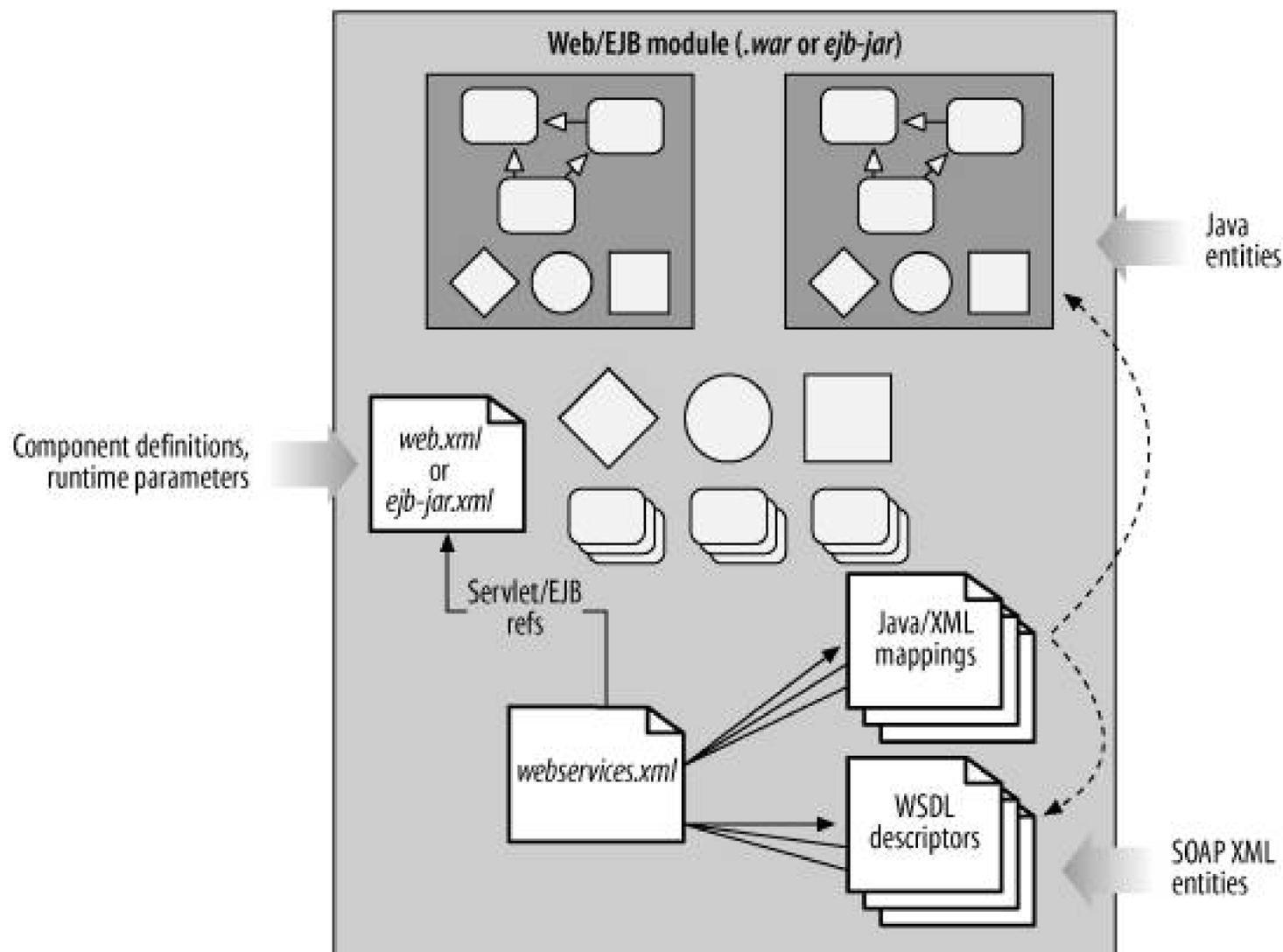
Now let's see how each type of JAX-RPC web service is deployed in detail. We first discuss web services that are deployed as simple Java classes within a web archive. In this section, we also cover many of the general details of J2EE web service deployment. We then look at deploying services implemented as EJB components.

12.6.1.1. Simple Java web services

Web services implemented using simple Java objects are deployed and managed within a web container. These service implementations are not, themselves, web components (they're just simple Java objects that follow the RMI programming model). But when you deploy them as web services, you create `<servlet>` entries in the *web.xml*/file for them, creating a virtual component for each web service. To deploy our Echo web service example, we would add the following entry to the *web.xml*/deployment descriptor for the web archive containing the web service:

```
<web-app . . . >
    . . .
    <!-- Echo SOAP service -->
    <servlet>
        <servlet-name>EchoWebService</servlet-name>
        <servlet-class>com.oreilly.jent.people.soap.Echo</servlet-class>
    </servlet>
    . . .
</web-app>
```

Figure 12-4. Deploying web services in a J2EE component archive



You do not need to provide a `<servlet-mapping>` entry for the virtual web component. These web service components will not have their URL entry points managed directly by the web container. Instead, the web service engine will be responsible for mapping endpoint URLs for services (published in their WSDL files) to the actual web service implementation.

Next, we need a `webservices.xml` deployment descriptor to include in the web archive. The `webservices.xml` file is placed in the `WEB-INF` directory of the archive, alongside the `web.xml` deployment descriptor. A complete web service deployment file for our `Echo` web service is shown in [Example 12-4](#).

Example 12-4. webservices.xml deployment descriptor for Echo service

```
<?xml version="1.0" encoding="UTF-8"?>
<webservices 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://www.ibm.com/webservices/xsd/j2ee_web_services_1_1.xsd"
  version="1.1">
  <webservice-description>
    <webservice-description-name>
```

```

    Echo Web Service
  </webservice-description-name>
  <wsdl-file>WEB-INF/wsdl/EchoWS.wsdl</wsdl-file>
  <jaxrpc-mapping-file>WEB-INF/Echo-mapping.xml</jaxrpc-mapping-file>
  <port-component>
    <port-component-name>Echo</port-component-name>
    <wsdl-port>EchoPort</wsdl-port>
    <service-endpoint-interface>
      com.oreilly.jent.people.soap.IEcho
    </service-endpoint-interface>
    <service-impl-bean>
      <servlet-link>EchoWebService</servlet-link>
    </service-impl-bean>
  </port-component>
</webservice-description>
</webservices>

```

The format of the deployment descriptor is specified by a standard XML Schema, referenced as a namespace in the root `<webservices>` element of the XML file. Each web service contained in the web archive needs to have a `<webservice-description>` entry in this file. In our case, we have only one web service, `Echo`, so we have a single description entry. The web service entry has a descriptive name, `<webservice-description-name>`, followed by several elements that specify the WSDL descriptor for the service, the Java/XML mapping file for the service, and information about each WSDL port in the web service. For each port in the web service, the `<webservice-description>` entry contains a `<port-component>` entry that specifies the name of the port in the WSDL file, the Java interface that the port implementation uses, and a reference to the web component in the `web.xml` descriptor that implements the port. In our case, we have a single port in the `Echo` web service, so there is a single `<port-component>` element.

In the war file, the WSDL descriptors for web services are stored in the `WEB-INF/wsdl` directory, and each `<webservice-description>` entry references the relevant WSDL file for its web service. In our case, the `Echo` service WSDL descriptor is expected to be in the `WEB-INF/wsdl/EchoWS.wsdl` file in the archive, as specified in the `<wsdl-file>` element in the `webservices.xml` file. As mentioned earlier, this WSDL file can be written by hand or (more likely) generated from the Java code that implements the web service using a tool provided by your web service engine.

Similarly, Java/XML mapping files are placed directly in the `WEB-INF` directory and are referenced in the `webservices.xml` file using the `<jaxrpc-mapping-file>` element. In our `Echo` web service, we've placed the mapping file in `WEB-INF/wsdl/Echo-mapping.xml` within the war file. The mapping file for our `Echo` service is shown in [Example 12-5](#).

Example 12-5. Minimal Java/XML mapping file for Echo service

```

<?xml version="1.0" encoding="UTF-8"?>
<java-wsdl-mapping
  version="1.1"
  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://www.ibm.com/webservices/xsd/j2ee_jaxrpc_mapping_1_1.xsd">
<package-mapping>
  <package-type>com.oreilly.jent.people.soap</package-type>
  <namespaceURI>http://soap.people.jent.oreilly.com/</namespaceURI>
</package-mapping>
</java-wsdl-mapping>

```

Each mapping file follows a standard XML Schema specified in the Web Services for J2EE standard. At a minimum, the mapping file is used to specify the conversion between your Java packages and XML namespaces in the WSDL descriptor. This is all we've done in the mapping file for the Echo service; using a `<package-mapping>` element to map any Java entities in the `com.oreilly.jent.people.soap` package should be mapped to the XML namespace <http://soap.people.jent.oreilly.com> in the WSDL.

A compliant web service engine should be able to automatically map everything else in the WSDL and your Java classes, assuming that:

- Only one service and one port are defined in the WSDL.
- The WSDL doesn't have any "overloaded" names (i.e., each operation on the port has a unique name; the port, service, binding, and data type names are all unique; etc.).
- All of the port's operations have one input and one output, and all of the messages used contain at most one part.
- All of the exceptions mapped to faults have no special attributes (beyond the typical string message), and the exceptions do not inherit from `RuntimeException` or `RemoteException`.
- All of the data types being mapped are within the standard JAX-RPC Java/XML mappings described earlier.

If your situation falls outside of this profile (e.g., you use Java or WSDL entities that are not covered by the JAX-RPC mappings, extra parameters on your application-specific exceptions need to be mapped, etc.), you can include explicit type mappings in your service's mapping file. These mappings can include Java types to WSDL data types, Java exceptions to SOAP fault types in the WSDL, and Java interfaces and methods to WSDL service ports and operations. If you do provide any mappings beyond the package mappings, however, you must provide a mapping for every entity in the WSDL descriptor for the web service.

Custom type mappings are specified using the following elements, all of which are peers of the `<package-type>` element in the mapping file:

```
<java-xml-type-mapping>
```

Each element specifies how to map a Java type to one or more XML types. The Java type itself is mapped to an XML data type in the WSDL, and any properties on the Java type are also mapped to XML elements within the main XML data type. In the case of our `PeopleFinder` web service, we could specify the mapping for the `Person` XML data type as follows:

```

<java-xml-type-mapping>
  <java-type>com.oreilly.jent.people.Person</java-type>
  <root-type-qname
    xmlns:typeNS="http://soap.people.jent.oreilly.com/types"
  >typeNS:Person</root-type-qname>
  <qname-scope>complexType</qname-scope>
  <variable-mapping>
    <java-variable-name>emailAddresses</java-variable-name>
    <xml-element-name>emailAddresses</xml-element-name>
  </variable-mapping>
  <variable-mapping>
    <java-variable-name>firstName</java-variable-name>
    <xml-element-name>firstName</xml-element-name>
  </variable-mapping>
  <variable-mapping>
    <java-variable-name>lastName</java-variable-name>
    <xml-element-name>lastName</xml-element-name>
  </variable-mapping>
</java-xml-type-mapping>

```

This entry maps the `Person` XML data type to the `com.oreilly.jent.people.Person` Java bean and then each subelement of the XML data type is mapped to properties on the Java bean, using `<variable-mapping>` elements.

```
<exception-mapping>
```

Each of these elements specifies how to map a Java exception to a fault type defined in the WSDL. Since faults in the WSDL also have a corresponding XML data type defined for them, also in the WSDL, each fault has two entries in the JAX-RPC mapping file. A `<java-xml-type-mapping>` element links the XML data type for the fault to the Java `Exception` class, and an `<exception-mapping>` element maps the WSDL fault element to the Java `Exception` class. In our `PeopleFinder` example, the `InvalidSearchException` fault could be mapped to the `com.oreilly.jent.people.InvalidSearchException` class using the following entry in the JAX-RPC mapping file:

```

<exception-mapping>
  <exception-type>
com.oreilly.jent.people.InvalidSearchException</exception-type>
  <wsdl-message xmlns:exMsgNS="http://soap.people.jent.oreilly.com/"
    >exMsgNS:InvalidSearchException</wsdl-message>
  <constructor-parameter-order>
    <element-name>message</element-name>
  </constructor-parameter-order>
</exception-mapping>

```

```
<service-interface-mapping>
```

This element maps a service defined in the WSDL to a Java interface that represents the service. Each port in the service can also be mapped using `<port-name>` elements within the `<service-interface-mapping>` element. The service definition in the PeopleFinder WSDL can be mapped to a `PeopleFinderWS` Java interface using the following mapping entry:

```
<service-interface-mapping>
  <service-interface>
    <com.oreilly.jent.people.soap.PeopleFinderWS</service-interface>
  <wsdl-service-name xmlns:serviceNS=
<"http://soap.people.jent.oreilly.com/">
    <serviceNS:PeopleFinderWS</wsdl-service-name>
  <port-mapping>
    <port-name>PeopleFinderPort</port-name>
    <java-port-name>PeopleFinderPort</java-port-name>
  </port-mapping>
</service-interface-mapping>
```

```
<service-endpoint-interface-mapping>
```

This element specifies how a particular port binding in the service's WSDL is to be mapped to corresponding Java entities. The port associated with the binding element in the WSDL descriptor is mapped to a Java interface, and each operation within the port is mapped to methods on the Java interface. The following mapping entry maps the binding for the `PeopleFinder` port to the `com.oreilly.jent.people.soap.PeopleFinder` interface, and all of the operations in the port are mapped to the methods on the `PeopleFinder` interface:

```
<service-endpoint-interface-mapping>
  <service-endpoint-interface>
    com.oreilly.jent.people.soap.PeopleFinder
  </service-endpoint-interface>
  <wsdl-port-type
    xmlns:portTypeNS="http://soap.people.jent.oreilly.com/">
    portTypeNS:PeopleFinder</wsdl-port-type>
  <wsdl-binding
    xmlns:bindingNS="http://soap.people.jent.oreilly.com/">
    bindingNS:PeopleFinderBinding</wsdl-binding>
  <service-endpoint-method-mapping>
    <java-method-name>findPeople</java-method-name>
    <wsdl-operation>findPeople</wsdl-operation>
    <method-param-parts-mapping>
      <param-position>0</param-position>
      <param-type>com.oreilly.jent.people.SearchArg[]</param-type>
    <wsdl-message-mapping>
      <wsdl-message
        xmlns:wSDLMsgNS="http://soap.people.jent.oreilly.com/">
        wSDLMsgNS:PeopleFinder_findPeople</wsdl-message>
      <wsdl-message-part-name>
```

```

        arrayOfSearchArg_1</wsdl-message-part-name>
        <parameter-mode>IN</parameter-mode>
    </wsdl-message-mapping>
</method-param-parts-mapping>
<wsdl-return-value-mapping>
    <method-return-value>com.oreilly.jent.people.Person[ ]
</method-return-value>
<wsdl-message
    xmlns:wsdlMsgNS="http://soap.people.jent.oreilly.com/">
    wsdlMsgNS:PeopleFinder_findPeopleResponse</wsdl-message>
    <wsdl-message-part-name>result</wsdl-message-part-name>
</wsdl-return-value-mapping>
</service-endpoint-method-mapping>
</service-endpoint-interface-mapping>

```

Once the WSDL for the web service(s), the *webservices.xml*/deployment descriptor, and the JAX-RPC mapping file have been created, the services are deployed within a web archive, by adding these files to the web archive in the appropriate places. The *webservices.xml*/file is placed in the *WEB-INF* directory of the archive, with the *web.xml*/file. The JAX-RPC mapping file is also placed in the *WEB-INF* directory. The WSDL for the service is typically placed in the *WEB-INF/wsdl* directory, although you can technically place your WSDL files anywhere in the archive.

12.6.1.2. EJB web services

Before a stateless session EJB can serve as a web service, its EJB deploymentdescriptor entry needs to have a `<service-endpoint>` entry added to it, to indicate the Java interface that its web service will expose. Here is the modified deployment descriptor for our PeopleFinder EJB component:

```

<?xml version="1.0" encoding="UTF-8"?>
<ejb-jar 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/ejb-jar_2_1.xsd"
    version="2.1">
    <enterprise-beans>
        <!-- Declare our PeopleFinder session bean -->
        <session>
            <ejb-name>PeopleFinder-EJB</ejb-name>
            <home>com.oreilly.jent.people.ejb.PeopleFinderHome</home>
            <remote>com.oreilly.jent.people.ejb.PeopleFinder</remote>
            <local-home>
                com.oreilly.jent.people.ejb.PeopleFinderLocalHome
            </local-home>
            <local>com.oreilly.jent.people.ejb.PeopleFinderLocal</local>

            <service-endpoint>
                com.oreilly.jent.people.ejb.PeopleFinder</service-endpoint>
            <ejb-class>com.oreilly.jent.people.ejb.PeopleFinderBean

```

```

        </ejb-class>
        <session-type>Stateless</session-type>
        <transaction-type>Container</transaction-type>
    </session>
</enterprise-beans>
</ejb-jar>

```

In this example, we've kept the client and home interfaces for our EJB, but if you are using the EJB only as a web service, they can be eliminated from the deployment entry for the bean.

Web services implemented as EJB components are deployed in much the same way as services implemented as simple Java objects. The web services are tied to their EJB components in a *webservices.xml* file, each service has a JAX-RPC mapping file and a WSDL file, and all of these files are deployed within the EJB archive. The *webservices.xml* file and the JAX-RPC mapping files are stored in the *META-INF* directory of the archive, while the WSDL descriptors for the services are typically stored in the *META-INF/wsdl* directory. Note that, like services packaged in web archives, the WSDL files aren't strictly required (according to the specifications) to be contained in the *META-INF/wsdl* directory in the EJB archive, but this is a common location in which to put them.

The key difference is in the *webservices.xml* file. In the case of a service implemented as a session EJB, the `<webservice-description>` entry will point to the appropriate EJB component in the EJB archive rather than to the virtual web component used for simple Java class implementations. [Example 12-6](#) shows the *webservices.xml* deployment file for the session EJB implementation of our PeopleFinder web service.

Example 12-6. Deployment descriptor for EJB-based web service

```

<?xml version="1.0" encoding="UTF-8"?>
<webservices 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://www.ibm.com/webservices/xsd/j2ee_web_services_1_1.xsd"
  version="1.1">
  <webservice-description>
    <webservice-description-name>
      PeopleFinder Web Service
    </webservice-description-name>

    <wsdl-file>META-INF/wsdl/PeopleFinderWS.wsdl</wsdl-file>
    <jaxrpc-mapping-file>
      META-INF/PeopleFinder-mapping.xml</jaxrpc-mapping-file>
    <port-component>
      <port-component-name>PeopleFinder</port-component-name>
      <wsdl-port>PeopleFinderPort</wsdl-port>

      <service-endpoint-interface>
        com.oreilly.jent.people.ejb.PeopleFinder
      </service-endpoint-interface>

```

```

    <service-impl-bean>
      <ejb-link>PeopleFinder-EJB</ejb-link>
    </service-impl-bean>
  </port-component>
</webservice-description>
</webservices>

```

In this case, we've linked the web service to the EJB named `PeopleFinder-EJB` in the archive, using the `<service-impl-bean>` element of the `<port-component>` section. We've also set the `<service-endpoint-interface>` value to be the remote client interface of the EJB.

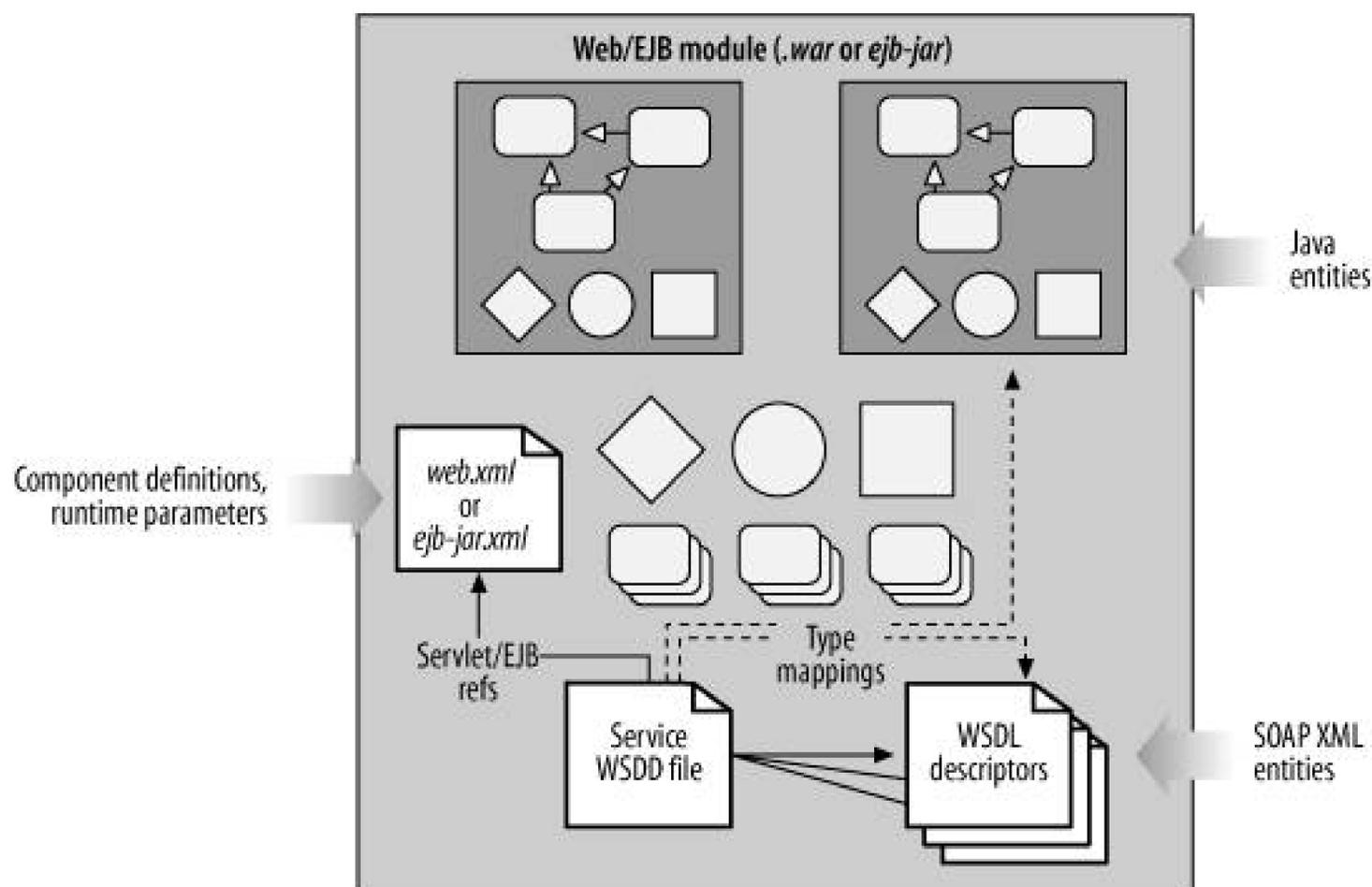
The JAX-RPC mapping file for an EJB-based web service is essentially the same as those for simple Java web services. The only difference is that the mapping of the service endpoint (in the `<service-endpoint-interface-mapping>` element) refers to the Java service endpoint interface exposed by the EJB component.

12.6.2. Axis Deployment Model

Prior to the availability of the Web Services for J2EE specification, Axis and other Java-based web service engine implementations needed to provide developers with a (nonstandard) deployment scheme that could be used to deploy web services to their environments. Here, we'll look at the Axis deployment scheme. We do this for a few reasons. The most important of them is the fact that Axis is a very popular engine for building and using web services in Java/J2EE environments. But in addition the Axis deployment scheme is a good example of a nonstandard approach to deploying web services, and it compares and contrasts nicely with the newer, standardized approach provided by JSR 109.

[Figure 12-5](#) depicts the model used by Axis for deploying web services. If you compare this to the J2EE standard model shown in [Figure 12-4](#), you'll see a lot of similarities. Instead of a `webservices.xml` deployment descriptor and JAX-RPC mapping files, Axis uses its own Web Services Deployment Descriptor (WSDD) format and a separate namespace/package mapping file. Like the `webservices.xml` file, the WSDD file for a web service specifies the name of the service and the Java component (simple Java class, EJB component, etc.) that implements it. The WSDD file also includes the Java/XML entity mapping details.

Figure 12-5. Axis deployment model



[Example 12-7](#) shows a WSDD file that can be used to deploy the simple Java version of our PeopleFinder web service to an Axis web service engine.

Example 12-7. Axis WSDD deployment descriptor for PeopleFinder service

```
<deployment xmlns="http://xml.apache.org/axis/wsdd/"
  xmlns:java="http://xml.apache.org/axis/wsdd/providers/java">
  <service name="peopleFinder" provider="java:RPC">
    <!-- Specify the implementing class for the service -->
    <parameter name="className"
      value="com.oreilly.jent.people.soap.PeopleFinderImpl"/>
    <!-- Specify the methods exposed to SOAP clients -->
    <parameter name="allowedMethods" value="*" />
    <!-- Specify that Person should be mapped using the Axis JavaBean
      mapping -->
    <beanMapping qname="ns:Person" xmlns:ns="people.jent.oreilly.com"
      languageSpecificType="java:com.oreilly.jent.people.Person"/>
    <!-- SearchArg objects should also be mapped using the JavaBean
      mapping -->
    <beanMapping qname="ns:SearchArg" xmlns:ns="people.jent.oreilly.com"
      languageSpecificType="java:com.oreilly.jent.people.SearchArg"/>
  </service>
</deployment>
```

The `<service>` element specifies the details for a particular web service (similar to a `<webservice-description>` entry in a *webservices.xml*/file). In our case, we specify that our web service is implemented as a simple Java class by setting the `provider` attribute on the `<service>` element to `java:RPC`. The child elements of the service element tell Axis how to deploy our web service. Among other things, this information contains a set of standard parameters for the web service and a set of optional Java/XML type mappings.

In our example, we first tell Axis the implementation class for the service, using the `className` parameter and setting its value to the fully qualified Java class name of the service implementation, `com.oreilly.jent.people.soap.PeopleFinderImpl`. Next, we tell Axis which methods on the implementation class should actually be exposed as operations in the web service. In our case, all of the `PeopleFinderImpl` methods are part of the web service, so we set the `allowedMethods` parameter to a wildcard, `*`.

After these parameters, we specify a few Java/XML type mappings for our service. Axis is able to automatically map basic Java data types (`int`, `long`, `float`, etc.) and the core Java data classes (`Integer`, `String`, `Float`, etc.) when they appear in methods on the service implementation class. But in cases in which the mapping is for a JavaBeans class or when a custom mapping is needed, you need to specify the mapping details in the WSDO using either a `<beanMapping>` or `<typeMapping>` element. In our case, the only Java types exposed in the `PeopleFinderImpl` interface are `Person` and `SearchArg`, and they both follow the JavaBeans pattern (in terms of property accessors, default constructor, etc.). So in both cases, we tell Axis to use its built-in JavaBeans mapper by using the `<beanMapping>` element in the WSDO file. In each case, we tell Axis the XML type and the JavaBeans class for the mapping.

If we had Java types or XML types that required custom mapping logic (e.g., they were not one of the standard supported data types that can be automatically mapped or we want to bypass the standard mapping), then you can specify a custom mapping in the WSDO file using a `<typeMapping>` element. The `<typeMapping>` element specifies the XML data type to be mapped (using its qualified name), the Java class it will be mapped to, and the class names for a serializer (to convert the Java type to XML) and a deserializer (to convert the XML into the Java type). The serializer and deserializer classes implement the `javax.xml.rpc.encoding.SerializerFactory` and `DeserializerFactory` interfaces, respectively, from JAX-RPC. Again, writing custom serializers/deserializers for Java types is outside the scope of this chapter. If you find that you need to support a nonstandard mapping for some reason, please consult the Axis documentation and code samples.

One deployment detail that isn't included in the Axis WSDO file is the mapping of XML namespaces to Java packages. This is handled in a separate namespace mapping file, which we discussed earlier when we discussed writing web service clients with Axis. These namespace/package mapping files are used by the *WSDL2Java* tool in Axis, to generate client-side stubs and/or starter code for the service implementation. A mapping file can also be used with Axis's *Java2WSDL* tool to create a WSDL file for a given Java service implementation class.

Chapter 13. Remote Method Invocation

This chapter examines the Java Remote Method Invocation (RMI) API Java's native scheme for creating and using remote objects. Java RMI provides the following elements:

- Support (API and runtime) for remote object implementations
- Support for client interfaces, or stubs, to remote objects
- A remote object registry for finding objects on the network
- A facility for automatically creating (activating) remote objects on demand
- A network protocol for communication between remote objects and their clients

Each of these elements has related Java interfaces within the `java.rmi` package and its subpackages and runtime support for the feature as part of the JVM services. Using these interfaces and runtime services, you can develop remote objects and the clients that use them as part of a distributed application that resides on hosts across the network.

[← PREV](#)

13.1. What's Covered Here?

This chapter covers the RMI API and runtime included in the JDK 1.5 (otherwise known as J2SE 5.0). In key areas, we'll highlight features or behavior that varies in earlier versions of the JDK. The most notable RMI feature introduced in JDK 1.5 is the dynamic generation of client stubs for RMI remote objects. This makes the use of the RMI compiler optional since stub classes can now be generated on demand at runtime by the JVM. If your distributed system includes clients running in 1.4 or older JVMs, however, you'll still need to generate stub classes with the RMI compiler so that these clients can interact with your remotely exported objects.

[← PREV](#)

13.2. Introduction to RMI

RMI is the distributed object system that is built into the core Java environment. You can think of RMI as a built-in facility for Java that allows you to interact with objects that are actually running in Java virtual machines on remote hosts on the network. With RMI (and other distributed object APIs we discuss in other chapters of this book), you can get a reference to an object that "lives" in a remote process and invoke methods on it as if it were a local object running within the same virtual machine as your code (hence the name "remote method invocation").

Another way to characterize RMI (and other remote object schemes) is in terms of the granularity of the distribution that it enables. The Java servlet and JSP APIs, described in [Chapters 3](#) and [4](#), allow you to distribute applications at the user interface level. Putting a servlet or JSP frontend on your server-side object model serves to export a web-based (typically HTML) interface to your application, which any remote web browser can access. The Java networking APIs, embodied in the [java.net](#) and [java.io](#) packages, allow you to open up very narrow, low-level data connections to your Java process, for simple data exchanges or "heartbeat" purposes (make a successful connection and transmit a few bytes to confirm that a process is alive). RMI and other remote object systems fall somewhere in between the two. They allow you to export functionality at the object/component level allowing remote clients to interact directly with individual objects, in the same way they do with local objects: using (remote) method calls.

RMI was added to the core Java API in Version 1.1 of the JDK, in recognition of the critical need for support for distributed objects in distributed application development. Prior to RMI and other remote object schemes, writing a distributed application involved basic socket programming, in which a "raw" communication channel was used to pass messages and data between two remote processes. Now, with RMI and distributed objects, you can "export" an object as a remote object, so that other remote processes/agents can access it directly as a Java object. So, instead of defining a low-level message protocol and data transmission format between processes in your distributed application, you can use Java interfaces as the "protocol" and the exported method arguments become the data transmission format. The distributed object system (RMI in this case) handles all the underlying networking needed to make your remote method calls work.

Java RMI is normally a Java-only distributed object scheme; if you are using the default RMI communication protocol (Java Remote Method Protocol, or JRMP), objects in an RMI-based distributed application have to be implemented in Java. Some other distributed object schemes, most notably CORBA, are language-independent, which means that the objects can be implemented in any language that has a defined binding. With CORBA, for example, bindings exist for C, C++, Java, Smalltalk, and Ada, among other languages.

The advantages of RMI primarily revolve around the fact that it is "Java-native." Since RMI is part of the core Java API and is built to work directly with Java objects within the Java VM, the integration of its remote object facilities into a Java application is almost seamless. And since Java RMI is built on the assumption that both the client and server are Java objects, RMI can extend Java's internal garbage-collection mechanisms to provide distributed garbage collection of remotely exported objects.

If you have a distributed application with heterogeneous platform components, some of which are written in Java and some of which aren't, you have a few choices. You can use RMI, wrapping the non-Java code with RMI-enabled Java objects using the Java Native Interface (JNI). Near the end of this chapter, we discuss this first option in some detail to give you a feeling for when this technique could be useful and when it wouldn't be. Another option is to use another object distribution scheme, such as CORBA, that supports language-independent object interfaces. [Chapter 14](#) covers the Java interface to CORBA that is included in the JDK. A third option involves using RMI/IIOP, which allows RMI objects to communicate directly with remote CORBA objects (implemented in any language supported by CORBA) over IIOP (Internet Inter-ORB Protocol). We discuss this option in some detail at the end of this chapter. Finally, a more recent and more popular option for connecting cross-platform distributed systems is to use SOAP-based web services. Standard support for web services in Java is provided by the JAX-RPC and SAAJ APIs, both of which are discussed in [Chapter 12](#).

13.2.1. RMI in Action

Before we start examining the details of using RMI, let's look at a simple RMI remote object at work. We will create an `Account` object that represents some kind of bank account and then use RMI to export it as a remote object so that remote clients (e.g., ATMs, personal finance software running on a PC, etc.) can access it to carry out transactions.

The first step is to define the interface for our remote object. [Example 13-1](#) shows the `Account` interface. You can tell that it's an RMI object because it extends the `java.rmi.Remote` interface. Another signal that this is meant for remote access is that each method can throw a `java.rmi.RemoteException`. The `Account` interface includes methods to get the account name and balance and to make deposits, withdrawals, and transfers.

Example 13-1. A remote account interface

```
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface Account extends Remote {
    // Get the name on the account
    public String getName( ) throws RemoteException;
    // Get current balance
    public float getBalance( ) throws RemoteException;
    // Take some money away
    public void withdraw(float amt)
        throws RemoteException, InsufficientFundsException;
    // Put some money in
    public void deposit(float amt) throws RemoteException;
    // Move some money from one account into this one
    public void transfer(float amt, Account src)
        throws RemoteException, InsufficientFundsException;
}
```

The next step is to create an implementation of this interface, which leads to the `AccountImpl` class shown in [Example 13-2](#). This class implements all the methods listed in the `Account` interface and adds a constructor that takes the name of the new account to be created. Notice that the `AccountImpl` class implements the `Account` interface, but it also extends the `java.rmi.UnicastRemoteObject` class. The `UnicastRemoteObject` class provides the basic remote functionality for RMI server objects.

Example 13-2. Implementation of the remote account interface

```
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

public class AccountImpl extends UnicastRemoteObject implements Account {
    // Our current balance
    private float mBalance = 0;
    // Name on account
    private String mName = "";
    // Create a new account with the given name
    public AccountImpl(String name) throws RemoteException {
        mName = name;
    }

    public String getName( ) throws RemoteException {
        return mName;
    }

    public float getBalance( ) throws RemoteException {
        return mBalance;
    }
    // Withdraw some funds
    public void withdraw(float amt)
        throws RemoteException, InsufficientFundsException {
        if (mBalance >= amt) {
            mBalance -= amt;
            // Log transaction...
            System.out.println("--> Withdrew " + amt +
                " from account " + getName( ));
            System.out.println("    New balance: " + getBalance( ));
        }
        else {
            throw new InsufficientFundsException("Withdrawal request of " +
                amt + " exceeds balance of "
                + mBalance);
        }
    }

    // Deposit some funds
    public void deposit(float amt) throws RemoteException {
        mBalance += amt;
    }
}
```

```

        // Log transaction...
        System.out.println("--> Deposited " + amt +
            " into account " + getName( ));
        System.out.println("    New balance: " + getBalance( ));
    }
    // Move some funds from another (remote) account into this one
    public void transfer(float amt, Account src)
        throws RemoteException, InsufficientFundsException {
        if (checkTransfer(src, amt)) {
            src.withdraw(amt);
            this.deposit(amt);
            // Log transaction...
            System.out.println("--> Transferred " + amt +
                " from account " + getName( ));
            System.out.println("    New balance: " + getBalance( ));
        }
        else {
            throw new InsufficientFundsException("Source account balance " +
                "is insufficient.");
        }
    }
    // Check to see if the transfer is possible, given the source account
    private boolean checkTransfer(Account src, float amt) {
        boolean approved = false;
        try {
            if (src.getBalance( ) >= amt) {
                approved = true;
            }
        }
        catch (RemoteException re) {
            // If some remote exception occurred, then the transfer is still
            // compromised, so return false
            approved = false;
        }
        return approved;
    }
}

```

Once the remote interface and an implementation of it are complete, you compile both Java files with your favorite Java compiler. If you are using JDK 1.5 or later for both the server and the client (and compiled the classes using a 1.5-compliant compiler), you can use them both at this point the stubs and skeletons needed to connect the client to the server will be dynamically generated at runtime. But if you're using an earlier JDK to run either the client or the server, you'll need to use the RMI stub/skeleton compiler to generate a client stub and a server skeleton for the `AccountImpl` object. The stub and skeleton handle the communication between the client application and the server object. The RMI compiler is called `rmic`, and you can invoke it for this example like so:

```
% rmic -d /home/classes AccountImpl
```

The stub and skeleton classes are generated and stored in the directory given by the `-d` option (`/home/classes`, in this case). This example assumes that the `AccountImpl` class and the `/home/classes` directory are already in your `CLASSPATH` before you run the RMI compiler.

There's just one more thing we need to do before we can actually use our remote object: register it with an RMI registry, so that remote clients can find it on the network. The utility class that follows, `AccountServer`, does this by creating an `AccountImpl` object and then binding it to a name in the local registry using the `java.rmi.Naming` interface:

```
import java.net.MalformedURLException;
import java.rmi.Naming;
import java.rmi.RMISecurityManager;
import java.rmi.RemoteException;

public class AccountServer {
    public static void main(String argv[]) {
        try {
            System.setSecurityManager(new RMISecurityManager( ));
            // Make an Account with a given name
            AccountImpl acct = new AccountImpl("JimF");

            // Register it with the local naming registry
            Naming.rebind("JimF", acct);
            System.out.println("Registered account as JimF.");

            // Keep the server process alive indefinitely
            Object dummy = new Object( );
            synchronized (dummy) {
                try {
                    dummy.wait( )
;
                }
                catch (InterruptedException ie) {}
            }
        }
        catch (RemoteException re) {
            System.err.println("Remote exception while " +
: "
                + re.getMessage( ));
        }
        catch (MalformedURLException mue) {
            System.err.println("Bad name given when binding server object: "
                + mue.getMessage( ));
        }
    }
}
```

After the `AccountServer.main()` method is done registering its `Account` object, it keeps the server process alive by going into an infinite `wait()` loop, waiting on a local object that is the target of our own synchronized block. In a real-world situation, you would probably have more elegant ways to keep the server alive and kill it off. Or perhaps not, since this approach is simple, but effective.

After you compile the `AccountServer` class, you can run its `main()` method to register an `Account` with the local RMI registry. First, however, you need to start the registry. With Sun's JDK, the registry can be started using the `rmiregistry` utility. On a Unix machine, this can be done like so:

```
objhost% rmiregistry &
```

Once the registry is started, you can invoke the `main()` method on the `AccountServer` class simply by running it:

```
objhost% java AccountServer
Registered account.
```

Now we have a remote `Account` object that is ready and waiting for a client to access it and call its methods. The following client code does just this, by first looking up the remote `Account` object using the `java.rmi.Naming` interface (and assuming that the `Account` object was registered on a machine named objhost.org), and then calling the `deposit` method on the `Account` object:

```
import java.rmi.Naming;

public class AccountClient {
    public static void main(String argv[]) {
        try {
            // Look up account object
            Account jimAcct = (Account)Naming.lookup("rmi://objhost.org/JimF");

            // Make deposit
            jimAcct.deposit(12000);

            // Report results and balance.
            System.out.println("Deposited 12,000 into account owned by " +
                jimAcct.getName( ));
            System.out.println("Balance now totals: " + jimAcct.getBalance( ));
        }
        catch (Exception e) {
            System.out.println("Error while looking up account:");
            e.printStackTrace( );
        }
    }
}
```

The first time you run this client, here's what you'd do:

```
% java AccountClient
Deposited 12,000 into account owned by Jim.Farley
Balance now totals: 12000.0
```

For the sake of this example, we have made two big assumptions about the environment used to run the code. First, we assume that the client process and the RMI registry have all the necessary classes available on their runtime `CLASSPATH` (i.e., the `Account` interface and, if needed, the stub classes generated from the `AccountImpl` implementation). The second assumption is that the JVMs being used for the RMI registry, the server process, and the client all have their security permissions set to allow the various network connections needed for the server to register the `Account` object and for the client to get a reference to the `Account` from the registry. Later in the chapter, we'll see how to deal with loading classes remotely and dynamically, when the client doesn't necessarily have them locally. We'll also discuss the security permissions needed to run RMI servers, clients, and registries. In the meantime, just to see the code running, make sure that all of the example classes are in the `CLASSPATH` of the client, the server process, and the RMI registry. Also, to avoid any potential security permission issues, you can make a special security policy file with the following contents:

```
grant {
    permission java.security.AllPermission;
};
```

Then run your client and server JVMs using this security policy:

```
> java -Djava.security.policy=allpermissions.policy . . .
```

This policy file will effectively disable all permission checks in the JVM, so it is definitely not recommended for use in production environments. But it's very handy for experimentation and debugging purposes.

13.2.2. RMI Architecture

Now that we've seen a complete example of an RMI object in action, let's look at what makes remote objects work, starting with an overview of the underlying RMI architecture. There are three layers that comprise the basic remote object communication facilities in RMI:

Stub/skeleton layer

Provides the interface that client and server application objects use to interact with each other.

Remote reference layer

The middleware between the stub/skeleton layer and the underlying transport protocol. This layer handles the creation and management of remote object references.

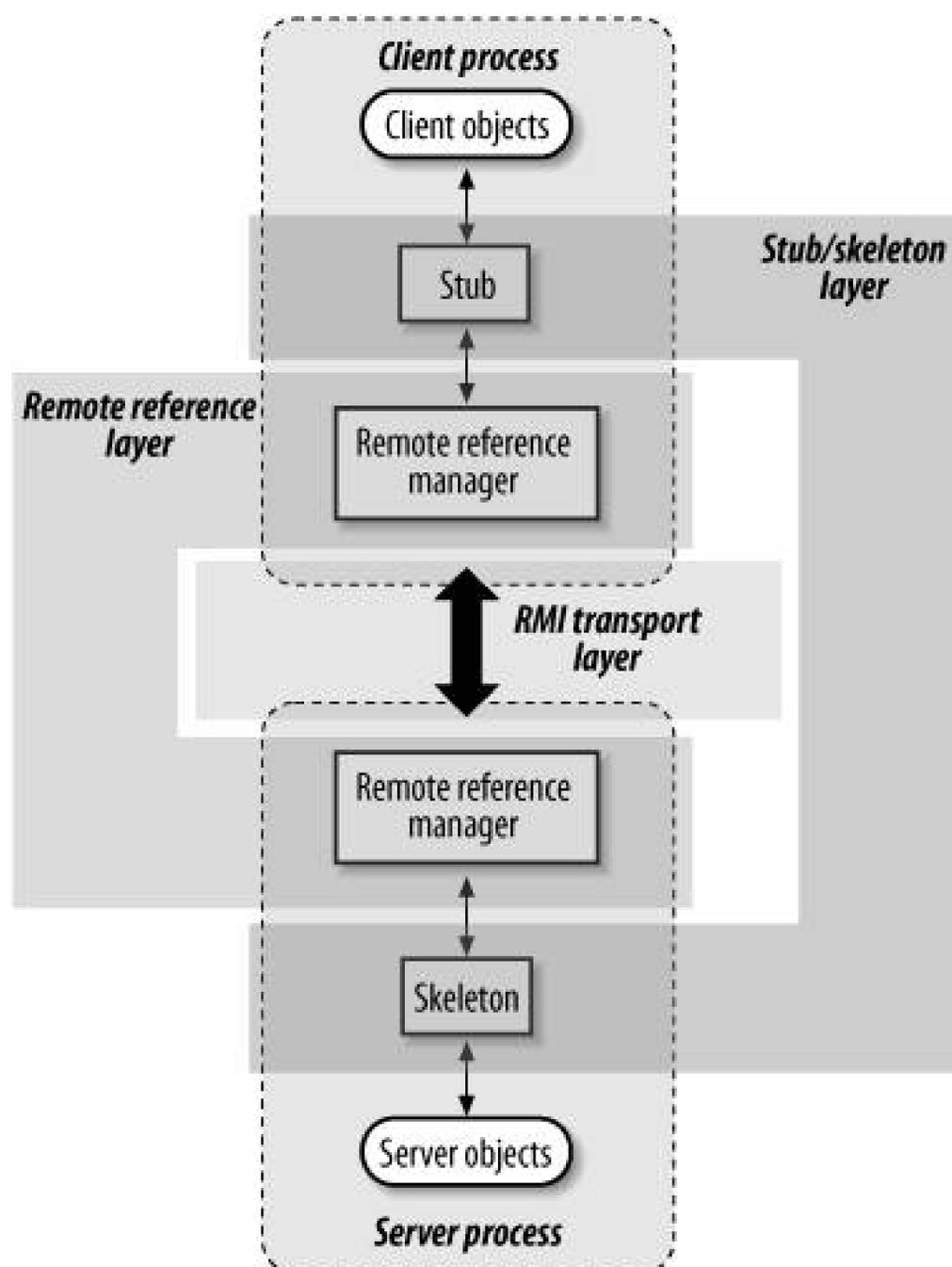
Transport protocol layer

The binary data protocol that sends remote object requests over the wire.

These layers interact with one another as shown in [Figure 13-1](#). In this figure, the server is the application that provides remotely accessible objects, while the client is any remote application that communicates with these server objects.

In a distributed object system, the distinctions between clients and servers can get pretty blurry at times. Consider the case in which one process registers a remote-enabled object with the RMI naming service, and a number of remote processes are accessing it. We might be tempted to call the first process the server and the other processes the clients. But what if one of the clients calls a method on the remote object, passing a reference to an RMI object that's local to the client. Now the server has a reference to and is using an object exported from the client, which turns the tables somewhat. The "server" is really the server for one object and the client of another object, and the "client" is a client and a server, too. For the sake of discussion, we refer to a process in a distributed application as a server or client if its role in the overall system is generally limited to one or the other. In peer-to-peer systems, in which there is no clear client or server, we refer to elements of the system in terms of application-specific roles (e.g., chat participant, chat facilitator).

Figure 13-1. The RMI runtime architecture



As you can see in [Figure 13-1](#), a client makes a request of a remote object using a client-side stub; the server object receives this request from a server-side object skeleton. A client initiates a remote method invocation by calling a method on a stub object. The stub maintains an internal reference to the remote object it represents and forwards the method invocation request through the remote reference layer by *marshaling* the method arguments into serialized form and asking the remote reference layer to forward the method request and arguments to the appropriate remote object. Marshaling involves converting local objects into portable form so that they can be transmitted to a remote process. Each object is checked as it is marshaled, to determine whether it implements the `java.rmi.Remote` interface. If it does, its remote reference is used as its marshaled data. If it isn't a `Remote` object, the argument is serialized into bytes that are sent to the remote host and reconstituted into a copy of the local object. If the argument is neither `Remote` nor `Serializable`, the stub throws a `java.rmi.MarshalException` back to the client.

If the marshaling of method arguments succeeds, the client-side remote reference layer receives the remote reference and marshaled arguments from the stub. This layer converts the client request into low-level RMI transport requests according to the type of remote object communication being used. In RMI, remote objects can (potentially) run under several different communication styles, such as point-to-point object references, replicated objects, or multicast objects. The remote reference layer is responsible for knowing which communication style is in effect for a given remote object and

generating the corresponding transport-level requests. In the version of RMI bundled with Sun's JVM implementations, the only communication style provided out of the box is point-to-point object references, so this is the only style we'll discuss in this chapter. For point-to-point communication, the remote reference layer constructs a single network-level request and sends it over the wire to the sole remote object that corresponds to the remote reference passed along with the request.

On the server, the server-side remote reference layer receives the transport-level request and converts it into a request for the server skeleton that matches the referenced object. The skeleton converts the remote request into the appropriate method call on the actual server object, which involves *unmarshaling* the method arguments into the server environment and passing them to the server object. As you might expect, unmarshaling is the inverse procedure to the marshaling process on the client. Arguments sent as remote references are converted into local stubs on the server, and arguments sent as serialized objects are converted into local copies of the originals.

If the method call generates a return value or an exception, the skeleton marshals the object for transport back to the client and forwards it through the server reference layer. This result is sent back using the appropriate transport protocol, where it passes through the client reference layer and stub, is unmarshaled by the stub, and is finally handed back to the client thread that invoked the remote method.

13.2.3. RMI Object Services

On top of its remote object architecture, RMI provides some basic object services you can use in your distributed application. These include an object naming/registry service, a remote object activation service, and distributed garbage collection.

13.2.3.1. Naming/registry service

When a server process wants to export some RMI-based service to clients, it does so by registering one or more RMI-enabled objects with its local RMI registry (represented by the `Registry` interface). Each object is registered with a name clients can use to reference it. A client can obtain a stub reference to the remote object by asking for the object by name through the `Naming` interface. The `Naming.lookup()` method takes the fully qualified name of a remote object and locates the object on the network. The object's fully qualified name is in a URL-like syntax that includes the name of the object's host and the object's registered name.

It's important to note that, although the `Naming` interface is a default naming service provided with RMI, the RMI registry can be tied into other naming services by vendors. Sun has provided a binding to the RMI registry through JNDI, for example. See [Chapter 9](#) for more details on how JNDI can be used to look up objects (remote or otherwise).

Once the `lookup()` method locates the object's host, it consults the RMI registry on that host and asks for the object by name. If the registry finds the object, it generates a remote reference to the object and delivers it to the client process, where it is converted into a stub reference that is returned to the caller. Once the client has a remote reference to the server object, communication between the client and the server commences as described earlier. We'll talk in more detail about the `Naming` and `Registry` interfaces later in this chapter.

13.2.3.2. Object activation service

The remote object activation service was introduced to RMI in JDK 1.2. It provides a way for server objects to be started on an as-needed basis. Without remote activation, a server object has to be registered with the RMI registry service from within a running Java virtual machine. A remote object registered this way is available only during the lifetime of the Java VM that registered it. If the server VM halts or crashes for some reason, the server object becomes unavailable and any existing client references to the object become invalid. Any further attempts by clients to call methods through these now-invalid references result in RMI exceptions being thrown back to the client.

The RMI activation service provides a way for a server object to be activated automatically when a client requests it. The activation service creates the server object dynamically within a new or existing virtual machine and obtains a reference to this newly created object for the client that caused the activation. A server object that wants to be activated automatically needs to register an activation method with the RMI activation daemon running on its host. We'll discuss the RMI activation service in more detail later in the chapter.

13.2.3.3. Distributed garbage collection

The last of the remote object services, distributed garbage collection, is a fairly automatic process that you as an application developer should never have to worry about. Every server that contains RMI-exported objects automatically maintains a list of remote references to the objects it serves. Each client that requests and receives a reference to a remote object, either explicitly through the registry/naming service or implicitly as the result of a remote method call, is issued this remote object reference through the remote reference layer of the object's host process. The reference layer automatically keeps a record of this reference in the form of an expirable "lease" on the object. When the client is done with the reference and allows the remote stub to go out of scope or when the lease on the object expires, the reference layer on the host automatically deletes the record of the remote reference and notifies the client's reference layer that this remote reference has expired. The concept of expirable leases, as opposed to strict on/off references, is used to deal with situations in which a client-side failure or a network failure keeps the client from notifying the server that it is done with its reference to an object.

When an RMI object has no further remote references recorded in the remote reference layer, then its eligibility for garbage collection is based solely on its local references, like any other Java object. If there are also no local references to the object (this reference list is kept by the Java VM itself as part of its normal garbage-collection algorithm), the object is marked as garbage and picked up by the next run of the system garbage collector.

13.3. Defining Remote Objects

Now that you have a basic idea of how Java RMI works, we can explore the details of creating and using distributed objects with RMI in more detail. As mentioned earlier, defining a remote RMI object involves specifying a remote interface for the object, then providing a class that implements this interface. The remote interface and implementation class are then used by RMI to generate a client stub and server skeleton for your remote object. The communication between local objects and remote objects is handled using these client stubs and server skeletons. The relationships among stubs, skeletons, and the objects that use them are shown in [Figure 13-2](#).

Figure 13-2. Relationships among remote object, stub, and skeleton classes

When a client gets a reference to a remote object (details on how this reference is obtained come later) and then calls methods on this object reference, there needs to be a way for the method request to get transmitted back to the actual object on the remote server and for the results of the method call to get transmitted back to the client. This is what the generated stub and skeleton

classes are for. They act as the communication link between the client and your exported remote object, making it seem to the client that the object actually exists within its Java VM.

Stubs and skeletons are generated from the classes that implement the remote RMI interfaces that you've exported. The process of generating these stub and skeleton classes is done either dynamically at runtime (in JDK 1.5 or later JVMs) or by using the RMI compiler. Whether you use the RMI compiler or use dynamic stub and skeleton generation, the role of these classes in the RMI runtime are the same. The stub and skeleton classes act as go-betweens for the client application and the actual server object, respectively. The client stub class implements the remote interface that you defined, and its implementations of each remote method package up (marshal) the method arguments provided by the client and transmit them to the server. The skeleton class is responsible for receiving the method arguments from the RMI runtime after they've been transmitted over the network. It unpackages (unmarshals) the method arguments and makes the corresponding method call on the actual remote object implementation on the server. Whatever response the method call generates (return data or an exception), the results are packaged and transmitted back to the remote client by the skeleton and the RMI runtime working together. The client stub method (which is still executing at this point) unpackages the results and delivers them to the client as the result of its remote method call.

As we saw earlier in the chapter, the first step in creating your remote objects is to define the remote interfaces for the types of objects you need to use in a distributed object context. This isn't much different from defining the public interfaces in a nondistributed application, with the following exceptions:

- Every object you want to distribute using RMI has to directly or indirectly extend an interface that extends the `java.rmi.Remote` interface.
- Every method in the remote interface has to declare that it throws a `java.rmi.RemoteException` or one of the parent classes of `RemoteException`.^[*]

^[*] Note that prior to Java 1.2, the RMI specification required that every method on a remote interface had to throw `RemoteException` specifically. In Java 1.2 and later, this restriction was loosened to allow any superclass of `RemoteException`. One reason for this change is to make it easier to define generic interfaces that support both local and remote objects.

RMI imposes the first requirement to allow it to differentiate quickly between objects that are enabled for remote distribution and those that aren't. As we've already seen, during a remote method invocation, the RMI runtime system needs to be able to determine whether each argument to the remote method is a `Remote` object or not. The `Remote` interface, which is simply a tag interface that marks remote objects, makes it easy to perform this check.

The second requirement is needed to deal with errors that can happen during a remote session. When a client makes a method call on a remote object, any number of errors can occur, preventing the remote method call from completing. These include client-side errors (e.g., an argument can't be marshaled), errors during the transport of data between client and server (e.g., the network connection is dropped), and errors on the server side (e.g., the method throws a local exception that needs to be sent back to the remote caller). The `RemoteException` class is used by RMI as a base exception class for any of the different types of problems that might occur during a remote method call. Any method you declare in a `Remote` interface is assumed to be remotely callable, so every method has to declare that it might throw a `RemoteException` or one of its parent interfaces.

In [Example 13-1](#) we saw the `Account` interface, a remote interface that declares six remote methods:

`getName()`, `getBalance()`, `withdraw()`, `deposit()` and `TRansfer()`. Since we want to use this interface in an RMI setting, we've declared that the interface extends the `Remote` interface. In addition, each method has arguments and return values that are either `Remote` or `Serializable`, and each method is declared as throwing a `RemoteException`.

With the remote interface defined, the next thing we need to do is write a class that implements the interface. We saw the implementation of the `Account` interface, the `AccountImpl` class, in [Example 13-2](#). This class has implementations of the six remote methods defined in the `Account` interface; it also has a nonremote method, `checkTransfer()`, to verify that a funds transfer between two accounts can be made. Notice that the `checkTransfer()` method doesn't have to be declared as throwing a `RemoteException`, since it isn't a remotely callable method. Only the methods that participate in the remote method protocol of the RMI runtime need to declare that they can throw `RemoteException` or one of its parent exceptions. The methods in your implementation class that participate in the RMI runtime include implementations of any methods declared in the remote interface (`withdraw()`, `transfer()`, etc. in our example) and, if you're extending `UnicastRemoteObject`, any constructors on your implementation class. Any other methods you define in your implementation class (like `checkTransfer()`) are considered nonremote (i.e., they are callable only from within the local Java virtual machine where the object exists).

Constructors are required to have `RemoteException` in their throws clause because an RMI implementation class that extends `UnicastRemoteObject` is "exported" and made accessible to remote method invocations when it is constructed. The `RemoteException` in the throws clause of the constructor is required to cover any problems that may occur during the RMI export process. All of the constructors on `UnicastRemoteObject` throw `RemoteException` because they all call the static `UnicastRemoteObject.exportObject()` methods internally, so the Java compiler will force you to include `RemoteException` in the throws clause of your implementation class.

13.3.1. Key RMI Classes for Remote Object Implementations

As you have probably noticed, our `AccountImpl` class also extends the `UnicastRemoteObject` class. This is a class in the `java.rmi.server` package that extends `java.rmi.server.RemoteServer`, which itself extends `java.rmi.server.RemoteObject`, the base class for all RMI remote objects. Four key classes are related to writing server object implementations:

`RemoteObject`

`RemoteObject` implements both the `Remote` and `Serializable` interfaces. Although the `RemoteObject` class is in the `java.rmi.server` package, it is used by both the client and server portions of a remote object reference. Both client stubs and server implementations are subclassed (directly or indirectly) from `RemoteObject`. A `RemoteObject` contains the remote reference for a particular remote object.

`RemoteObject` is an abstract class that reimplements the `equals()`, `hashCode()`, and `toString()` methods inherited from `Object` in a way that makes sense and is practical for remote objects. The `equals()` method, for example, is implemented to return `true` if the internal remote references of the two `RemoteObject` objects are equal (i.e., if they both point to the same server object).

RemoteServer

`RemoteServer` is an abstract class that extends `RemoteObject`. It defines a set of static methods that are useful for implementing server objects in RMI, and it acts as a base class for classes that define various semantics for remote objects. In principle, a remote object can behave according to a simple point-to-point reference scheme; it can have replicated copies of itself scattered across the network that need to be kept synchronized or any number of other scenarios. JDK 1.1 supported only point-to-point, nonpersistent remote references with the `UnicastRemoteObject` class. JDK 1.2 introduced the RMI activation system, so current versions of the core Java API include another subclass of `RemoteServer`: `Activatable`.

UnicastRemoteObject

This is a concrete subclass of `RemoteServer` that implements point-to-point remote references over TCP/IP networks. These references are nonpersistent: remote references to a server object are valid only during the lifetime of the server object. Before the server object is created (inside a virtual machine running on the host) or after the object has been destroyed, a client can't obtain remote references to the object. In addition, if the virtual machine containing the object exits (intentionally or otherwise), any existing remote references on clients become invalid and generate `RemoteException` objects if used.

Activatable

This concrete subclass of `RemoteServer` is part of the RMI object activation facility and can be found in the `java.rmi.activation` package. It implements a server object that supports persistent remote references. If a remote method request is received on the server host for an `Activatable` object and the target object is not executing at the time, the object can be started automatically by the RMI activation daemon.

13.4. Creating the Stubs and Skeletons

As mentioned earlier, JVMs in the 1.4 or earlier versions of Java require that you generate stub/skeleton classes using the RMI compiler. Once the interface and implementation classes have been compiled using the standard Java compiler, the RMI compiler (*rmic*, in the Sun JDK) is used to generate the stub and skeleton classes depicted earlier in [Figure 13-1](#). In its simplest form, you can run *rmic* with the fully qualified class name of your implementation class as the only argument. For example, once we've compiled the `Account` and `AccountImpl` classes, we can generate the stubs and skeletons for the remote `Account` object with the following command (Unix version):

```
% rmic AccountImpl
```

If the RMI compiler is successful, this command generates the stub and skeleton classes, `AccountImpl_Stub` and `AccountImpl_Skel`, in the current directory. The *rmic* compiler has additional arguments that let you specify where the generated classes should be stored, whether to print warnings, and so on. For example, if you want the stub and skeleton classes to reside in the directory `/usr/local/classes`, you can run the command using the `-d` option:

```
% rmic -d /usr/local/classes AccountImpl
```

This command generates the stub and skeleton classes in the specified directory. A full description of the *rmic* utility and its options is given in [Appendix F](#).

13.5. Accessing Remote Objects as a Client

Now that we've defined a remote object interface and its server implementation and generated the stub and skeleton classes that RMI uses to establish the link between the server object and the remote client, it's time to look at how you make your remote objects available to remote clients.

13.5.1. The Registry and Naming Services

The first remote object reference in an RMI distributed application is typically obtained through the RMI registry facility and the `Naming` interface. Every host that wants to export remote references to local Java objects must be running an RMI registry daemon of some kind. A registry daemon listens (on a particular port) for requests from remote clients for references to objects served on that host. The standard JDK distribution provides an RMI registry daemon, `rmiregistry`. This utility simply creates a `Registry` object that listens to a specified port and then goes into a wait loop, waiting for local processes to register objects with it or for clients to connect and look up RMI objects in its registry. You start the registry daemon by running the `rmiregistry` command, with an optional argument that specifies a port to listen to:

```
objhost% rmiregistry 5000 &
```

Without the port argument, the RMI registry daemon listens on port 1099. Typically, you run the registry daemon in the background (i.e., put an `&` at the end of the command on a Unix system or run `start rmiregistry [port]` in a command prompt on a Windows system) or run it as a service at startup.

Once the RMI registry is running on a host, you can register remote objects with it using one of these classes: the `java.rmi.registry.Registry` interface, the `java.rmi.registry.LocateRegistry` class, or the `java.rmi.Naming` class.

A `Registry` object represents an interface to a local or remote RMI object registry. The `bind()` and `rebind()` methods can register an object with a name in the local registry, where the name for an object can be any unique string. If you try to `bind()` an object to a name that has already been used, the registry throws an `AlreadyBoundException`. If you think that an object may already be bound to the name you want to register, use the `rebind()` method instead. You can remove an object binding using the `unbind()` method. Note that these three methods (`bind()`, `rebind()`, and `unbind()`) can be called only by clients running on the same host as the registry. If a remote client attempts to call these methods, the client receives a `java.rmi.AccessException`. You can locate a particular object in the registry using the `lookup()` method, while `list()` returns the names of all the objects registered with the local registry. Note that only `Remote` objects can be bound to names in the `Registry`. `Remote` objects are capable of supporting remote references. Standard Java classes aren't, so they can't be exported to remote clients through the `Registry`.

The `LocateRegistry` class provides a set of static methods a client can use to get references to local and remote registries, in the form of `Registry` objects. There are four versions of the static `getRegistry()` method, so that you can get a reference to either a local registry or a remote registry running on a particular host, listening to either the default port (1099) or a specified port. There's also a static `createRegistry()` method that takes a port number as an argument. This method starts a registry running within the current Java VM on the given local port and returns the `Registry` object it creates.

Using the `LocateRegistry` and `Registry` interfaces, we can register one of our `AccountImpl` remote objects on the local host with the following code:

```
AccountImpl server = new AccountImpl("Jim.Farley");
Registry localRegistry = LocateRegistry.getRegistry( );
try {
    localRegistry.bind("JimF", server);
}
catch (RemoteException re) { // Handle failed remote operation }
catch (AlreadyBoundException abe) { // Already one there }
catch (AccessException ae) { // Shouldn't happen, but... }
```

If this operation is successful (i.e., it doesn't raise any exceptions), the local registry has an `AccountImpl` remote object registered under the name `JimF`. Remote clients can now look up the object using a combination of the `LocateRegistry` and `Registry` interfaces or take the simpler approach and use the `Naming` class.

The `Naming` class lets a client look up local and remote objects using a URL-like naming syntax. The URL of a registered RMI remote object is typically in the format shown in [Figure 13-3](#). Notice that the only required element of the URL is the actual object name. The protocol defaults to `rmi:`, the hostname defaults to the local host, and the port number defaults to 1099. Note that the default `Naming` class provided with Sun's JDK accepts only the `rmi:` protocol on object URLs. If you attempt to use any other protocol, a `java.net.MalformedURLException` is thrown by the `lookup()` method.

Figure 13-3. Anatomy of an RMI object URL

If we have a client running on a remote host that wants to look up the `AccountImpl` we registered, and the `AccountImpl` object is running on a host named rmiremote.farley.org, the client can get a remote reference to the object with one line of code:

```
Account jimsAccount =
    (Account)Naming.lookup("rmi://rmiremote.farley.org/JimF");
```

If we have a client running on the same host as the `AccountImpl` object, the remote reference can be retrieved using the degenerate URL:

```
Account jimsAccount = (Account)Naming.lookup("JimF");
```

Alternatively, you can use the `LocateRegistry` and `Registry` interfaces to look up the same object, using an extra line of code to find the remote `Registry` through the `LocateRegistry` interface:

```
Registry rmtRegistry = LocateRegistry.getRegistry("rmireMOTE.farley.org");
Account jimsAccount =
    (Account)rmtRegistry.lookup("JimF");
```

When you look up objects through an actual `Registry` object, you don't have the option of using the URL syntax for the name, because you don't need it. The hostname and port of the remote host are specified when you locate the `Registry` through the `LocateRegistry` interface, and the RMI protocol is implied, so all you need is the registered name of the object. With the `Naming` class, you can reduce a remote object lookup to a single method call, but the name must now include the host, port number, and registered object name, bundled into a URL. Internally, the `Naming` object parses the host and port number from the URL for you, finds the remote `Registry` using the `LocateRegistry` interface, and asks the `Registry` for the remote object using the object name in the URL.

The principal use for the `Registry` and `Naming` classes in an RMI application is as a means to bootstrap your distributed application. A server process typically exports just a few key objects through its local RMI registry daemon. Clients look up these objects through the `Naming` facility to get remote references to them. Any other remote objects that need to be shared between the two processes can be exported through remote method calls.

13.5.2. Remote Method Arguments and Return Values

As already mentioned, a critical element of executing a remote method call is the marshaling and unmarshaling of the method arguments and, once the method has executed, the reverse marshaling and unmarshaling of the method's return value. RMI handles this process for you automatically, but you need to understand how different types of objects are transmitted from the method caller to the server object and back again. More importantly, you need to know which types of objects can't be used in remote method calls at all.

When you call a method on a remote object, the arguments to the method have to be serializable. That is, they need to be primitive Java data types (like `int`, `float`, etc.) or Java objects that implement `java.io.Serializable`. The same restriction applies to the return value of the remote method. This restriction is enforced at runtime, when you actually make the remote method call, rather than at compile time.

The RMI stub and skeleton layer decides how to send method arguments and return values over the network, based on whether a particular object is `Remote`, `Serializable`, or neither:

- If the object is a `Remote` object, a remote reference for the object is generated, and the reference is marshaled and sent to the remote process. The remote reference is received on the other end and converted into a stub for the original object. This process applies to both method arguments and return values.
- If the object is `Serializable` but not `Remote`, the object is serialized and streamed to the remote process in byte form. The receiver converts the bytes into a copy of the original object.
- If the method argument or return value is not serializable (i.e., it's not a primitive data type or an object that implements `Serializable`), the object can't be sent to the remote client, and a `java.rmi.MarshalException` is thrown.

The principal difference between remote and nonremote objects is that remote objects are sent *by reference*, while nonremote, serializable objects are sent *by copy*. In other words, a remote reference maintains a link to the original object it references, so changes can be made to the original object through the remote stub. If the server object calls update methods on an argument to a remote method and you want the updates to be made on the original object on the client side, the argument needs to be a `Remote` object that automatically exports a stub to the server object. Similarly, if the return value of a remote method call is intended to be a reference to an object living on the server, the server implementation needs to ensure that the object returned is a `Remote` object.

It's important to consider, whenever you pass either a remote reference or a serializable object from one VM to another in a remote method argument, whether the classes needed to unmarshal the method argument are available on the other side of the method call. If the receiver of the remote method argument has the necessary class definitions (the remote interface and stub class for remote references or the actual class definition for serializable objects) in its local `CLASSPATH` or some other local `ClassLoader` context, then you're fine. If it doesn't, you need to configure things so that the receiver of the method call can dynamically load the needed classes from the caller. We discuss how this happens in "[Dynamic Classloading](#)" later in this chapter.

13.5.3. Factory Classes

When a reference to a remote object is obtained through the RMI registry and then used to request additional remote references, the registered remote object is often referred to as a *factory class*.

Factory classes are useful in distributed applications that use remote objects, because in most cases you can't predict beforehand the kind and number of remote objects that will need to be shared between two processes. To make a remote object visible to clients through the RMI registry service, you need to explicitly create the object inside a Java VM on the server and then register that object using the `bind()` or `rebind()` method on the `Registry`. Using remote references obtained through method calls on factory objects, however, the client application can dynamically request the creation of new remote objects, without the objects being registered individually with the server registry.

As an example, suppose we're building a remote banking system, using the `Account` object we've been working with throughout this chapter. We want to set up a centralized server that provides account services to remote clients running on PCs, embedded in ATMs, and the like. On the server,

we could run an RMI registry, create an `Account` object for every account we have on record, and register each one with the RMI registry service using the account name. In this scheme, registering accounts with the RMI registry goes something like this:

```
Registry local = LocateRegistry.getRegistry( );
local.bind("Abrams, John", new AccountImpl("John Abrams"));
local.bind("Barts, Homer", new AccountImpl("Homer Barts"));
.
.
.
```

As you can imagine, this is quite unwieldy in practice. Starting the server can take a long time, as thousands of accounts need to be registered, many of them unnecessarily, since many accounts may not see any activity before the next downtime. More importantly, accounts that are created or closed during the server's lifetime somehow need to be added to or removed from the RMI registry, as well as to and from the bank's database of accounts. A much more sensible approach is to define a factory class for `Account` objects, along the lines of the following interface:

```
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface AccountManager extends Remote {
    public Account getAccount(String name) throws RemoteException;
    public boolean newAccount(Account s) throws RemoteException;
}
```

`AccountManager` lets a client ask for an account by name, using the `getAccount()` remote method. The method returns a reference to an `Account` object that corresponds to the account. Once the client has the `Account` reference, transactions against the account can be done through method calls on the `Account` object. The `AccountManager` also has a `newAccount()` method that allows clients to add new accounts to the manager's underlying database.

The server implementation of the `getAccount()` method simply needs to look up the named account in the account database, create an `AccountImpl` object to represent the account, and return the object to the remote client as a remote reference. Since `Account` objects are `Remote` objects, the RMI remote reference layer automatically creates a remote reference for the `Account` object, and the client that called the `getAccount()` method receives a stub for the `Account` object on the server.

Using the factory object to find accounts is more manageable than using the RMI registry. The bank maintains a database of accounts and their status, so the server implementation of the `AccountManager` can access that database directly to find accounts and create corresponding `Account` remote objects. Trying to keep the RMI registry in sync with the bank database makes the registry an unnecessary shadow of the main database of accounts, giving the bank two databases to maintain.

13.6. Dynamic Classloading

The RMI runtime system has a dynamic classloading facility that loads the classes it needs while executing remote method calls. In some situations, you don't need to worry much about how your application classes are obtained by the various agents in an RMI application. This is especially true if you have direct access to all hosts involved in the distributed system (i.e., if you can install your application classes in the local `CLASSPATH` for each machine participating in the application). For instance, when discussing the earlier `Account` example, we assumed all the relevant classes (`Account`, `AccountImpl`, stub, and skeleton classes) were installed on both the client and the server. However, if your distributed application involves remote agents running on hosts that aren't directly under your control, you need to understand how RMI loads classes at runtime, so you can ensure that each remote agent can find the classes it needs in order to run.

As with any Java application, the Java runtime system is responsible for loading the classes needed to initiate an RMI session. Starting an interaction with a remote object means loading the RMI API classes themselves, as well as the base interface for the remote object and the stub class for the remote interface. On the server side, the skeleton class for the remote object and the actual implementation class need to be loaded in order to run the server object that is being remotely exported.

The classes that are referenced directly by a given Java class are normally loaded by the same classloader that loaded the class itself. So, in an RMI client that does `Naming` lookup to find a remote object, the stub interface for the remote object is loaded using the classloader for the class doing the lookup. If the RMI client is a Java application (started using the `java` command to invoke the `main()` method on an object), the default (local) classloader tries to find the remote interface locally, from the local `CLASSPATH`. If the RMI client is an applet loaded in a web page, the `AppletClassLoader` tries to look for the remote interface on the applet's host, in the codebase of the applet.

The RMI runtime system provides its own classloader, the `RMIClassLoader`, to augment the default classloading process just described. The `RMIClassLoader` loads stubs and skeleton classes for remote interfaces, as well as the classes for objects used as remote method arguments or return values. These classes usually aren't explicitly referenced by your RMI application itself, but they are needed by the RMI runtime system for generating remote references and marshaling/unmarshaling method arguments and return values.

When it's loading the bytecodes for class definitions, the RMI runtime system first attempts to use the default classloader for the local context (i.e., an `AppletClassLoader` for an applet or the system classloader for a Java application). If the referenced class isn't found using the default local classloader, the `RMIClassLoader` tries to load the class bytecodes remotely according to the procedures explained next.

13.6.1. Configuring Clients and Servers for Remote Classloading

When the RMI runtime system marshals a remote object stub, method argument, or return value, it encodes a URL in the marshaled bytestream to tell the process on the receiving end of the stream where to look for the class file for the marshaled object. If the class for the object being marshaled was loaded by a nondefault classloader (e.g., the `AppletClassLoader` or the `RMIClassLoader`), the codebase of that classloader is encoded in the marshaled stream. If the class was loaded by the default classloader from the local `CLASSPATH`, the value of the `java.rmi.server.codebase` property for the Java VM marshaling the object is sent in the stream. This property is not set by default in the Java VM, so you need to make sure that it's set to a URL that points to the location of the necessary class files. One way to do this is to include a command-line argument when starting the Java VM, as in:

```
% java -Djava.rmi.server.codebase=http://objhost.org/classes RMIProcess
```

Here we start a Java process with its codebase set to <http://objhost.org/classes/>. This means that any remote process that needs to load classes for objects received from this process during an RMI session should use this HTTP URL in order to find them (if the classes can't be found on the local `CLASSPATH`, that is). This applies either if `RMIProcess` is serving remote objects itself through an RMI registry or if `RMIProcess` is passing objects into methods it is calling on other remote objects. In the first case, a remote client that needs to load the stub classes for the objects exported by `RMIProcess` uses the codebase to find these classes. In the second case, a remote process uses the codebase to load the classes for method arguments that `RMIProcess` is passing into remote method calls it makes.

If an RMI runtime system is trying to unmarshal an object stub, method argument, or return value and it doesn't find the class using the default classloader (e.g., the system classloader, which looks on the local `CLASSPATH` first), the `RMIClassLoader` can use the URL in the marshal stream to look for the class bytecodes remotely. The `RMIClassLoader` takes the URL from the marshaled bytestream and opens a URL connection to the specified host to load the needed classes. If both the local class search and this remote URL search fail to find the required classes, the unmarshal operation generates an exception, and the remote method call fails.

Note that in order for a Java runtime system to even attempt to load classes remotely, it has to have a security manager installed that allows remote classloading. The `java.rmi.RMISeccurityManager` can be used for this. In both your RMI object server and clients, include the following line before any RMI calls:

```
System.setSecurityManager(new RMISeccurityManager( ));
```

If you don't set the security manager, the Java VM is allowed to look for classes locally, and your RMI calls will work only if all of the required classes can be found on the local `CLASSPATH`.

Another issue with dynamically loading remote classes is that the default Java security policy doesn't allow all the networking operations required to resolve a class from a remote host. So, if you have an RMI client or server that needs to resolve classes remotely, you need to use a policy file that opens up network permissions to allow this. We don't go into all the details of network policies or the syntax of the security policy file, [\[2\]](#) but at minimum, you need to add the following line to the policy file on the RMI client:

[2] For details on Java code-level security policies and files, see *Java Security* by Scott Oaks (O'Reilly).

```
permission java.net.SocketPermission "objhost.org", "accept,connect";
```

This line allows the client to connect to any port on objhost.org and to accept connections from objhost.org on any of those ports. If you want to make this a bit more restrictive, since we're using HTTP as the protocol for downloading classes, you really just need port 80 access to/from objhost.org, so use this line in the policy file instead:

```
permission java.net.SocketPermission "objhost.org:80", "accept,connect";
```

The policy file change is needed in order to bypass the stricter rules imposed by the `RMISecurityManager` and allow the RMI runtime to create the network connections it needs. You can either add this permission to the default policy file used by your Java runtime (if you are using Sun's JRE, this is in `JAVA_HOME/lib/security/java.policy`, where `JAVA_HOME` is the root of your Java runtime installation), or you can create a custom policy file (perhaps by copying and editing the default one) and use it specifically for this application. If you've made a modified policy file, you can specify it on the command line when you start your RMI process, in a similar way to setting the `codebase` property in our earlier example:

```
% java -Djava.security.policy=mypolicy.txt RMIProcess
```

As a simple example, suppose we want to use our earlier `Account` example to export an `Account` object on one host and access that `Account` on another host, where the only class available locally is the `Account` interface class itself. On the server, we start an RMI registry [3] and run the `AccountServer` class as before, but since we want remote clients to be able to load the stub classes remotely, we need to set the `codebase` property to where the clients can find these classes:

[3] Note that in order for the RMI registry to recognize and pass along the `codebase` property you specify, it has to be started in such a way that it can't find any of the remotely loaded classes on its `CLASSPATH`. So start your RMI registry with a `CLASSPATH` that doesn't include the stub and skeleton classes, and then run your RMI server with a `CLASSPATH` that includes all required classes.

```
% java -Djava.rmi.server.codebase=http://objhost.org/classes/ AccountServer
Registered account.
```

We've set the `codebase` to <http://objhost.org/classes/>, so we have to make sure that an HTTP server is running on the objhost.org machine and that the necessary class files (e.g., the `AccountImpl` stub class) are in the `classes` directory of that HTTP server's document root.

Now, it may seem strange that we need to provide an HTTP server in order to download classes

between two RMI processes. If the RMI registry can provide remote objects to remote callers, why can't it provide class definitions as well? The reason that the designers of the RMI specification did not tie dynamic classloading to the RMI registry is because they did not want the RMI registry to be a strictly required element of the RMI architecture. In order to leave open the possibility of using other naming services to register RMI remote objects (CORBA, LDAP, custom systems), they needed to keep the RMI registry an optional component. Therefore, dynamic classloading had to be a separate, but portable, process, hence the HTTP process that exists in the specification. Luckily, as of JDK 1.4, the ability to customize the behavior of the `RMIClassLoader` was added to RMI, which opens up the possibility of implementing other dynamic classloading schemes (if the HTTP scheme doesn't fit your needs). See the RMI Javadoc for `RMIClassLoader` and `RMIClassLoaderSpi` (both in the `java.rmi.server` package) for details.

Now we can run the `AccountClient` class on the remote client as before, but the client's host machine doesn't have the stub class for the `AccountImpl` remote object available locally. When the `AccountClient` tries to look up the remote `Account` object, we want the stub class to be loaded remotely. Two simple changes to our `Account` example make this possible. First, add a line to the `AccountClient.main()` method that sets the `RMISecurityManager`, in order to allow for remote classloading:

```
import java.rmi.Naming;
import java.rmi.RMISecurityManager;

public class AccountClient {
    public static void main(String argv[]) {
        try {
            // Set the RMI security manager,
            // in case we need to load remote classes
            System.setSecurityManager(new RMISecurityManager( ));

            // Lookup account object
            Account jimAcct = (Account)Naming.lookup("rmi://objhost.org/JimF");
            .
            .
            .
        }
    }
}
```

The other change is to use a more lenient policy file when running `AccountClient` so the necessary network operations can be performed. Again, we don't discuss the syntax of the policy file here, but assuming you've put the required policy settings into a file named `rmipolicy.txt`, you can start the client like so:

```
% java -Djava.security.policy=rmipolicy.txt AccountClient
Deposited 12,000 into account owned by JimF
Balance now totals: 12000.0
```

You'll notice that the `-Djava.rmi.server.codebase=` option isn't needed here, because the client isn't providing class definitions to the `Account` server. If we were passing arguments into remote methods

called on the `Account` server, and the classes for these arguments weren't available on the client, then we would have to set up the `AccountClient` in a similar way to what we just described for the `Account` server (i.e., set up an HTTP-accessible codebase and export it to the server by setting the `java.rmi.server.codebase` property of the client's VM).

13.6.2. Loading Classes from Applets

Virtually all the steps we've outlined for running an RMI client to allow it to remotely load classes apply to applets as well. The only difference is that the classes for applets are loaded using an `AppletClassLoader`, which checks the applet's codebase for any classes required to run the applet. The default security policy for applets already allows for remote loading of classes, since this is how an applet works in the first place, so there's no need to change the security policy when using RMI within an applet. All you need to do to ensure that the applet finds the remote interface and stub class for the RMI object is to put them in the server directory that corresponds to the applet's codebase.



13.7. Remote Object Activation

Automatic activation of remote objects was added to RMI as of Java 1.2. The activation subsystem in RMI provides you with two basic features: the ability to have remote objects instantiated (activated) on demand by client requests and the ability for remote object references to remain valid across server crashes, making the references persistent. These features can be quite useful in certain types of distributed applications.

For example, think back to the `AccountManager` class we discussed when we talked about factory objects. We might not want to keep the `AccountManager` running on our server 24 hours a day; perhaps it consumes lots of server resources (memory, database connections, etc.), so we don't want it running unless it is being used. Using the RMI activation service, we can set up the `AccountManager` so that it doesn't start running until the first client requests an `Account`. In addition, after some period of inactivity, we can have the `AccountManager` shut down to conserve server resources and then reactivate the next time a client asks for an `Account`.

If a remote object is made activatable, it can be registered with the RMI registry without actually being instantiated. Normally, RMI remote objects (based on the `UnicastRemoteObject` interface) provide only nonpersistent references to themselves. Such a reference can be created for a client only if the referenced object already exists in a remote Java VM. In addition, the remote reference is valid only during the lifetime of the remote object. The remote object activation service adds support for persistent remote references that can be created even if the remote object is not running at the time of the request. This can persist beyond the lifetime of an individual server object.

The key features provided by the RMI activation service include:

- The ability to automatically create remote objects, triggered by requests for references to these objects.
- Support for activation groups, in which groups of activatable remote objects are executed in the same Java VM, which is automatically started by the activation service if needed.
- The ability to restart remote objects if they exit or are destroyed due to a system failure of some kind. This can add a certain degree of fault tolerance to RMI applications.

In the RMI activation system, activatable objects belong to activation groups, and each activation group runs within its own Java VM. If you don't group your activatable objects, instead simply assigning a new activation group to each activatable object you create, each object then runs inside a separate Java VM.

You typically define an activatable remote object by:

- Subclassing your remote object implementation from the `Activatable` class provided in the `java.rmi.activation` package

- Providing activation constructors in the server implementation
- Registering the object and its activation method with the activation service

If you want remote clients to directly access your activatable object, you also need to register the object with the RMI registry, so that it can be found by name on the network. You can register an activatable class with the registry without actually creating an instance of the remote object, as we'll see shortly.

You can also create an activatable object without subclassing the `Activatable` class. This might be necessary if you need to extend another class and the Java single-inheritance limit keeps you from also extending `Activatable`. For most of this section, we'll just discuss the case in which you're subclassing `Activatable`; we'll mention this other approach only when needed.

13.7.1. Persistent Remote References

The primary difference between an activatable remote object and a nonactivatable one is that a remote reference to an activatable object doesn't need to have a "live" object behind it. If an activatable object is not running (e.g., it hasn't been constructed yet, it has been garbage-collected by its Java VM, or its VM has exited), a remote reference to the object can still be exported to a client. The client receives a stub, as usual, and can make remote method invocations through the stub. When the first method is invoked, the activation service running on the server sees that the object is not active and goes about activating the object for the client. If the object doesn't have a VM to run in, the activation system starts one. The object is then activated using information that has been registered with the activation system. This information includes the object's class name, a URL that can load the class bytecodes if they're not found in the local `CLASSPATH`, and data to pass into the object's activation constructor. Once the object has been activated, the method invocation takes place, and the results are marshaled and sent back to the client.

As long as the object stays running, future method requests are handled as usual. If the object stops running for some reason (e.g., it is garbage-collected, or its VM dies), the next method request by a client triggers the activation service again, and the object is reactivated. This is what is meant by persistent remote references: remote references to activatable objects can persist across multiple lifetimes of the actual server object.

13.7.2. Defining an Activatable Remote Object

Naturally, before you can register and use an activatable object with the RMI activation system, you need to define the remote interface and the server implementation for the object. The `java.rmi.activation` package provides the classes you need to define an activatable remote object. You usually define a remote object as activatable by subclassing it from `Activatable` and defining a special constructor that activates the object. You also have to register the object with the activation service on the server host.

Other than that, the implementation of an activatable remote object is similar to that of a nonactivatable one. Start with a remote interface that contains the methods you want to export from your object. The interface should extend `Remote`, and each method should throw a `RemoteException` (or, as of Java 1.2, any parent of `RemoteException`). The server implementation implements this

interface and extends a concrete implementation of the `java.rmi.server.RemoteServer` class. Since you're defining an activatable remote object, you typically extend `java.rmi.activation.Activatable` directly and use its constructors to initialize, register, and activate your remote object. If you choose not to extend `Activatable` directly, you have to use the static `exportObject()` methods on the `Activatable` class to register your object with the activation runtime system.

13.7.2.1. The `Activatable` class

The `Activatable` class has four constructors. Here are signatures for two of them:

```
protected Activatable(String src, MarshalledObject data,
    boolean restart, int port) throws RemoteException
protected Activatable(String src, MarshalledObject data,
    boolean restart, int port, RMIClientSocketFactory csfactory,
    RMIServerSocketFactory ssfactory) throws RemoteException
```

These two constructors are *initialization* constructors. Use them when you decide to proactively create one of your remote objects and register it with the RMI activation service. In this case, the object already exists when a client first makes a method request on it, but if the object is destroyed, the next client request causes the object to be reactivated. These constructors register an object with the local activation service and export the object so that it can receive remote method requests. Both constructors have the following arguments in common:

- The `String` parameter is a URL that indicates where class bytecodes required by this object can be located. This information is exported to a remote client so it can dynamically load classes required to unmarshal method return values, for example. If a null value is passed in for this argument, remote clients of the activatable object can't dynamically load classes and therefore need the required classes available locally.
- The `MarshalledObject` parameter provides initialization data for the object; this parameter is necessary because data is typically sent from the activation daemon's VM to the VM designated to run the activatable object and the two might not be the same (more on this later).
- The `boolean` flag indicates whether the object should be automatically re-created when its home VM or its activation group is restarted (e.g., after a server restart).
- The `int` parameter specifies the port on which the object is exported. A port of zero tells the RMI runtime system to export the object on a random open port.

The second initialization constructor takes custom client and server socket factories that create socket communications between the server and the clients of the object. Customized socket factories were added to RMI in Java 1.2.

The other two `Activatable` constructors have the following signatures:

```
protected Activatable(ActivationID id, int port) throws RemoteException
```

```
protected Activatable(ActivationID id, int port,
    RMIClientSocketFactory csfactory, RMIServerSocketFactory ssfactory)
    throws RemoteException
```

These constructors are *(re)activation* constructors. The activation system uses them to activate a remote object that has received a remote method request, but isn't currently active. The `ActivationID` is a persistent ID issued by the activation system for the remote object, and the port number is the port that exports the remote object. The second constructor again takes custom server and client socket factories.

The `Activatable` class also has a set of `exportObject()` methods that correspond to the constructors just described. You can use these methods when an activatable object doesn't directly extend the `Activatable` class. You call the appropriate `exportObject()` methods from within the constructors of the class, so they serve the same function as calling the `Activatable` constructors during initialization of an `Activatable` subclass.

13.7.2.2. Implementing an activatable object

You can implement an activatable remote object in two ways: derive the remote object from the `Activatable` class directly and make the required calls to the `Activatable` constructors in its constructors or have the class implement a `Remote` interface and make the required calls to the static `exportObject()` methods in its constructors.

In either case, when the activation system activates a remote object, it looks for a constructor on the class that takes two arguments: an `ActivationID` and a `MarshaledObject`. The activation system calls this constructor, passing in an `ActivationID` it generates for the object and the `MarshaledObject` registered for the activatable object by the first constructor we just discussed.

This means you have to provide a constructor with this signature in your implementation of an activatable object. In this constructor, you should call either one of the (re)activation constructors on the `Activatable` parent class (if your class extends `Activatable`) or the corresponding `Activatable.exportObject()` method (if you didn't extend `Activatable`). In this call, you pass on the `ActivationID` issued by the activation system and specify the port for the exported remote object (a port number of 0 causes the object to be exported on a random open port).

In addition to this required constructor, you can define other constructors for your remote object implementation as needed. If you want your object to be reactivatable, any additional constructors should call one of the initialization constructors on `Activatable` (using `super()`) or the corresponding `exportObject()` method, passing in a valid source URL and a `MarshaledObject` to be used as an argument if the object is reactivated. If the object is destroyed at some point and a subsequent remote method request is received for it, the activation system reactivates the object by calling the required (re)activation constructor on the object's class, passing in this `MarshaledObject` argument.

[Example 13-3](#) shows a partial listing of an activatable implementation of the `Account` interface from [Example 13-1](#). The primary differences between this implementation and the nonactivatable one in [Example 13-2](#) are that this new implementation extends the `java.rmi.activation.Activatable` class instead of `UnicastRemoteObject`, and its constructors support the activation system. We've also defined a helper class, `AccountState`, that is used to wrap the account's state information (e.g., its

name and balance) so that it can be held within a `MarshaledObject` for initialization data. (This is described later in this section.)

Example 13-3. An activatable version of the account

```
import java.rmi.server.UnicastRemoteObject;
import java.rmi.activation.*;
import java.rmi.MarshaledObject;
import java.rmi.RemoteException;
import java.util.List;
import java.util.ListIterator;
import java.io.IOException;

public class ActivatableAccountImpl extends Activatable implements Account {
    // Our current balance
    private float mBalance = 0;
    // Name on account
    private String mName = "";

    // "Regular" constructor used to create a "preactivated" server
    public ActivatableAccountImpl(String name)
        throws RemoteException, ActivationException, IOException {
        // Register and export object (on random open port)
        super(null, new MarshaledObject(new AccountState(name, 0f)), false, 0);
        mName = name;
    }
    // Constructor called by the activation runtime to (re)activate
    // and export the server
    protected ActivatableAccountImpl(ActivationID id, MarshaledObject arg)
        throws RemoteException, ActivationException {
        // Export this object with the given activation ID, on random port
        super(id, 0);
        System.out.println("Activating an account");
        // Check incoming data (account state) passed in with activation request
        try {
            Object oarg = arg.get( );
            if (oarg instanceof AccountState) {
                AccountState s = (AccountState)oarg;
                // Set our name and balance based on incoming state
                mName = s.name;
                mBalance = s.balance;
            }
            else {
                System.out.println("Unknown argument type received on activation: " +
                    oarg.getClass( ).getName( ));
            }
        }
        catch(Exception e) {
            System.out.println("Error retrieving argument to activation");
        }
    }
}
```

```

    }
}
// Remainder of implementation is identical to AccountImpl
// .
// .
// .
}

// Define a structure to hold our state information in a single marshalled
// object, so that we can register it with the activation system
class AccountState {
    public float balance = 0f;
    public String name = null;
    public AccountState(String n, float b) {
        name = n;
        balance = b;
    }
}
}

```

The first constructor for `ActivatableAccountImpl` is a public one, used to construct an account for a user with the given name. The constructor registers the new object with the activation system by calling the constructor of its parent, `Activatable`. As we discussed earlier, the first argument to the `Activatable` constructor is the URL for the codebase (which is null, in this case, to indicate that we don't support dynamic classloading). The second argument, the (re)initialization data for the activatable object, is created by wrapping the state of the account (the name and balance, held in an `AccountState` object, a utility class defined at the end of the code listing) in a `MarshaledObject`. If the `Account` needs to be reactivated later, the activation daemon will use this `MarshaledObject` to initialize the reactivated `Account`, as explained later. The third and fourth arguments indicate that we want the object activated only on demand and that it can be exported on any random available port on the local machine.

The second constructor is used by the activation system and must be found on your implementation class when the activation system looks for it during a reactivation attempt. If an object of this type needs to be activated (or reactivated after a crash of some sort), this constructor is called to create the remote object. The constructor takes an `ActivationID` argument, issued by the activation system and the `MarshaledObject` that was previously registered for the object with the activation system. Our `ActivatableAccountImpl` constructor exports the object by calling the second constructor on the `Activatable` class, then initializes itself with the data from the `MarshaledObject`.

13.7.3. Registering Activatable Objects

There are several ways to register an activatable object with its local activation system. In each case the activation system needs to be told how to create (or re-create) the object. The information the activation system needs to activate an object is encapsulated in the `ActivationDesc` class. An `ActivationDesc` object contains the name of the class for the remote object, a URL with the network location of the bytecodes for the class, a `MarshaledObject` to be used as the initialization data for the object, and the group assignment for the object.

The simplest way to register an `Activatable` object is to create an instance of the object. In our example, we've derived our server implementation from the `Activatable` class, so the public constructor on the `ActivatableAccountImpl` class registers the object by calling the necessary constructor on `Activatable`. Thus, we can create and register one of these as follows:

```
// Make an activation group for the object
ActivationGroupDesc gdesc = new ActivationGroupDesc(null, null);
ActivationGroupID gid = ActivationGroup.getSystem( ).registerGroup(gdesc);
ActivationGroup.createGroup(gid, gdesc, 0);

// Make an Account object, which registers itself with activation system
Account account = new ActivatableAccountImpl(accountName);

// Register with naming service
LocateRegistry.getRegistry( ).rebind(accountName, account);
```

The first four lines are required to create an activation group for our activatable object. We'll talk more about activation groups shortly. For now, all you need to know is that this code creates the default activation group for the current VM. Any remote object that isn't specifically assigned to a group is placed in this default group.

The activatable object itself is created by calling the public `ActivatableAccountImpl` constructor. This constructor registers the object with the activation system by calling the appropriate `Activatable` constructor, as we've already discussed. Since we haven't specified an activation group for the object, it is placed in the default group we just created. If we hadn't created that default group, the activation system would throw an exception here, when the object is registered.

Aside from the creation of the activation group, this example looks a lot like our other examples of registering RMI objects. The difference here is that if the process dies off at some point, the activation system can reactivate the activatable object in a new Java VM using the information provided in the `ActivationDesc` for the object. In this case, we're relying on the `Activatable` constructor (which is called by our `ActivatableAccountImpl` constructor) to create and register an `ActivationDesc` for our object using the information we passed into it when we called it from our constructor.

When an object needs to be activated, the activation system first looks up the `ActivationDesc` for the object and then looks for the class referenced in the `ActivationDesc`, using the codebase URL specified in the `ActivationDesc` to load the class bytecodes, if necessary. Once the class has been loaded, the activation system creates an instance of the class by calling the activation constructor, which takes an `ActivationID` and a `MarshaledObject` as arguments. The `ActivationID` is issued by the activation system, and the `MarshaledObject` contains the data previously registered with the `ActivationDesc`. In our activatable `Account` in [Example 13-3](#), the activation system calls the second constructor on our `ActivatableAccountImpl` class. The new object passes the `ActivationID` up to the `Activatable` constructor so that it can be recorded, and the account state (name and balance) is pulled from the `MarshaledObject`. The `Activatable` constructor takes care of creating and registering an `ActivationDesc` for the object and exporting the object with the activation system.

13.7.3.1. Registering an activatable object without instantiating

A more complicated but often more useful way to register a remote object is to create an `ActivationDesc` for it and then register this information directly with the activation system, *without* creating an instance of the object. The static `Activatable.register()` method accepts an `ActivationDesc` object and registers it with the activation system directly. Here's how we can do that

```
// Make a codebase and activation argument for the object
String src = "http://objhost.org/classes";
MarshaledObject actArg = new MarshaledObject("MyAccount");

// Create the ActivationDesc and get a stub for the object
ActivationDesc desc =
    new ActivationDesc("com.oreilly.jent.rmi.ActivatableAccountImpl",
        src, actArg);
Account accountStub =
    (Account)Activatable.register(desc);
```

When we create the `ActivationDesc` for the object, we specify the name of the class to use for creating the object, a codebase for finding the class, and a `MarshaledObject` that is passed to the object when it's activated. The `ActivationDesc` is used in the call to the `Activatable.register()` method, which returns a `RemoteStub` for the activatable object. Since we know this stub is for an object that implements the `Account` interface, we can safely cast it to an `Account`. We can also use this reference to register the remote object with the local RMI naming registry:

```
LocateRegistry.getRegistry( ).bind("Account", accountStub);
```

Although it is not shown here, note that you also have to create an activation group for the object, just as we did in our earlier example, before you can register it with the activation service.

So, to recap, we've registered a remote object with the activation system and the RMI naming registry without actually creating the object itself. When a client tries to look up the object, it gets back a remote stub, with no active object behind it on the server. When the client calls a method on the stub, however, the activation system on the server creates the object, using the information in the `ActivationDesc` we provided.

13.7.3.2. Passing data with the MarshaledObject

The sole option for passing arguments to activatable objects before they are activated is through the `MarshaledObject` contained within the `ActivationDesc` for the object. However, once the `ActivationDesc` is registered with the activation system, you can't dynamically update the contents of the `MarshaledObject`. This is a serious problem in our example, if you've been following it closely. We wrap the account state (the account owner's name and the account balance) and register it with the activation system just once, when we create or register the object. As clients interact with the

`Account`, the balance is going to change, but our registered `MarshaledObject` is not going to change along with it. If the VM of the `Account` crashes and the `Account` is reactivated using the `MarshaledObject` registered for it, the balance will be set back to the value it was when the object was first registered with the activation system.

There are a number of ways to get around this problem and make the initialization arguments more dynamic. One is to bundle a filename or URL into the `MarshaledObject`, rather than the data itself. We could then keep the data behind the file or URL updated with the current state of the `Account`. At the point that the object is activated, it can read data from the file or URL and use the up-to-date data during activation. Alternatively, if this were a real-world example and our account information were stored persistently in a database, we would just need to pass the account ID in the initialization parameters, and the `Account` object could then read the current state of the `Account` from the database.

13.7.4. Activation Groups

Every activatable RMI object belongs to an activation group. Each group of activatable objects runs within the same Java VM on the server host. In essence, activation groups are a way of defining collections of activatable remote objects that should share the same physical address space. We've already seen how to set up an activation system, since we had to do this before registering our activatable object with the activation system. In this section, we'll take a look at creating activation groups in a bit more detail and discuss what the activation group is actually doing for you.

Activation groups in RMI are more than just a way of organizing remote objects. Each activation group is responsible for monitoring, activating, and reactivating the objects it contains. Internally, remote requests from clients are intercepted by an `ActivationMonitor` that coordinates the active/inactive status of server objects with the `ActivationGroup`. If an object needs to be activated, an `Activator` is invoked to make the object active within the `ActivationGroup`. Note that you don't normally need to interact with the underlying components of the activation system. Simply set up your `ActivationGroup` objects and assign activatable objects to them; the activation system does the rest for you.

An `ActivationGroup` is created when the first object in the group needs to be activated. The `Activator` is responsible for creating a VM for the `ActivationGroup` to run in and for starting the `ActivationGroup` using the information in the registered object's `ActivationGroupDesc`, if it has one. If the remote object doesn't have a specified group, a default one is created. The new `ActivationGroup` object is then told to activate the requested remote object by calling its `newInstance()` method. The arguments the `Activator` passes into this method are the `ActivationID` for the new object and the `ActivationDesc` that the `Activator` has registered for the object.

The `ActivationDesc` gives an `ActivationGroup` everything it needs to activate the remote object. The `ActivationGroup` takes the class name for the object and looks for the class bytecodes. First it checks the local `CLASSPATH`, and if that pulls up nothing, it uses the `URL` in the `ActivationDesc` to load the class from the given URL. Once the class is loaded, an instance of the class is created by calling the activation constructor on the class (e.g., the constructor that has an `ActivationID` argument and a `MarshaledObject` argument). The `ActivationID` and `MarshaledObject` come from the call to the `newInstance()` method. The new, active remote object is returned to the `Activator` as a serialized `MarshaledObject`. This is done for two reasons. First, the `Activator` runs in a separate Java VM, so the active object reference needs to be transferred from one VM to another, and the easiest way to do this is to serialize it and transmit it in that form. Second, since the object has been bundled into a

`MarshaledObject`, the `Activator` doesn't need to load the object's bytecodes unless absolutely necessary. In most cases, the `Activator` doesn't need to interact directly with the object itself, so it doesn't need to waste time loading unnecessary bytecodes.

Each `ActivationGroup` has an `ActivationMonitor` associated with it. The `ActivationGroup` has to tell the `ActivationMonitor` whenever an object becomes active or inactive. An activatable object is responsible for informing its `ActivationGroup` when it becomes active and inactive, by calling the group's `activeObject()` and `inactiveObject()` methods, respectively. The `ActivationGroup`, in turn, passes the information on to the `ActivationMonitor` by calling identical methods on the monitor object. When the object becomes inactive, the `ActivationMonitor` makes note of it and arranges for the object to be reactivated the next time a method request comes in for it. If an entire `ActivationGroup` becomes inactive, the `ActivationMonitor` is informed through its `inactiveGroup()` method. The next request for an object in that group causes the `Activator` to re-create the group.

13.7.4.1. Registering activation groups

An `ActivationGroup` is registered with the activation system in roughly the same way as an activatable object. You have to create an `ActivationGroupDesc` object that contains the name of the class for the group, the URL where the class bytecodes can be loaded, and a `MarshaledObject` that is given to the `ActivationGroup` as initialization data. Unlike activatable objects, though, the class of a group has to be a concrete subclass of `ActivationGroup`. Register the `ActivationGroupDesc` by calling the static `ActivationSystem.registerGroup()` method, passing in the `ActivationGroupDesc`. The `ActivationSystem` returns an `ActivationGroupID` that can assign specific objects to the group.

13.7.4.2. Assigning activatable objects to groups

Assign an activatable object to a group by specifying the group ID in the `ActivationDesc` registered with the activation system. The `ActivationGroupID` returned by the `ActivationSystem.registerGroup()` method can be passed into the `ActivationDesc` constructor.

Before you can register a remote object with the activation system, you need to create a group for it. For our activatable `Account` example, we can run Java code along the following lines on the object server (note that we've left out the exception handling):

```
// Make an activation group for the object
ActivationGroupDesc gdesc = new ActivationGroupDesc(null, null);
ActivationGroupID gid =
    ActivationGroup.getSystem( ).registerGroup(gdesc);
ActivationGroup.createGroup(gid, gdesc, 0);

// Set up ActivationDesc for object
String codebaseURL = "http://objhost.org/classes";
String accountName = "Fred";
MarshaledObject activationArg = new MarshaledObject(accountName);
ActivationDesc desc =
    new ActivationDesc(gid, "com.oreilly.jent.rmi.ActivatableAccountImpl",
        codebaseURL, activationArg);
```

```
Account accountRef = (Account)Activatable.register(desc);
LocateRegistry.getRegistry( ).rebind(accountName, accountRef);
```

Here we're using the `ActivatableAccountImpl` class and registering a remote object with the activation system without actually instantiating it. Before we register our remote object, we create an `ActivationGroupDesc`, then use it to register and create a new activation group with the activation system. After we create the activation group (using the `ActivationGroup.createGroup()` method), we use the `ActivationGroupID` for our new group to make an `ActivationDesc` for our remote object. We use that to register the object with the activation system. The activation system generates a remote stub for our object, and we register that with the RMI naming registry.

Since each `ActivationGroup` is started within its own VM if it's initially activated by the activation system, grouping objects is a convenient way to partition your remote objects into shared address spaces on your server.

13.7.5. The Activation Daemon

The heart of the RMI activation system is the activation daemon, which runs on the host for an activatable object. The activation daemon is responsible for intercepting remote method requests on activatable objects and orchestrating the activation of the object, if needed.

The activation daemon provided with the Java SDK, `rmid`, runs a Java VM that includes a `java.rmi.activation.Activator` object. The `Activator` is responsible for keeping a registry of activatable objects, along with the information needed to activate them. This information is in two parts: an `ActivationDesc` object and an optional `ActivationGroupDesc`. The `ActivationGroupDesc` identifies the group of activatable objects to which the object should be added and describes how to start the group if it doesn't exist. The `ActivationDesc` includes all information needed to activate the object itself. An activatable object has to be registered with the activation system in one of the ways described earlier to be started automatically by the `Activator`.

If a remote method request is received by the RMI runtime system on a host, and the target object hasn't been created yet, the `Activator` is asked to activate it. The `Activator` looks up the `ActivationDesc` (and `ActivationGroupDesc`, if present) for the object. If the object has an `ActivationGroup` assigned to it and the `ActivationGroup` doesn't exist yet, a Java VM is started for the group, and the `ActivationGroupDesc` data is used to start an `ActivationGroup` object within the new VM. If the object has no `ActivationGroup` associated with it, it's given its own `ActivationGroup` running in its own VM. The group is then asked to start the requested object, using the `ActivationDesc` object registered for the object. Once the `ActivationGroup` activates the object within its VM, the `Activator` is notified, and the now-active remote reference is returned to the RMI runtime system. The RMI runtime system forwards the remote method request through the reference to the object, and the return value is exported back to the client as usual.

13.7.5.1. The daemon's dual personality

When you start the `rmid` daemon, it creates an `Activator` and then listens on the default port of 1098 for activation requests. There is also a `-port` command-line option that lets you specify a

different port for the VM to use. In addition to running the `Activator`, the `rmi` daemon also runs its own RMI `Registry`. If needed, you can register local objects with the daemon's internal `Registry` by specifying the daemon's port when you call the `bind()` or `rebind()` method of the `Registry`. For example, if `rmi` is running on its default port of 1098:

```
RemoteObject server = ...
Registry local = LocateRegistry.getRegistry(1098);
local.bind(server, "Server");
```

This way, you can consolidate your activation system and your naming service into one VM on your server.



13.8. RMI and Native Method Calls

RMI is a Java-only remote object scheme, so it doesn't provide a direct connection between objects implemented in different languages, as CORBA does. But, using Java's Native Interface API, it is possible to wrap existing C or C++ code with a Java interface and then export this interface remotely through RMI.

To demonstrate, let's suppose we have some (legacy) native code that implements the withdrawal, deposit, and transfer operations for a particular banking system. We can create an implementation of our `Account` interface that uses this native code to implement the `deposit()`, `withdraw()`, and `Transfer()` methods on our remote interface. The implementation for a `NativeAccountImpl` is shown in [Example 13-4](#). The only significant difference between this implementation and our original `AccountImpl` is that the `deposit()`, `withdraw()`, and `Transfer()` methods are declared native, so the method bodies are not provided here.

Example 13-4. Remote object using a native method implementation

```
import java.rmi.server.UnicastRemoteObject;
import java.rmi.RemoteException;
import java.util.List;
import java.util.ListIterator;

public class NativeAccountImpl extends
    UnicastRemoteObject implements Account {
    // Our current balance
    private float mBalance = 0;
    // Name on account
    private String mName = "";
    // Create a new account with the given name
    public NativeAccountImpl(String name) throws RemoteException {
        mName = name;
    }

    public String getName() throws RemoteException {
        return mName;
    }
    public float getBalance() throws RemoteException {
        return mBalance;
    }
    // Withdraw some funds
    native public void withdraw(float amt)
        throws RemoteException, InsufficientFundsException;
    // Deposit some funds
```

```

native public void deposit(float amt) throws RemoteException;
// Move some funds from another (remote) account into this one
native public void transfer(float amt, Account src)
    throws RemoteException, InsufficientFundsException;

// Remainder of implementation is identical to AccountImpl...

```

We can compile this RMI class and generate the stubs and skeletons for it using the RMI compiler, just like with our other RMI examples. But once this is done, we need to provide native implementations for the `withdraw()`, `deposit()`, and `transfer()` methods. To start, we can generate a C/C++ header file for the native methods using the *javah* tool:

```
% javah -jni -d . NativeAccountImpl
```

The `-jni` option tells the *javah* tool to generate JNI-compliant header files (as opposed to header files based on the earlier native method interface that shipped with Java 1.0). Invoking this command generates a JNI C/C++ header file that looks something like the following:

```

/* DO NOT EDIT THIS FILE -- it is machine generated */
#include <jni.h>
/* Header for class com_oreilly_jent_rmi_NativeAccountImpl */

#ifdef _Included_com_oreilly_jent_rmi_NativeAccountImpl
#define _Included_com_oreilly_jent_rmi_NativeAccountImpl
#ifdef _ _cplusplus
extern "C" {
#endif
// NOTE: Some preprocessor directives deleted for clarity...
/*
 * Class: com_oreilly_jent_rmi_NativeAccountImpl
 * Method: withdraw
 * Signature: (F)V
 */
JNIEXPORT void JNICALL Java_com_oreilly_jent_rmi_NativeAccountImpl_withdraw
    (JNIEnv *, jobject, jfloat);

/*
 * Class: com_oreilly_jent_rmi_NativeAccountImpl
 * Method: deposit
 * Signature: (F)V
 */
JNIEXPORT void JNICALL Java_com_oreilly_jent_rmi_NativeAccountImpl_deposit
    (JNIEnv *, jobject, jfloat);

/*

```

```

* Class: com_oreilly_jent_rmi_NativeAccountImpl
* Method: transfer
* Signature: (FLcom/oreilly/jent/rmi/Account;)V
*/
JNIEXPORT void JNICALL Java_com_oreilly_jent_rmi_NativeAccountImpl_transfer
(JNIEnv *, jobject, jfloat, jobject);

#ifdef _ _cplusplus
}
#endif
#endif

```

The only details worth noting in this header file are the inclusion of the *jni.h* header file (which is provided with the Java SDK) and the actual method declarations. The *jni.h* header file provides declarations and definitions for all of the data structures and utility methods provided by the JNI API. The method declarations have a signature that corresponds to the native methods declared on our Java class. When you invoke the `withdraw()` method on the `NativeAccountImpl`, for example, the Java VM looks for a native method that matches the signature (shown here) for the `Java_com_oreilly_jent_rmi_NativeAccountImpl_withdraw` function.

Now all we need to do is implement the C/C++ functions declared in our JNI-generated header file. This is where we tie our Java methods to some legacy native code. In this case, suppose the native code is wrapped up in a single C/C++ function called `dbInvoke()`. This function is available in a native library on the server platform (e.g., a DLL file on Windows or a shared library on Unix). We want to use our Java method to invoke this native function to perform various operations on the database holding our account information, so we can implement the `Java_NativeAccountImpl_withdraw()` function along these lines:

```

#include <jni.h>
#include "com_oreilly_jent_rmi_NativeAccountImpl.h"

JNIEXPORT void JNICALL Java_com_oreilly_jent_rmi_NativeAccountImpl_withdraw
(JNIEnv * env, jobject me, jfloat amt) {
    // Call the native method that will withdraw the money from the account

    // First, get the name of the account holder from the account object
    jclass acctCls =
        (*env)->FindClass(env, "com/oreilly/jent/rmi/NativeAccountImpl");
    jmethodID nameMID = env.GetMethodID(env, acctCls,
                                        "getName", "( )Ljava/lang/String;");
    jobject nObj = (*env)->CallObjectMethod(env, me, nameMID);
    const char* name = (*env)->GetStringUTFChars(env, (jstring)nObj, 0);

    // Now call the native function to withdraw money from the account
    // stored in the non-JDBC database...
    int err = dbInvoke("update ACCOUNT ...");
    if (err > 0) {
        // IF an error occurs, assume it's a balance problem and throw an
        // InsufficientFundsException
    }
}

```

```

jclass excCls =
    (*env)->FindClass(env,
                      "com/oreilly/jent/rmi/InsufficientFundsException");
    (*env)->ThrowNew(env, excCls, "Error during withdrawal.");
}
}

```

The first part of the function gets the name of the `Account` that is the target of the withdrawal. From native code, this simple method call is rather convoluted: first, we locate the Java class definition through the JNI environment, then we get a handle on the method we want to call, and finally we call the method on the reference to the `NativeAccountImpl` object passed in as a function argument. The Java `String` return value of the `getName()` method is returned to us in the native environment as a `jobject` structure, which we convert to a C/C++ char array. When this is complete, we can then invoke the native `dbInvoke()` function to pass SQL commands to the account database. If some kind of error occurs during the database call, then we throw an exception back to the Java environment.

Once we compile this C/C++ code (linking with the native library that contains the `dbInvoke()` function), we can export remote `NativeAccountImpl` objects. Then remote clients can call the `withdraw()`, `deposit()`, or `Transfer()` methods. These remote method calls in turn cause the invocation of native code on the server, when the object implementation calls its native methods.

Note that in order for the server object to find its native method, the native library containing the `dbInvoke()` function has to be loaded into the server object's VM using the `System.loadLibrary()` method. You can do this either in the application code that uses the native method or by adding a static initializer to the class. You can have the library loaded automatically when the `NativeAccountImpl` class is referenced:

```

static { System.loadLibrary("nativeDB"); }

```

The `System.loadLibrary()` method automatically converts the library name that you provide to a platform-specific filename. So if the previous example is run on a Solaris machine, the Java VM looks for a library file named `libnativeDB.so`. On a Windows machine, it looks for `nativeDB.dll`.

13.8.1. RMI with JNI Versus CORBA

There are pros and cons to using RMI and JNI to export legacy native code using Java remote objects, as opposed to using CORBA. With CORBA, a CORBA object implemented in the same language as the native code (C/C++ for our example) is created and exported on the server. Remote Java clients can get a Java stub to this CORBA object using Java IDL, or any third-party Java CORBA implementation (see [Chapter 14](#) for details).

One obvious advantage of the CORBA approach is that you don't need to have Java on the server. Since this is presumably a legacy server, perhaps a mainframe of some sort, finding a stable Java VM and development kit for the platform may be a problem. If a Java implementation isn't available or if installing additional software on the legacy server isn't desirable, CORBA is your only option.

An advantage of the RMI/JNI approach is that you're running Java at both ends of the remote communication and avoiding the use of CORBA entirely. CORBA is a rich distributed object API, but it may be overkill for your application. Using the simpler RMI API and keeping your code development strictly in Java (with some minimal C/C++ to interface to the legacy code) might be an advantage to you in this case.



13.9. RMI Over IIOP

Another approach for connecting RMI objects to non-Java objects is the ability for RMI objects to communicate directly with remote CORBA objects using IIOP, the CORBA network interface protocol.^[4] The standard RMI implementation provided with Java uses an RMI-specific protocol, JRMP, to communicate over the network. RMI/IIOP allows RMI objects to use the CORBA network protocol, IIOP, to communicate with other objects. This means that an RMI object using RMI/IIOP can communicate with a remote CORBA object, regardless of the implementation language of the CORBA object. Likewise, a CORBA object can interact with your Java RMI objects directly. This really gives you the best of both worlds, since you can then implement your remote clients using RMI and use either CORBA or RMI/JNI on the server to interface to any native legacy code.

^[4] The RMI/IIOP tools and classes became a standard part of the Java core environment as of JDK 1.3. If you are using an earlier version of Java, you need to download the RMI/IIOP tools and libraries from <http://java.sun.com/products/rmi-iiop>.

In order to convert your RMI objects to use IIOP, you need to make some changes:

- Any implementation classes should extend `javax.rmi.PortableRemoteObject`, rather than `java.rmi.server.UnicastRemoteObject`.
- All your stub and skeleton classes need to be regenerated using the updated `rmi` compiler provided with the RMI/IIOP installation. This updated compiler has an `-iiop` option that produces stubs and ties (*ties* refers to skeletons in the CORBA vernacular). These stubs and ties handle the link between client and server objects, but use IIOP rather than JRMP.
- All use of the RMI `Naming` registry has to be converted to use of JNDI to talk to a CORBA Naming Service. Objects that you export are bound to names in the CORBA Naming Service through the JNDI context, and remote objects you look up are accessed from the Naming Service through the JNDI context.
- Instead of using the standard Java casting operator on remote objects you look up, use the `javax.rmi.PortableRemoteObject.narrow()` method.

To give you a taste for how to use RMI/IIOP with your RMI classes, let's convert our first `Account` example to use RMI/IIOP. First, we need to update the `AccountImpl` class to extend `PortableRemoteObject`. The following fragment of the `IIOPAccountImpl` class does that:

```
import javax.rmi.PortableRemoteObject;
import java.rmi.RemoteException;
import java.util.List;
import java.util.ListIterator;
public class IIOPAccountImpl extends PortableRemoteObject implements
Account {
    // Remainder of implementation is identical
```

We can compile the updated `IIOPAccountImpl` using the regular Java compiler, then use the extended `rmic` compiler included with RMI/IIOP to generate IIOP stubs and ties:

```
% rmic -iiop -d /home/myclasses IIOPAccountImpl
```

This generates an `IIOPAccountImpl_Stub` class and an `IIOPAccountImpl_Tie` class, which act as the IIOP stub and tie for the remote object.

In the CORBA world, remote objects are looked up using the CORBA Naming Service, so we need to update the `AccountServer` class to use JNDI to register an `Account` object with a CORBA Naming Service, rather than the RMI registry. The updated `IIOPAccountReg` class looks like this:

```
import javax.naming.*;
import java.rmi.*;

public class IIOPAccountReg {
    public static void main(String argv[]) {
        try {
            // Make an Account with a given name
            IIOPAccountImpl acct = new IIOPAccountImpl("Jim.Farley");

            // Get a reference to CORBA Naming Service using JNDI
            Hashtable props = new Hashtable( );
            props.put("java.naming.factory.initial",
                    "com.sun.jndi.cosnaming.CNCtxFactory");
            props.put("java.naming.provider.url", "iiop://objhost.org:900");
            Context ctx = new InitialContext(props);

            // Register our Account with the CORBA Naming Service
            ctx.rebind("JimF", acct);
            System.out.println("Registered account.");
        }
        catch (Exception e) {
            e.printStackTrace( );
        }
    }
}
```

Refer to [Chapter 7](#) for details on the properties used to create the JNDI context and what they mean. All you need to glean from this is that we're trying to connect to a naming service running on objhost.org, listening to port 900. Once we are connected, we register the new `IIOPAccountImpl` object with the naming service using the `Context.rebind()` method.

Finally, we need to update our client so that it works with RMI/IIOP. Instead of using an RMI registry

to look up the remote `Account` object, the client needs to use JNDI to connect to the same CORBA Naming Service that now hosts our `Account` object and ask for the `Account` by name. The updated `IIOPAccountClient` is shown here. Notice that we've also changed the client to use the `PortableRemoteObject.narrow()` method, instead of just casting the object returned from the lookup:

```
import javax.naming.*;
import java.rmi.RMISecurityManager;

public class IIOPAccountClient {
    public static void main(String argv[]) {
        try {
            // Lookup account object
            Hashtable props = new Hashtable( );
            props.put("java.naming.factory.initial",
                    "com.sun.jndi.cosnaming.CNCtxFactory");
            props.put("java.naming.provider.url", "iiop://objhost.org:900");
            Context ctx = new InitialContext(props);
            Account jimAcct =
                (Account)PortableRemoteObject.narrow(ctx.lookup("JimF"),
                                                    Account.class);

            // Make deposit
            jimAcct.deposit(12000);
            // Report results and balance.
            System.out.println("Deposited 12,000 into account owned by " +
                               jimAcct.getName( ));
            System.out.println("Balance now totals: " + jimAcct.getBalance( ));
        }
        catch (Exception e) {
            System.out.println("Error while looking up account:");
            e.printStackTrace( );
        }
    }
}
```

In order to register the server object, we need a CORBA Naming Service running, just as we need an RMI registry with standard RMI. The RMI/IIOP package includes a special naming service that is started using the `tnameserv` utility. This tool is similar to the naming service provided with Java IDL (and discussed in [Chapter 14](#)), but this version is a CORBA Naming Service that also provides JNDI access. On objhost.org, we need to start the naming service like so:

```
objhost% tnameserv -ORBInitialPort 900
```

Now we can run `IIOPAccountReg` to register the `Account` object with the naming service, then run our `IIOPAccountClient` to access the `Account` and make a deposit. All network communications are now taking place using IIOP rather than the RMI protocol.

13.9.1. Accessing RMI Objects from CORBA

Since our `Account` object is now speaking IIOP, we can also access it from other, non-Java CORBA clients. First, we need to get an IDL interface for the `Account` interface, which can be done using the `rmic` compiler provided with RMI/IIOP. The `-idl` option generates an IDL mapping of a Java RMI interface using the Java-to-IDL mapping defined by the CORBA standard. With this IDL mapping, we can generate language-specific stubs that allow any CORBA client to talk to our Java remote object. See [Chapter 14](#) for more details on using IDL and generating language-specific interfaces from it.



Chapter 14. Java IDL (CORBA)

The Java IDL API provides an interface between Java programs and distributed objects built using the Common Object Request Broker Architecture (CORBA). CORBA is a standard defined by the Object Management Group (OMG). It describes architecture, interfaces, and protocols that distributed objects can use to interact with each other. Part of the CORBA standard is the Interface Definition Language (IDL), which is an implementation-independent language for describing the interfaces of remote-capable objects. Standard mappings are defined by the OMG for converting IDL interfaces into C++ classes, C code, and Java classes, among others (a complete list is provided later in the chapter). These generated classes use the underlying CORBA framework to communicate with remote clients and give you the basis for implementing and exporting your own distributed objects. Java IDL is Sun's implementation of the standard IDL-to-Java mapping and is provided by Sun's JDK in the `org.omg.CORBA` package, the `org.omg.CosNaming` package, and other `org.omg.*` packages.

Like RMI, Java IDL gives you a way to access remote objects over the network. It also provides the tools you need to make your objects accessible to other CORBA clients. If you export a Java class using Java IDL, it's possible to create an instance of that class and publish it through a naming/directory service. A remote client can find this object, call methods on it, and receive data from it, just as if it were running on the client's local machine. Unlike RMI, however, objects that are exported using CORBA can be accessed by clients implemented in any language with an IDL binding (not just Java).

The CORBA standard is extensive, to say the least. In addition to the basic remote object architecture and the syntax of IDL, it also includes specifications for several distributed object services, like an object naming service, a security policy service, and persistent object services. It would be foolhardy to attempt to cover all these topics completely in one chapter, so we won't. Instead, we'll just cover the basic features of the CORBA architecture and the IDL syntax. A full reference for IDL syntax is provided in [Appendix G](#). We'll also look at the Naming Service, which is key to almost every CORBA application, because it provides a standard way to find remote CORBA objects on the network. After that, we'll take a look at the Java IDL API and the *idlj* compiler and how they work together to give you an interface from your Java code to CORBA objects and services. They also provide the tools you need to create your own CORBA objects, implemented in Java.

The rest of this chapter is broken down roughly into three parts. In the first part, we'll look at an overview of the CORBA architecture and how it allows you to create, export, access, and manage remote objects. In the second part, we'll explore the details of creating your own CORBA objects. Finally, we'll look at how clients can remotely access your CORBA objects.

14.1. A Note on Evolving Standards

At the time CORBA support was first made part of the core Java API, the CORBA specification and the IDL-to-Java binding for CORBA were in a bit of flux. The server-side object adaptor interface (the interface between remote object implementations and the core CORBA object services) had been altered significantly by the OMG in Version 2.3 of the CORBA specification. The Basic Object Adaptor (BOA) interface had been replaced by the Portable Object Adaptor (POA), which, as the name suggests, provides a portable implementation interface for CORBA server-side objects. This filled a gap in the specification left by the BOA that led to vendor-specific extensions and, therefore, CORBA server objects that were dependent on particular vendor Object Request Broker (ORB) implementations. The standard IDL-to-Java mapping, however, took some time to be updated to support the new POA, and JDK 1.2 was released before the new version of the Java mapping. As a stopgap measure, the first version of Java IDL in JDK 1.2 used a server-side object adaptor interface based on an `ImplBase` scheme. By the time JDK 1.4 was introduced in beta in 2001, the POA-compatible version of the IDL-to-Java mapping had been released, and the Java IDL packages, as well as the IDL-to-Java compiler in JDK 1.4, were based on this mapping. The Java 5.0 environment updated its Java IDL support only incrementally. Both JDK 1.4 and Java 5.0 support the CORBA 2.3.1 specification.

Another relatively recent addition to the CORBA specifications is the Interoperable Naming Service (INS) interface, which adds additional utilities and functionality on top of the standard CORBA Naming Service. INS was incorporated into the CORBA 2.3 specification, and support for it in Java IDL was introduced in JDK 1.4.

In this chapter, we discuss only the POA version of Java IDL. If you are using JDK 1.4 or later, you are using a POA-compatible mapping of the CORBA interfaces. If you are using JDK 1.3 or JDK 1.2, you are using the "pre-POA" version of the IDL-to-Java mapping that Sun used prior to adding the POA support. Most of the examples in the chapter would need to be revised somewhat to work in a pre-POA environment. We also limit our discussion to the more recent INS interface rather than the original Naming Service. Again, if you are using JDK 1.4 or later, you have access to the INS interface and the Naming Service provided with the Java IDL. The ORB supports these extended features. If you are using a prior version of the JDK, you have to use the original Naming Service interface, and the examples here will need to be revised to work in your environment.

14.2. The CORBA Architecture

At its core, the CORBA architecture for distributed objects shares many features with the architecture used by Java RMI. A description of a remote object is used to generate a client stub interface and a server skeleton interface for the object. A client application invokes methods on a remote object using the client stub. The method request is transmitted through the underlying infrastructure to the remote host, where the server skeleton for the object is asked to invoke the method on the object itself. Any data resulting from the method call (return values, exceptions) is transmitted back to the client by the communication infrastructure.

But that's where the similarities between CORBA and RMI end. CORBA was designed from the start to be a language-independent distributed object standard, so it is much more extensive and detailed in its specification than RMI is (or needs to be). For the most part, these extra details are required in CORBA because it needs to support languages that have different built-in features. Some languages, like C++, directly support objects, while others, like C, don't. The CORBA standard needs to include a detailed specification of an object model so that non-object-oriented languages can take advantage of CORBA. Java includes built-in support for communicating object interfaces and examining them abstractly (using Java bytecodes and the Java Reflection API). Many other languages don't. So the CORBA specification includes details about a Dynamic Invocation Interface and a Dynamic Skeleton Interface (DSI), which can be implemented in languages that don't have their own facilities for these operations. In languages that do have these capabilities, like Java, there needs to be a mapping between the built-in features and the features as defined by the CORBA specification.

The rest of this section provides an overview of the major components that make up the CORBA architecture: IDL, which is how CORBA interfaces are defined; the ORB and Object Adaptor, which are responsible for handling all interactions between remote objects and the applications that use them; the Naming Service, which is a standard service in CORBA that lets remote clients find remote objects on the network; and the inter-ORB communication protocol, which handles the low-level communication between processes in a CORBA context.

14.2.1. Interface Definition Language

IDL provides the primary way of describing data types in CORBA. IDL is independent of any particular programming language. Mappings, or bindings, from IDL to specific programming languages are defined and standardized as part of the CORBA specification. Standard bindings for C, C++, Smalltalk, Ada, COBOL, Lisp, Python, and Java have been approved by the OMG. [Appendix G](#) contains a complete description of IDL syntax.

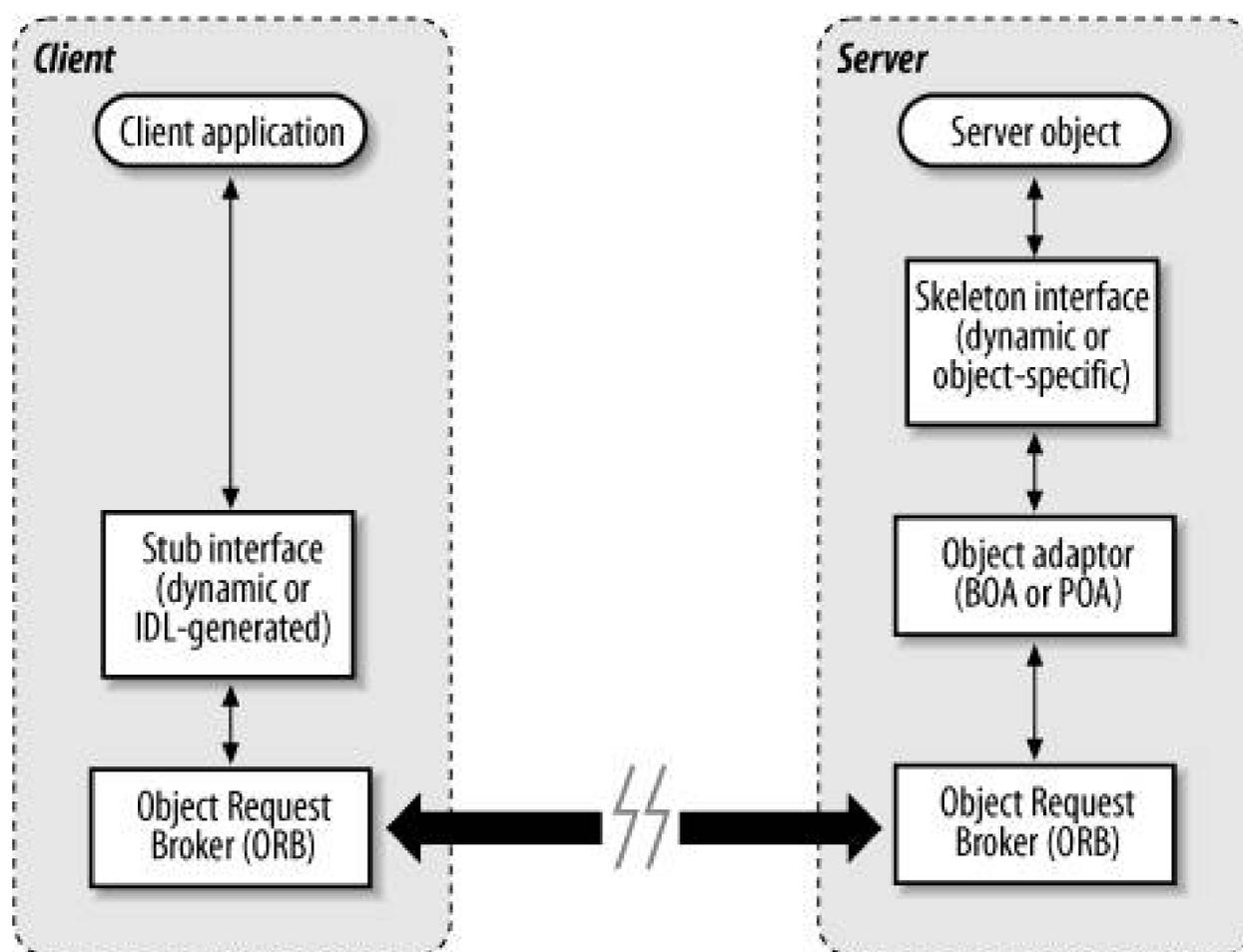
The central CORBA functions, services, and facilities, such as the ORB and the Naming Service, are also specified in IDL. This means that a particular language binding also specifies the bindings for the core CORBA functions to that language. Sun's Java IDL API follows the Java IDL mapping defined by the OMG standards. This allows you to run your CORBA-based Java code in any compliant Java implementation of the CORBA standard, provided you stick to standard elements of the Java binding. Note, however, that Sun's implementation includes some nonstandard elements; they are highlighted

in this chapter as appropriate.

14.2.2. The Object Request Broker and the Object Adaptor

The core of the CORBA architecture is the ORB, as shown in [Figure 14-1](#). Each machine involved in a CORBA application must have an ORB running in order for processes on that machine to interact with CORBA objects running in remote processes. Object clients and servers make requests through their ORBs; the ORB is responsible for making the requests happen or indicating why they can't. The client ORB provides a stub for a remote object. Requests made on the stub are transferred from the client's ORB to the ORB servicing the implementation of the target object. The request is passed on to the implementation through an object adaptor and the object's skeleton interface.

Figure 14-1. Basic CORBA architecture



The skeleton interface is specific to the type of object that is exported remotely through CORBA. Among other things, it provides a wrapper interface that the ORB and object adaptor can use to invoke methods on behalf of the client or as part of the lifecycle management of the object. The object adaptor provides a general facility that "plugs" a server object into a particular CORBA runtime environment. Older versions of the CORBA specification and Java IDL supported a BOA interface, while newer versions (CORBA 2.3 and later, JDK 1.4 and later) support a POA interface. All server objects can use the object adaptor to interact with the core functionality of the ORB, and the ORB in turn can use the object adaptor to pass along client requests and lifecycle notifications to the server object. Typically, an IDL compiler is used to generate the skeleton interface for a particular IDL interface; this generated skeleton interface will include calls to the object adaptor that are supported

by the CORBA environment in use.

14.2.3. The Naming Service

The CORBA Naming Service (sometimes abbreviated to COSNaming, from CORBA Object Services, Naming) provides a directory naming structure for remote objects. The CORBA Naming Service is one of the naming and directory services supported by JNDI, so the concepts used in its API are similar to the general model of `Contexts` and `DirContexts` used in JNDI.

The naming tree always starts with a root node, and subnodes of the object tree can be created by an application. Actual objects are stored by name at the leaves of the tree. [Figure 14-2](#) depicts an example set of objects registered within a Naming Service directory.^[*] The fully qualified name of an object in the directory is the ordered list of all of its parent nodes, starting from the root node and including the leaf name of the object itself. So, the full name of the object labeled "Fred" is "Living thing," "Animal," "Man," "Fred," in that order.

[*] Example adapted from *Categories*, by Aristotle. Please pardon the categorization "man," as opposed to "human." This is the typical translation of Aristotle's original Greek, perhaps because political correctness wasn't in fashion in 350 B.C.

Figure 14-2. A naming directory

Each branch in the directory tree is called a *naming context*, and leaf objects have *bindings* to specific names. Each node in the naming directory is represented by an `org.omg.CosNaming.NamingContext` object. Each `NamingContext` can be asked to find an object within its branch of the tree by asking for the object by name, relative to that particular naming context. You can get a reference to the root context of the naming directory from an ORB using the `resolve_initial_references()` method. Once you have a reference to the root of the naming directory, you can perform lookups of CORBA objects, as well as register your own CORBA objects with the Naming Service. We'll see more concrete details of utilizing the CORBA Naming Service in ["Putting It in the Public Eye"](#) later in this chapter.

14.2.4. Inter-ORB Communication

The CORBA standard includes specifications for inter-ORB communication protocols that can transmit object requests between various ORBs running on the network. The protocols are independent of the particular ORB implementations running at either end of the communication link. An ORB implemented in Java can talk to another ORB implemented in C, as long as they're both compliant with the CORBA standard and use the same CORBA communication protocol. The inter-ORB protocol is responsible for delivering messages between two cooperating ORBs. These messages might be method requests, return types, or error messages. The inter-ORB protocol also deals with differences between the two ORB implementations, like machine-level byte ordering and alignment. As a CORBA application developer, you shouldn't have to deal directly with the low-level communication protocol between ORBs. If you want two ORBs to talk to each other, you need to ensure that they are compatible in terms of CORBA compliance levels (do they support similar levels of the CORBA specification?) and that they both speak a common, standard inter-ORB protocol.

The Internet Inter-ORB Protocol (IIOP) is an inter-ORB protocol based on TCP/IP. TCP/IP is by far the most commonly used network protocol on the Internet, so IIOP is the most commonly used CORBA communication protocol. Other standard CORBA protocols are defined for other network environments, however. The DCE Common Inter-ORB Protocol (DCE-CIOP), for example, allows ORBs to communicate on top of DCE-RPC.

14.3. Creating CORBA Objects

Now that we have a better understanding of the various components of the CORBA architecture, let's walk through the creation of CORBA objects using Java IDL. In order to distribute a Java object over the network using CORBA, you have to define your own CORBA-enabled interface and its implementation. This involves doing the following:

- Writing an interface in the CORBA IDL
- Generating a Java base interface, plus a Java stub and skeleton class, using an IDL-to-Java compiler
- Writing a server-side implementation of the Java base interface

We'll walk through these steps one by one, starting with a quick primer on CORBA IDL, followed by the requirements for creating a Java implementation of an IDL-defined remote object. A full reference for IDL syntax is provided in [Appendix G](#). The following primer covers the basics needed to follow the rest of this chapter.

14.3.1. An IDL Primer

The syntax of both Java and IDL were modeled to some extent on C++, so there are a lot of similarities between the two in terms of syntax. Interfaces in IDL are declared much like classes in C++ and, thus, classes or interfaces in Java. The major differences between IDL and Java are:

- IDL is a declaration language. In IDL, you declare only the names and types for interfaces, data members, methods, method parameters, and the like. Method implementations are created in the implementation language you choose (in this case Java), after you've used an IDL compiler to convert your IDL interface to your target language.
- IDL, like C++, includes nonclass data structure definitions, like structs, unions, and enumerations.
- Method parameters in IDL include modifiers that specify whether they are input, output, or input/output variables. In Java, all primitive data types are passed by value, and all object data types are passed by reference.
- An IDL file can include multiple public interfaces. Only a single public class can be defined in a given Java file (although Java does allow for multiple innerclasses within a single public class definition and multiple nonpublic classes per file).
- Modules, which are similar to Java packages, can be nested within other modules in the same IDL file, and interfaces in multiple distinct modules can be defined in the same IDL file. In Java,

you can define a class only within a single package in a single Java file.

14.3.1.1. Modules

Modules are declared in IDL using the `module` keyword, followed by a name for the module and an opening brace that starts the module scope. Everything defined within the scope of this module (interfaces, constants, other modules) falls within the module and is referenced in other IDL modules using the syntax `modulename::x`. Suppose that you want all your classes to be contained in a module called `corba`, which is part of a larger module called `jent` (an abbreviation of the title of this book). In IDL this is declared as follows:

```
// IDL
module com {
    module oreilly {
        module jent {
            module corba {
                interface NeatExample ...
            };
        };
    };
};
```

If you want to reference the `NeatExample` interface in other IDL files, you use the syntax `com::oreilly::jent::corba::NeatExample`, which may look familiar to readers who have done C++ programming. Java programmers should note the semicolons following the closing braces on the module definitions, which are required in IDL but not in Java. A semicolon is also required after the close of an interface definition.

14.3.1.2. Interfaces

Interfaces declared in IDL are mapped into classes or interfaces in Java. As mentioned earlier, IDL is used only to declare modules, interfaces, and their methods. Methods on IDL interfaces are always left abstract, to be defined in the programming language you use to implement the interfaces.

The declaration of an interface includes an interface header and an interface body. The header specifies the name of the interface and the interfaces it inherits from (if any). Here is an IDL interface header:

```
interface PrintServer : Server { ...
```

This header starts the declaration of an interface called `PrintServer` that inherits all the methods and

data members defined in the `Server` interface. An IDL interface can inherit from multiple interfaces; simply separate the interface names with commas in the inheritance part of the header.

14.3.1.3. Data members and methods

The interface body declares all the data members (or attributes) and methods of an interface. Data members are declared using the `attribute` keyword. At a minimum, the declaration includes a name and a type (see [Appendix G](#) for a complete list of the basic data types available in IDL and the mapping to Java types). The declaration can optionally specify whether the attribute is read-only or not, using the `readonly` keyword. By default, every attribute you declare is readable and writable (for Java, this means that the IDL compiler generates public read and write methods for it). Here is an example declaration for a read-only `string` attribute:

```
readonly attribute string myString;
```

You declare a method by specifying its name, return type, and parameters, at a minimum. You can also optionally declare exceptions the method might raise, the invocation semantics of the method, and the context for the method call (see [Appendix G](#) for more details). Here is the declaration for a simple method that returns a `string`:

```
string parseString(in string buffer);
```

This declares a method called `parseString()` that accepts a single `string` argument and returns a `string` value.

14.3.1.4. A complete IDL example

Now let's tie all these basic elements together. Here's a complete IDL example that declares a module within another module, which itself contains several interfaces:

```
module OS {
  module services {
    interface Server {
      readonly attribute string serverName;
      boolean init(in string sName);
    };
    interface Printable {
      boolean print(in string header);
    };

    interface PrintServer : Server {
      boolean printThis(in Printable p);
    };
  };
};
```

```

    };
};
};

```

The first interface, `Server`, has a single read-only `string` attribute and an `init()` method that accepts a `string` and returns a `boolean`. The `Printable` interface has a single `print()` method that accepts a string header. Finally, the `PrintServer` interface extends the `Server` interface (hence inheriting all its methods and attributes) and adds a `printThis()` method that accepts a `Printable` object and returns a `boolean`. In all cases, we've declared our method arguments as input-only (i.e., pass-by-value), using the `in` keyword.

14.3.2. Turning IDL into Java

Once you've described your remote interfaces in IDL, you need to generate Java classes that act as a starting point for implementing those remote interfaces in Java using an IDL-to-Java compiler. Every standard IDL-to-Java compiler can generate the following Java classes from an IDL interface:

- A *Java interface* with the same name as the IDL interface (e.g., `Server`). This interface includes Java method declarations that are a mapping of the operations declared in the IDL interface. An `interfaceNameOperations` interface (e.g., `ServerOperations`) is generated that contains these method declarations. The object's mapped Java interface extends this interface. This same operations interface is extended by the server-side skeleton interfaces.
- A *helper* class whose name is the name of the IDL interface with `Helper` appended to it (e.g., `ServerHelper`). The primary purpose of this class is to provide a static `narrow()` method that can safely cast CORBA `Object` references to the Java interface type. The helper class also provides other useful static methods, such as `read()` and `write()` methods that allow you to read and write an object of the corresponding type using I/O streams.
- A *holder* class whose name is the name of the IDL interface with `Holder` appended to it (e.g., `ServerHolder`). This class is used when objects with this interface are used as `out` or `inout` arguments in remote CORBA methods. Instead of being passed directly into the remote method the object is wrapped with its holder before being passed. When a remote method has parameters that are declared as `out` or `inout`, the server-side method has to be able to update the argument it is passed and return the updated value. The only way to guarantee this, even for primitive Java data types, is to force `out` and `inout` arguments to be wrapped in Java holder classes, which are filled with the output value of the argument when the method returns. Also, `inout` arguments are initialized with the desired input value before the remote method is called in Java.
- A *client stub* class, called `_interface-nameStub`, that acts as a client-side implementation of the interface. This stub class implements the generated Java interface for the object, but simply knows how to convert client method calls into ORB requests that are forwarded to the actual remote object. The stub class for an interface named `Server` is called `_ServerStub`.
- These classes comprise the "outward-facing" mapping of the CORBA object's interface (the interfaces that clients of the object use directly). The IDL-to-Java compiler can also generate

server-side skeleton classes you can use to implement the server-side implementation of the remote CORBA interface.

- A server skeleton class named *interfaceNamePOA* (e.g., *ServerPOA*), which implements the generated *interfaceNameOperations* interface and extends the POA-related server-side interfaces. The *interfaceNameOperations* interface contains Java mappings of all the methods declared in the IDL definition.

So, in addition to generating a client-side Java mapping of the IDL interface and some helper classes for the Java interface, the IDL-to-Java compiler also creates subclasses that act as an interface between a CORBA client and the ORB and between the server-side implementation and the ORB. [Appendix H](#) provides a complete reference for Sun's *idlj* compiler. We will use this IDL-to-Java tool in the examples in this chapter. Remember, though, that any Java mapping of the CORBA standard should include its own IDL-to-Java compiler to generate these Java classes from the IDL interfaces you write. In addition, the Java that these tools generate should be compliant with the standard IDL mapping for Java, published by the OMG in the CORBA standard. Tools developed before the POA-compliant IDL-to-Java mapping will tend to use some nonstandard server-side object adaptor styles, while those developed after the POA was introduced should generate POA-compliant server skeleton classes. So when using third-party Java-based CORBA tools and ORBs, it's important to understand which version of the core CORBA spec and which IDL-to-Java mapping they support.

14.3.2.1. The delegation server-side model

The scheme described earlier (which will be demonstrated in [Example 14-1](#)) for the generated server-side code for a CORBA object is called an *inheritance-based model*. It depends on the server-side implementation directly extending a generated class (*interfaceNamePOA*). There is another option, called the *delegation model*, available to you in terms of how you "plug" your server code into the CORBA environment.

The delegation model is based on a scheme in which a server-side delegate is generated by the IDL compiler. This delegate extends the generated skeleton class and implements each of the mapped remote methods by delegating the incoming method request to a delegate object. This delegate object needs to implement the *interfaceNameOperations* interface generated by the IDL compiler, but it doesn't have to extend a concrete or abstract base class. This can prove to be useful in cases in which you have a preexisting Java class with its own inheritance scheme and want to "export" this class through CORBA for remote access. Because Java prohibits multiple inheritance of concrete classes, you don't have the option of extending both the existing class and the generated skeleton class. With the delegation model, you can define a simple delegate class that extends the preexisting Java class and implements the *interfaceNameOperations* interface generated by the compiler. You can then "publish" an instance of the original class by creating an instance of the new delegate class and an instance of the generated delegation-based server object and associating the server object with your delegate.

We won't provide full details of the delegation model here, but it's important to realize that this option exists. It may prove useful in some situations. There are options available on the IDL-to-Java compiler to instruct it to generate delegation-based server objects (also referred to as *xies*). These compiler options are documented in [Appendix H](#).

14.3.2.2. A simple server class

The IDL interface shown in [Example 14-1](#) is the IDL equivalent of the `Account` class we defined in [Example 13-1](#). The interface, named `Account`, is declared within some nested modules (`com`, `oreilly`, `jent`, `corba`) and declares methods similar to the `Account` example in [Chapter 13](#). Since this is IDL, the various method argument and return value data types are represented using IDL data types (e.g., `string` instead of `String` and so forth), and method arguments are declared with `in`, `inout`, or `out` modifiers.

Example 14-1. An account interface defined in IDL

```

module com {
    module oreilly {
        module jent {
            module corba {
                // Forward-declare the Account interface,
                // for the typedefs below
                interface Account;

                // Declare some useful typedefs: a list
                // of Accounts and of floats
                typedef sequence<Account> AccountList;
                typedef sequence<float> floatList;

                exception InsufficientFundsException {};

                interface Account {
                    // Get the name of the account owner
                    string getName( );
                    // The account balance
                    float getBalance( );
                    // Withdraw funds from the account
                    void withdraw(in float amt)
                        raises (InsufficientFundsException);
                    // Deposit funds to the account
                    void deposit(in float amt);
                    // Transfer funds from the source account
                    // to this account
                    void transfer(in float amt, in Account src)
                        raises (InsufficientFundsException);
                    // Similar to above, but perform transfers
                    // from a series of source accounts
                    void transferBatch(in floatList amts,
                                      in AccountList srcs)
                        raises (InsufficientFundsException);
                };
            };
        };
    };
};

```

We can run the *idlj* compiler on this IDL interface using the following command line (Windows version):

```
C:\>idlj -fall Account.idl
```

This command creates the five Java classes described in the previous sections: a Java version of the interface, a helper class, a holder class, a client stub, and a server skeleton. The `fall` option tells the compiler to generate both client-side and server-side mapping interfaces (see [Appendix H](#) for complete details on the command-line arguments for *idlj*).

The compiler creates the Java interface shown in [Example 14-2](#), in a file named *Account.java*. This interface doesn't have much meat to it, because all of the method declarations generated in the `AccountOperations` interface are extended by this interface, which is shown in [Example 14-3](#). The interface declaration in the IDL file is mapped directly to the `Account` Java interface declaration, with the interface extending the `AccountOperations` and `org.omg.CORBA.Object` interfaces. The module declarations in the IDL file have been mapped into a `com.oreilly.jent.corba` package statement at the beginning of all the generated Java files. The IDL data types have been converted into the equivalent Java data types, and, since they don't require any special handling in a remote method call, the `in` method parameters in IDL are mapped into regular Java input arguments.

Example 14-2. Java mapping of the Account interface

```
package com.oreilly.jent.corba;

/**
 * com/oreilly/jent/corba/Account.java .
 * Generated by the IDL-to-Java compiler (portable), version "3.1"
 * from src/idl/Account.idl
 * Sunday, September 11, 2005 10:40:07 PM EDT
 */

public interface Account
    extends AccountOperations, org.omg.CORBA.Object,
        org.omg.CORBA.portable.IDLEntity
{
} // interface Account
```

Example 14-3. AccountOperations Java interface

```
package com.oreilly.jent.corba;
```

```

/**
 * com/oreilly/jent/corba/AccountOperations.java .
 * Generated by the IDL-to-Java compiler (portable), version "3.1"
 * from src/idl/Account.idl
 * Sunday, September 11, 2005 10:40:07 PM EDT
 */

public interface AccountOperations
{

    // Get the name of the account owner
    String getName ();

    // The account balance
    float getBalance ();

    // Withdraw funds from the account
    void withdraw (float amt)
        throws com.oreilly.jent.corba.InsufficientFundsException;

    // Deposit funds to the account
    void deposit (float amt);

    // Transfer funds from the source account to this account
    void transfer (float amt, com.oreilly.jent.corba.Account src)
        throws com.oreilly.jent.corba.InsufficientFundsException;

    // source Accounts
    void transferBatch (float[] amts, com.oreilly.jent.corba.Account[] srcs)
        throws com.oreilly.jent.corba.InsufficientFundsException;
} // interface AccountOperations

```

14.3.2.3. The helper class and narrowing references

The compiler also generates a helper class, called `AccountHelper`. We won't provide the full code listing for this generated class here, but it can be found in the downloadable source code examples available for this book, along with all of the other Java interfaces generated by the IDL-to-Java compiler from this IDL interface.

The helper class is a standalone utility class that doesn't extend any other interfaces:

```

abstract public class AccountHelper { . . . }

```

As mentioned earlier, the helper class has static methods that let you read and write `Account` objects to and from CORBA I/O streams:

```
public static
    com.oreilly.jent.corba.Account read (org.omg.CORBA.portable.InputStream
        istream)
public static void write (org.omg.CORBA.portable.OutputStream ostream,
    com.oreilly.jent.corba.Account value)
```

a `type()` method that provides the `TypeCode` for the mapped `Account` class:

```
synchronized public static org.omg.CORBA.TypeCode type ( )
```

and, most importantly, a `narrow()` method that safely narrows a CORBA `org.omg.CORBA.Object` reference into an `Account` reference:

```
public static com.oreilly.jent.corba.Account narrow
    (org.omg.CORBA.Object obj)
```

Object narrowing is CORBA's equivalent to directly casting object references we discuss why narrowing is necessary later in this chapter in "Remote Object References and Narrowing." In the implementation of the `narrow()` method, the helper class converts a CORBA `Object` reference to a reference to a specific type. If the CORBA object can't be narrowed to the requested type (e.g., the passed object reference is a null reference, the object's `TypeCode` doesn't match the `TypeCode` of the narrowed type), then the `narrow()` method throws a `BAD_PARAM` exception.

14.3.2.4. The holder class

The compiler generates a holder class for the `Account` class, which implements the CORBA `Streamable` interface:

```
public final class AccountHolder
    implements org.omg.CORBA.portable.Streamable { . . . }
```

The holder class is a wrapper used when `Account` objects are called for as `out` or `inout` arguments in an IDL method. All holder classes implement the `Streamable` interface from the `org.omg.CORBA.portable` package, which includes implementations of the `_read()` and `_write()` methods of the `Streamable` interface:

```
public void _read (org.omg.CORBA.portable.InputStream i)
```

```
public void _write (org.omg.CORBA.portable.OutputStream o)
```

This allows holders to be transmitted in remote method calls using these `_read()` and `_write()` methods; these methods handle whatever serialization the object needs. This functionality is similar to that provided by Java serialization, but CORBA needs its own scheme because it is independent of any particular language and needs to provide this serialization even if the target language doesn't provide this feature natively.

A holder contains a single instance of the corresponding CORBA object (an `Account`, in this example) as a data member:

```
public com.oreilly.jent.corba.Account value = null;
```

This instance is initialized in the constructor of the holder:

```
public AccountHolder (com.oreilly.jent.corba.Account initialValue)
{
    value = initialValue;
}
```

When a holder object is passed into a remote method call as an `inout` argument, its `_write()` method is invoked. This method takes the object instance contained by the holder class, serializes it, and streams it through the ORB to the remote object server. When the remote method call returns, the holder's `_read()` method is invoked to read the (possibly updated) object from the remote object server, and the holder object replaces its internal value with the updated object.

As an example of using the holder class, let's define another IDL interface that includes a method that uses an `Account` as an `inout` parameter:

```
// IDL
interface AccountManager {
    boolean updateAccount(inout Account account);
};
```

The `AccountManagerOperations` Java interface generated from this IDL interface uses the `AccountHolder` class as the type for the corresponding Java method parameter:

```
// Java
public interface AccountManagerOperations
{
```

```

        boolean updateAccount (AccountHolder account);
    } // interface AccountManagerOperations

```

14.3.2.5. The client stub

The *idlj* compiler generates a Java client stub (`_AccountStub`) for our CORBA interface. The client stub implements the generated `Account` Java interface and acts as a client-side proxy for a remote `Account` object:

```

public class _AccountStub
    extends org.omg.CORBA.portable.ObjectImpl
    implements com.oreilly.jent.corba.Account

```

When a client acquires a reference to a remote `Account` object (through any of the methods we'll describe in "[Finding and Using Remote Objects](#)" later in this chapter), it actually receives an instance of this client stub class. The stub has implementations of all the methods from the interface, as mapped into the `AccountOperations` interface that is the parent of the `Account` interface. The stub class serves as a proxy for a remote server object, and each method implementation in the stub generates a request to the ORB to make a remote method call on the corresponding server-side object. The method arguments are bundled up ("marshaled") and passed along with the request to the ORB. We're not going to go into the details of the stub's method implementations, because you shouldn't have to worry much about them under normal conditions. But it is enlightening to look at an example to see how your remote objects do what they do in detail, using the core CORBA functions. As an example, here is the generated client stub implementation of the `getName()` method from our `Account` interface:

```

// Get the name of the account owner
public String getName ( )
{
    org.omg.CORBA.portable.InputStream $in = null;
    try {
        org.omg.CORBA.portable.OutputStream $out = _request ("getName", true);
        $in = _invoke ($out);
        String $result = $in.read_string ( );
        return $result;
    } catch (org.omg.CORBA.portable.ApplicationException $ex) {
        $in = $ex.getInputStream ( );
        String _id = $ex.getId ( );
        throw new org.omg.CORBA.MARSHAL (_id);
    } catch (org.omg.CORBA.portable.RemarshalException $rm) {
        return getName ( );
    } finally {
        _releaseReply ($in);
    }
}

```

```
} // getName
```

As mentioned, when a Java client gets a reference to a remote `Account` object, it is given one of these stub objects. The client can make method calls on the stub object, and the stub converts these calls into corresponding requests to the ORB to invoke the methods on the remote object and send back the results.

14.3.2.6. Server skeletons

The `idlj` compiler generates server implementation skeleton classes that follow the POA specification. The base class for the server implementation of our `Account` interface is called `AccountPOA`:

```
public abstract class AccountPOA
    extends org.omg.PortableServer.Servant
    implements com.oreilly.jent.corba.AccountOperations,
        org.omg.CORBA.portable.InvokeHandler { . . . }
```

The particulars of the implementation of this skeleton class follow the POA specification, in terms of how method invocations are mapped to actual operations on the server object. This helps to guarantee that your CORBA object implementations will migrate easily between different CORBA ORE implementations.

14.3.3. Writing the Implementation

So, we've written an IDL interface and generated the Java interface and support classes for it, including the client stub and the server skeleton. Now we need to create concrete server-side implementations of all of the methods on our interface. We do this by subclassing from the server-side skeleton class generated by the `idlj` compiler. For our example, we need to subclass `AccountPOA`. We also need to implement the various methods defined on the `Account` IDL interface and mapped through the `idlj` compiler into the `AccountOperations` Java interface. The `AccountImplPOA` class in [Example 14-4](#) shows the implementation of our server object.

As you look through the implementation class, you'll notice that the method implementations are similar to the RMI version of the `Account` server implementation example in [Chapter 13](#). The only real difference is that this `AccountImpl` class extends the generated `AccountPOA` class (and, through it, the generated `AccountOperations` interface and the CORBA `Servant` class), while the RMI server implementation implements the RMI `Account` interface and extends `java.rmi.server.UnicastRemoteObject`. So in the same way that the two remote object schemes are analogous to each other in terms of functionality, the particulars of the server implementations in each case are analogous as well.

Example 14-4. Server implementation of the Account CORBA object

```
public class AccountImplPOA extends AccountPOA {
    // Our current balance
    private float mBalance = 0;
    // Name on account
    private String mName = "";
    // Create a new account with the given name
    public AccountImplPOA(String name) {
        mName = name;
    }

    public String getName( ) {
        return mName;
    }
    public float getBalance( ) {
        return mBalance;
    }
    // Withdraw some funds
    public void withdraw(float amt) throws InsufficientFundsException {
        if (mBalance >= amt) {
            mBalance -= amt;
            // Log transaction...
            System.out.println("--> Withdrew " + amt
                + " from account " + getName( ));
            System.out.println("    New balance: " + getBalance( ));
        }
        else {
            throw new InsufficientFundsException("Withdrawal request of " + amt +
                " exceeds balance of " + mBalance);
        }
    }
    // Deposit some funds
    public void deposit(float amt) {
        mBalance += amt;
        // Log transaction...
        System.out.println("--> Deposited " + amt +
            " into account " + getName( ));
        System.out.println("    New balance: " + getBalance( ));
    }
    // Move some funds from another (remote) account into this one
    public void transfer(float amt, Account src)
        throws InsufficientFundsException {
        if (checkTransfer(src, amt)) {
            src.withdraw(amt);
            this.deposit(amt);
            // Log transaction...
            System.out.println("--> Transferred " + amt +
                " from account " + getName( ));
            System.out.println("    New balance: " + getBalance( ));
        }
        else {

```

```
        throw new InsufficientFundsException("Source account balance is less
                                             " + "than the requested transfer.");
    }
}
// Make several transfers from other (remote) accounts into this one
public void transferBatch(float[] amts, Account[] srcs)
    throws InsufficientFundsException {
    // Iterate through the accounts and the amounts to be
    // transferred from each
    for (int i = 0; i < amts.length; i++) {
        float amt = amts[i];
        Account src = srcs[i];
        // Make the transaction
        this.transfer(amt, src);
    }
}
// Check to see if the transfer is possible, given the source account
private boolean checkTransfer(Account src, float amt) {
    boolean approved = false;
    if (src.getBalance( ) >= amt) {
        approved = true;
    }
    return approved;
}
}
```

14.4. Putting It in the Public Eye

We still need to do some work to make the Java implementation of our IDL interface available to remote clients. We must instantiate one or more instances of our CORBA server implementation and connect them to an ORB on the server so that they can receive remote method requests. Then clients need to somehow obtain remote references to our server objects. A client obtains a remote reference in two fundamental ways: it can get an initial object reference (usually to a CORBA service of some kind, like a Naming Service) using the `ORB.resolve_initial_references()` method and somehow find a reference to the remote object through method calls on the initial object, or it can get a "stringified" reference to the remote object (either an Interoperable Object Reference (IOR) or an Interoperable Naming Service (INS) URL) and use the local ORB to convert it to a live object reference.

For the first case, a remote object needs to be registered in some way with a server-side ORB and probably with a CORBA service of some kind. Connecting your CORBA object to an ORB enables it to accept remote method requests from clients. Registering your object with a CORBA service makes it easier for remote clients to find your object in the first place.

In order for you to register a remote object, you first have to get a reference to an ORB. We'll look at how to do this, then look at registering the remote object with a Naming Service (the most common way to "publish" remote objects in a CORBA context). Finally, we'll look at two ways that clients can use stringified references to access your CORBA object.

14.4.1. Initializing the ORB

Since the ORB is so central to everything in a CORBA environment, the first thing any CORBA process needs to do (whether it's a CORBA server or client process) is get a reference to an ORB that it can use to find other objects, access CORBA services, and handle remote method calls. A CORBA participant initializes its ORB reference by calling one of the static `init()` methods on the `ORB` interface. Each of these methods returns an `ORB` object that can find CORBA objects and services, among other things. The standard `init()` methods provided on an ORB are as follows (Sun's Java IDL supports all of these standard initialization methods):

```
public static ORB ORB.init( )
```

Returns a shared (static) ORB instance, called the "singleton ORB." Each call within the same runtime environment of this version of `init()` returns the same ORB reference. If used within an applet context, the ORB has limited abilities. The singleton ORB is used mainly by IDL-generated classes to perform actions like create `TypeCode` objects that identify the types of CORBA objects.

```
public static ORB ORB.init(String[] args, Properties props)
```

Creates a new ORB using the given arguments and properties, as discussed in the following paragraphs.

```
public static ORB ORB.init(Applet applet, Properties props)
```

Creates a new ORB within an applet context. The applet's codebase and host are used by the ORB as the source of various services, such as the Naming Service.

There are two standard properties defined for an ORB that can be set in the call to `init()`, using either the `String` arguments array or a `Properties` object. These are:

```
org.omg.CORBA.ORBSingletonClass
```

Specifies the class to use for the singleton ORB instance returned by the no-argument version of `init()`.

```
org.omg.CORBA.ORBClass
```

Creates ORB instances when the other versions of `init()` are called.

In both cases, the class name you provide has to implement the `org.omg.CORBA.ORB` interface. You can use these two properties to specify a custom ORB implementation, if needed. You may want to override the default ORB implementation (`com.sun.CORBA.iiop.ORB` in Java IDL) with one of your own that has particular performance characteristics, for example. Or you may be running your CORBA code within an applet and want to ensure that a valid ORB is available no matter what browser version your applet encounters.

Sun's Java IDL also adds two nonstandard properties: `ORBInitialHost` and `ORBInitialPort`. By default, each `ORB.init()` method initializes an ORB that looks for its services (naming service, for example) locally. Java IDL adds these two nonstandard properties to allow your local ORB to defer its services (naming, trading, and so on) to a remote ORB running on a given host and listening on a given port. Be careful before you decide to depend on these properties in your application or applet. They are honored only within Sun's Java IDL implementation of the CORBA standard. If you want your CORBA application to be portable to any implementation of the standard IDL-to-Java binding and you want to use a remote Naming Service, you should stick to using a stringified reference to the remote object, obtained through a secondary communication channel (as we'll discuss shortly).

Any of these properties can be specified within a `Properties` object, as a command-line option to a Java application or in a system properties file. As an example, if you want to specify a different host to use for finding services like the Naming Service, one way to do this is to specify the host explicitly in the code that initializes the ORB, using a `Properties` object:

```
Properties orbProps = new Properties( );
orbProps.put("org.omg.CORBA.ORBInitialHost", "remote.orb.com");
ORB myOrb = ORB.init((String[])null, orbProps);
```

Alternately, you can take command-line arguments passed into your Java code and pass them to the `ORB.init()` method to be parsed. Say we have a class named `InitRemote` with a main method implemented as follows:

```
public class InitRemote {
    public static void main(String[] argv) {
        try {
            ORB myOrb = ORB.init(argv, null);
            ...
        }
    }
}
```

In this case, we can specify any ORB properties on the command line using specific argument names. Notice that when specifying these, you should omit the `org.omg.CORBA` package prefix on the property names:

```
orbhost% java InitRemote -ORBInitialHost remote.orb.com
```

Note that you can use the second `ORB.init()` method with both a `String` arguments array and a `Properties` list specified, even though the examples here haven't shown that.

14.4.2. Registering with a Naming Service

One way to make a server object available to remote clients is to register it with a local CORBA Naming Service under a specific name. A remote client can then get a reference to the root `NamingContext` for the Naming Service and look up the server object by name.

Here are the basic steps that create and register a CORBA object with the ORB:

1. Initialize an ORB reference.
2. Create an instance of your CORBA server implementation, and get a reference from it that can be registered with the Naming Service.
3. Get a reference to the root `NamingContext` for the Naming Service from the ORB.
4. Construct a name for the object that reflects where you want it to sit in the Naming Service hierarchy.

5. Bind your object to that name in the Naming Service, using the root `NamingContext`.

[Example 14-5](#) shows the code listing for `AccountInitPOA`. Since we're dealing with an ORB using the POA, and the POA is acting as the mediator between our server object and the ORB, we need to use the POA to obtain a reference to our server object that can be used for registration with the Naming Service (among other things).

The `main()` method creates an instance of our `AccountImplPrePOA` implementation and then registers the object with the Naming Service. We get a reference to the POA from the ORB using the `resolve_initial_references()` method (this time using the standard name for the POA, "RootPOA"). We then make sure that the POA is active by getting its POA manager and calling its `activate()` method every POA has a `POAManager` that maintains the runtime state of the POA. Finally, we ask the POA for a usable reference to our server object by calling its `servant_to_reference()` method and converting the resulting `Object` to an `Account` reference.

Example 14-5. AccountInitPOA.java

```
import org.omg.CORBA.*;
import org.omg.CosNaming.*;
import org.omg.PortableServer.POA;

public class AccountInitPOA {
    public static void main(String[] args) {
        try {
            // Initialize an ORB reference
            ORB myORB = ORB.init(args, null);

            // Create an instance of an Account server implementation
            AccountImplPOA acct = new AccountImplPOA(args[0]);

            // Get the root Portable Object Adapter (POA)
            POA rootPOA = (POA)myORB.resolve_initial_references("RootPOA");

            // Activate the POA manager
            rootPOA.the_POAManager( ).activate( );

            // Get an object reference from the implementation
            org.omg.CORBA.Object obj = rootPOA.servant_to_reference(acct);
            Account acctRef = (Account)AccountHelper.narrow(obj);

            // Get the root name context (use the INS interface, so that we can use
            // the simpler name construction process)
            org.omg.CORBA.Object objRef =
                myORB.resolve_initial_references("NameService");
            NamingContextExt nc = NamingContextExtHelper.narrow(objRef);

            // Register the local object with the Name Service.
            // Use the Interoperable Naming Service
            interface to simplify matters
            // and to support URL-formatted names (e.g. "JohnS",
```

```

// "corbaname://orbhost.com#/JohnS", etc.).
NameComponent[] name = nc.to_name(args[0]);
nc.rebind(name, acctRef);

System.out.println("Registered account under name " + args[0]);

// Go into a wait state, waiting for clients to connect
myORB.run( );
}
catch (Exception e) {
    System.out.println(
        "An error occurred while initializing server object:");
    e.printStackTrace( );
}
}
}

```

Once we have our usable object reference, we invoke the `INSNamingContextExt` interface to bind our server object to a name in the Naming Service. We'll get into the details of the INS interface shortly.

With the object registered, we then invoke the `run()` method on the ORB. Calling this method causes the current thread to wait until the ORB shuts down, which leaves our server object active to respond to client requests delivered to it from the ORB.

Before running this utility object to create and register our CORBA object, we need to start a Naming Service someplace. A Naming Service daemon listens for Naming Service requests on a specific port and provides access to the named object directory it manages. In Java IDL, the Naming Service is started using the `tnameserv` command:

```
objhost% tnameserv &
```

With that done, we can run our initialization method to register our server object with the ORB:

```
objhost% java com.oreilly.jent.corba.AccountInitPOA JimF
Registered account under name JimF
```

Now that we've seen the basic steps involved in creating and registering an object with a CORBA Naming Service, let's look in more detail at binding objects and new contexts within the Naming Service.

14.4.3. Adding Objects to a Naming Context

As was discussed earlier, a CORBA Naming Service maintains a hierarchy of named objects that can be looked up by remote clients. Initially, a CORBA naming directory is empty, with only its root `NamingContext` and no objects. The `bind()` method on a `NamingContext` object binds a server object to a name within the context. The `bind_new_context()` method creates new subcontexts within a given `NamingContext`. Using a file directory analogy, calling `bind_new_context()` on a `NamingContext` object is like making a new subdirectory from the current directory, while calling `bind()` puts a new file into the current directory.

The Java IDL mapping uses arrays of `NameComponent` objects to represent the names of subcontexts within a naming directory. Each `NameComponent` represents a component of the path to the named object. A `NameComponent` contains `id` and `kind` string fields that serve to label the component in the path. Only the `id` field is significant in determining name uniqueness. So a `NameComponent` with an `id` set to "student" and `kind` set to an empty string conflicts with a `NameComponent` with an "student" and a `kind` of "doctoral", if both `NameComponent` objects are relative to the same subcontext. The `NameComponent` class has a constructor that takes the `id` and `kind` values as arguments. Here's how to create a single `NameComponent`:

```
NameComponent comp1 = new NameComponent("student", "doctoral");
```

A complete name path can be composed as an array of these objects:

```
NameComponent path[] = { comp1, comp2, ... };
```

The `bind()` method takes two arguments: an array of `NameComponent` objects as the relative name for the object you're putting into the Naming Service and the server object itself. If you're binding a server object using the root context of the Naming Service, the name is also the absolute name of the object in the overall naming directory. If an object is already bound to the name, you can use the `rebind()` method with the same arguments, causing the existing object bound to that name to be replaced by the new object. Note that since the Naming Service is a CORBA service that can be accessed remotely by other CORBA clients, the objects it contains need to be exportable to these remote clients. This means that only `org.omg.CORBA.Object` references can be bound to names within a `NamingContext`.

The following code snippet binds a few of our `Account` objects to names within the root context of a Naming Service:

```
// Get the root naming context
ORB myORB = ORB.init(...);
org.omg.CORBA.Object ref =
    myORB.resolve_initial_references("NameService");
NamingContext rootNC = NamingContextHelper.narrow(ref);

// Create a few Accounts
Account acct1 = new AccountImplPOA("JohnSmith");
Account acct2 = new AccountImplPOA("MaryJones");
```

```

// Bind them to names in the Naming Service
NameComponent name1 = new NameComponent("Smith,J", "");
NameComponent path1[] = { name1 };
NameComponent name2 = new NameComponent("Jones,M", "");
NameComponent path2[] = { name2 };
rootNC.bind(path1, ref1);
rootNC.bind(path2, ref2);

```

Before you can bind an object to a name with multiple components, all the subcontexts (subdirectories) have to be created using the `bind_new_context()` method on a `NamingContext`. The `bind_new_context()` method takes an array of `NameComponent` objects as the relative path of the new context and a reference to the `NamingContext` object to bind to that location in the overall directory. A new `NamingContext` object can be created from an existing one by calling its `new_context()` method. If a context already exists at the target name, you can use the `rebind_context()` method to replace the existing context with a new one. This is useful for emptying out an entire subcontext without removing each object individually.

Here is an example that binds some `Account` objects within various subcontexts representing different branches of a bank, using the standard Naming Service interfaces; `NameComponent` and `NamingContext` (both from the `org.omg.CosNaming` package):

```

// Initialize the ORB and get the POA and root naming context, as before
ORB myORB = ORB.init(...);
POA rootPOA = ...;
NamingContext rootNC = ...;

// Create the components to the subcontext paths
NameComponent branchComp = new NameComponent("bankBranches", "");
NameComponent cambridgeComp = new NameComponent("Cambridge", "");
NameComponent bostonComp = new NameComponent("Boston", "");

// Create a new context, bind it to the path "bankBranches"
NamingContext ctx = rootNC.new_context( );
NameComponent path[] = { branchComp };
rootNC.bind_context(path, ctx)

// Create another context, bind it to the path "bankBranches, Cambridge"
NamingContext cambridgeCtx = rootNC.new_context( );
path = { branchComp, cambridgeComp };
rootNC.bind_context(path, cambridgeCtx)

// Create another context, bind it to the path "bankBranches, Boston"
NamingContext bostonCtx = rootNC.new_context( );
path = { branchComp, bostonComp };
rootNC.bind_context(path, bostonCtx);

// Now we can bind Accounts to a name within any of the new subcontexts.
// Create a few Account server objects, and get usable client references
// from the POA.

```

```

Account johnAcct = new AccountImplPOA("JohnSmith");
Account johnRef =
    AccountHelper.narrow(rootPOA.servant_to_reference(johnAcct));
Account maryAcct = new AccountImplPOA("MarkJones");
Account maryRef =
    AccountHelper.narrow(rootPOA.servant_to_reference(maryAcct));

// Bind each Account to a name in the appropriate branch path. Assume
// that John has his account out of the Cambridge branch, Mary has hers
// out of the Boston branch.
NameComponent johnComp = new NameComponent("Smith,J", "");
NameComponent johnPath[] = { branchComp, cambridgeComp, johnComp };
rootNC.bind(johnPath, johnRef);
NameComponent maryComp = new NameComponent("Jones,M", "");
NameComponent maryPath[] = { branchComp, bostonComp, maryComp };
rootNC.bind(maryPath, maryRef);

```

If you try to bind an object or a subcontext to a name within a context that hasn't been created yet, an `org.omg.CosNaming.NamingContextPackage.NotFound` exception is thrown.

Note that names used in the `bind()` or `rebind()` methods are relative to the `NamingContext` object that they're called on. This means we can bind Mary's `Account` object in the previous example to the same absolute name within the directory by replacing the last two lines of the example with the following:

```

NameComponent maryRelativePath[] = { maryComp };
bostonCtx.bind(maryRelativePath, maryRef);

```

The `bostonCtx` context is bound to the `{ "bankBranches", "Boston" }` subdirectory, so binding an object to the name `{ "Jones,M" }` within this context is equivalent to binding it to the full path `{ "bankBranches", "Boston", "Jones,M" }` from the root context. You can use similar shorthand when binding new contexts within a directory. In other words, you can bind a context to a relative name within a subcontext, instead of an absolute name within the root context.

14.4.3.1. A slightly simpler approach: The Interoperable Naming Service

The Interoperable Naming Service (INS) interfaces make both the binding and lookup of objects in a Naming Service a bit easier. In our previous examples, we bound objects to specific locations in the naming hierarchy by first creating the subcontexts (subdirectories) we needed, then binding the remote objects to names within those contexts. In each step, we needed to construct fully qualified names for both the contexts and the objects we were binding. Prior to the introduction of the INS, the only way to do this was the way shown in earlier examples: manually construct an array of `NameComponent`s representing the intended path to the object. Using the INS interfaces and their support for new types of stringified object references and URLs, it's possible to create new contexts and bind objects in a slightly simpler way. We can convert stringified object references directly to

`NameComponent` arrays using the `to_name()` method on the `org.omg.CosNaming.NamingContextExt` interface. For example, assuming that we've created the necessary subcontexts already, we could bind John Smith's account to the same path as before by specifying the full path as a string and converting it using `to_name()`:

```
NamingContextExt rootNC = ...;  
String johnURL = "bankBranches/Cambridge/Smith,J";  
NameComponent johnPath[] = rootNC.to_name(johnURL);  
rootNC.rebind(johnPath, johnRef);
```

[← PREV](#)

14.5. Finding and Using Remote Objects

Now that we've seen how to implement a CORBA remote object and register it with an ORB, we can turn our attention to the heart of the matter: client applications that want to use the object. As we said earlier, every CORBA process needs a reference to an ORB. Once a client application has access to an ORB, next it must find remote objects to interact with. But before we can discuss finding remote objects, we need to talk a bit about what remote object references look like in the CORBA environment.

The whole point of CORBA is to be able to distribute objects across the network and use them from any point. In order for a local process to make requests of a remote object, it needs to have some kind of reference to that remote object. This object reference needs to contain enough information for the local ORB to find the ORB serving the target object and send the request to the remote ORB using an agreed-upon protocol.

14.5.1. Remote Object References and Narrowing

In most situations, a CORBA client has a reference to a remote object in the form of an object stub. The stub encapsulates the actual object reference, providing what seems like a direct interface to the remote object within the local environment. If the client is implemented in C++, Java, or some other object-oriented language, the object stub is a native object in that language. Other nonobject languages represent remote object references in whatever way is dictated in the CORBA language binding for that language.

CORBA includes its own root object class, because some object-programming languages may have different inheritance structures. In the Java binding for CORBA, all CORBA object references (local or remote) implement the `org.omg.CORBA.Object` interface. So, when a client of a remote CORBA object receives a stub for the object, it actually gets an `org.omg.CORBA.Object` that serves as a proxy for the remote object. The `org.omg.CORBA.portable.ObjectImpl` class provides default client-side implementations for the methods defined on `org.omg.CORBA.Object`. Java stubs (generated from the IDL-to-Java compiler, as discussed earlier) for CORBA objects are actually subclassed from the `ObjectImpl` class. Internally, `ObjectImpl` deals with delegating client requests on the object stub to the proper target object, whether it's a local or remote object. `ObjectImpl` implements the `org.omg.CORBA.Object` interface and inherently extends the `java.lang.Object` class, so it provides a joining point between the CORBA and Java object environments.^[*]

[*] In pre-POA versions of Java IDL (JDK 1.2 and 1.3), the server skeletons generated for CORBA interfaces were also subclassed from `ObjectImpl`. In POA-compliant versions, the skeletons are subclassed from `org.omg.PortableServer.Servant`, which provides an alternative implementation of the `org.omg.CORBA.Object` interface, as well as a number of methods specific to servant objects.

A reference to an `org.omg.CORBA.Object` instance that is connected to a remote object is actually all a client needs to invoke methods on a remote object. Using the Dynamic Invocation Interface defined by the CORBA standard, you can create method requests and send them to the remote object directly through the generic `Object` interface, as we'll discuss later in this chapter. If your client has

the actual Java interface for the remote object available at compile time, however, you probably want to convert the `Object` reference into a reference of that type, so you can use the interface to call remote methods directly.

Converting an `org.omg.CORBA.Object` to a specific remote interface is done by narrowing the object reference to the corresponding interface type, using type-specific helper classes. We've already seen how the Java IDL compiler, *idlj*, creates a helper class from an IDL interface (e.g., `AccountHelper`). The helper class includes a `narrow()` method that safely converts an `org.omg.CORBA.Object` reference to a reference of the given type. If the object reference you pass into the `narrow()` method is not the type the helper expects, an `org.omg.CORBA.BAD_PARAM` exception is thrown. This is a `RuntimeException`, so it doesn't have to be caught by your code, unless you're trying to test the type of a CORBA reference for some reason.

This `narrow()` operation highlights one of the key differences between RMI and CORBA. In the Java environment, class bytecodes are portable, and all remote object types are objects that can be specified by their full class names. An RMI client can automatically download the bytecodes for a remote stub from the object server, if the class for the stub can't be found locally (see [Chapter 13](#) for more details on the mechanics of remote class loading). CORBA is a language-independent remote object scheme, so there is no portable way to specify a remote object's native type when a client obtains a stub reference. In Java, you could do this using the same remote classloading scheme as RMI, but there isn't a similar scheme available in C or C++, for example. As a result, the stub reference is initially represented by a basic object type that knows how to forward method requests to its server object, but doesn't necessarily satisfy the mapped interface for the remote object type the client is expecting. The client application is forced to "cast" this stub to the correct local type, using the appropriate `narrow()` method. In the Java mapping of IDL, this means calling the `narrow()` method on the corresponding helper class. The `narrow()` method converts the reference, making a type-specific stub interface that also includes the remote object reference information needed to forward method requests to the actual object implementation.

14.5.2. Accessing Remote Objects

With that background material out of the way, let's discuss actually finding remote object references. There are many ways that an object reference can find its way through the ORB into a client application, but they all boil down to one of these methods:

- Getting an initial reference directly from the ORB
- Getting an object reference through a method call on another remote object reference
- Using a stringified object reference obtained through a secondary channel and converting it to a live object reference

14.5.3. ORB Initial Object References

In addition to providing core object communication services, an ORB can also provide additional services, such as a Naming Service, a Trading Service, or a Security Service. These services are represented as CORBA objects and are typically available through the ORB directly, based on how it

is configured. The ORB interface provides the `resolve_initial_references()` method for obtaining references to the objects that represent these services. Each CORBA service is represented by one or more object interfaces, and these objects can be asked for using standard names. As we saw earlier when we registered CORBA objects, the standard name for the Naming Service is `"NameService"`, and the object representing access to the Naming Service is a `NamingContext` or `NamingContextExt` object. These represent the root of the naming hierarchy.

Once you've initialized your ORB reference (as described in ["Initializing the ORB"](#) earlier in this chapter), you can ask the ORB for a list of the names of the initial objects it has available, using the `list_initial_references()` method:

```
String names[] = myORB.list_initial_references( );
```

This method returns an array of `String` objects that contains the names of all initial objects in the ORB. These names can then be used to get references to the objects through the `resolve_initial_references()` method.

Here's how we used `resolve_initial_references()` to obtain a reference to the Naming Service in our earlier examples:

```
ORB myORB = ORB.init(...);
org.omg.CORBA.Object nameRef =
    myORB.resolve_initial_references("NameService");
```

Although the `list_initial_references()` and `resolve_initial_references()` methods are a standard element of the ORB interface, how a particular ORB implements these initial object references is not standardized. Sun's Java IDL implementation, for example, stores an ORB's initial object references as root objects in its internal Naming Service.

14.5.4. Getting Objects from Other Remote Objects

In addition to getting remote objects directly from an ORB reference, a client can obtain remote objects from other remote objects. The most common variation on this approach is to get a reference to a Naming Service object and then look up objects in the naming directory by name. Another variation (that we won't cover in detail in this section) is to obtain an application-specific object reference, either directly from the ORB or through the Naming Service, and use this initial reference to request other objects. An object used in this way in a distributed application is sometimes called a *factory* object.

14.5.4.1. Naming Service lookups

Once you have a `NamingContext` reference that points to a position (either the root or a subdirectory)

in a Naming Service, you can look up objects within the context by passing names to its `resolve()` method. As before, when we were binding objects, a name is represented by an ordered array of `NameComponent` objects. Each `NameComponent` (both the `id` field and the `kind` field) must exactly match the path to an object within the context in order to successfully find the object. If an object is not found at a specified name, an `org.omg.CosNaming.NamingContextPackage.NotFound` exception is thrown.

So, if a client wants to find the account for Mary Jones, which we stored in the last binding example, it needs to do the following:

```
// Set up the full path to the account
NameComponent comp1 = new NameComponent("bankBranches", "");
NameComponent comp2 = new NameComponent("Cambridge", "");
NameComponent acctName = new NameComponent("Jones,M", "");
NameComponent acctPath[] = { comp1, comp2, acctName };
// Find the object in the directory
org.omg.CORBA.Object acctRef = rootNC.resolve(acctPath);
Account acct = AccountHelper.narrow(acctRef);
```

We're assuming that `rootNC` is a reference to the root context of the Naming Service holding the `Account` object references. Note the use of the `narrow()` method on `AccountHelper` to "cast" the generic object reference to an `Account` object.

In addition to finding references to objects (leaf nodes) in the Naming Service, you can also use the `resolve()` method on a `NamingContext` to get a reference to a subcontext (a subdirectory). Just use the path to the context itself and `narrow()` it to a `NamingContext` reference:

```
NameComponent cambridgePath[] = { comp1, comp2 };
org.omg.CORBA.Object ncRef = rootNC.resolve(cambridgePath);
NamingContext cambridgeContext = NamingContextHelper.narrow(ncRef);
```

14.5.5. Object URLs

As we've seen, Sun's implementation of Java IDL provides a nonstandard way to initialize an ORB to reference a remote Naming Service, so that one of the ORB's initial references is to the root context of the remote Naming Service. But what do you do if you want an object from a remote Naming Service and your Java IDL implementation doesn't provide a way to directly initialize a reference to the remote service? Or, worse yet, what if the object that you want isn't stored in a Naming Service or available through any other CORBA service? How can your client get a reference to the object?

The CORBA standard comes to the rescue again. CORBA defines several ways to represent *stringified object references*, or remote object URLs. In other words, you can fully specify a specific CORBA server object using a URL syntax, which is very similar to the URL syntax used to specify Java RMI objects in an RMI registry (see [Chapter 13](#) for more details on RMI registry URLs). These object URLs are called "stringified object references" in the CORBA specifications, because that's exactly what

they are: strings that can be sent around the network (in email and in configuration files, for example) and used by clients to obtain a living reference to the object that the URL specifies. So these object URLs are called "stringified object references" in the same way that you could call HTML document URLs "stringified document references."

In these URL schemes, the objects being referenced need to be hosted behind an ORB someplace on the network, and they must have a server process running that is listening for remote method requests for that object. In some cases, the objects and their request servers can just be connected directly to the ORB; in other cases, they have to be bound to a name in a Naming Service, with the Naming Service acting as the method request mediator.

Three basic forms of object URLs are defined in the CORBA standards:

Interoperable Object References (IORs)

These URLs are based on a syntax for representing a remote object reference in the form of a printable, but not human-readable, string of characters. This stringified object reference includes enough information for a remote CORBA client to locate the object's home ORB and convert the string to a runtime stub reference to the object. Two methods on the ORB interface, `object_to_string()` and `string_to_object()`, let you convert a CORBA object reference to an IOR and back into an object reference. An object doesn't need to be bound to a name in a Naming Service in order to have an IOR; it simply needs to be exported through an ORB using an object adaptor, like the POA.

corbaloc URLs

These URLs are a human-readable URL format for object references. They are similar in purpose to IORs: an object needs to be exported through an ORB for remote access in order to have a corbaloc URL associated with it. So corbaloc URLs point to a single object accessible directly through an ORB, without the help of a service of some kind. The same restrictions apply for making servants accessible through corbaloc URLs as for IORs: the servant object needs to be connected to an ORB through some kind of object adaptor, like the POA.

corbaname URLs

These URLs are extensions to the corbaloc syntax to allow for references to objects that are registered with a CORBA Naming Service. The URL format includes additional information that specifies the object's location (name) in the Naming Service.

[Example 14-6](#) shows `AccountInit`, a server that creates an instance of our `AccountImplPOA` and makes it available for clients to access. In its `main()` method, it initializes the ORB, then creates an instance of our `Account` servant. We're assuming that a POA-compliant version of Java IDL is being used, so the next thing to do is get a reference to the root POA (using `resolve_initial_references()`) and activate the POA manager, then ask the POA to activate our servant object. By activating our servant object this way, we're guaranteed that any "direct" IIOP requests (through either an IOR or a corbaloc URL) will be accepted immediately. Note that in order for the generated IOR to be acceptable to another ORB, both your ORB and the remote ORB have to be using the same inter-ORB communication protocol (IIOP, DCE-CIOP, and so on). In this example, we'll assume that our client and object server are both running IIOP.

Example 14-6. Initializing an account/getting stringified references

```

import org.omg.CORBA.*;
import org.omg.CosNaming.*;
import org.omg.PortableServer.POA;
import java.net.InetAddress;
public class AccountInit {

    public static void main(String[] args) {
        try {
            // Initialize an ORB reference
            ORB myORB = ORB.init(args, null);

            // Create an instance of an Account server implementation
            AccountImplPOA acct = new AccountImplPOA(args[0]);

            // Get the root Portable Object Adapter (POA)
            POA rootPOA = (POA)myORB.resolve_initial_references("RootPOA");
            // Activate the POA manager
            rootPOA.the_POAManager( ).activate( );
            // Activate our servant, so that corbaloc and IOR requests will work
            // immediately
            rootPOA.activate_object(acct);

            // Get an object reference from the implementation
            org.omg.CORBA.Object obj = rootPOA.servant_to_reference(acct);
            Account acctRef = (Account)AccountHelper.narrow(obj);

            // Get the root name context (use the INS interface, so that we can use
            // the simpler name construction process)
            org.omg.CORBA.Object objRef =
                myORB.resolve_initial_references("NameService");
            NamingContextExt nc = NamingContextExtHelper.narrow(objRef);

            // Register the local object with the Name Service.
            // Use the Interoperable Naming Service interface to simplify matters,
            // and to support URL-formatted names (e.g. "JohnS",
            // "corbaname://orbhost.com#JohnS", etc.).
            NameComponent[] name = nc.to_name(args[0]);
            nc.rebind(name, acctRef);

            System.out.println("Registered account under name " + args[0]);
            System.out.println("New account created and registered.  URLs are: ");
            System.out.println("\nIOR");
            System.out.println("\t" + myORB.object_to_string(acctRef));
            System.out.println("\ncorbaname");
            System.out.println("\tcorbaname:iiop:" +
                InetAddress.getLocalHost( ).getHostName( ) +
                "#" + args[0]);
        }
    }
}

```

```

        // Go into a wait state, waiting for clients to connect
        myORB.run( );
    }
    catch (Exception e) {
        System.out.println(
            "An error occurred while initializing server object:");
        e.printStackTrace( );
    }
}
}

```

Next, we ask the POA to generate an `Account` reference for us from the servant object; we use this reference to register the `Account` with the Naming Service. The request to convert the reference to an `Account` reference is necessary because the object implementation class doesn't directly extend the mapped object interface.

Now that the server object is both connected to the ORB through the POA and registered with a Naming Service, we can generate various URLs for this object that clients can use to get direct references to the object. First we get the IOR of the object, using the `object_to_string()` method on the ORB. Then we create the corbaname URL for the object, using the local hostname, the port number of the ORB, and the name the user passed in on the command line and used to register the object with the Naming Service. Each of these is printed to the console, and we can distribute these URLs to the clients that need them. Note that we aren't creating a corbaloc URL for our object. This is because the server-side creation of corbaloc URLs is not standardized; CORBA vendors currently provide nonstandard ways to generate these in their implementations.

To demonstrate using these URLs, [Example 14-7](#) shows a simple client for our `Account` server. This client uses a command-line interface and allows the user to perform transactions on a given `Account`, by specifying the `Account` to use (using its object URL), a transaction type ("deposit", "withdrawal", or "balance"), and an optional amount (for deposits and withdrawals). After doing the standard ORB initialization, the client first checks the name of the `Account` object (the first command-line argument). If the name of the `Account` is an object URL (IOR, corbaloc, or corbaname), then the client simply uses the `string_to_object()` method on the ORB to get a reference to the specified `Account` object and narrows the returned `Object` to an `Account` reference using the `AccountHelper`. If some other string is passed in, the client assumes the string is a name to be used to do a lookup in the Naming Service accessible from the ORB and performs the lookup using the standard Naming Service operations (described earlier). Once the client has the reference to the remote `Account` object, the rest is easy just parse the remaining command-line options given by the user and perform the requested transaction on the account.

Example 14-7. A client for the account servant

```

import org.omg.CORBA.*;
import org.omg.CORBA.ORBPackage.*;
import org.omg.CosNaming.*;
import java.util.*;

```

```

public class AccountClient {
    public static void main(String args[]) {
        // Initialize the ORB
        ORB orb = ORB.init(args, null);
        org.omg.CORBA.Object ref = null;
        // The object name passed in on the command line
        String name = args[0];
        Account acct = null;
        // See if a stringified object reference/URL was provided
        if (name.startsWith("corbaname") || name.startsWith("corbaloc") ||
            name.startsWith("IOR")) {
            System.out.println("Attempting to lookup " + args[0]);
            ref = orb.string_to_object(args[0]);
            acct = AccountHelper.narrow(ref);
        }
        // Otherwise, do a traditional Naming Service lookup using the
        // services being referenced by our local ORB
        else {
            try {
                ref = orb.resolve_initial_references("NameService");
            }
            catch (InvalidName invN) {
                System.out.println("Couldn't locate a Naming Service");
                System.exit(1);
            }
            NamingContext nameContext = NamingContextHelper.narrow(ref);
            NameComponent comp = new NameComponent(args[0], "");
            NameComponent path[] = {comp};
            try {
                ref = nameContext.resolve(path);
                System.out.println("ref = " + ref);
                acct = AccountHelper.narrow(ref);
            }
            catch (Exception e) {
                System.out.println("Error resolving name against Naming Service");
                e.printStackTrace();
            }
        }

        if (acct != null) {
            // We managed to get a reference to the named account; now check the
            // requested transaction from the command line
            String action = args[1];
            float amt = 0.0f;
            if (action.equals("deposit") || action.equals("withdrawal")) {
                amt = Float.parseFloat(args[2]);
            }
            System.out.println("Got account, performing transaction...");
            try {
                // Did user ask to do a deposit?
                if (action.equals("deposit")) {
                    acct.deposit(amt);
                }
            }
        }
    }
}

```

```

        System.out.println("Deposited " + amt + " to account.");
        System.out.println("New balance = " + acct.getBalance( ));
    }
    // Did user ask to do a withdrawal?
    else if (action.equals("withdrawal")) {
        acct.withdraw(amt);
        System.out.println("Withdrew " + amt + " from account.");
        System.out.println("New balance = " + acct.getBalance( ));
    }
    // Assume a balance inquiry if no deposit or withdrawal was given
    else {
        System.out.println(
            "Current account balance = " + acct.getBalance( ));
    }
}
catch (InsufficientFundsException ife) {
    System.out.println("Insufficient funds for transaction.");
    System.out.println("Current account balance = " + acct.getBalance( ));
}
catch (Exception e) {
    System.out.println("Error occurred while performing transaction:");
    e.printStackTrace( );
}
}
else {
    System.out.println("Null account returned.");
    System.exit(1);
}
}
}
}

```

To run this example, we first have to have a Naming Service running somewhere, then initialize an `Account` object for the client to use:

```

orbhost% tnameserv &
Initial Naming Context:
IOR:0000000000000002b49444c3a6f6d672e6f72672f436f734e616d696e672f4e616d696
e67436f6e746578744578743a312e300000000000100000000000007c00010200000000
0a3132372e302e302e3100041a00000035afabcb0000000020363e1ad7000000010000000
0000000010000000d544e616d65536572766963650000000000000040000000003000000
0000000100000001000000200000000000010001000000020501000100010020000101090
000000100010100
TransientNameServer: setting port for initial object references to: 900
Ready.
orbhost% java com.oreilly.jent.corba.AccountInit JimF
Registered account under name JimF
New account created and registered. URLs are:

IOR

```

```
IOR:000000000000002349444c3a6f7265696c6c792f6a656e742f636f7262612f4163636
f756e743a312e3000000000001000000000000068000102000000000a3132372e302e30
2e310004660000021afabcb000000020363f810b0000001000000000000000000000
40000000030000000000001000000100000020000000000010001000000205010001
0001002000101090000000100010100
```

```
corbaname
    corbaname:iiop:orbhost#JimF
```

To perform transactions on this newly created and registered `Account` object, we run the `AccountClient`, passing in one of these object URLs as the "name" of the `Account`. Here we'll use the corbaname URL.

```
myclient% java com.oreilly.jent.corba.AccountClient corbaname:iiop
:orbhost#JimF
Attempting to lookup corbaname:iiop:orbhost#JimF
Got account, performing transaction...
Deposited 1000.0 to account.
New balance = 1000.0
```

14.6. What If I Don't Have the Interface?

In the examples we've seen so far, we've always assumed that the Java interfaces for the remote objects are available at compile time. But what happens if they aren't? You might get a reference to a CORBA `Object` from a Naming Service, for example, and not know what interface that object implements or (more likely) not have that Java interface in the client JVM. We mentioned earlier that you can use an `org.omg.CORBA.Object` reference directly to make requests and exchange data with its remote object now we'll briefly look at how the Dynamic Invocation Interface (DII) makes that possible.

The CORBA standard actually defines two complementary APIs for this purpose. The DII is used by a CORBA client to make remote method requests of a server object, while the DSI can be used by a server-side skeleton to forward method invocations to its server implementation object when it doesn't have the actual servant interface available. Both of these APIs perform the same essential function: providing a dynamic interface to an object whose interface is not known at compile time.

The DII and DSI may seem like sidebar topics in the CORBA world, but in reality they are at the heart of CORBA and how it works. When we generate Java stubs and skeletons from IDL interfaces, the generated code uses the DII and DSI to execute remote method calls. The details of how this is done are shielded from you, the developer, by the Java interface you use to interact with the remote object. But it's still worthwhile to understand how CORBA objects implement their distributed nature, especially in situations in which the Java interface for the remote object is not there and you need to deal directly with these details.

On a more pragmatic note, in some cases you might find yourself needing the DII or the DSI. In all the examples we've gone through in this chapter so far, we've assumed that both the server and client JVMs had all of the relevant interfaces available locally. For clients, we assumed that somehow they had acquired the IDL-generated Java interface for our objects (`Account`, etc.). It is safe to assume you have access to the client machines directly or can provide an easy way for users to download the required classes themselves as part of some installation process. Since the clients will obviously need to acquire your client code in the first place, it's usually possible to provide the other CORBA client classes as well, and the DII isn't needed. But if not (perhaps you want to minimize class downloads for some reason, for example), then it may be useful to use the DII to create a client that operates without the need for the object interfaces themselves.

In this section, we take a look at how the DII works and how you might use it in a client. We won't cover the DSI in this book, since its practical uses are even more limited for the average developer. Note, however, that the API of the DSI is analogous to that of the DII, so you shouldn't have much trouble mapping the following explanation to the DSI as well.

14.6.1. Dynamic Invocation Interface

The DII provides abstract representations of remote method requests and their arguments. In simple terms, this means it includes objects that represent remote method requests and parameters that

are passed with these method requests. Methods on these objects allow you to set the parameters to the request, make the request, and get the results. DII's central classes are:

Request

A request to invoke a method on a remote object. Created by the client and issued through the ORB to the server object.

NamedValue

A named parameter to a method request. Conceptually, this is a name tied to an `Any` value. The name of the value must match the name of the parameter as specified in the IDL interface the remote object satisfies.

NVList

A list of `NamedValue` parameters used to represent an argument list passed into a remote method request.

Any

A general argument value. An `Any` object can contain the Java equivalent of any basic IDL type or an `Object` that can be described in IDL.

Context

A list of `NamedValue` objects used to specify any details of the client environment that shouldn't be passed as method arguments.

Once you get an `org.omg.CORBA.Object` reference to a remote object (using any of the approaches we've already covered), you can create and issue a method request to the object by building a parameter list for the method call, making a `NamedValue` object to hold the result, making a `Context` object and putting any useful environment values in it, and then using all of these items to create a `Request` object that corresponds to a particular method on the object. In general, the various `create_XXX()` methods on the `ORB` interface are used to construct all of these elements of a DII request, except for the `Request` itself. That is created by calling `_create_request()` on the `Object` that is the target of the remote method call.

[Example 14-8](#) shows a version of our `Account` client that uses DII calls to invoke the `deposit()` or `getBalance()` methods on a remote `Account` object (the case for the `withdraw()` method is very similar to the `deposit()` case, so it's omitted here for the sake of brevity.) The client is structured very much like the client in [Example 14-7](#), except that the actual calls to the remote `Account` object are constructed as DII calls. For the deposit requests, the method has a single float argument and no return value, so the call to create the `Request` includes a null `NamedValue` for the result and an `NVList` containing a single `NamedValue` holding the amount to be deposited. In the case of requests for the `Account` balance, the `getBalance()` method has no arguments and a single float return value, so the creation of the `Request` includes a null `NVList` for the arguments and a `NamedValue` containing an `Any`

object intended to hold a floating-point value.

Example 14-8. Account client DII.java

```
import org.omg.CORBA.*;
import org.omg.CosNaming.*;

public class AccountClientDII {
    public static void main(String args[]) {
        ORB myORB = ORB.init(args, null);
        try {
            // The object name passed in on the command line
            String name = args[0];
            org.omg.CORBA.Object acctRef = null;
            if (name.startsWith("corbaname") || name.startsWith("corbaloc") ||
                name.startsWith("IOR")) {
                System.out.println("Attempting to lookup " + args[0]);
                acctRef = myORB.string_to_object(args[0]);
            }
            else {
                System.out.println("Invalid object URL provided: " + args[0]);
                System.exit(1);
            }

            // Make a dynamic call to the doThis method
            if (acctRef != null) {
                // We managed to get a reference to the named account; now check the
                // requested transaction from the command line
                String action = args[1];
                float amt = 0.0f;
                if (action.equals("deposit")) {
                    amt = Float.parseFloat(args[2]);
                }
                System.out.println("Got account, performing transaction...");
                try {
                    // Did user ask to do a deposit?
                    if (action.equals("deposit")) {
                        // The following DII code is equivalent to this:
                        // acct.deposit(amt);

                        // First build the argument list. In this case, there's a single
                        // float argument to the method.
                        NVList argList = myORB.create_list(1);
                        Any arg1 = myORB.create_any( );
                        // Set the Any to hold a float value, and set the value to the
                        // amount to be deposited
                        arg1.insert_float(amt);
                        NamedValue nvArg =
                            argList.add_value("amt", arg1, org.omg.CORBA.ARG_IN.value);
```

```

// Java IDL doesn't implement the get_default_context( ) operation
// on the ORB, so we just set the Context to null
Context ctx = null;
// Create the request to call the deposit( ) method
Request depositReq =
    acctRef._create_request(ctx, "deposit", argList, null);

// Invoke the method...
depositReq.invoke( );
System.out.println("Deposited " + amt + " to account.");
}
else {
// The following DII code is equivalent to this:
//    acct.balance( );

// No argument list is needed here, since the getBalance( ) method
// has no arguments. But we do need a result value to hold the
// returned balance
Any result = myORB.create_any( );
// Set the Any to hold a float value
result.insert_float(0.0f);
NamedValue resultVal =
    myORB.create_named_value("result", result,
        org.omg.CORBA.ARG_OUT.value);
// Java IDL doesn't implement the get_default_context( ) operation
// on the ORB, so we just set the Context to null
Context ctx = null;
// Create the request to call getBalance( )
Request balanceReq =
    acctRef._create_request(ctx, "getBalance", null, resultVal);

// Invoke the method...
balanceReq.invoke( );
System.out.println("Current account balance: " +
    result.extract_float( ));
}
}
catch (Exception e) {
    System.out.println("Error occurred while performing transaction:");
    e.printStackTrace( );
}
}
else {
    System.out.println("Null account returned.");
    System.exit(1);
}
}
catch (Exception e) {
    e.printStackTrace( );
}
}
}
}

```

Again, note that in most situations you will actually have the Java interface for the remote object available in your client along with its helper class, so you'll be able to narrow the `Object` reference to a specific type and call its methods directly. One exception might be if you're building some kind of software development tool and you want to provide a dynamic execution utility for the CORBA code being developed. The previous example demonstrates how a CORBA method call can be carried out at this lower level, in case you ever find it necessary to do so. And when you're trying to fix a problem with your CORBA application, it's always better to understand what's going on under the hood, so to speak.



Chapter 15. JavaMail

The JavaMail APIs provide a platform- and protocol-independent framework to build Java-based mail and messaging applications with a particular focus on Internet email. JavaMail can be used to build mail clients, but, in an enterprise setting, the primary application for JavaMail is to add electronic mail capability to large applications. For example, a Java servlet can use JavaMail to implement a web-based mail reading system, send confirmations and notifications, receive commands from users, or even communicate with middleware.

Sun included a basic SMTP email class in the first version of the JDK. However, because this class could only send email, and was available only with Sun's implementation of the JDK, developers who wished to include more advanced electronic mail capability into their applications generally needed to start from scratch. The JavaMail API, first introduced in early 1998 and now in Version 1.3, fills this niche, giving Java applications a complete mail and messaging framework. JavaMail has been part of J2EE since Version 1.2, and JavaMail 1.3 is included in J2EE 1.4.

Strictly speaking, the JavaMail API isn't really an email API at all. The JavaMail classes provide an interface to a generic set of messaging functions for creating, storing, and transporting messages. Developers can extend these classes to implement custom transport protocols and develop standardized mail applications. As an add-on, the distribution includes implementations of popular mail-related protocols and a set of helper classes specifically designed to work with email on the Internet. Since JavaMail includes built-in support for Multipurpose Internet Mail Extensions (MIME^[*])-formatted messages, this communication can be extremely rich, opening up a range of new possibilities for large-scale applications.

[*] MIME was originally defined in RFCs 1521 and 1522. These have been replaced by RFCs 2045 through 2049. The complete set, along with additional information, related RFCs, and helpful links, is available at <http://www.nacs.uci.edu/indiv/ehood/MIME/MIME.html>.

15.1. Email and JavaMail

The vast majority of JavaMail applications use the system to interact with Internet email. While it's safe to assume that anyone reading this book has some conceptual familiarity with electronic mail, a quick overview of Internet-based email from a technical perspective is in order. If you would like more details about email services, see *Programming Internet Email* by David Wood (O'Reilly).

The primary mechanism for email transport over the Internet is the Simple Mail Transfer Protocol (SMTP), as defined in RFC 822. SMTP clients, such as desktop email programs, send messages by connecting to another SMTP server and transmitting a series of headers followed by a message body. Once the SMTP server receives a message, it is either stored for access by local users or forwarded to another SMTP server nearer to its destination. Most ISP and corporate mail servers will accept messages only for delivery to local users or from local users for delivery elsewhere. In the latter case the SMTP server is responsible for determining the appropriate destination for the message.

If the SMTP server determines that a message is for local delivery (for instance, mail.oreilly.com handles incoming email for O'Reilly Media), it handles the message in a number of ways. Early SMTP-based email systems just stored the incoming messages in the Unix mailbox format for retrieval off the local disk by mail reader software. However, as systems grew and mail services became decoupled from other systems, a need arose for a standardized way to access centralized "post office" systems that pooled the incoming mail for an entire domain (or, in the case of many ISPs, multiple domains). At a minimum, this involved providing access to Unix mailboxes over the Internet. POP, which stands for Post Office Protocol, was the first successful attempt to do this. A mail client (such as Outlook, PINE, or Eudora) can use POP to connect to a POP server and download the messages waiting for a particular user. POP has subsequently been replaced with POP-3 (which is universally supported) and IMAP, a more flexible successor that has rapidly gained acceptance since Version 4 was introduced in 1994.

[Figure 15-1](#) shows the path of a typical email message from the sender's email client to the recipient's email client. Note that this is a very simple case, with two SMTP servers relaying mail over the long haul (like AT&T long distance) and a or IMAP server at the receiving end (acting like a Baby Bell). Depending on server and client configurations, there may be only one SMTP server involved, or there may be several as messages are routed from ISP to ISP.

Figure 15-1. Basic email routing

15.1.1. JavaMail Layers

At this point, we'll give you a brief overview of the JavaMail layers. The abstract layer of the JavaMail APIs can be divided into three categories: *sessions*, *services*, and *messages*. Sessions contain information about the current user, mail hosts, and protocols. They serve as factories for various types of objects. Messages are exactly that: text sent through the mail system. Services allow you to do useful things with messages and are subdivided (as of JavaMail 1.2) into two categories: *stores* and *transports*. Stores act as containers for sets of messages, while transports route messages from one site to another.

As we mentioned earlier, most of the JavaMail API consists of abstract classes. When necessary, however, implementations of those classes are provided by an *implementation layer*. Implementation layers may include SMTP, IMAP, NNTP, or POP-3 protocols or a message storage system. The implementation layer handles the actual message processing tasks.

15.1.2. Installing and Configuring JavaMail

If you're using J2EE 1.3 or 1.4, which include JavaMail 1.2 and 1.3 respectively, you need not download JavaMail. Otherwise, you can download the standalone JavaMail 1.3 distribution, suitable for inclusion in any application, from <http://java.sun.com/products/javamail>. You will also need the JavaBeans Activation Framework, which contains the `javax.activation` classes used to handle certain content formatting tasks. JAF is available from <http://java.sun.com/products/javabeans/jaf>.

If you're not in a full J2EE configuration, install JavaMail by adding JAF's `activation.jar` and JavaMail's `mail.jar` to your system `CLASSPATH` or your JDK's `lib` directory. If you look at the JavaMail archive, you'll notice some additional jars: `mailapi.jar`, `imap.jar`, `smtp.jar`, and `pop3.jar`. The `mail.jar` file contains the contents of all four files, but you can use the smaller jars to selectively include service implementations in your environment. Using the smaller jar files helps reduce loading time for applets. Note that some J2EE application servers may package the JavaMail and JAF classes differently. Consult your server's documentation for details.

By simply adding these jar files to the `CLASSPATH`, most JavaMail programs should compile and execute. However, some additional configuration options are available. JavaMail allows you to set a number of system properties to provide email-related system defaults. In addition to the eight properties listed in [Table 15-1](#), each service has its own set of specific properties. Note that you don't need to set any of these to use the APIs.

Table 15-1. Core JavaMail environment properties

Property	Description	Default
----------	-------------	---------

Property	Description	Default
<code>mail.store.protocol</code>	Default message store protocol (such as POP or IMAP).	First message store specified in the JavaMail configuration files
<code>mail.transport.protocol</code>	Default message transport protocol (such as SMTP).	First transport specified in the JavaMail configuration files
<code>mail.host</code>	Default host. JavaMail providers will attempt to connect to this system if no other host is specified.	The local machine
<code>mail.user</code>	Default username to use when connecting to a provider.	<code>user.name</code>
<code>mail.protocol.host</code>	Default host for a particular protocol. To set the default IMAP host, use <code>mail.imap.host</code> . Overrides <code>mail.host</code> .	Value of <code>mail.host</code>
<code>mail.protocol.user</code>	Default user for a particular protocol. To set the default IMAP user, use <code>mail.imap.user</code> .	Value of <code>mail.user</code>
<code>mail.from</code>	Return address of the current user.	username@host
<code>mail.debug</code>	Whether JavaMail should operate in debug mode. Debug mode can also be toggled via the <code>setDebug()</code> method of <code>Session</code> .	<code>false</code>

15.1.2.1. Provider registries

JavaMail is designed to allow third parties to provide service implementations. JavaMail identifies the installed providers via the *javamail.providers* and *javamail.address.map* registry files. When starting up a JavaMail session, the system searches for *javamail.providers* files in the following locations:^[*]

[*] Note that under JDK 1.1, only one *META-INF/javamail.providers* file can be found due to classloader limitations.

- *JAVA_HOME/lib/javamail.providers*
- *META-INF/javamail.providers* (in application jar files)
- *META-INF/javamail.default.providers* (in application jar files; generally found only in the JavaMail *mail.jar* file)

The first entries found provide the defaults unless specifically changed by the application; if two POP3 providers are found, the first one identified becomes the default.

The *javamail.providers* file is formatted like this:

```
protocol=imap; type=store;
class=com.sun.mail.imap.IMAPStore;
vendor=SunMicrosystems,Inc;
version=1.2;
```

The semicolon is the field delimiter. The *javamail.address.mapfile* is located in the same fashion as *javamail.providers* (including the *javamail.default.address.mapfile* in *mail.jar*) and contains mappings of address types to transport services. For example, messages to RFC 822-formatted addresses are transported via the SMTP protocol, and this relationship is indicated by a line in the *javamail.address.mapfile* like this one:

```
rfc822=smtp
```

Unless you are developing your own service providers, you probably won't need to alter any of these files.

15.1.3. The Mail Session

As we mentioned earlier, the JavaMail API provides a `Session` class that fills two roles. First, it stores messaging-related properties, both global and user-specific, and provides this information to most new JavaMail objects via their constructors. Second, it provides factory methods for `Store` and `transport` objects.

The JavaMail API provides two types of `Session` objects: the default session and local sessions. The default session is created only once using the static `Session.getDefaultInstance()` method. There are two overloaded versions of the `getDefaultInstance()` method: one accepts a `java.util.Properties` object, and one accepts a `Properties` object and a `javax.mail.Authenticator`. The `Properties` object can be used to set JavaMail system properties.^[*] The `Authenticator` can also be used to limit access to the default session; if the first call to `getDefaultInstance()` includes an `Authenticator`, all subsequent calls must pass either the same instance of the `Authenticator` object or an instance from the same class loader as the original `Authenticator` object. Note that no `Authenticator` implementations are provided with the basic JavaMail distribution; you have to create your own by subclassing the abstract `javax.mail.Authenticator` class provided by the API.

[*] The `Properties`-only version of this method was added in the JavaMail 1.2 release, so if you are running against a stock J2EE 1.2 installation or the JavaMail 1.1 release, you won't find it. Everything else in this chapter should apply equally to Versions 1.1 and 1.2.

The `Session.getInstance()` method behaves the same as `getDefaultInstance()`, but also provides a local instance of the `Session` object that is not shared across the JVM. With this approach, the `Authenticator` is used only to determine usernames and passwords for services that require them.

The following code sample retrieves a default session, using the JVM system properties to define JavaMail properties and using an `Authenticator` implementation called `myAuthenticator`. It then tries to retrieve a second reference to the default session without using an authenticator. If this fails

(which, in this case, it will), the code gets a local instance instead.

```

Session session = Session.getDefaultInstance(System.getProperties( ),
                                             new MyAuthenticator( ));

// Try to get it again, without the authenticator; get local
// instance on failure
Session sess2 = null;
try {
    sess2 = Session.getDefaultInstance(System.getProperties( ), null);
} catch (SecurityException se) {
    sess2 = Session.getInstance(System.getProperties( ), null);
}

```

15.1.3.1. Authenticators

Besides controlling access to the default session, `javax.mail.Authenticator` objects are also used to provide usernames and passwords to messaging services such as mail servers. This is accomplished by extending the abstract `Authenticator` class and overriding the `getPasswordAuthentication()` method, which returns a JavaMail `PasswordAuthentication` object:

```

class BasicAuthenticator extends Authenticator {
    protected PasswordAuthentication getPasswordAuthentication( ) {
        return new PasswordAuthentication("scott", "tw7182");
    }
}

```

The `PasswordAuthentication` object is simple: it just serves as a container for the two strings. In most applications, these would be entered by the user via a dialog box or retrieved from some centralized directory service. Since `Authenticator` objects are global to the session, the base `Authenticator` class contains a number of final methods, which can be used to provide information about the particular authentication request—for instance, using `getRequestingProtocol()` and `getRequestingSite()`, the `getPasswordAuthentication()` code can determine whether to provide the IMAP login information for *mailserver.myco.com* or the POP3 login information for *pop.prep.edu*.

15.1.3.2. Providers

A `Session` itself doesn't provide any mail-handling capabilities; it simply provides the means to access other messaging services. Generally, you will know what sort of messaging protocols and services are available before sitting down to code your application and won't need to determine them on the fly. However, the `javax.mail.Provider` object provides a convenient view of the implementation details for particular services: the protocol (such as SMTP), the service type (message storage or message transport), vendor, version, and implementing class name.

For most applications, you don't actually need to worry about the `Provider` object, because you don't need access to the object itself to access the features it provides. However, if you do need one, the `getProviders()` method of `Session` will allow you to interrogate the current session to determine which service providers are installed. This method returns an array of `Provider` objects describing the different protocol implementations available to the session. To retrieve the `Provider` object for a particular protocol, call the `getProvider(String)` method and pass it the protocol name, as in the following:

```
mySession.getProvider("smtp");
```

By default, provider information is specified in the `javamail.providers` and `javamail.default.providers` resource files. This means that if you are using the providers in the Sun JavaMail distribution, you don't have to do anything to obtain access to the standard services. If you want to programmatically install your own implementation of a particular protocol, you can pass a vendor-supplied `Provider` object to the `setProvider()` method of `Session`.

15.1.3.3. URL names

JavaMail provides a number of mechanisms for identifying services, including the system properties discussed earlier in this chapter. JavaMail services can also be identified via URLs in the form `protocol://server`. An IMAP URL, for example, might look like `imap://mailserver.cs.edu`. The `javax.mail.URLName` object is used to encapsulate information about a connection to a mail service. This can be quite convenient, since it allows all necessary information about a connection to be assembled in a centralized location. Since service URLs are protocol-specific, this can include an arbitrary *range of precision* IMAP URLs, for instance, allow the specification of particular messages and folders within the IMAP message store.^[*]

[*] RFC 2192 contains full information about the IMAP URL scheme. It can be found online at <http://www.faqs.org/rfcs/rfc2192.html>.

A `URLName` object is only a container for connection information hence the "Name." It doesn't actually connect to the service it identifies. Instead, it can be passed to various methods of `Session`, including `getFolder()`, `getStore()`, and `gettransport()`.

`URLName` objects can be retrieved from existing service connections via the `getURLName()` method of `Service` (we'll look at specific services later in this chapter). They can also be created directly via three constructors:

```
URLName(String url)
URLName(java.net.URL url)
URLName(java.lang.String protocol, java.lang.String host, int port,
        java.lang.String file, java.lang.String username,
        java.lang.String password)
```

The first two are fairly straightforward, accepting a raw URL in a `String` format or encapsulated as a `java.net.URL` object. The third option the detailed constructor allows you to build up the URL from its individual components, including some components (i.e., password) that might not be included in the actual URL at all. Not all options apply to all protocols, so to keep the default value for each field, pass in `null`. The obvious exception is the port parameter, which accepts `-1` as its default value indicator.

The other way to associate login information with a `URLConnection` (particularly if the `URLConnection` was created by some mechanism other than the detailed constructor) is to associate a `PasswordAuthentication` object with the URL. This is done at the session level by using `Session.setPasswordAuthentication(URLConnection, PasswordAuthentication)` and `Session.getPasswordAuthentication(URLConnection)`. Once a `PasswordAuthentication` object has been associated with a `URLConnection`, it will be used to handle authentication for future connections to that URL.

[← PREV](#)

15.2. Creating and Sending Messages

JavaMail messages are encapsulated within the abstract `Message` class. Since JavaMail is a transport-independent API, the default message class is very generic. A standard `Message` has three attributes: a subject, a set of headers, and content. The `Message` class defines `get-` and `set-` methods for each of these. In addition, the `Message` class implements the `Part` interface, which defines a set of methods for dealing with message headers in an abstract fashion and for associating content with the message. The actual message content is contained in a `javax.activation.DataHandler` object, part of the JavaBeans Activation Framework.

Message origins and destinations are set via `Address` objects. Since `Address` is abstract, subclasses are provided to handle particular address types. The basic JavaMail implementation includes an `InternetAddress` class for SMTP mail and a `NewsAddress` class for NNTP (Usenet) news. Note that, because there is no NNTP service provider included with the JavaMail distribution, the `InternetAddress` class is of greater use to most programmers.

The easiest way to create an `InternetAddress` is to instantiate a new object, passing an RFC-822-formatted (`user@host`) address into the constructor:

```
InternetAddress addr = new InternetAddress("adams@whitehouse.gov");
```

If the address format is invalid, an `AddressException` is thrown. You can also supply a personal name as a second parameter:

```
InternetAddress addr = new InternetAddress(
    "adams@whitehouse.gov", "John Adams");
```

Error-handling behavior for `InternetAddress` objects is a little odd. The basic constructor throws an exception if the address provided doesn't conform to the RFC 822 rules. However, if you supply a personal name, no `AddressException` is thrown, even if the address provided is improperly formatted.

JavaMail 1.3 introduced a `strict` flag to the `InternetAddress` constructor. However, there appears to be a bug in the reference implementation that causes this flag to be ignored. You can get around this by creating an empty `InternetAddress` object and setting the `address` property separately. If you subsequently want to validate the address, you can call the `newValidate()` method. To summarize:

```
// No exception:
InternetAddress address = new InternetAddress("test@", "John Test");
// Exception! (JavaMail 1.3.2, J2EE 1.4.1 RI)
```

```

InternetAddress address2 = new InternetAddress("test@", false);

InternetAddress ad1 = new InternetAddress( );
ad1.setAddress("bob@"); // doesn't throw exception
ad1.validate( ); // throws exception

```

In most cases, this behavior does not make any difference you generally want strict parsing, since if JavaMail cannot understand the address, the mail server probably won't either. But some applications (such as email to fax gateways) do break the RFC 822 rules, so you should know how to handle cases like this.

Back to the matter at hand. The JavaMail distribution comes with one `Message` implementation: the `MimeMessage` class, which handles Internet email. Instantiating a simple `MimeMessage` is easy:

```

Message msg = new MimeMessage(session);
msg.setFrom(new InternetAddress("tip@house.gov"));
msg.setRecipients(Message.RecipientType.TO,
                  InternetAddress.parse("mike@state.ma.us", false));
msg.setText("We need to discuss a highway project.");

```

Additional `set` methods can be called to set other attributes of the message, such as the subject, header fields, and so on. The `Part` interface also offers ways to set message content. The simplest is to use the `setText()` method, which sets the message content to a text string, with the content MIME type set to `text/plain`. The more fine-grained approach is to create a JAF `DataHandler` object, assign it content and MIME type, and use the `setDataHandler()` method to associate the `DataHandler` with the message. The equivalent of the `setText()` call would be:

```

javax.activation.DataHandler dh = new javax.activation.DataHandler(
    "We need to discuss a highway project.", "text/plain");
msg.setDataHandler(dh);

```

Other content types besides plain text are possible. To send HTML-formatted mail, pass the HTML into a `DataHandler` with the content type set to `text/html`.

15.2.1. Sending Messages

Once a message has been created, sending it is simple. The `Address` object identifies the type of address used (Internet email or Usenet news) so that the JavaMail API can determine the appropriate transport mechanism. The simplest way to do this is to call the static `send(Message)` method of `transport`. This approach uses the default session to determine mail servers (via the environment properties) and appropriate protocols (via `javamail.address.map`).

To obtain a little more control, you can retrieve a protocol-specific `Transport` object from a `Session`:

```
Transport t = session.getTransport("smtp");
t.send(msg);
```

[Example 15-1](#) puts all the pieces together, retrieving the default session, assembling a simple message, and using the default transport mechanism to send it.

Example 15-1. Sending an SMTP mail message

```
import javax.mail.*;
import javax.mail.internet.*;
import java.util.Properties;

public class MailSend {
    public static void main(String[] args) {
        Properties props = System.getProperties();
        props.put("mail.smtp.host", "mail.college.edu");
        Session session = Session.getDefaultInstance(props, null);

        try {
            Message msg = new MimeMessage(session);
            msg.setFrom(new InternetAddress("you@yourhost.com"));
            msg.setRecipients(Message.RecipientType.TO,
                InternetAddress.parse("me@myhost.com", false));
            msg.setSubject("Test Message");
            msg.setText("This is the sample Message Text");
            msg.setHeader("X-Mailer", "O'Reilly SimpleSender");
            Transport.send(msg);
        } catch (AddressException ae) {
            ae.printStackTrace(System.out);
        } catch (MessagingException me) {
            me.printStackTrace(System.out);
        }
    }
}
```

15.3. Retrieving Messages

In addition to allowing programs to create and send messages, the JavaMail API also provides facilities to retrieve incoming messages. This is done via a "message store." For standard Internet mail, the message store is generally held on a mail server and accessed via a mail client protocol.

JavaMail 1.2 and later distributions include message store implementations for POP-3 and IMAP mail client protocols. POP-3 is an older protocol that is still used by most ISPs and many corporate environments. It allows clients to list all available messages, retrieve them, and delete them. POP, which stands for Post Office Protocol, was designed to work like a post office, acting as a central clearinghouse for incoming messages and providing a mechanism to deliver those messages to the recipient.

POP-3 stops at the point of message delivery. IMAP, however, is a newer protocol that allows more complex message handling. IMAP clients can organize messages into different "folders" on the server based on user commands or built-in server filtering rules. In some implementations, the IMAP server is intended as the final repository for all of a user's messages. The mail client simply accesses the IMAP folders and retrieves message information as required.

15.3.1. Message Stores

Message stores are accessed via implementations of the `Store` object. The `getStore()` method of `Session` is used to retrieve a `Store` implementation. `getStore()` provides a variety of ways to specify the desired message retrieval method, using the JavaMail system properties to greater or lesser degrees. Calling `getStore()` without any arguments will return a message store based completely on JavaMail system properties. You can also pass `getStore()` a text string specifying the protocol to use, a `Provider` object, or a `URLName` object that completely defines the connection. If the message store provider requested is unavailable, a `NoSuchProviderException` is thrown:

```
Session session = Session.getDefaultInstance(
    System.getProperties( ), null);

try {
    // Get the default message store
    Store defaultStore = session.getStore( );

    // Get a store using the default server for IMAP
    Store imapStore = session.getStore("imap");

    // Get the default IMAP store based on a Provider
    Store imapStore2 = session.getStore(session.getProvider("imap"));

    // Get an IMAP store based on a URLName
```

```

        Store namedStore = session.getStore(
            new URLName("imap", "imap.my.com", -1, null, "user", "pw"));
    } catch (NoSuchProviderException nsp) {
    }

```

Once you have a `Store` object, you need to connect to it before retrieving messages. `Store` extends the abstract `Service` class, which provides generic connection services for both `Store` and `transport` objects, as well as hooks to the JavaMail event model. To connect to the store, use the `connect()` method, which comes in three variants. The parameterless version can be used with message stores that don't require authentication or with authenticated stores that have already had the username and password specified via a `URLName` or an `Authenticator`. The other two versions allow you to specify the mail host, username, and password or the mail host, port, username, and password. To use the default value for any of these parameters, pass `null` (for object parameters) or `-1` (for integer parameters.)

Between the system properties, the `getStore()` method, and the `connect()` method, JavaMail provides a variety of ways to specify the attributes of message stores. This is because different protocols require different information to successfully connect to a server. The information that is most directly provided always overrides the system or store defaults.

15.3.2. Handling Incoming Messages

Message stores provide the most general collection of messages possible. However, it is not a good idea to assume that every message in a system will be collected in the same place. JavaMail supports grouping of messages into folders. Every message store contains a default folder, represented by a `Folder` object. The default folder generally contains one or more subfolders that contain actual messages. The default folder is retrieved via the `getDefaultFolder()` method of a `Store` object. Subfolders can then be retrieved by calling the `getFolder()` method on the default folder:

```

Folder defaultFolder = myStore.getDefaultFolder( );
Folder inboxFolder = defaultFolder.getFolder("INBOX");

```

In this example, we first retrieve the default folder from the message store and INBOX folder. This is by the way, the standard hierarchy for both IMAP and POP-3 message stores; actual messages are stored, by default, in the INBOX folder. IMAP message stores allow users to configure a broader range of folders internally, either via a mail client or by configuring settings on the mail server. These folders can be nested indefinitely.

After obtaining a folder, you can open it with the `Folder.open()` method. Folders can be opened in either read-only or read/write mode, depending on whether the `Folder.READ_ONLY` or `Folder.READ_WRITE` constant is passed to the `open()` method. For example:

```

// Open a folder for read-write access:

```

```
inboxFolder.open(Folder.READ_WRITE);
```

[Example 15-2](#) connects to a message store located at *mail.college.edu*, via the IMAP and POP-3 protocols and using an `Authenticator` to handle usernames and passwords.

Example 15-2. IMAP and POP message retrieval

```
import javax.mail.*;

public class MailCheck {

    static String mailhost = "mail.college.edu";

    public static void main(String[] args) {

        Session session = Session.getDefaultInstance(System.getProperties( ),
            new myAuthenticator( ));

        try {
            Store imapStore = session.getStore(new URLName("imap://" + mailhost));
            imapStore.connect( );
            System.out.println(imapStore.getURLName( ));
            // Get a default folder, and use it to open a real folder
            Folder defaultFolder = imapStore.getDefaultFolder( );
            defaultFolder = defaultFolder.getFolder("INBOX");
            defaultFolder.open(Folder.READ_WRITE);

            Message[] msgs = defaultFolder.getMessages( );
            if(msgs != null)
                for (int i = 0; i < msgs.length; i++) {
                    System.out.println(msgs[i].getSubject( ));
                }

            // Now, get it with POP3, using authenticator
            Store popStore = session.getStore("pop3");
            popStore.connect(mailhost, null, null);

            // Get a default folder, and use it to open a real folder
            Folder defaultPopFolder = popStore.getDefaultFolder( );
            defaultPopFolder = defaultPopFolder.getFolder("INBOX");
            defaultPopFolder.open(Folder.READ_ONLY);

            msgs = defaultPopFolder.getMessages( );
            if(msgs != null)
                for (int i = 0; i < msgs.length; i++) {
                    System.out.println(msgs[i].getSubject( ));
                }
        }
    }
}
```

```

        } catch (MessagingException me) {
            me.printStackTrace(System.out);
        }
    } // End main( )
}

class myAuthenticator extends Authenticator {
    protected PasswordAuthentication getPasswordAuthentication( ) {
        return new PasswordAuthentication("mailname", "test");
    }
}

```

15.3.3. Searches and Message Management

JavaMail includes a mechanism for programmatically generating search criteria for finding messages within folders. Searches are built via the `SearchTerm` object. `SearchTerm` is an abstract class with a single method, `match(Message)`, which returns `true` or `false` depending on whether the message matches the term's criteria. JavaMail provides a number of `SearchTerm` implementations that allow for searching on various aspects of a message. `SubjectTerm`, for instance, does substring matching on the Subject header. The following code will search a folder for all messages with "Urgent" in the subject line:

```

SearchTerm t = new SubjectTerm("Urgent");
Message[] msgs = folder.search(t);

```

More complex searches use the `AndTerm`, `OrTerm`, and `NotTerm` objects, which allow chaining and nesting of search criteria. Here's how to tell when we haven't gotten something to our editor on time

```

SearchTerm t = new AndTerm(new FromStringTerm("oreilly.com"),
    new SubjectTerm("late"));
Message[] msgs = folder.search(t);

```

The search will match messages from *oreilly.com* in which the subject contains the word "late." You can nest `AndTerm` and `OrTerm` objects as deeply as you want, although performance will obviously suffer with more complex searches.

The default implementation of the search functionality will rely on the data provided via `Message` objects to perform searches. This has the advantage of working regardless of whether the underlying message store supports search functionality and allows programmers to easily define their own search terms by extending the `SearchTerm` class. If the underlying message store does support search functionality, the implementation has the option of translating the search, wherever possible, into native terms.

Unsolicited commercial email (spam) is of great concern to many readers, so we offer our contribution to cleaner inboxes. [Example 15-3](#) will connect to a mailbox and perform a set of subject searches for strings that generally indicate less than desirable content. Once identified, a copy of the message is sent to a separate address for monitoring purposes, and the original is deleted.

Example 15-3. SpamCleaner.java

```
import javax.mail.*;
import javax.mail.internet.*;
import javax.mail.search.*;

import java.util.Properties;

public class SpamCleaner {

    static String mailhost = "imap.mycompany.com";
    static String smtphost = "mail.mycompany.com";
    static String redirectEmail = "spamlog@mycompany.com";

    static String[] badSubjects = {"XXX", "MAKEMONEYFAST", "!!!!"};

    public static void main(String[] args) {

        Properties props = new Properties( );
        props.setProperty("mail.smtp.host", smtphost);
        Session session = Session.getDefaultInstance(props, null);

        try {
            Store imapStore = session.getStore("imap");
            imapStore.connect(mailhost, "scott", "yu7xx");

            Folder defaultFolder = imapStore.getDefaultFolder( );
            Folder inboxFolder = defaultFolder.getFolder("INBOX");
            inboxFolder.open(Folder.READ_WRITE);

            // Assemble some search criteria, using nested OrTerm objects
            SearchTerm spamCriteria = null;
            for(int i=0; i<badSubjects.length; i++) {
                SubjectTerm st = new SubjectTerm(badSubjects[i]);
                if(spamCriteria == null)
                    spamCriteria = st;
                else
                    spamCriteria = new OrTerm(spamCriteria, st);
            }

            Message[] msgs = inboxFolder.search(spamCriteria);

            if(msgs != null)
                for (int i = 0; i < msgs.length; i++) {
```

```

// Redirect the message
Message m = new MimeMessage((MimeMessage)msgs[i]);
m.setRecipient(Message.RecipientType.TO,
    new InternetAddress(redirectEmail));
// Clear out the cc: field so we don't loop or anything
m.setHeader("CC", "");
Transport.send(m);

// Now delete the original
System.out.println(
    "Deleting Message "+msgs[i].getMessageNumber()+" : "
    + msgs[i].getSubject());
msgs[i].setFlag(Flags.Flag.DELETED, true);
}

// Close inbox folder, with "Expunge" flag set to true
inboxFolder.close(true);

imapStore.close();
} catch (MessagingException me) {
    me.printStackTrace(System.out);
}
} // End main()
}

```

[Example 15-3](#) also shows how to delete a message from an IMAP message store. Rather than calling a delete method on a folder or message object, we call the `setFlag()` method of `Message` and set the deletion flag (identified by the `Flags.Flag.DELETED` constant) to true. You can examine all the flags set on a particular message by calling the `getFlags()` method of `Message`, which returns a `Flags` object. However, in most cases, the ability to set the `DELETED` flag is all you need.

Note that marking a message as deleted doesn't actually delete it. Messages aren't removed from the message store until the store is expunged. This can be done either by calling the `expunge()` method of `Folder` or passing a Boolean `true` value to the `close()` method of `Folder`. Once a message has been expunged, the message numbers within the folder (returned by the `getMessageNumber()` method of `Message`) may change.

[Example 15-3](#) shows how to clone a message. JavaMail 1.2 added a new constructor to the `MimeMessage` object, which causes the new `MimeMessage` to be populated with all the fields and content from the original. The example does something dangerous in assuming that the message retrieved by the `search()` function will actually be a `MimeMessage`. This is fine when using the Sun service implementations, but a more general version of this program would have to be more careful, either constructing the new `Message` object by hand or trying to modify and resend the copy pulled from the message store (which is generally not allowed).

If you are working with a message store that supports modification of message content, the loop in [Example 15-3](#) could be replaced with this instead:

```

for (int i = 0; i < msgs.length; i++) {

```

```
// Alter the message
System.out.println("Altering Message "+msgs[i].getMessageNumber( )+": "
    + msgs[i].getSubject( ));
msgs[i].setSubject("SPAM:" + msgs[i].getSubject( ));
msgs[i].saveChanges( );
}
```

Rather than deleting the message, this code alters the subject line and inserts the word "SPAM:" before the original subject. The `saveChanges()` method writes the changes back to the message stores. However, we can't use this approach with this code because incoming IMAP messages are read-only.

[← PREV](#)

15.4. Multipart Messages

Up until now, we haven't discussed viewing or creating messages except at a very simplistic level. Let's close this chapter by taking a look at the `MimeMessage` class, the standard `Message` implementation for Internet email, and the `Multipart` class, which allows MIME-formatted messages that contain multiple parts (for instance, text and HTML versions of content and file attachments).

15.4.1. Displaying Multipart Messages

`MimeMessage` objects are either single part or multipart. A single-part message has a content-type attribute matching the content of the message (often "text/plain"); the actual message content is stored in the message's content attribute, accessed via the `getContent()` method.

Multipart messages of content types begin with "multipart" and contain a `Multipart` object as their content. The `Multipart` object contains a series of `BodyPart` objects (in the case of a MIME-formatted message, these will be `MimeBodyPart` objects and can be cast as such). Each `BodyPart` has its own content type and associated content.

The `getContent()` and `getContentType()` methods of `Message` are actually inherited from the `Part` interface, which is also implemented by `BodyPart` and `MimeBodyPart`. `Part` provides another useful method, `isMimeType()`, which allows testing for particular content types. The `isMimeType()` method will compare only the primary and subcontent types (the types immediately before and after the / character). So, `isMimeType("text/plain")` will return true if the MIME type is "text/plain" or "text/plain; charset=USASCII." You use the wildcard character `*` in place of the subtype (for example, "text/*") to match all data with a particular primary type.

[Example 15-4](#) is an enhanced version of the mail viewer from [Example 15-2](#) (although it just pulls from a single source). It starts by connecting to a POP server and downloading copies of the messages in the inbox. For each message, the program displays subject and senders and determines whether the message is multipart or not. If it is, the `Multipart` object is retrieved and each individual part is processed. Textual content is written to the screen. Nontextual content with a filename associated with it, such as a file attachment, is written to disk and the path is displayed.

Example 15-4. Multipart display

```
import javax.mail.*;
import javax.mail.internet.*;

import java.io.*;

public class MimeShow {
```

```

static String mailhost = "pop.company.com";

public static void main(String[] args) {

    Session session = Session.getDefaultInstance(
                                                System.getProperties( ), null);

    try {
        Store popStore = session.getStore("pop3");
        popStore.connect(mailhost, "username", "password");

        // Get a default folder, and use it to open a real folder
        Folder defaultPopFolder = popStore.getDefaultFolder( );
        defaultPopFolder = defaultPopFolder.getFolder("INBOX");
        defaultPopFolder.open(Folder.READ_ONLY);

        Message[] msgs = defaultPopFolder.getMessages( );
        if(msgs != null)
            for (int i = 0; i < msgs.length; i++) {
                // Display the message envelope
                System.out.println("\n\r-----");
                System.out.println("Subject: " + msgs[i].getSubject( ));
                Address[] from = msgs[i].getFrom( );
                if(from != null)
                    for(int a = 0; a < from.length; a++)
                        System.out.println("From: " + from[a]);

                // Display the content
                if (msgs[i].isMimeType("text/plain")) {
                    System.out.println((String)msgs[i].getContent( ));
                } else if (msgs[i].isMimeType("multipart/*")) {
                    Multipart mp = (Multipart)msgs[i].getContent( );
                    int count = mp.getCount( );
                    for(int m = 0; m < count; m++) {
                        showBodyPart((MimeBodyPart)mp.getBodyPart(m));
                    }
                } else if (msgs[i].isMimeType("message/rfc822")) {
                    System.out.println("Nested Message");
                } else {
                    System.out.println(msgs[i].getContent( ).toString( ));
                }
            }
        } // end for

    } catch (MessagingException me) {
        me.printStackTrace(System.out);
    } catch (IOException ie) {
        ie.printStackTrace(System.out);
    }
} // End main( )

// Show or save a MIME body part; very simple
public static void showBodyPart(MimeBodyPart p) {
    try {

```

```

String contentType = p.getContentType( );
System.out.println("-- MIME Part: " + contentType);

if(contentType.startsWith("text/"))
    System.out.println((String)p.getContent( ));
else if(p.getFileName( ) != null) {

    File f = new File(p.getFileName( ));
    FileOutputStream fos = new FileOutputStream(f);
    byte[] b = new byte[1024];
    InputStream is = p.getInputStream( );
    fos.write(b);
    is.close( );
    fos.close( );
    System.out.println("Attachment saved as " + f.getAbsolutePath( ));
} else {
    System.out.println(
        "Cannot display this content type and no filename supplied.");
}
} catch(Exception e) {
    System.out.println("Exception Caught: " + e.getMessage( ));
}
} // End showBodyPart( )
} // End class

```

15.4.2. Sending Multipart Messages

Creating a multipart message isn't any more complex than creating a regular MIME message. First, instantiate a `MimeMultipart` object, which will serve as a container for the various parts of the message. For each part, create a `MimeBodyPart` and set its content via the `setContent()` method. Add each body part to the `MimeMultipart` object using the `addBodyPart()` method. Finally, add the `Multipart` object as the content of a `MimeMessage`, and send that message normally.

There are, of course, a few wrinkles in the process, particularly when dealing with file attachments. To attach a file to a message, you need to specify the filename when creating the `MimeBodyPart` (via the `setFileName()` method) and provide the actual file data. Note that the content is provided via the JavaBeans Activation Framework's `DataHandler` object, which we discussed earlier in this chapter. `DataHandler` objects can be fed via `DataSource` objects. JAF provides two of these: `FileDataSource` and `URLDataSource`, which are more or less as advertised one retrieves data from a file, the other from a URL. In addition to retrieving raw bytes, the `DataSource` will also provide the `DataHandler` with an appropriate content type. Here's how:

```

File f = new File("foo.jpg");
MimeBodyPart mbp = new MimeBodyPart( );
mbp.setFileName(f.getName( ));
mbp.setDataHandler(new DataHandler(new FileDataSource(f)));

```

[Example 15-5](#) puts the pieces together. It sends an email to root@company.com from logs@company.com, with a brief text message and a file attachment containing */var/logs/today.log*.

Example 15-5. Sending a multipart message

```
import javax.mail.*;
import javax.mail.internet.*;
import javax.activation.*;
import java.io.File;
import java.util.Properties;

public class MimeAttach {

    public static void main(String[] args) {
        try {
            Properties props = System.getProperties( );
            props.put("mail.smtp.host", "mail.company.com");
            Session session = Session.getDefaultInstance(props, null);

            Message msg = new MimeMessage(session);
            msg.setFrom(new InternetAddress("logs@company.com"));
            msg.setRecipient(Message.RecipientType.TO,
                new InternetAddress("root@company.com"));
            msg.setSubject("Today's Logs");

            Multipart mp = new MimeMultipart( );
            MimeBodyPart mbp1 = new MimeBodyPart( );
            mbp1.setContent("Log file for today is attached.", "text/plain");
            mp.addBodyPart(mbp1);

            File f = new File("/var/logs/today.log");
            MimeBodyPart mbp = new MimeBodyPart( );
            mbp.setFileName(f.getName( ));
            mbp.setDataHandler(new DataHandler(new FileDataSource(f)));
            mp.addBodyPart(mbp);

            msg.setContent(mp);
            Transport.send(msg);
        } catch (MessagingException me) {
            me.printStackTrace( );
        }
    }
}
```

JavaMail's support for multipart messages opens up some interesting possibilities for using email

within your enterprise applications. With a few POP or IMAP mailboxes, you can write software that will exchange messages without any user intervention. This can be used to synchronize configuration files, exchange XML documents, or even handle more mundane tasks such as generating automated status emails.



Chapter 16. Transactions

Transactions and transactional support are crucial for most J2EE applications. J2EE provides thorough and solid support for transactions in the form of JTA. This chapter begins with a brief overview of transaction terminology, including ACID properties and transaction isolation levels as well as the concepts of local and distributed transactions. It then describes various distributed transaction scenarios, such as reading a message from a queue and updating information in a database table within a single transaction. The chapter covers the programmatic transaction model provided by the JTA but also describes the declarative transaction model used with J2EE components.

Transaction support is an essential part of many Java and J2EE APIs such as JDBC and JMS. This chapter covers core concepts about transactions, but you'll find more detailed information about JDBC and JMS in [Chapters 8](#) and [11](#), respectively. Since EJBs, described in [Chapter 6](#), represent the standard persistence model for J2EE, EJBs have extensive support for JTA-based transactions. Hibernate, the open source persistence API discussed in [Chapter 20](#), also provides support for JTA-based transactions.

16.1. Transaction Overview

Before delving into the details of how transactions work, let's define some basic terminology. A *transaction* is a unit of work, a sequence of operations that succeed or fail as a single unit. Transactions guarantee that this unit of work either fully completes or is fully rolled back. Transactions are essential to the successful and robust working of a critical class of enterprise applications: those that make use of shared networked resources like databases, messaging services and the like.

[Figure 16-1](#) shows a simple conceptual model of a transaction. A transaction has three main operations: `begin()`, `commit()`, and `rollback()`. You start a transaction with the `begin()` operation, perform a series of steps, and successfully commit with the `commit()` operation. You roll back the transaction by invoking the `rollback()` operation if a failure occurs.

Figure 16-1. Conceptual model of a transaction

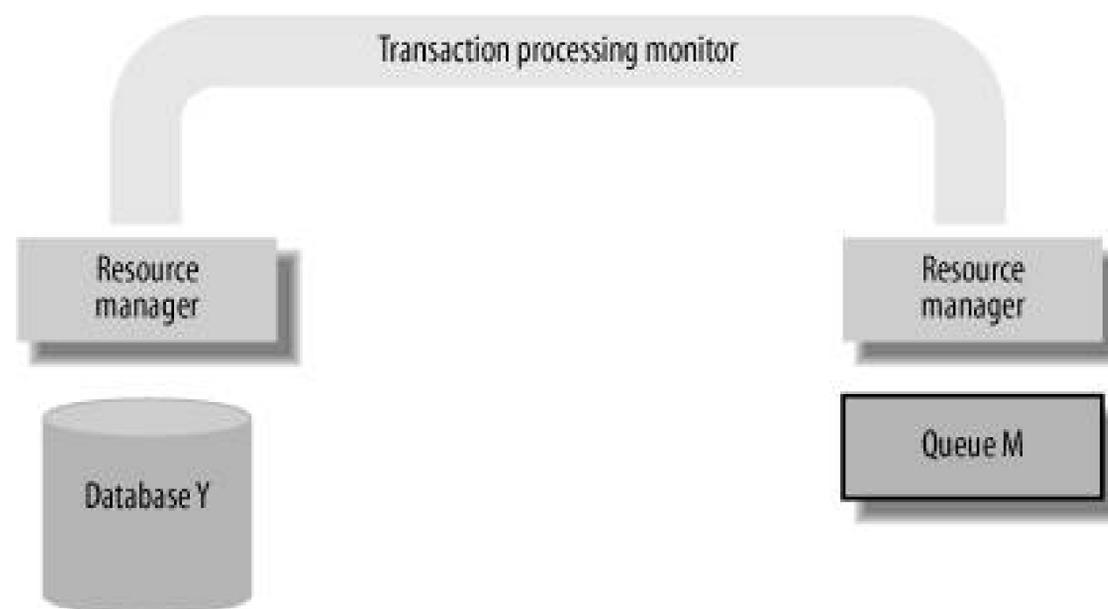
Consider the following example. You want to move money from your checking account into your savings account. There is an operation to debit the money from the checking account and another operation to credit the money into the savings account. Both the operations must either succeed together or fail together. You group the debit and credit operations together in a single transaction. The complete sequence for this transaction is begin, debit checking account, credit savings account, commit if successful or roll back if it fails. Therefore, the entire transaction either succeeds or fails as a single unit. Without a transaction, failure or success of just one operation in this two-operation transaction will leave the system and the accounts in an inconsistent state. The application is unreliable if transactions are not implemented correctly.

Transactions enable a simplified programming model because the developers do not need to consider and program for all the error and exception conditions that can occur when using shared networked resources. When these errors and exception conditions do occur, transactions help with failure recovery by restoring the shared resource to a consistent state. The application simply receives and deals with a corresponding exception generated from the underlying transactional service(s), keeping with the general approach taken with other Java runtime services. This simplified model frees the developers from dealing with complex concurrency issues across multiple users and across multiple, potentially distributed, applications accessing the same networked resource at the same time.

Transactions work closely with one or more *resources*, as we've already mentioned. Most often, the resource is a database. Popular databases include Oracle, IBM DB2, Sybase, and MySQL. Other types of resources such as message-oriented middleware (MOM) destinations can also be transactional. A transaction does not directly interact with a resource. A *resource manager* manages the resource; the resource manager is built into the software that manages the resource (the database or MOM). A transaction is coordinated with the resource managers using a *transaction manager*, also known as a *transaction processing (TP) monitor*. Traditional standalone TP monitors include BEA Tuxedo, IBM TXSeries, and IBM CICS. But transaction managers can also be found embedded in resource managers themselves, such as some relational databases and messaging services and within J2EE containers and application servers. [Figure 16-2](#) depicts the relationships among resources, resource managers, and TP monitors.

The transaction manager works by passing the same *transaction context* to the different resource managers participating in the transaction. The transaction manager keeps *transaction logs* of the work it has asked the resource managers to do.

Figure 16-2. Transaction processing monitor



16.1.1. Transaction Properties

Transactions have four main properties, referred to as ACID properties. Transactions are Atomic, Consistent, Isolated, and Durable.

Atomic

Implies that the transaction is a unit of work consisting of many operations as a single unit and that it succeeds or fails as one unit of work. In our bank transfer example, the debit operation and the credit operation are an atomic unit.

Consistent

Means that the effect of the transaction should leave the system in a known and stable state. The state of the system must be consistent regardless of whether the transaction succeeds or fails. In our example, the bank accounts must be in a consistent state whether the debit-credit transaction succeeds or fails. If the transaction succeeds, the accounts should reflect the adjusted amounts. If it fails, the transaction manager restores the accounts to their states before the transaction occurred.

Isolated

Implies that transactions do not intersperse intermediate updates with other transactions. The transactions execute serially even if clients perform them concurrently. Thus, concurrent transaction updates are *isolated*. In our example, if another transaction is acting on the same accounts while our debit-credit transaction is in progress, the other transaction must either complete (succeed or fail) before our transaction is executed or our transaction's operations must be committed or rolled back before the other transaction. The operations of the two transactions must not be interspersed.

Durable

Means that the effects of a transaction persist. In our banking example, after the successful completion of the debit-credit transaction, the effect of the transaction on the accounts is permanent.

16.1.2. Transaction Isolation Problems and Levels

Isolation, the "I" in ACID, can have various levels. Some breaches in strict transaction isolation might be necessary, or even required, in some contexts. Transaction isolation levels refer to the degree to which these isolation violations are allowed or prevented. The following section describes transaction isolation violations and the isolation levels that define the allowed isolation violations in a given transaction context.

16.1.2.1. Transaction isolation violations

It's easier to understand the various levels if you first understand what could go wrong if we didn't use various isolation levels to protect the transaction. Consider the following potential isolation violations that can arise in some transaction contexts:

Dirty read

A dirty read occurs when a transaction reads data that another transaction has changed but not yet committed.

Nonrepeatable read

Nonrepeatable reads occur when the same transaction reads the same data twice with different results. Consider the following scenario. In transaction A, you read data item X. Before transaction A completes, transaction B updates and commits the data in item X. A later operation within transaction A reads the same data again and sees the updated value of the data. The first data read in transaction A is nonrepeatable because transaction B's update has been allowed to intrude into transaction A's context.

Phantom read

A phantom read occurs when you read data, look again, and it is gone, or when you read data, look again, and new data suddenly appears. Consider the following scenario. In transaction A, you execute a query and obtain a result set. Before transaction A completes, transaction B inserts a row that matches the query used in transaction A, and this insert is allowed to be committed before transaction A completes. If transaction A executes the same query again, it will obtain a different set of data. This is a phantom read situation because, despite the supposed isolation of transactions, new data not created by transaction A has appeared in the underlying resource.

16.1.2.2. Transaction isolation levels

A transaction isolation level describes the degree to which the access to a resource manager by a transaction is isolated from the access to the resource manager by other concurrently executing transactions. This section describes the various transaction isolation levels, as well as the transaction isolation violations that they prevent or allow.

Read uncommitted

No isolation is provided when a transaction can read uncommitted data. This transaction isolation level is not useful for most business applications that require the ACID transaction properties. However, some data warehouse applications, especially those that principally read rather than update data, find this transaction isolation level useful. Dirty read, nonrepeatable read, and phantom read isolation violations are allowed at this isolation level.

Read committed

With this isolation level, a transaction cannot read uncommitted data and therefore dirty reads do not happen. Nevertheless, nonrepeatable reads and phantom reads can occur. This isolation level is the default in many popular databases such as Oracle.

Repeatable read

With this isolation level, dirty reads and nonrepeatable reads do not occur. Phantom reads can still occur.

Serializable

With this isolation level, no isolation violations are allowed. Dirty reads, nonrepeatable reads, and phantom reads do not occur.

[Table 16-1](#) summarizes the relationship between transaction isolation violations and transaction isolation levels.

Table 16-1. Transaction isolation violations and levels

Isolation level	Dirty read	Nonrepeatable read	Phantom read
Read uncommitted	Possible	Possible	Possible
Read committed		Possible	Possible
Repeatable read			Possible
Serializable			

Given its strict enforcement of transaction isolation, should you always select serializable as the transaction isolation level? The answer is no, because transaction isolation level and concurrency are inversely related. The higher the isolation level, the less concurrency you get. With less concurrency, the performance of the system suffers and the system supports fewer users. Therefore, you need to find a balance between the safety of higher isolation levels and the greater performance possible at lower isolation levels. This choice also has an impact on the complexity of your code. With the highest isolation levels, you need to worry less about concurrency issues in your code because the transaction manager is dealing with them for you. With lower isolation levels, you need to worry about the isolation violations that you're letting through, consider whether they might actually occur or not, and potentially deal with them yourself in your application code.

In many cases, the isolation level decision has been made for you. If your application interacts with a single relational database, for example, its transaction manager will have a default isolation level setting that has been optimized by the vendor for typical situations.

16.1.3. Transaction Models

A variety of different transaction models are available. To further complicate matters, each has some slight variations as well. The *flat transaction model* (FTM) is the simplest and most common. It specifies that only one transaction is active at any given time. Can you start another transaction at the same time? This can be handled in two ways. First, you can disallow starting another transaction. This is often too rigid, however. A second commonly implemented strategy is to suspend the current transaction and start a new one. Once the new transaction finishes, the original transaction resumes. In this scenario, the second transaction might commit while the first transaction rolls back. The outcome of the second transaction remains successfully committed.

Variations on the flat transaction model are quite common. One popular variation in the JDBC API is the use of *savepoints*. With savepoints, the resource periodically saves the work done within a transaction. Not surprisingly, the savepoint is the point at which the save occurs. The transaction can

be partially rolled back to a specific savepoint. For more details, see [Chapter 8](#).

Other transaction models, such as the *nested transaction model* (NTM) and *saga*, were created to address some of the shortcomings of the flat transaction model. In NTM, transactions are nested, forming a tree of transactions with multiple transactions active at the same time. You can emulate nested transactions with savepoints. A saga is a sequence of flat transactions that work as a single unit. In the event of a rollback, the partial work is undone by applying compensating transactions to effectively undo each transaction in the sequence. The flat transactions in a saga are not isolated from one another.

A slight variation on these transaction models is the use of *chained transactions*. Some resource managers can be configured to automatically begin a new transaction once the current one is either committed or rolled back. This ensures that all operations performed by a client happen within a transaction, even if the client doesn't explicitly start one. Both the JDBC and JMS APIs implicitly assume a chained transaction model since they expose commit operations in their APIs, but the start of a transaction is assumed in the connection process. In JMS, for example, the client makes a `Connection` and creates a `Session` from it, which implicitly begins a transaction with the underlying message service. A similar model exists in JDBC for creating `Connections` and `Statements`.

Having said all this, the JTA itself remains fairly agnostic about the transaction model implemented by the underlying transaction manager. JTA implementations can support flat or nested transaction models, for example. If an application attempts to use the JTA to create nested transactions and the underlying manager supports only a flat model, a suitable `Exception` is thrown. In practice, most JTA implementations support only a flat transaction model, since this is sufficient for nearly all application contexts. In addition, the EJB specification requires only a flat transaction model for the EJB container's transaction manager. This decision was made in recognition of the fact that flat transactions are sufficient for the majority of applications out there and to allow vendors to more easily implement the transactional capabilities of an EJB container with existing transaction managers.

16.1.4. Distributed Transactions and the JTA

A *local transaction* is a transaction that communicates with just one resource with no transaction monitor involved. The resource manager manages a local transaction at the connection level. An application making a query to a single relational database, for example, can be managed by the RDBMS resource manager as a local transaction. On the other hand, a *distributed* or *global transaction* is a transaction that covers multiple resource managers and is coordinated by a transaction monitor. An application that wants to perform a transaction involving operations against several different database engines and a messaging service, for example, requires a distributed transaction.

Note that it's not the number or distribution of the clients that determines the need for a distributed transaction. When two different components in an application ask the application server to coordinate a transaction that accesses the same resource manager, a distributed transaction is not required. A single resource manager is involved in the transaction, so the application server can manage the request using a local transaction instead of a distributed transaction.

Various protocols and standards support, coordinate, and implement distributed transactions. The most popular standard is the Open Group's XA Protocol (X/Open XA). Others include the Open Systems Interconnect (OSI) transaction Processing protocol defined by the International Organization

for Standards (ISO), and Systems Network Architecture (SNA) LU 6.2 defined by IBM.

The application or the application server leverages the JTA API programmatically or declaratively to manage distributed transactions. In addition to its general-purpose transaction API (found in the `javax.transaction` package), the JTA also provides a standard Java mapping of the X/Open XA protocol in its `javax.transaction.xa` package. This organization of the JTA interfaces allows the JTA to support both XA-based transactional systems as well as those that don't support X/Open XA.

A popular implementation of the JTA by application server vendors uses the Java Transaction Service (JTS). However, JTS usage is not mandatory in J2EE. JTS provides Java bindings for the CORBA Object Transaction Service (OTS). Since JTS is basically "under the covers" of the application server, you as a developer need know only about the JTA, which we describe in this chapter.

16.1.5. Distributed Transaction Scenarios

There are many common distributed transaction scenarios. One such scenario is a distributed transaction that spans a database and message queue. Others we discuss include those that use two or more databases or message servers. Yet another scenario is a distributed transaction initiated by a J2EE client that invokes different J2EE components.

16.1.5.1. Database and message queue

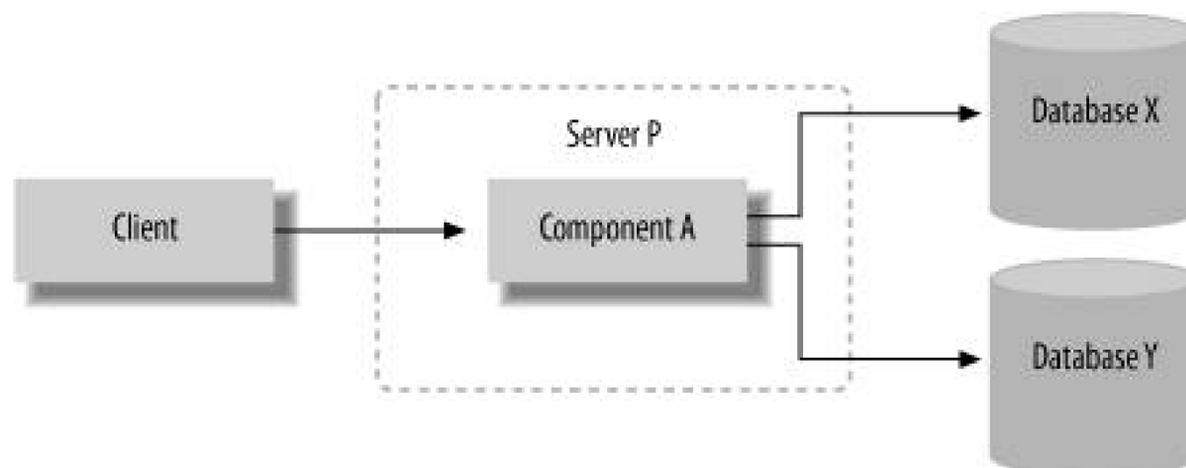
In the distributed transaction scenario depicted in [Figure 16-3](#), a transaction spans a database and a message queue. This scenario is one of the most common distributed transactions in a J2EE application. For example, an application receives a message on a queue and processes the message. The application saves the outcome of processing the message to the database. Both the operations, one on the database and the other on the queue, are part of the same distributed transaction. If the processing of the message fails midstream, for example, we want both the relational database and the message queue to be returned to their states before the transaction was attempted.

Figure 16-3. Distributed transaction: Database and message queue

16.1.5.2. Multiple databases

In the scenario depicted in [Figure 16-4](#), a transaction spans two different databases. An example of this scenario is a transaction that spans updates to a database maintaining the customer master as well as updates to another database maintaining the order master. Both the operations, one on the customer master and the other on the order master, are part of the same distributed transaction.

Figure 16-4. Distributed transaction: Multiple databases



16.1.5.3. Multiple application servers

In the scenario depicted in [Figure 16-5](#), a transaction spans two different application server instances. An example of this scenario is a tax manager component that runs inside one application server that invokes methods of a tax service component running inside another server. The first application server passes along the transaction context to the other server so that the operations that run on the second application server participate in an overall distributed transaction. Both application servers follow the JTA specification, ensuring that the transaction is interoperable between the different server instances, even if the application servers are from different vendors.

16.1.5.4. Client demarcation

In the scenario depicted in [Figure 16-6](#), the client starts a transaction. The client then invokes components (EJBs, web components, etc.) running on one or more application servers. The client container propagates the transaction to the various components that the client invokes. This scenario is similar to the multiple application server scenario, except that an external client initiates the transaction context rather than code running on one of the application servers.

Figure 16-5. Distributed transaction: Multiple application servers

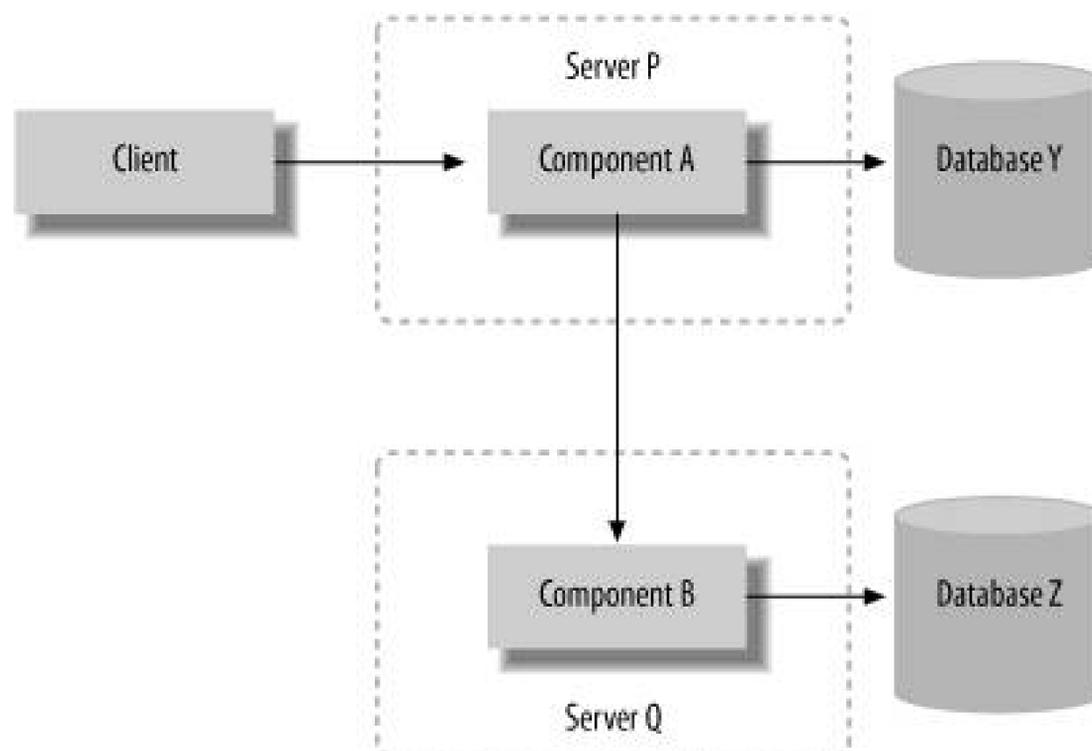


Figure 16-6. Distributed transaction: Client demarcation

These distributed transaction scenarios illustrate the necessity of distributed transaction support in J2EE. The J2EE application server becomes the coordinator of distributed transactions. It coordinates the transaction across different resources, servers, and components. But how does a distributed transaction work under the covers? What is the protocol that the transaction coordinator uses to coordinate disparate resources? The protocol that the transaction coordinator uses is known as *two-phase commit*, which we discuss next.

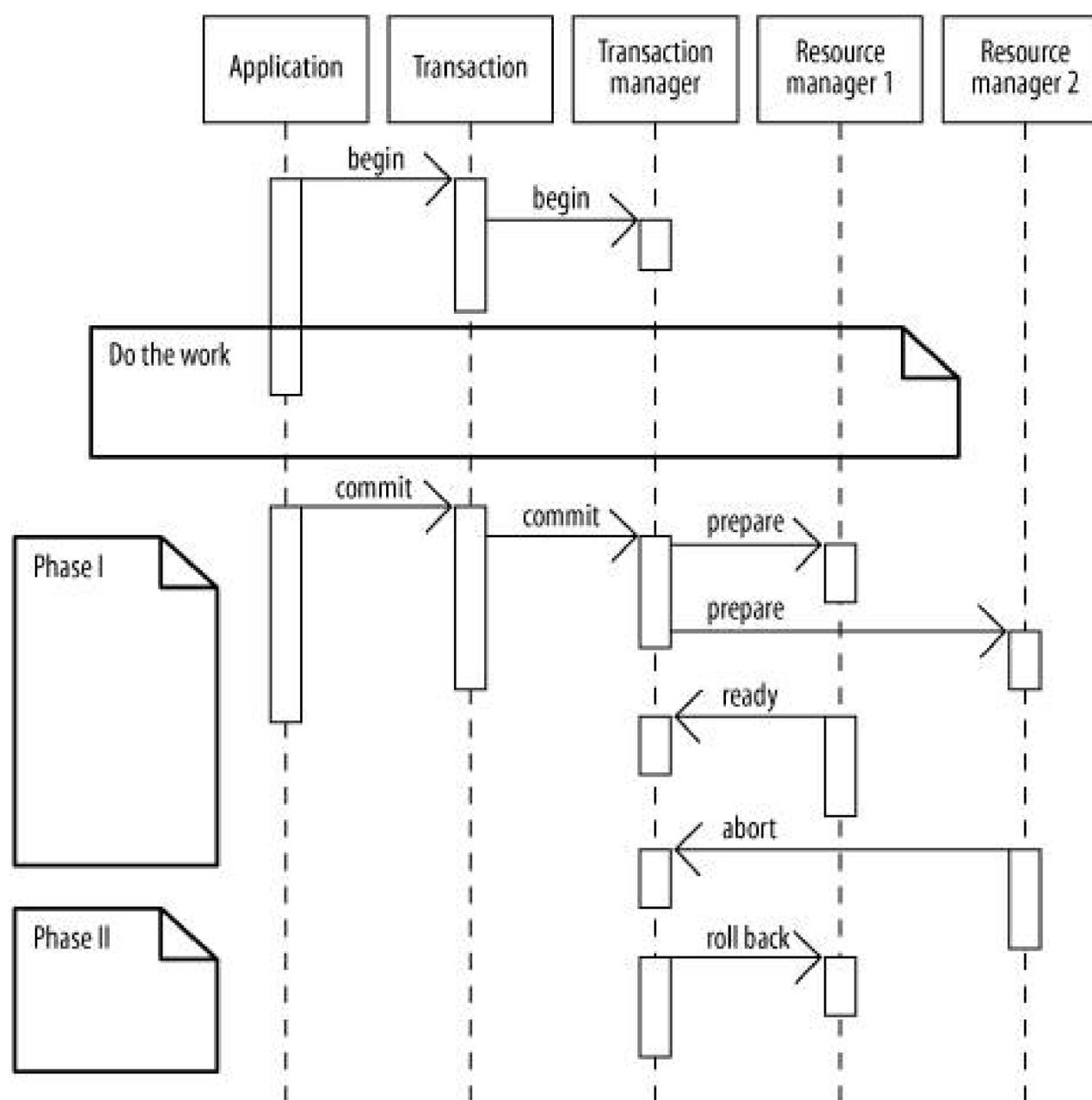
16.1.6. Two-Phase Commit

Two-phase commit is the protocol used to implement distributed transactions between different resource managers. The two-phase commit protocol ensures the consistency and durability of distributed transactions by coordinating the states of the various resource managers involved in the transaction. The two-phase commit protocol kicks in when a commit or rollback is initiated on a distributed transaction. The commit or rollback can be requested by either the application itself or by the underlying transaction manager on behalf of the application.

As the name implies, there are two distinct phases in the two-phase commit protocol. The first phase is known as the *prepare phase*, and the second phase is known as the *commit phase* or *rollback phase*. In the prepare phase, the transaction manager sends a prepare (to commit) message to every resource participating in the transaction. Each resource replies back with either a ready response or an abort response. In the second phase, if any resource responds with abort, the transaction manager rolls back the whole transaction as depicted in [Figure 16-7](#).^[*] If all the resources respond as ready, the transaction manager commits the entire transaction as depicted in [Figure 16-8](#). Instead of issuing a commit, the application may issue a rollback as depicted in [Figure 16-9](#).

[*] This figure and the two that follow use the UML sequence diagram format.

Figure 16-7. Rollback in the first phase of a two-phase commit

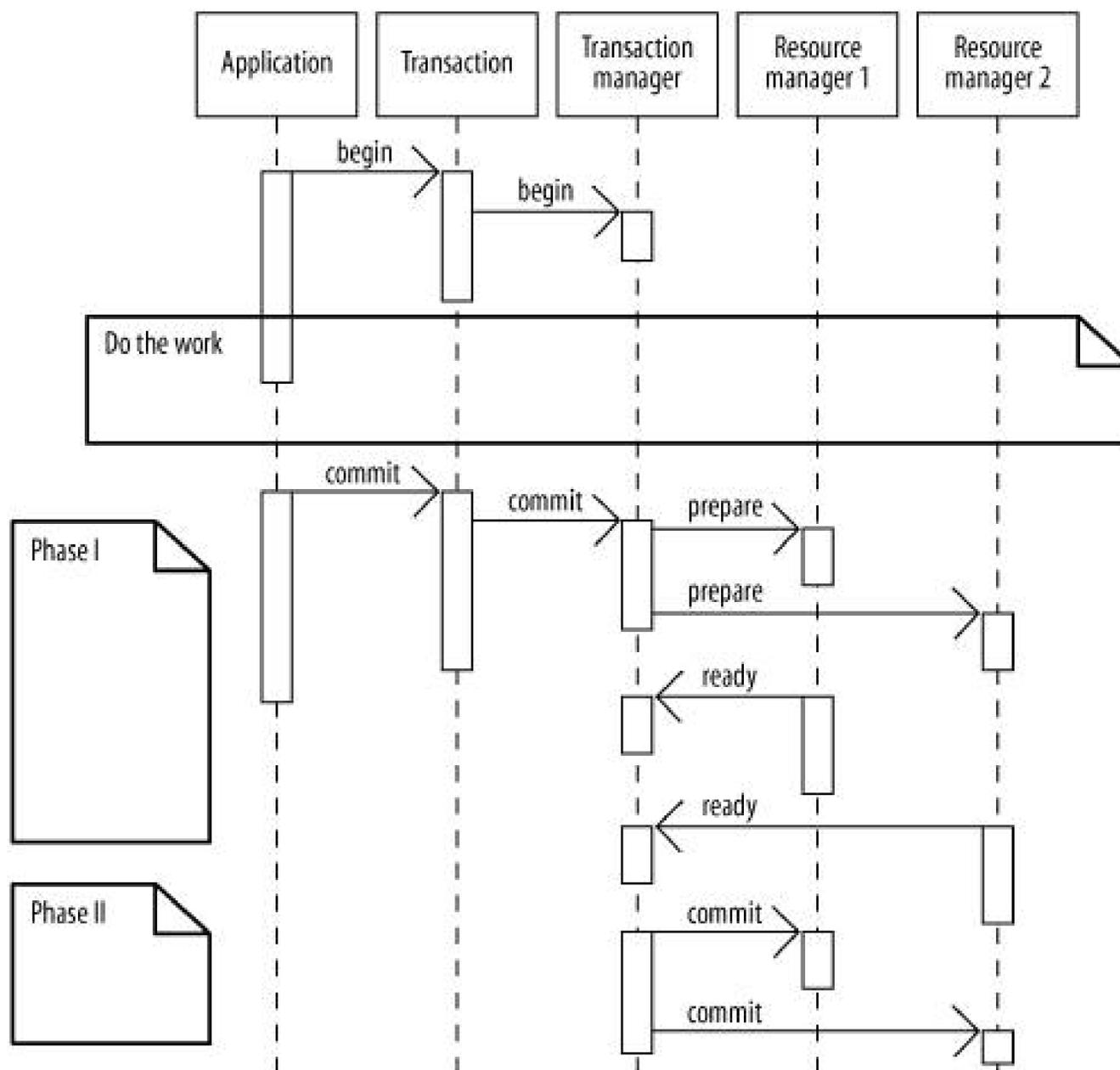


What happens if the transaction manager never receives a response from one or more of the underlying resource managers? Every transaction has a timeout value, settable using the `transactionManager.setTimeout()` method. If a response is not received from a resource manager before this timeout expires, the transaction manager rolls back the entire

transaction.

Even the protection of the two-phase commit protocol can't prevent unrecoverable issues during a transaction. One or more of the resource managers involved in a distributed transaction might lose communications with the transaction manager and decide to make a local decision to commit or roll back. A unilateral decision by a resource manager to commit or roll back is called a *heuristic decision*, and it can

Figure 16-8. Successful commit in two-phase commit



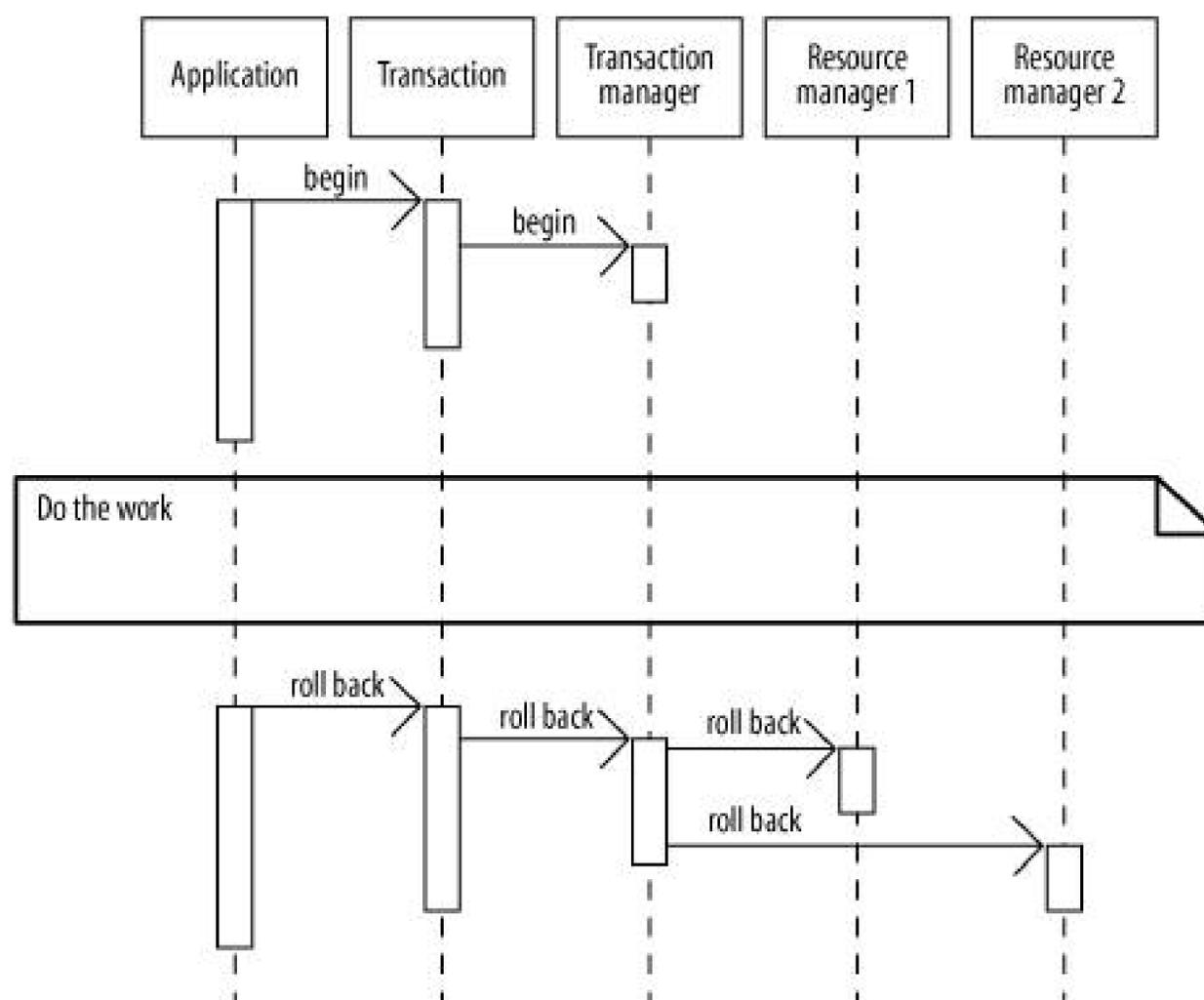
render a distributed transaction inconsistent. An even more catastrophic situation, like a system crash on the transaction manager or one of the resource managers, could prevent even a heuristic decision from being made. In these rare situations, you have to resort to the usual array of disaster recovery measures. Transaction logs from the resource managers and transaction manager can be reviewed to restore a consistent overall state, and/or data backups can be used to restore the distributed system to a previous stable state.

16.1.6.1. Emulating two-phase commit

The resources participating in a two-phase commit must be XA resources in other words, they must support the X/Open XA distributed transaction protocol. Can a non-XA resource participate safely in a distributed transaction? Normally, the answer is no, but there are workarounds.

Consider the following real-world example in which a non-XA resource needs to participate in a distributed transaction. You encounter a situation in which the program receives a message from a queue and updates data in a MySQL database. However, MySQL does not support XA and certain older versions of MySQL do not support transactions at all. For the data source that connects to the MySQL database, the transaction manager (in the container and/or application server) can emulate the two-phase commit protocol by responding on behalf

Figure 16-9. Transaction rollback in two-phase commit



of the underlying non-XA resource manager during two-phase commits. When emulating two-phase commit, a transaction manager may, for example, always return a *ready* response during the prepare phase on behalf of the resource manager. However, the guarantees provided by the two-phase commit protocol are diminished in this situation. The non-XA resource might be left in an inconsistent state with compromised data integrity (it might not have been ready when the transaction manager said it was). Obviously, if you are using a container or application server that supports two-phase commit emulation, you should use it with care. In addition, some emulating transaction managers put restrictions on their support for this, such as allowing one and only one non-XA resource to participate in a distributed transaction.



16.2. Programmatic Transactions Versus Declarative Transactions

You can approach coding transactions in two basic ways: programmatically or declaratively. When possible, declarative transactions are a better choice since they allow the container to manage the transaction for you, and this saves you from having to put JTA transaction management calls into your application code. But as we'll see, not every scenario can be adapted to use declarative transactions. We'll start by examining programmatic transaction management, then discuss how declarative transaction management can supplant some programmatic calls.

16.2.1. Programmatic Transactions

Typically, you implement transactions in an application using one of the resource-specific transaction APIs such as JDBC or JMS. If the application requires distributed transactions, you implement the transaction code using JTA.

The JTA can be used in a number of programming contexts, including web and EJB components, Swing- or SWT-based desktop clients, and many others. The application code begins a transaction and then performs some operations against one or more resource managers (databases accessed via JDBC, message servers accessed through JMS, EJB components accessed via the EJB API, etc.). At some point, the application decides to either commit or roll back the transaction. The container managing the application's runtime environment, including its transaction manager, propagates the transactional context to the different components and resources that the client directly or indirectly invokes. [Figure 16-6](#) depicts client-managed transaction management.

The JTA exposes these capabilities to applications through the `javax.transaction.UserTransaction` interface. [Example 16-1](#) illustrates client-managed transaction management in a generic client application. A JTA client acquires a `UserTransaction` from the transaction manager through a JNDI lookup using a standardized name, `java:comp/UserTransaction`. The JNDI context used has to be the same context accessed by the transaction manager handling the distributed transaction (e.g., the application server managing web or EJB components). Once the client has a `UserTransaction`, a transaction is explicitly started by invoking the `begin()` method of a `javax.transaction.UserTransaction` instance. The transaction is committed by invoking the `commit()` method and rolled back by invoking the `rollback()` method of the `UserTransaction` instance. The same `UserTransaction` reference can be used to start additional transactions if needed. If the underlying transaction manager supports nested transactions, you can begin a nested transaction by calling `begin()` on the `UserTransaction` before committing or rolling back the current transaction. If the transaction manager doesn't support nested transactions, attempting to create one this way will result in a `javax.transaction.NotSupportedException`.

Example 16-1. Client-managed transaction demarcation

```

InitialContext aInitialContext = new InitialContext( );
UserTransaction aUserTransaction = (UserTransaction)
    aInitialContext.lookup("java:comp/UserTransaction");
try {
    aUserTransaction.begin( );

    PersonHome aPersonHome = (PersonHome)
        aInitialContext.lookup("java:comp/env/ejb/Person");
    Person p = aPersonHome.create(aPersonValue.getEmail( ),
        aPersonValue.getPassword( ));
    p.setWorkPhone(aPersonValue.getWorkPhone( ));
    p.setHomePhone(aPersonValue.getHomePhone( ));

    sLogger.info(" created a person with id : " + p.getPersonId( ));
    aUserTransaction.commit( );
} catch (Exception e) {
    try {
        aUserTransaction.rollback( );
    } catch (Exception aException) {
        String aMessage =
            "Exception occurred in client command.execute transaction rollback";
        sLogger.log(Level.SEVERE, aMessage, aException);
        throw aException;
    }
    String aMessage = "Exception occurred in client command.execute ";
    sLogger.log(Level.SEVERE, aMessage, e);
    throw e;
}

```

16.2.2. Declarative Transactions

Declarative transaction management is similar in concept to declarativesecurity management as discussed in [Chapter 10](#). In your component deployment descriptors, you declare whether and where the underlying transaction manager should intercept resource manager requests made by the application code and perform the necessary transaction management (begin, suspend, commit, or roll back transactions). Declarative transaction management can also include the specification of transactional parameters, such as isolation levels. Declarative transaction management facilities are available in EJB deployment descriptors and in certain Hibernate configuration parameters. Refer to [Chapter 6](#) and [Chapter 20](#), respectively, for specific details on declaratively configuring transactions in these contexts. [Appendix A](#) provides details on the structure of EJB deployment descriptors as well.

Declarative transaction management can lead to much cleaner separation of business logic and transaction code, and applications written to depend on declarative transaction management can be much easier to maintain and evolve. All of the complexity of transaction management is ceded to the transaction manager, leaving strictly business-related code in the application. However, as mentioned earlier, in some circumstances, declarative transactions are insufficient and programmatic transactions are required.



16.3. Optimistic Concurrency

Consider the following archetypical example. A user selects a widget to edit. The application displays details about the widget for editing. The user changes the widget's attributes and submits the changes. This process involves two distinct transactions. The application obtains the widget for the user to edit in the first transaction. The application updates the widget in the database using a second transaction, committing the first user's changes. Now, what if another user edited the same widget before the second transaction of the first user was committed. The second user's changes are lost! This application behavior is often referred to as the *last update/write wins policy*. It's also sometimes called the *lost update problem*. It's likely that this behavior is not intentional since the changes are lost silently.

Why don't transaction isolation levels help here? Transaction isolation levels apply only to concurrent transactions. In our case, we're talking about distinct transactions that conflict with each other in terms of outcomes. If the application performed both operations within a single transaction, isolation levels could be used to ensure that two users working concurrently would not conflict with each other. But with the work broken down into two separate transactions for each user, the lost update problem is a possibility.

One simple and potentially elegant solution to this problem is optimistic concurrency. Before every update, you simply ensure that another transaction has not updated the data. There are many ways to ensure this behavior. The most common way is to use a version identifier. Revisiting the archetypical example, a user selects a widget to edit. The application displays the details about the widget to the user for editing. Additionally, the application also keeps track of the version identifier associated with the widget, perhaps using a field in the relational database storing the widget data. The user changes the widget's attributes and submits the changes along with the original version identifier. The changes are successful only if the submitted version identifier matches the version identifier in the database. Upon successful update, the version identifier is incremented. However, what if a second user edits the same widget before the first user submits her changes? The action by the second user updates the version identifier. When the first user submits the changes, the application rejects the update since the version identifiers do not match. This prevents the first user from overriding the changes made by the second user. Thus, the changes made by the second user are not lost! Typically, the first user is presented with the state of the widget as-is (or a merged state) and is asked to submit the update again.

Using a version identifier is a very common technique. Other techniques, with their own advantages and disadvantages, are possible including using a timestamp, using checksums, and checking all the data in the previous row.

This technique is called "optimistic" concurrency because the application is designed to be optimistic that the likelihood of conflict is minimal by splitting its operations into two separate, independent transactions. An alternative approach is pessimistic concurrency: you assume that the likelihood of contention for data is highly likely, and you prevent the conflict explicitly by using a single transaction and high isolation levels or by locking the data under contention when it is accessed by the first user.

The J2EE specifications do not specify a standard implementation of optimistic concurrency. Instead, your application code implements it, using approaches such as the one just described. In addition,

the application can delegate the concurrency-handling logic to the database, using database triggers and/or stored procedures. Application server vendors may also provide proprietary extensions that implement and support optimistic concurrency.



16.4. EJB Transaction Management

In this section, we discuss some extended topics related to transaction management with EJB components. This material complements the introductory section on EJB transaction management found in [Chapter 6](#). If you are specifically concerned with transaction management related to EJBs, we recommend that you read "Transaction Management" in [Chapter 6](#) since it introduces the basic concepts and some general rules related to both declarative and programmatic transaction management within EJBs. Once you have that material under your belt, if you feel you need to know more, come back to this section for additional coverage.

If you're using strictly declarative transaction management within EJBs, this is referred to as Container-Managed Transaction Demarcation (CMTD). The container is responsible for starting and committing or rolling back the transactions required by the EJB based on the directives specified in the deployment descriptor as well as some programmatic hints by the developer in the EJB code. The container leverages JTA for the transaction management under the covers.

Using programmatic transaction management within EJB components is a bit of a special case, because the EJB container has a default assumption that it is managing transactions on behalf of its components. Programmatic transaction management within EJB components is sometimes referred to as Bean-Managed Transaction Demarcation (BMTD). If an EJB developer chooses to use BMTD in a bean, then the bean is responsible for transaction operations. The transaction operations can be performed through resource-specific APIs such as JDBC and JMS, or they can be done explicitly using JTA. When BMTD is in use, the container's transaction manager still handles all the two-phase commit issues related to distributed transactions that we discussed earlier, but it assumes that the application code is handling the start, commit, and rollback of transactions. As mentioned in [Chapter 6](#), BMTD is not allowed for entity beans, even those that manage their own persistence.

16.4.1. BMTD with JTA

Using bean-managed transaction management doesn't necessarily mean that you are using JTA calls in your EJB code. You could simply be using specific resource manager APIs, like JDBC or Hibernate for databases or JMS for message servers. But if you do find the need to access and use a JTA `UserTransaction` from your BMTD beans (or use any other aspects of the JTA, for that matter), you need to consider a few things.

As we saw in [Example 16-1](#), you can obtain the JTA transaction from the JNDI context of the EJB container. EJB also allows you to access the `UserTransaction` using the `getUserTransaction()` method of the `javax.ejb.EJBContext`, as shown in [Example 16-2](#) for a session EJB.

Example 16-2. Accessing the UserTransaction from the EJBContext

```
SessionContext mSessionContext = . . . ;
```

```
UserTransaction aUserTransaction = mSessionContext.getUserTransaction( );
// Now use the UserTransaction to begin/commit/roll back transactions
. . .
```

If you use BMTD in your bean, it's important to be mindful of how the container is going to manage clients that make requests of your EJB from within their own transactions. If the client starts its own transaction and invokes business methods on your BMTD bean, the EJB container will automatically suspend the client's transaction during the request. Once the request completes, the client's transaction is resumed. So if your EJB performs no JTA operations during the request, the entire request operates without any transaction context. But if your EJB starts a new transaction by calling the `begin()` method on a `UserTransaction`, a new transaction is started and the request is processed under this new transaction context.

Since you are using the JTA API to manage transaction boundaries, the specification prohibits invoking resource-specific APIs. The EJB specification prohibits invoking the `commit()` or `rollback()` methods of the `java.sql.Connection` interface. Similarly, you are not allowed to invoke the `commit()` and `rollback()` methods of the `javax.jms.Session` interface.

The specification also prohibits the `getRollbackOnly()` and the `setRollbackOnly()` methods of `javax.ejb.EJBContext`, because they are related to CMTD, not BMTD.

In addition to these general behaviors and rules for BMTD beans, there are some special caveats related to BMTD for particular types and subtypes of EJBs.

16.4.1.1. Stateless session beans and message-driven beans

Since a stateless session bean by definition does not maintain state information on behalf of the client, a transaction is not allowed to span methods of a stateless bean. The bean method that starts a transaction must complete the transaction by calling the `commit()` method or the `rollback()` method. If you fail to do so, the container rolls back the transaction, discards the stateless bean instance, and throws an appropriate exception back to the client (`java.rmi.RemoteException` for remote clients, `javax.ejb.EJBException` for local clients).

This rule also applies to message-driven beans. If you start a transaction in the `onMessage()` method of a message-driven bean with BMTD, you must call the `commit()` method or the `rollback()` method within the `onMessage()` method to complete the transaction. Otherwise, the container rolls back the transaction and discards the instance of the bean. No exception is thrown back to the client, obviously, since the client has indirectly invoked the bean by sending a message to a JMS destination.

16.4.1.2. Stateful session beans

The behavior of BMTD with a stateful session bean is a bit more involved. Unlike a stateless session bean, a transaction started in one method can be left active at the end of the method, and it remains associated with the stateful session bean instance across subsequent method calls. The transaction remains in effect for the bean until the `commit()` method or the `rollback()` method is called. As mentioned earlier in this section, a container will always suspend any client transactions during

requests to BMTD beans. So as long as the transaction started by the bean remains active, all requests made by clients will operate under this transaction context, until it is committed or rolled back.

16.4.2. Container-Managed Transaction Demarcation (CMTD)

When an EJB is using CMTD, it is prohibited from invoking JTA operations in any contexts in which they might interfere with the container's transaction management activities. A bean with CMTD must not invoke the `getUserTransaction()` method of `javax.ejb.EJBContext`. If it does, the container will throw a `java.lang.IllegalStateException`. Additionally, the bean must not invoke any `UserTransaction` methods from within its business methods, message-listener callbacks, or `ejbTimeout()` callbacks. In these same situations, the bean is forbidden from calling `commit()` or `rollback()` on a `JDBC Connection` or `JMS Session`.

16.4.2.1. Inducing rollbacks in CMTD

Although application code is not allowed to directly interfere with transaction management under CMTD, your EJB can induce a transaction rollback by calling the `setRollbackOnly()` method of the `javax.ejb.EJBContext`. At the end of the transaction, the container automatically rolls back instead of committing the transaction, regardless of the state of the transaction. You can query whether the current transaction has been marked for rollback by calling the `getRollbackOnly()` method of the `EJBContext`. Calling the `getRollbackOnly()` method can be useful for avoiding expensive chunks of transactional code in your EJB. If the results of the code are going to be rolled back regardless, you can skip the processing then and there.

The container will also roll back a transaction when the EJB code throws a runtime exception. The container catches the runtime exception and rolls back the transaction. Note that the exception must be a runtime exception, not a general application exception. A common runtime exception thrown in these situations is `javax.ejb.EJBException`.

16.5. Some Common Programming Scenarios

In this section, we'll look at some of the most common transaction scenarios and we'll examine ways to leverage transactions effectively.

The subsection "Using a Session Façade" provides a concrete example of how to use CMTD effectively with an EJB component by minimizing the number of unnecessary transactions and increasing performance.

The subsection "Using Transaction Attributes with CMTD" illustrates an approach for utilizing transaction attributes with CMTD EJBs. This approach can help avoid resorting to BMTD in some situations.

The subsection "Distributed Transactions Involving a JMS Destination and Database" provides a detailed example of a distributed transaction using both JMS and JDBC resources.

16.5.1. Using a Session Façade

Consider the code fragment in [Example 16-3](#), which uses a `Person` EJB component. Assume that the EJB has been deployed using the `Required` transaction attribute applied to all of its methods. How many transactions does the user invoke in total? The answer is four. Each EJB method, `create()`, `setWorkPhone()`, `setHomePhone()`, and `getPersonId()`, is invoked in a different transaction. This is most likely not the intended behavior. This code is not only very expensive but also may be incorrect. For example, if there is some error setting the home phone number, the client might expect the creation of a person bean instance to fail.

Example 16-3. Direct usage

```
PersonHome aPersonHome =
    (PersonHome) aInitialContext.lookup("java:comp/env/ejb/Person");
Person p = aPersonHome.create
    (aPersonValue.getEmail( ), aPersonValue.getPassword( ));
p.setWorkPhone(aPersonValue.getWorkPhone( ));
p.setHomePhone(aPersonValue.getHomePhone( ));

sLogger.info("Created a person with id : " + p.getPersonId( ));
```

The most common solution to this problem is using the *session façade* pattern. In brief, this pattern involves creating a stateless session bean that acts as a simple frontend, or façade, to hide the complexity behind it. You refactor the code from the client into a stateless session bean as illustrated

in [Example 16-4](#). This session bean would also be deployed using the `Required` transaction attribute on its methods. If a client invokes the `createPersonSameTransaction()` method, a transaction begins and the methods of the `Person` bean, `create()`, `setWorkPhone()`, `setHomePhone()`, and `getPersonId()`, are now invoked in one single transaction.

Example 16-4. Code fragment from session façade

```
public PersonValue createPersonSameTransaction(PersonValue aPersonValue) {
    return createPerson(aPersonValue);
}

protected PersonValue createPerson(PersonValue aPersonValue) {
    sLogger.info("5 creating person in manager cmd bean ...");

    try {
        InitialContext aInitialContext = new InitialContext( );
        PersonHome aPersonHome =
            (PersonHome) aInitialContext.lookup("java:comp/env/ejb/Person");
        Person p = aPersonHome.create(aPersonValue.getEmail( ),
                                     aPersonValue.getPassword( ));
        p.setWorkPhone(aPersonValue.getWorkPhone( ));
        p.setHomePhone(aPersonValue.getHomePhone( ));

        sLogger.info("Created a person with id : " + p.getPersonId( ));
        aPersonValue.setPersonId(p.getPersonId( ));
    } catch (Exception e) {
        String aMessage =
            "exception in manager cmd bean.createPersonNewTransaction ";
        sLogger.log(Level.SEVERE, aMessage, e);
        throw new EJBException(aMessage, e);
    }

    return aPersonValue;
}
```

Is there another way to approach this problem? An alternative is to use client-managed transaction demarcation as illustrated in [Example 16-5](#). The EJB methods are called directly from the client, but the client begins and ends its own singular transaction. [Example 16-5](#) illustrates this approach. While this does solve the problem, the client code has become more complex by taking on programmatic transaction management. Unless this is strictly necessary (e.g., the client has very specific transaction boundary needs that don't apply in general), the session façade approach is preferable.

Example 16-5. Client-managed transaction demarcation usage

```
InitialContext aInitialContext = new InitialContext( );
```

```

UserTransaction aUserTransaction = (UserTransaction)
    aInitialContext.lookup("java:comp/UserTransaction");
try {
    aUserTransaction.begin( );

    PersonHome aPersonHome =
        (PersonHome) aInitialContext.lookup("java:comp/env/ejb/Person");
    Person p = aPersonHome.create(aPersonValue.getEmail( ),
        aPersonValue.getPassword( ));
    p.setWorkPhone(aPersonValue.getWorkPhone( ));
    p.setHomePhone(aPersonValue.getHomePhone( ));

    sLogger.info(" created a person with id : " + p.getPersonId( ));
    aUserTransaction.commit( );
} catch (Exception e) {
    try {
        aUserTransaction.rollback( );
    } catch (Exception aException) {
        String aMessage =
            "Exception occurred in client command.execute transaction rollback";
        sLogger.log(Level.SEVERE, aMessage, aException);
        throw aException;
    }
    String aMessage = "Exception occurred in client command.execute ";
    sLogger.log(Level.SEVERE, aMessage, e);
    throw e;
}

```

16.5.2. Using Transaction Attributes with CMTD

Consider the following example. In a very simple object model, there is a `Person` EJB and a `PersonStatus` bean. You would like to create a `Person` with `PersonStatus`. However, you would like the creation of `Person` to succeed regardless of whether the `PersonStatus` is valid. Suppose this part of the EJB is implemented in the form of a `createPersonAndStatus()` method, as shown in [Example 16-6](#).

Example 16-6. Create Person and PersonStatus

```

public PersonValue createPersonAndStatus(
    PersonValue aPersonValue,
    PersonStatusValue aPersonStatusValue) {
    aPersonValue = createPersonNewTransaction(aPersonValue);
    aPersonStatusValue = createPersonStatus(aPersonValue,
        aPersonStatusValue);
    return aPersonValue;
}

```

If your bean is using CMTD and you wanted to enforce the requirements as detailed, you might set the transaction attribute for the `createPersonAndStatus()` method to `Required` and the transaction attribute for `createPersonNewTransaction()` method to `RequiresNew`. You might expect that this would cause a new transaction to be created for the `createPersonNewTransaction()` method, suspending the one on `createPersonAndStatus()`.

But does the code fragment in [Example 16-6](#) work as intended? Is the `createPersonNewTransaction()` method executed in a new transaction? No! The container uses the same transaction for both methods because it never has an opportunity to intercept the second method call and start a new transaction. [Example 16-7](#) illustrates how to invoke the second EJB method through the container, allowing it to apply the intended transaction attributes.

Example 16-7. Create Person with interception

```
public PersonValue createPersonAndStatusNew(
    PersonValue aPersonValue,
    PersonStatusValue aPersonStatusValue
) throws RemoteException {
    aPersonValue = ((ManagerCMTD) mSessionContext.getEJBObject( ))
        .createPersonNewTransaction(aPersonValue);

    aPersonStatusValue = createPersonStatus(aPersonValue,
        aPersonStatusValue);

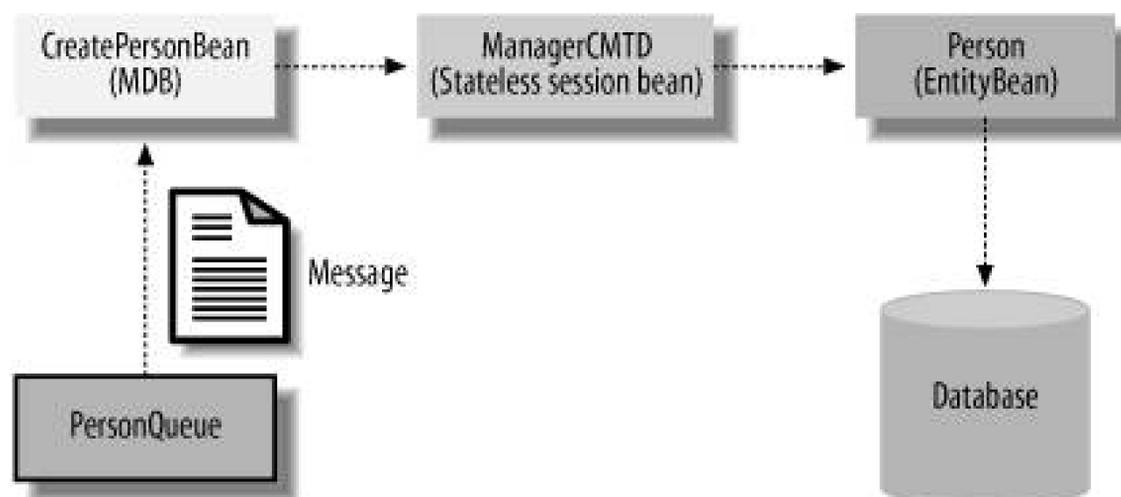
    return aPersonValue;
}
```

Using the session context, you obtain a client reference to the `EJBObject` being invoked. You cast the `EJBObject` stub to the correct type (`ManagerCMTD`, in this example) and then invoke the `createPersonNewTransaction()` method. Since you invoke the method on a client stub acquired from the container (just like any other client), the method request will be intercepted by the container, and it has an opportunity to apply the requested transaction attributes.

16.5.3. Distributed Transactions Involving a JMS Destination and Database

As discussed in the "[Database and message queue](#)" section earlier in this chapter, a distributed transaction involving a message destination and a database is a common use of distributed transactions in enterprise applications. Here is a detailed example of such a case. In this example, the application retrieves and processes a message from a queue and inserts the data into the database. The entire process, as illustrated in [Figure 16-10](#), from the message retrieval to the database insert, happens in one transaction.

Figure 16-10. Distributed transaction detailed example



An instance of a message-driven bean, `CreatePersonBean` retrieves a message from `PersonQueue`. The message is of type `javax.jms.ObjectMessage`. In the `onMessage()` callback, the message-driven bean retrieves an instance of `PersonValue` from the object message. The bean invokes the `createPersonSameTransaction()` method on the `ManagerCMTD` stateless session bean as shown in [Example 16-8](#). The `ManagerCMTD` bean persists the data to the database using the `Person` entity bean.

Example 16-8. CreatePersonBean message-driven bean

```

ObjectMessage aObjectMessage = (ObjectMessage) aMessage;
PersonValue aPersonValue = (PersonValue) aObjectMessage.getObject();
InitialContext aInitialContext = new InitialContext();
ManagerCMTDHome aManagerCMTDHome =
    (ManagerCMTDHome) aInitialContext.lookup
        ("java:comp/env/ejb/ManagerCMTD");
ManagerCMTD aManagerCMTD = aManagerCMTDHome.create();
aPersonValue = aManagerCMTD.createPersonSameTransaction(aPersonValue);
sLogger.info("Created a person with id : " + aPersonValue.getPersonId());
  
```

To ensure that the message retrieval happens in a transaction, you use CMTD for the `CreatePersonBean` as shown in an excerpt from the EJB deployment descriptor in [Example 16-9](#). If there are any errors or exceptions in the entire process, the entire set of operations in the transaction is rolled back, including message retrieval from the JMS destination. If configured this way, the container redelivers the message so that the application makes another attempt to process the message.

Example 16-9. CMTD configuration for CreatePersonBean

```
<container-transaction>
  <description>
    Specify all create person bean methods to run in a transaction
  </description>
  <method>
    <description>All methods of create person Bean</description>
    <ejb-name>CreatePerson</ejb-name>
    <method-name>*</method-name>
  </method>
  <trans-attribute>Required</trans-attribute>
</container-transaction>
```



16.6. Transaction Best Practices

We recommend the following transaction management best practices:

- Use declarative transaction management rather than programmatic transactions whenever possible.
- When using declarative transactions with EJBs, consider using `Required` as the first choice for a transaction attribute.
- When using distributed transactions, use XA-enabled resources and XA-enabled drivers whenever possible. Be aware of the limitations of emulating XA for a non-XA resource.
- Avoid mixing direct transaction management using JTA with implicit transaction management done through a resource-specific API such as JDBC or JMS since this complicates the transaction management picture.
- Consider leveraging the session façade design pattern for creating coarse-grained transactions.
- Minimize the use of client-managed transactions wherever possible. The session façade can be useful here.
- Carefully evaluate the need and implementation of an optimistic concurrency strategy.

Transactions help you to manage very complex, and even unpredictable, system behavior when dealing with shared networked resources. But as this chapter has demonstrated, you must learn to leverage transactions in the most efficient and straightforward manner possible. To that end, having a solid understanding of transaction terms and concepts such as ACID properties, local versus distributed transactions, the flat transaction model, two-phase commit, isolation problems and levels and optimistic concurrency is essential.

Part II: Open Source Enterprise Tools

The chapters in [Part II](#), like the ones in [Part I](#), serve as both introduction and tutorial for key enterprise tools. But in this part, each chapter covers a tool that has become a "best practice" among enterprise Java developers, without actually being a standard Java or J2EE application. Some of these tools will never be standards, some will advise future standards, some are already being adapted into Java standards. But regardless, they are invaluable tools and have proven their worth through their wide adoption in the Java community.

[Chapter 17](#), *Ant*

[Chapter 18](#), *JUnit and Cactus*

[Chapter 19](#), *Struts*

[Chapter 20](#), *Hibernate*

[Chapter 21](#), *Annotations with XDoclet and J2SE Metadata*

Chapter 17. Ant

Enterprise software development involves a (sometimes complex) combination of source code, configuration files, HTML pages, images, and various other assets that need to be processed, combined, and deployed in order to create a running application. In addition to compiling and running code, a number of other tasks need to be performed to manage an enterprise project: generating documentation, running tests against code, interacting with source control systems, and more.

Over the years, several tools have emerged to help manage the various repetitive tasks involved with developing software. Earlier examples include *make* and *gnumake*, command-line tools that have been used to manage building and packaging code, primarily on Unix systems. *make* and its variants utilize a text-based command file, a *makefile*, that you use to specify how various tasks (compiling code, generating executables, etc.) are to be carried out, as well as dependencies between various tasks (e.g., an executable should be generated only if all of the compiled code is up-to-date).

In Java development projects, Ant has emerged as the de facto tool of choice for these types of tasks. Integrated development environments (IDEs) typically provide build/deploy tools, but Ant supports these and other development tasks in an independent manner, separate from any specific IDE. This, along with the nearly ubiquitous support in Java IDEs for editing and exercising Ant buildfiles, has made Ant a very popular tool for Java development projects. In this chapter, we provide a brief overview of the basics of Ant and how to write an Ant build script, followed by specific details on how Ant can assist in enterprise development projects, in terms of building, assembling, and deploying J2EE components as well as other key development tasks.

This tutorial is meant to be just that: a tutorial. We don't attempt to cover all aspects of the Ant build model, and we provide details on only a subset of the core tasks that are bundled with Ant. For more complete coverage, we refer the reader to the official documentation on the Ant project site (<http://ant.apache.org>) and to *Ant: The Definitive Guide* by Steve Holzner, an O'Reilly book devoted to all things Ant.

[← PREV](#)

17.1. What Version Is Covered Here?

The examples in this chapter were developed and tested using Ant 1.6.5, downloadable from the Ant project site at Apache (<http://ant.apache.org>). Most of these examples should work with earlier versions of Ant, but in some cases modifications to the scripts will be necessary to make them backward-compatible, since some core tasks have changed from previous minor versions (1.5.x, 1.4.x), and the property-handling mechanism in Ant has also evolved somewhat from earlier versions. Wherever possible, we've highlighted aspects of the sample buildfiles that are specific to Ant 1.6 and therefore would need to be ported to older versions.

[← PREV](#)

17.2. Ant Overview

Ant was developed in part as an improvement to previous build tools like *make*. At a high level, the improvements it offers include an improved command file format (using XML as the basis for its buildfiles) and the ability to extend the tool using Java code rather than shell script commands. In the past, many developers found the text format of *make* and its variants to be somewhat arcane and error-prone (to the point that people developed tools like *imake*, which provided a metalanguage for the generation of makefiles). An XML format is much more directly accessible to the average developer, the syntax rules are clearer since they are based on general XML syntax, and the plethora of XML tools can be used directly to edit Ant buildfiles.

The ability to extend the build tool using Java code is also an obvious advantage, at least to Java developers. As we'll see later in this section, tasks in Ant are invoked using XML elements in the buildfile with a core set of tasks provided with the tool. Custom Ant tasks can also be written using Java code. This allows for more portable extensibility than earlier shell command systems, since the syntax of shell commands and their arguments tends to be platform-dependent.

Before we look at the model Ant uses for managing development tasks, let's get familiar with the terminology of Ant.

Ant is driven from a *buildfile* that you write. This buildfile defines a set of project-specific *targets* that represent specific goals or results that you can request Ant to accomplish. Typical targets might be to compile a set of Java code, to create a jar archive from a set of files, or to check in a set of files to a source control system like CVS, Perforce, or Subversion. Targets can be hierarchical in nature (i.e., one target may depend on the results of other targets, and a target may directly invoke other targets in order to accomplish its goal). A target that generates a jar file may depend on another target that compiles a set of Java files into classes, and it may invoke another target that moves all of the necessary assets into a temporary area before creating the archive.

Ant executes *tasks* in order to complete specific targets. A task in Ant can be just about any action that you can execute from Java, or the system, or both: compile a Java file, move a file from one directory to another, open a URL and save the contents, and so on. Ant provides a set of core tasks that perform very common operations, like running the Java compiler on a set of Java files, moving files, and deleting files. You can also write your own custom Ant tasks if necessary, as mentioned earlier.

Looking at Ant's build model from the bottom up, it consists of tasks, which are well-defined, general purpose utilities. A set of core tasks is included natively in Ant, and custom tasks can also be defined if needed. Tasks are strung together to define how various targets are accomplished. A project's targets are defined in a buildfile, which is named *build.xml* by default, although the file can be named anything you like.

17.3. Ant Fundamentals

Before we look at using Ant for tasks specific to enterprise application development, let's run through the basic mechanics of Ant, for readers who haven't used Ant before. If you've used Ant before, you can skip this section and jump right to "[Enterprise Tasks](#)" later in this chapter.

17.3.1. Buildfiles

When you execute Ant, you specify a command file that it should use to load the definitions of targets. The buildfile uses an XML format to define the targets and the tasks involved in running each target. The root XML element of the buildfile is always a `<project>` element, which defines the name of the project and the default target that the buildfile will execute (if you don't specify one when invoking Ant). The `<project>` element in Ant buildfiles supports the attributes listed in [Table 17-1](#).

Table 17-1. Project element attributes

Attribute name	Purpose	Required
<code>name</code>	Descriptive name of the project, sometimes used by IDEs and other tools to describe a particular project buildfile. The project name can be accessed in the buildfile using the <code>ant.project.name</code> built-in property.	No
<code>default</code>	Name of the default target to be invoked when Ant is invoked on this buildfile with no specific target specified.	Yes
<code>basedir</code>	The root directory where Ant will execute all targets and tasks in this buildfile. This base directory will be the starting point for any relative paths used in the buildfile. The base directory can also be specified using the <code>basedir</code> built-in property. This same property can be used to access the project's base directory in the buildfile. If the base directory isn't specified, it defaults to the directory containing the buildfile.	No

Within the root `<project>` element, you define the targets for your project, with each target consisting of a series of tasks to execute in order to satisfy the overall target. [Example 17-1](#) shows a very basic Ant buildfile that contains a single target, `hello`, that simply prints a message to the console.

Example 17-1. An extremely simple Ant buildfile

```
<!-- Simple Ant buildfile; default (and only) target just
      prints "Hello, world" to the console -->
<project name="JEnt-Ant-simple" default="hello">
  <target name="hello">
    <echo message="Hello, world"/>
  </target>
</project>
```

To execute targets from this buildfile, we would usually place it in a directory with a default filename of *build.xml* and invoke Ant on the command line from the directory with the *build.xml* file:

```
> ant hello
Buildfile: build.xml

hello:
    [echo] Hello, world

BUILD SUCCESSFUL
Total time: 1 second
```

In this degenerately simple Ant buildfile, the project is named `JEnt-Ant-simple`, according to the `name` attribute on the `<project>` element. The `default` attribute on the `<project>` element is used to specify the name of the default target that is invoked if this buildfile is run with no target specified. Within the `<project>` element, we've defined a single target using a `<target>` element. The target is named using the `name` attribute. Finally, within the target, we've defined the tasks that should be executed when this target is invoked. In this case, we're using a built-in Ant task named `echo`, which echoes a text message to the console.

When we invoked the `ant` command, Ant first looked in the current directory for the default buildfile, *build.xml*. Once the file was found, Ant parsed the buildfile and searched for a target with a name of `hello`, the target specified on the command line. Once it found that target in our buildfile, it began executing the tasks defined within that target. In our case, Ant had to execute only one task, the `echo` task that prints our message to the console.

Since we specified that the `hello` target should be the default, we could also invoke the target by just running Ant with no arguments:

```
> ant
```

This should result in the same console output. If we named our buildfile something other than the default of *build.xml*, we could specify the name of the buildfile with the `-buildfile` option:

```
> ant -buildfile build.xml hello
```

The command-line arguments `-file` and `-f` are synonyms for the `-buildfile` argument.

17.3.2. Defining Targets, Their Tasks, and Their Relationships

As we saw in our very simple example, targets are defined in the Ant buildfile as child `<target>` elements of the root `<project>` element. The `<target>` element supports the attributes shown in [Table 17-2](#).

Table 17-2. Target element attributes

Attribute name	Purpose	Required
<code>name</code>	The name of the target, used to reference this target elsewhere in the buildfile and to invoke it from the command line.	Yes
<code>depends</code>	A list of targets (comma-delimited) on which this target depends. When this target is invoked, its list of predecessor targets will be invoked before the tasks in this target are invoked. The predecessor targets are executed in the order they are listed in the <code>depends</code> attribute, and the dependencies are handled recursively.	No
<code>if</code>	This target will be executed only if the property named by this attribute has been set.	No
<code>unless</code>	This target will not execute if the property named by this attribute has been set.	No
<code>description</code>	A readable description of the target, sometimes used by IDEs and other tools to display the contents of a buildfile.	No

A buildfile can simply contain a flat collection of independent targets, but often you will want to make use of target dependencies in order to make things more efficient.

A slightly more involved (and more useful) buildfile is shown in [Example 17-2](#). This sample buildfile demonstrates basic use of target dependencies.

Example 17-2. Simple target dependencies

```
<!-- A simple buildfile that compiles a set of Java sources
      from the src directory, placing the results in the classes
      directory, and can also create a jar from the classes -->
<project name="JEnt-Ant-basic" default="archive">
```

```

<target name="compile"
  description="Compile all code in src dir to the classes dir">
  <!-- Create the classes dir, if needed -->
  <mkdir dir="classes"/>
  <!-- Compile the java code into classes -->
  <javac srcdir="src" destdir="classes"/>
</target>
<target name="archive" depends="compile"
  description="Create an archive from the contents of the classes dir">
  <jar destfile="classes.jar" includes="classes/**"/>
</target>
<target name="clean"
  description="Remove all artifacts created from other targets">
  <!-- Remove the classes dir created by the compile target -->
  <delete dir="classes"/>
  <!-- Remove the jar generated by the archive target -->
  <delete file="classes.jar"/>
</target>
</project>

```

We've defined three targets here: `compile`, `archive`, and `clean`. The `compile` target runs `javac` on a set of code in the `src` directory and puts the generated classes into the `classes` directory. It does this by first using the built-in `mkdir` task to create the `classes` directory (if it doesn't exist) and then running the built-in `javac` task, which runs the Java compiler on the source code directory specified in the `srcdir` attribute and puts the results in the directory named by the `destdir` attribute. Additional details on these built-in tasks can be found in ["Core Tasks"](#) later in this chapter.

The `archive` target creates a jar file from all of the classes in the `classes` directory, using the built-in `jar` task to create the archive. We specified that the `archive` target depends on the `compile` target using the `depends` attribute on the target definition. This ensures that whenever we run the `archive` target, the source code is compiled first and then the archive is created. With this dependency in place, if we invoke the `archive` target:

```
> ant archive
```

Ant detects the dependency, runs the `compile` target, and then runs the tasks in the `archive` target:

```

Buildfile: build.xml
compile:
  [mkdir] Created dir: classes
  [javac] Compiling 1 source file to /jent/examples/ant/ex-basic/classes
archive:
  [jar] Building MANIFEST-only jar: /jent/examples/ant/ex-basic/classes.jar
  [jar] Building jar: /jent/examples/ant/ex2-jar/classes.jar
BUILD SUCCESSFUL
Total time: 7 seconds

```

Finally, we've included a `clean` target that removes all artifacts generated by the other targets in the buildfile. It uses the built-in `delete` task to remove the `classes` directory created by the `compile` target and the jar archive created by the `archive` target. It's good practice to define a target like this in your project buildfiles to make it easy to clear out the cruft left from using the buildfile. This can be helpful, for example, in situations in which you're using a source control system. A developer may check out the project files (including the Ant buildfile), use Ant to build classes and deploy components as he works on the code, and at some point he'll want to check in his changes. He will want to avoid checking in development artifacts, like class files and archives created for deployment purposes. The `clean` target allows you and other developers working on the project to clear out any transient files before checking in their updates.

17.3.3. Properties

So far, all of the targets and tasks we've defined in our examples have been explicitly specified all of the filenames and directory locations have been hardcoded into the buildfile itself. Similar to writing software, the use of literals like this is not a good long-term approach to managing buildfiles. It makes your buildfiles inflexible, and when a change is necessary, it typically needs to be made in several places, which makes errors more likely. In our previous example, the jar file created by the `archive` target is named `classes.jar`. If we wanted to change that to something else (if, for example, another project or library named its archive by the same name), we'd have to make the change to both the `jar` task in the `archive` target (the `destfile` attribute would have to be changed), and the second `delete` task in the `clean` target would need to be changed as well.

Ant supports the use of properties to set configurable parameters used in buildfiles. Properties have a name and a string value, and once they are set, they can be used anywhere a string literal is needed, such as an attribute value on a target or task.

17.3.3.1. Setting properties

Properties can be set in a number of different ways. The most straightforward way is to use the Ant `<property>` element to set a property directly in the buildfile:

```
<project name="sample" default="compile">
  <property name="java.src.dir" value="src"/>
  . . .
```

Note that the `property` task normally sits outside any target definitions so that property values can be used in any targets that need them. You can include the `property` task in your target definitions, but these property settings won't be processed until the target is invoked.

Another way to manage properties is to place them in a separate Java-style properties file, and import the entire file into the buildfile using a different variation of the `property` task. Here is how you would import a properties file using the local path to the file:

```
<project name="sample" default="compile">
  <property file="build.properties"/>
  . . .
```

In this example, the file must be named *build.properties* and located in the same directory as the buildfile. The `property` task also allows you to refer to a properties file using a Java-style resource name (located using the Java classpath) or (as of Ant 1.6) using a URL that refers to a properties file

You can also load all of the current user's system environment variables using the `property` task:

```
. . .
  <property environment="env"/>
  . . .
```

This task defines an Ant property for each environment variable found in the user's system settings. You refer to these properties using the prefix that you specified in the `environment` attribute. In our example, we specified `env` as the prefix, so an environment variable named `CLASSPATH` would be referenced as `env.CLASSPATH` in our buildfile.

You can load multiple sets of properties, if needed, using multiple instances of the `property` task. This can be used for managing distinct collections of properties or overlapping properties settings that allow individual developers to override default project properties. It's important to remember the precedence rules for Ant properties in these cases, though. Properties are set in the order that they are encountered in parsing the buildfile, but the first value encountered for a property wins over any subsequent ones. In other words, properties in Ant are immutable once they're set, their value can't be changed. This can lead to potentially confusing situations, such as:

```
. . .
  <property name="foobar" value="firstvalue"/>
  <property name="foobar" value="secondvalue"/>
  . . .
```

Because of the precedence rules in Ant, the property named `foobar` ends up with the value `firstvalue` in this case because it encounters the first property task and sets the `foobar` property there and the second property setting is ignored. As long as you remember the immutable nature of properties, it isn't hard to keep these precedence rules straight.

17.3.3.2. Referencing properties

Regardless of how a property is set, it can be referenced within the buildfile using the syntax `${property name}`. With our earlier `java.src.dir` property set, for example, we can reference it

elsewhere in the buildfile:

```
. . .
<target name="compile">
  <javac srcdir="${java.src.dir}" destdir="classes"/>
. . .
```

We can use properties to improve our example buildfile, as shown in [Example 17-3](#). Here, we're loading a set of properties from a file named *build.properties*, shown in [Example 17-4](#) and then using these properties in place of the string literals we were using in the previous version. The source directory, for example, is referenced in the `compile` and `clean` targets using the value of the `java.src.dir` property. With these changes to our example buildfile, we can now make changes to the project directory structure and reflect them in one place, the *build.properties* file, rather than searching through the buildfile for all places affected by a change.

Example 17-3. Simple buildfile using properties

```
<!-- A simple buildfile that compiles a set of Java sources
      from the src directory, placing the results in the classes
      directory, and can also create a jar from the classes -->
<project name="JEnt-Ant-props" default="archive">
  <property file="build.properties"/>
  <target name="compile"
    description="Compile all source code into the classes dir">
    <!-- Create the classes dir, if needed -->
    <mkdir dir="${java.classes.dir}"/>
    <!-- Compile the java code into classes -->
    <javac srcdir="${java.src.dir}" destdir="${java.classes.dir}"/>
  </target>
  <target name="archive" depends="compile"
    description="Create an archive from the contents of the classes dir">
    <jar destfile="${ant.project.name}.jar"
      includes="${java.classes.dir}/**"/>
  </target>
  <target name="clean"
    description="Remove all artifacts created in other targets">
    <!-- Remove the classes dir created by the compile target -->
    <delete dir="${java.classes.dir}"/>
    <!-- Remove the jar generated by the archive target -->
    <delete file="${ant.project.name}.jar"/>
  </target>
</project>
```

Example 17-4. build.properties file loaded by buildfile

```
# Location of the Java source, relative to the base directory
# of the project
java.src.dir = src

# Location of the generated Java classes, relative to the
# base directory of the project
java.classes.dir = classes
```



17.4. Core Tasks

We've already used a few of the core tasks included with Ant in our earlier examples. Our `compile` target uses the `javac` task, which invokes the Java compiler, and our `archive` target uses the `jar` task, which creates a jar archive from a set of files. In addition to these Java-specific tasks, we've also seen some general-purpose tasks, like the `mkdir` task that creates a directory and the `delete` task for deleting files and directories.

Many, many other tasks bundled with the Ant distribution can be combined in a virtually infinite set of combinations to do what you need to do in your development process. We don't intend to document all of them here again, for complete coverage of the core Ant tasks, go to the Ant project site (<http://ant.apache.org>) and pick up *Ant: The Definitive Guide* (O'Reilly). In the following sections, we briefly discuss some general functional areas that are encountered in nearly every development project and some of the common ways that Ant supports these needs with its core tasks. These tasks are the workhorse tasks in Ant that you'll find yourself using over and over again.

17.4.1. File and Directory Tasks

At the heart of any software development effort is the need to do things to files and directories. Files need to be moved or copied from one place to another, directories need to be created or deleted, sets of files need to be pulled together into archives in various formats. Naturally, Ant provides numerous built-in tasks that support these needs.

17.4.1.1. Basic entities

Ant provides some fundamental entities that are used in various tasks to represent files, sets of files, and paths. Chief among them is the `<fileset>`. A `<fileset>` defines a set of files starting from a base directory. A `<fileset>` is typically defined as a child element of a task, to indicate a set of files to be moved, filtered, or otherwise manipulated by the task. The files to be included in the `<fileset>` can be specified explicitly by naming them or by using filters. Filters are defined using attributes on the `<fileset>` element, child elements of the `<fileset>`, or both. This entry defines a `<fileset>` containing all of the files in the `/tmp` directory, for example:

```
...
<fileset dir="/tmp" includes="*" />
...
```

The `*` matches any set of zero or more characters.

A `<fileset>` that includes everything in `C:/Windows`, excluding files ending in `.ini`, could be defined

like this:

```
. . .
<fileset dir="C:/Windows">
  <exclude name="*.ini"/>
</fileset>
. . .
```

As a final example, if you wanted to include all of the files, recursively, that end in *.java* or *.xml*, but wanted to exclude Ant buildfiles, you would write something like this:

```
. . .
<fileset dir="${basedir}">
  <include name="**/*.java"/>
  <include name="**/*.xml"/>
  <exclude name="**/build.xml"/>
</fileset>
. . .
```

The expression **** is a shortcut for any sequence of directories or subdirectories. So ***/*.java* matches any file at any level within the chosen base directory that ends in *.java*.

To make file selections more modular, Ant also includes the concept of a `<patternset>`. A `<patternset>` is a set of file inclusion and exclusion filters that can be given a name and then referenced by that name whenever they are needed. We might define a `<patternset>` at the top of our buildfile, for example, to select all of the Java files in a given directory:

```
. . .
<patternset id="java.source.patternset">
  <include name="**/*.java"/>
</patternset>
. . .
```

We can then use this pattern elsewhere in our buildfile when we need to operate only on Java files under a base directory. A `<patternset>` can be used in any `<fileset>` to act as the filter for selecting the files in the set:

```
. . .
<fileset dir="${java.src.dir}/java">
  <patternset refid="java.source.patternset"/>
</fileset>
. . .
```

We can also use a `<patternset>` in various tasks that support them as child elements for filtering an activity. The `zip` task is one of these, since it allows the use of a `<patternset>` to specify what files to include in the zip file it generates:

```
. . .
<zip destfile="${ant.project.name}-src.zip" basedir="${java.src.dir}/java">
  <patternset refid="java.source.patternset"/>
</zip>
. . .
```

Note that in both of the previous two examples, we could have also defined the `<patternset>` in place rather than referencing the previously defined `<patternset>` by name.

17.4.1.2. Creating, moving, copying, and deleting

New directories are created using the `mkdir` task:

```
. . .
<mkdir dir="/tmp/stuff"/>
. . .
```

Copying files and directories is done with the `copy` task. You can copy a file from one place to another:

```
. . .
<copy file="${ant.project.name}.ear"
      todir="${jboss.home}/server/default/deploy"/>
. . .
```

This will create a copy with the same filename as the original file. You can also create a copy with a different name:

```
. . .
<copy file="config/app.properties"
      tofile="config/app.properties-orig"/>
```

In addition to the single `file` attribute, the `copy` task also accepts `<fileset>` child elements that specify the set of files to be copied. So you can, for example, copy the contents of an entire directory to another location:

```

. . .
<copy todir="/tmp">
  <fileset dir="${java.src.dir}" includes="**"/>
</copy>
. . .

```

The `copy` task has a number of other attributes to control how it behaves, such as whether existing files should be overwritten in the destination (`overwrite`) and whether source directories should be flattened out when the files are copied to a destination file (`flatten`).

Use the `move` task to move and rename files. To rename a single file, for example:

```

. . .
<move file="${ant.project.name}.war" tofile="${ant.project.name}-old.war"/>
. . .

```

To move a file from one location to another, preserving its original name:

```

. . .
<move file="${ant.project.name}.ear"
      todir="${jboss.home}/server/default/deploy"/>
. . .

```

Note that this is similar to our earlier example of using the `copy` task, but in this case, the file doesn't remain in its original location.

You can also move whole sets of files from one location to a new directory, using nested `<fileset>` elements. To move an entire directory from one place to another:

```

. . .
<move todir="${jboss.home}/server/default/deploy">
  <fileset dir="war-tmp"/>
</move>
. . .

```

To move a subset of files from one directory to another, you would use a `<fileset>` with

include/exclude filters and `<patternset>` elements. The following example moves all of the XML files from the project directory to a directory in */tmp*.

```

. . .
<mkdir dir="/tmp/xml-files"/>
<move todir="/tmp/xml-files">
  <fileset dir="${basedir}">
    <include name="**/*.xml"/>
  </fileset>
</move>
. . .

```

This entry preserves the subdirectory structures of the original directory. If there were a file *config/web.xml* in the original directory, for example, it will be copied into a *config/web.xml* file in the new directory.

Optional attributes on the `move` task mirror those on the `copy` task. They allow you to control things such as whether the moved files are flattened into a single directory (`flatten`) and whether files of the same name in the destination directory should be overwritten (`overwrite`).

Deleting files and directories is done with the `delete` task. Deleting a file is simple:

```

. . .
<delete file="${ant.project.name}.ear"/>
. . .

```

Deleting a directory is very similar:

```

. . .
<delete dir="/tmp/xml-files"/>
. . .

```

This deletes the directory and all of its contents.

To delete a subset of files and directories in an existing directory, you use nested `<fileset>` elements with include/exclude filters. To delete all of the files ending in *.tmp* in a directory, for example:

```

. . .
<delete>
  <fileset dir="${basedir}">
    <include name="**/*.tmp"/>
  </fileset>
</delete>

```

. . .

17.4.1.3. Making archives

Ant includes several tasks for dealing with file archives of different formats, including Java archives (jars), tar files, zip files, and bzip2 files.

Java archives can be created and unpacked using the `jar` and `unjar` tasks. Creating a jar archive involves specifying the files to include in the archive and where the archive file should be created. The simplest use of the `jar` task is to simply jar up an entire directory. This example creates a jar from the contents of our project's `classes` directory, and names the jar file after the name of the project:

. . .

```
<jar basedir="classes" destfile="${ant.project.name}-lib.jar"/>
```

. . .

You can, of course, be more selective about what goes into the jar, using all the filtering tools we've seen earlier. The `jar` task has `includes` and `excludes` attributes that can be used to do basic filters on what's pulled from the `basedir`. The following creates a jar from the contents of `classes` but leaves out any Java properties files:

. . .

```
<jar basedir="classes" destfile="${ant.project.name}-lib.jar"
  excludes="**/*.properties"/>
```

. . .

Nested `<fileset>` elements can also be used to do more elaborate filtering of the jar contents.

The `jar` task also includes ways to manage the *MANIFEST* file for the archive. You can either include a full *MANIFEST* file using the `manifest` attribute, or you can use the task to construct a manifest on the fly using a `manifest` child element, specifying the contents of the manifest using attributes and child elements on the nested `manifest` element.

The `unjar` task is used to extract files from a jar archive. To extract all of the contents into a destination directory:

. . .

```
<unjar src="${ant.project.name}-lib.jar" dest="${tmpdir}/jar-contents"/>
```

. . .

You can operate on multiple archives concurrently by specifying one or more `<fileset>` elements as

child elements of the `unjar` task. The following example extracts the contents of all of the Xerces libraries in our library directory into the project's `classes` directory:

```

. . .
<unjar dest="classes">
  <fileset dir="${lib.dir}/xerces-2.6.2">
    <include name="*.jar"/>
  </fileset>
</unjar>
. . .

```

You can use nested `<patternset>` elements to select specific subsets of the archive contents to extract. The following example pulls only the classes that are not in packages starting with `javax` from the Xerces libraries:

```

. . .
<unjar dest="classes">
  <fileset dir="${lib.dir}/xerces-2_6_2">
    <include name="*.jar"/>
  </fileset>
  <patternset>
    <exclude name="javax/**"/>
  </patternset>
</unjar>
. . .

```

Ant provides support for a number of other popular archive formats as well. The `tar/untar` tasks support tar archives, the `zip/unzip` tasks support archives in the zip format, the `gzip/gunzip` tasks support gzipped archives, and the `bzip2/bunzip2` tasks support archives in the bzip2 format. We don't provide examples for all of them since they largely work in a similar fashion to the `jar/unjar` tasks. There are some minor format-specific distinctions among the tasks to watch out for. For example, only the `jar` task deals with archive manifests since the other archives don't have manifests, and the `tar` task has a `longfile` option that lets you pick one of several ways `tar` can deal with long filenames.

17.4.2. Basic Java Tasks

Naturally, Ant provides strong support for basic Java activities, such as compiling code and running Java standalone applications. When using the Java-related tasks in Ant, it's important to understand how paths and, specifically, classpaths, are specified in Ant.

17.4.2.1. Paths and classpaths

Path lists are a common construct in most operating systems. Lists of directory paths are used to specify where to locate executable programs and support libraries. In Java, the classpath specifies where to locate classes at runtime. Ant supports both general paths and Java classpaths with the `<path>` and `<classpath>` entities.

A general-purpose path is defined using the `<path>` element. The directories and files included in the path are specified using nested `<pathelement>`, `<fileset>`, `<dirset>`, or `<filelist>` elements. The `<pathelement>` element is used to explicitly include a single entry on the path, either a file or a directory. This example defines a `<path>` containing the `/usr/bin` and `/usr/local/bin` directories:

```
. . .
<path id="exec.path">
  <pathelement location="/usr/bin"/>
  <pathelement location="/usr/local/bin"/>
</path>
. . .
```

You can also define colon- or semicolon-separated lists of entries as a property and include these in your path using a `<pathelement>` with a `path` attribute. This entry defines the same `<path>` as the previous example:

```
. . .
<property name="tmp.path" value="/usr/bin:/usr/local/bin"/>
<path id="exec.path2">
  <pathelement path="{exec.path2}"/>
</path>
. . .
```

When defining path properties like this, you don't need to worry about using the path separator that's appropriate for the current operating system. Ant will automatically convert the path to the proper format for the current environment (e.g., semicolons for Windows, colons for Unix).

A `<fileset>`, `<dirset>`, or `<filelist>` is used to add one or more entries to the path in a batch. This example defines a `<path>` with the project `classes` directory and any jars in the Xerces directory:

```
. . .
<path id="xml.path">
  <pathelement location="classes"/>
  <fileset dir="{lib.dir}/xerces-2_6_2">
    <include name="*.jar"/>
  </fileset>
</path>
. . .
```

It's important to note when using a `<fileset>`, `<dirset>`, or `<filelist>` in a path (either a general path or a classpath) that the order in which the matching entries are added to the path is undefined. So if the order is important in your case (e.g., you want one library to take precedence over another because of potentially overlapping class definitions), you should use explicit `<pathelement>` entries instead.

A `<path>` can be composed of other `<path>`s as well, using references to their IDs. We can construct another `<path>` that includes everything in our previous example `<path>`, plus the Xalan jars:

```
. . .
<path id="extended.xml.path">
  <pathelement refid="xml.path"/>
  <fileset dir="{lib.dir}/xalan-2_6_0/bin">
    <include name="*.jar"/>
  </fileset>
</path>
. . .
```

Classpaths are defined the same as paths. For example:

```
. . .
<classpath id="xml.class.path">
  <pathelement refid="xml.path"/>
  <fileset dir="{lib.dir}/xalan-2_6_0/bin">
    <include name="*.jar"/>
  </fileset>
</classpath>
. . .
```

The difference between a `<path>` and a `<classpath>` is that a `<path>` can be defined outside of any target definitions and referenced as needed in your buildfile targets. A `<classpath>` is always a child element of a task, used to specify where the task will locate its classes.

17.4.2.2. Compiling Java code

Java code is compiled in Ant using the `javac` task. At heart, the `javac` task compiles a set of Java source files and generates a set of classes in a destination directory. At its simplest, using the `javac` task requires specifying only a source directory and a destination directory:

```
. . .
<javac srcdir="{java.src.dir}" destdir="classes"/>
. . .
```

In this case, the `javac` task recursively searches the source directory for Java files (those ending in `.java`) that are newer than their equivalent class files in the destination directory. "Equivalence" is defined simply as the `.class` file with a package and base name that matches the directory and base filename of the `.java` file. The `javac` task does not (by default) attempt to parse the source files to do more refined equivalence checking.

If you want to compile source code from multiple source directories, you can use nested `<src>` elements instead of the `srcdir` attribute. For example:

```

. . .
<javac destdir="classes">
  <src path="${common.src.dir}"/>
  <src path="${java.src.dir}"/>
</javac>
. . .

```

This entry compiles all of the Java files found in the two source directories into the `classes` directory.

You can be more selective in terms of the source files to be compiled as well. `includes` and `excludes` attributes on the task can be used to specify filters to be applied to the contents of the source directories, or you can use nested `<include>/<exclude>` elements.

It's usually necessary to specify the classpath to be used when compiling Java code. By default, the `javac` task uses the classpath used to invoke Ant, which often doesn't include key libraries needed for your project. You can specify the compile classpath using the `classpath` attribute and a path string, for example:

```

. . .
<javac destdir="classes"
  srcdir="${java.src.dir}"
  classpath="${lib.dir}/lib1.jar;${lib.dir}/lib2.jar" />
. . .

```

or you can refer to a path defined elsewhere, using the `classpathref` attribute:

```

. . .
<path id="proj.compile.classpath">
  <pathelement location="${lib.dir}/lib1.jar"/>
  <pathelement location="${lib.dir}/lib2.jar"/>
</path>
. . .
<javac destdir="classes" srcdir="${java.src.dir}"
  classpathref="proj.compile.classpath" />

```

. . .

or you can use a nested `<classpath>` element, defining the path structure directly within the `javac` task invocation:

```
. . .
<javac destdir="classes" srcdir="${java.src.dir}">
  <classpath>
    <pathelement location="${lib.dir}/lib1.jar"/>
    <pathelement location="${lib.dir}/lib2.jar"/>
  </classpath>
</javac>
. . .
```

You can use numerous other options to control the compilation process. The `compiler` attribute specifies which Java compiler should be used, for example. By default, the compiler associated with the JVM that is running Ant will be used. The following example specifies that a Java 1.4 compiler should be used, if available on the system:

```
. . .
<javac destdir="classes" srcdir="${java.src.dir}" compiler="javac1.4"/>
. . .
```

If you want to specify a particular compiler for all invocations of the `javac` task, you can use the built-in property `build.compiler`. If a compatible compiler cannot be found in the user's environment when the task runs, an error will be generated.

Another important option on the `javac` task is whether to generate debug information in the class files, using the `debug` attribute. This is important when dealing with development environments in which an IDE will need this debug environment in order to provide step-through debugging features. To include debug information in the class files, set the `debug` attribute to `on`:

```
. . .
<javac destdir="classes" srcdir="${java.src.dir}" debug="on"/>
. . .
```

Many other options are available, allowing you to specify compiler arguments, how much memory for the compilation JVM to use, and so on.

17.4.2.3. Running Java programs

You can also run compiled Java code, using the `java` task. This comes in handy when a Java utility provided by a tool or vendor hasn't been wrapped by an Ant task or when you want to invoke some project code from the Ant buildfile, perhaps to test that a build completed successfully.

If you simply want to invoke the main method of a Java class:

```
. . .  
<java classname="org.jaf.MyApp" />  
. . .
```

If you have a jar archive with a manifest that specifies a main class, you can invoke its main method like so:

```
. . .  
<java jar="${lib.dir}/app1.jar" />  
. . .
```

These simplistic versions of the `java` task are usually not sufficient. For one thing, you often need to provide a classpath that includes all of the required libraries for the Java application. This is done roughly the same way as specifying a classpath for the `javac` task. You can specify an explicit classpath as a string using the `classpath` attribute:

```
. . .  
<java classpath="${lib.dir}/lib1.jar" classname="org.jaf.MyApp" />  
. . .
```

or you can refer to a predefined path using the `classpathref` attribute:

```
. . .  
<path id="proj.run.classpath">  
  <pathelement location="${lib.dir}/lib1.jar" />  
</path>  
. . .  
<java classpathref="proj.run.classpath" classname="org.jaf.MyApp" />  
. . .
```

or you can use an embedded `<classpath>` element, specifying the classpath explicitly or by referencing paths defined elsewhere in the buildfile:

```

. . .
<javac classname="org.jaf.MyApp">
  <classpath>
    <pathelement location="${lib.dir}/lib1.jar"/>
  </classpath>
</javac>
. . .

```

It's also likely that you will need to provide command-line arguments when running the Java application. You may need to provide arguments that are passed to the Java class in its invocation arguments, or you may need to pass arguments to the JVM to control its runtime configuration. The former is done using nested `<arg>` elements on the `java` task:

```

. . .
<java classname="org.jaf.MyApp" classpathref="proj.run.classpath">
  <arg value="-noAudio"/>
  <arg line="-user Jim"/>
</java>
. . .

```

JVM arguments are specified the same way, except you use a `<jvmarg>` nested element instead:

```

. . .
<java classname="org.jaf.MyApp" classpathref="proj.run.classpath">
  <arg value="-user Jim"/>
  <jvmarg value="-Dorg.jaf.MyApp.noAudio=true"/>
</java>
. . .

```

17.4.3. Miscellaneous Utilities

Besides the categories discussed in the previous sections, you'll find yourself using some other miscellaneous tasks in Ant over and over.

17.4.3.1. Ant's echo task: `System.out.println()`

Possibly the most commonly used task in Ant is the `echo` task, which prints a message to the standard output. This is useful both for diagnostic messages and for debugging Ant buildfiles that seem to be misbehaving:

```

. . .
<echo message="Compiling a whole mess o' Java code. . ."/>
<javac srcdir="${java.src.dir}" destdir="classes"/>
. . .

```

17.4.3.2. Calling external programs

We saw earlier how Java programs can be invoked from Ant. You can also invoke other external programs, using the `exec` task. This task should be used with caution since it often leads to buildfiles that are not easily portable to different platforms (or even the same platform with a different configuration). Still, the `exec` task is handy in a pinch.

The command to execute is specified with the `executable` attribute:

```

. . .
<property name="browser.exec" value="/usr/local/bin/mozilla"/>
<exec executable="${browser.exec}"/>
. . .

```

Command-line arguments are specified using nested `<arg>` elements, similarly to the `java` task:

```

. . .
<property name="browser.exec" value="/usr/local/bin/mozilla"/>
<property name="project.doc.dir" value="docs"/>
. . .
<exec executable="${browser.exec}">
  <arg value="${project.doc.dir}/index.html"/>
</exec>
. . .

```

Environment variables can be set for the executable using nested `<env>` elements. If, for example, you need to specify the path to be used to find the executable:

```

. . .
<exec executable="mozilla">
  <env key="PATH" path="/usr/local/bin:/usr/bin"/>
</exec>
. . .

```

To help make the use of `exec` more portable, you can use the `os` attribute to control when the

command will be executed. If the current platform type (as specified by the JVM's `os.name` system property) is listed in the comma-delimited list specified in the `os` attribute, Ant will attempt to execute the command; otherwise, it will skip the `exec` task entirely. This can be used to create a sort of conditional, platform-specific invocation of a particular type of command, for example:

```
. . .
<property name="win.browser.exec" value="mozilla.exe"/>
<property name="unix.browser.exec" value="mozilla"/>
<property name="win.browser.path" value="C:\Progra~1\Mozilla"/>
<property name="unix.browser.path" value="/usr/bin:/usr/local/bin"/>
. . .
<exec os="Windows 2000,Windows XP" executable="{win.browser.exec}">
  <env key="PATH" path="{win.browser.path}"/>
</exec>
<exec os="Linux,Solaris,FreeBSD,Mac OS X"
  executable="{unix.browser.exec}">
  <env key="PATH" path="{unix.browser.path}"/>
</exec>
. . .
```

If this example is run on a Windows 2000 or Windows XP machine, the first `exec` task will be executed, since the `os.name` Java system property will match one of the entries in the `os` attribute. This `exec` task uses the Windows-specific versions of the relevant properties, set to reasonable defaults for a typical Windows installation of Mozilla. The second `exec` task will not run because the `os.name` value will not match any of the entries in its `os` attribute.

17.4.3.3. Importing custom Ant tasks

Many application servers and development tools provide their own custom Ant tasks that can be used in your buildfiles. A J2EE server may provide a task that automatically deploys a component module to its server, for example. You may decide to write your own custom tasks as well and therefore need to import them into your buildfiles.

External tasks are imported into a buildfile using the `taskdef` task. If the classes defining the task are on the runtime classpath, all you need to do to import a task is to specify the name of the task and the class that implements it. This example imports a task provided by the BEA WebLogic server distribution:

```
. . .
<taskdef name="wldeploy"
  classname="weblogic.ant.taskdefs.management.WLDeploy"/>
. . .
```

If the classes that implement the imported tasks aren't on the runtime classpath, you can specify the classpath that `taskdef` should use, with a nested `classpath` element:

```

. . .
<taskdef name="wldeploy"
  classname="weblogic.ant.taskdefs.management.WLDeploy">
  <classpath>
    <pathelement path="{weblogic.home}/server/lib/weblogic.jar"/>
  </classpath>
</taskdef>
. . .

```

17.4.3.4. Invoking targets

Sometimes it's very useful to call Ant targets programmatically from other targets in your buildfile. Making one task depend on another using the `depends` attribute is a limited way to do this. But the target named in the `depends` attribute will always be run before the dependent target, and sometimes you need to check some conditions or set some property values before the supporting target is called. In these situations, you can use the `antcall` task to explicitly invoke a target from anywhere within another target.

Invoking a target is a simple matter of naming the target as an attribute of the `antcall` task:

```

. . .
<target name="main-target">
  <!-- Perform some other tasks -->
  <antcall target="sub-target"/>
</target>
. . .

```

It's important to remember that, when you use the `antcall` task in a target, Ant executes the secondary target as if the entire buildfile were reinvoked. By default, all of the property values that have been set at the point of the `antcall` execution are inherited by the new Ant runtime, which effectively makes the new target run as if it were running within the original runtime. If for some reason you don't want the new target execution to inherit the current property settings, you can use the `inheritAll` attribute:

```

. . .
<antcall target="sub-target" inheritAll="false"/>
. . .

```

You can also explicitly pass specific property values into the new target using nested `<param>` elements:

```
. . .  
<antcall target="sub-target">  
  <param name="param1" value="foo"/>  
  <param name="param2" value="bar"/>  
</antcall>  
. . .
```

This use of `antcall` makes the target call similar to a method call, a very effective tool indeed.



17.5. Enterprise Tasks

In addition to the basic Java development tasks we've discussed in the previous sections, enterprise application development requires special support in terms of managing component development and testing; assembling and deploying applications; preserving platform portability in the face of different application server vendors; and managing multiple application environments.

In the following sections, we'll explore each of these areas in terms of the Ant tasks that are relevant to each and look at some approaches to designing your Ant buildfiles to support each of these enterprise development issues.

17.5.1. Assembling Components

When developing J2EE components, you need to perform many of the basic Java development tasks we've already seen: compile Java code into classes, construct a jar file from a set of files, move the jar file from one place to another, and so forth. Some complexities are introduced when working with enterprise components, however. Component archives need to be assembled using specific file layouts. Depending on the application server being used, additional processing of your Java classes or component archives may be necessary as well.

The jar file formats used for component assemblies are more complicated than the simple class hierarchies used for libraries. As discussed in [Chapter 2](#) and detailed in [Appendix A](#), component archives follow a specific structure that dictates where each asset (deployment descriptors, libraries, classes, etc.) must reside in the archive structure. You could use the basic tasks described in the earlier sections to construct component archives, of course. You could use the `mkdir` and `copy` tasks to construct the required archive directory structure in a temporary location and then use the `jar` task to construct the actual component archive file. But this would require you to correctly interpret the component archive format and map that into your buildfile yourself (e.g., libraries in a web archive go into the `WEB-INF/lib` directory, compiled classes go into the `WEB-INF/classes` directory, etc.). Luckily, Ant provides tasks that create J2EE component archives directly, allowing you to simply specify the required elements of the archive. The Ant tasks assemble these assets into the standard archive structure for you and provide some other useful shortcuts as well.

17.5.1.1. Web components

The core of Ant's support for assembling web component archives is the `war` task. This task allows you to create a web archive by specifying the elements that should be included using task attributes and subelements in the task declaration. The attributes on the task element specify the name of the war file to be created and the deployment descriptor for the module. The task supports a set of child `<fileset>` elements that allows you to specify the HTML, JSP, and other web assets to be put into the archive's root directory; the files (other than the deployment descriptor) to be put into the `WEB-INF` directory; the Java classes and other classpath-accessible files to be put into the `WEB-INF/classes` directory; and the full libraries to be put into the `WEB-INF/lib` directory.

In our example buildfile, we already have an `archive` target that creates a simple jar file from the contents of the project's `classes` directory. In [Example 17-5](#), we show a new `create-wars` target that uses the contents of the `classes`, `config`, and `web` directories in the project to construct a web component archive. To support this target, we need two new properties in our `build.properties` file to specify the location of the configuration and web content directories:

```
. . . .
# Location of application configuration files
config.dir = config

# Location of all web-related content (HTML files, etc.)
web.dir = web
```

This maintains our pattern of keeping all project configuration details in one place and referencing these properties where needed in the buildfile.

Example 17-5. Target for assembly of web archives

```
. . . .
<!-- Create the web archives needed for this application -->
<target name="create-wars"
    description="Create the web archives needed for this application"
    depends="compile">
    <!-- Delete any existing war files -->
    <delete>
        <fileset dir=".">
            <include name="*.war"/>
        </fileset>
    </delete>
    <!-- Define a pattern for the set of files within the web dir
        that should be included in the root of the archive. -->
    <patternset id="war.files">
        <include name="**"/>
    </patternset>
    <!-- Define a pattern for the set of classes from the classes
        dir that should be included in the archive -->
    <patternset id="war.classes">
        <include name="**"/>
    </patternset>
    <!-- Define a set of libraries to be included in the archive -->
    <patternset id="war.libs">
        <include name="xercesImpl.jar"/>
        <include name="xml-apis.jar"/>
        <include name="xmlParserAPIs.jar"/>
    </patternset>
    <!-- Invoke the war task -->
```

```

    <war warfile="${proj.name}.war"
        webxml="${config.dir}/web.xml">
        <!-- Put any files specified by the war.files patternset in the root
            of the web archive. -->
        <fileset dir="${web.dir}">
            <patternset refid="war.files"/>
        </fileset>
        <!-- Include any classes specified by the war.classes patternset -->
        <classes dir="${java.classes.dir}">
            <patternset refid="war.classes"/>
        </classes>
        <!-- Include any libraries specified by the war.libs patternset -->
        <lib dir="${lib.dir}/xerces-2_6_2">
            <patternset refid="war.libs"/>
        </lib>
    </war>
</target>
. . .

```

We've made the `create-wars` target depend on the `compile` target to ensure that all of the project code is compiled before we try to assemble our web module(s). In the target itself, we first clear out any war files left over from previous runs of this target using the `delete` task. Since we intend this target to be the creator of all web archives for the project, we can safely delete all of the files with the war suffix. Once this is done, we define a few `<patternset>` elements that specify the set of general web assets and Java classes to be included in the archive. Here we're simply including the entire contents of their respective directories in the `<patternset>` since we have all of our web content and classes in singular directories, and we want them included in their entirety. But if that changes and we need to filter elements from each directory or we need to include contents from multiple subdirectories in the project, we simply need to adjust the `<patternset>` elements to suit the new project file structure.

The invocation of the `war` task itself is fairly straightforward. We specify the name for the war file using the `warfile` attribute on the task element using the `proj.name` property as the base for the filename. We also specify the deployment descriptor for the archive in the attributes, setting the `webxml` attribute to the `web.xml` file in our `config` directory. We then use child elements of the `war` task to specify files to be included in the root of the archive as well as Java classes that should be included in the archive. We use the `<patternset>` elements defined earlier in the target to define the files to be included in each case.

17.5.1.2. EJB components

Assembling EJB component archives is slightly more complicated than assembling web archives, due to the application server-specific details that are often required to deploy an EJB module. As discussed in [Chapter 2](#), these can include vendor-specific deployment files or the need to generate vendor-specific support classes for the EJB components in your module.

Ant provides support for creating EJB modules from project assets through the `ejbjar` task, which is included in its bundled set of optional tasks. The task searches recursively through the directory specified by its `descriptorDir` attribute, looking for relevant EJB deployment descriptors. For each

EJB deployment descriptor that it finds, `ejbjar` builds an EJB archive by parsing the EJB deployment descriptor, determining the classes that are required for the EJB components. These classes are copied from the directory specified in the `srcdir` attribute. Depending on how the `ejbjar` task is configured, it can also locate vendor-specific configuration files in the `descriptor` directory and include those in their appropriate locations in the archive. The `ejbjar` task can also generate vendor-specific support classes, if configured to do so. Once this is all accomplished, the task writes out a full EJB module archive to the location specified by the `destdir` attribute.

As opposed to the `war` task we saw in the previous section, the `ejbjar` task is categorized as an optional task in Ant because it requires external libraries to operate. In the case of `ejbjar`, the task requires the Jakarta Byte Code Engineering Library (BCEL) in your classpath when you run Ant since it uses this library to calculate class dependencies for your EJB components as it determines which classes need to be included in your EJB archive. The BCEL can be downloaded from the Apache Jakarta project site (<http://jakarta.apache.org/bcel>). In order to use the `ejbjar` task, either you'll need to have this library on your classpath when you invoke Ant or you'll have to copy the library into the Ant `lib` directory.

The `ejbjar` task can be used either to create generic EJB archives, containing only the standard EJB deployment descriptor and your classes or to generate EJB modules targeted for a specific application server. To start, let's define a target that simply generates a generic EJB module from your project assets, shown in [Example 17-6](#). The `create-ejbjars` target simply invokes the `ejbjar` task, indicating with the task attributes that deployment descriptors will be found in the project configuration directory (using the `descriptor` attribute), that Java classes for the components are to be found in the project's `classes` directory (using the `srcdir` attribute), and that the generated EJB module file should be written in the project's base directory (using the `destdir` attribute). The other attributes we're using on the `ejbjar` task specify how the task should generate the module archive, in terms of the name of the jar file (`naming`), how to calculate dependent classes that need to be included in the archive (`dependency`), and so forth.

Example 17-6. Target for generation of EJB modules

```

. . .
<!-- Create any ejb modules required for this application -->
<target name="create-ejbjars"
    description="Generates an ejb-jar archive for this application"
    depends="compile">
    <!-- Invoke the ejbjar task, looking for deployment descriptors
        in the project config directory. We are using "directory"-based
        naming here, with the name of the directory containing the EJB
        deployment descriptor used as the basis for the ejb-jar filename. -->
    <ejbjar descriptor="${config.dir}"
        srcdir="${java.classes.dir}"
        destdir="${basedir}"
        naming="directory"
        genericjarsuffix=".jar"
        flatdestdir="true"
        dependency="super">
    <!-- Use local copies of descriptor DTDs, in case we're not online -->
    <dtd publicid=
        "-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans 2.0//EN"

```

```

        location="${lib.dir}/j2ee/dtd/ejb-jar_2_0.dtd"/>
    <!-- Parse only EJB descriptors -->
    <include name="**/*ejb-jar.xml"/>
</ejbjar>
</target>
. . .

```

We're using preexisting project properties, `java.classes.dir` and `config.dir`, to indicate where the `ejbjar` task should find the assets for the EJB module. But in order for the task to work as intended, we have to put our EJB deployment descriptors into subdirectories of our project's configuration directory. We've specified that `ejbjar` should use "directory" naming for the modules it generates, so the name of the subdirectory should be the name we want for the EJB module file. The name of this file is important, since application deployment descriptors need to reference the file once it's included in an application archive. If we had EJB components that we wanted assembled into an archive named *ProfileEJB*, for example, and assuming that our `config.dir` property is set to `config`, our configuration directory structure would need to look like [Figure 17-1](#).

Figure 17-1. Sample directory structure for `ejbjar` task

We've also included a few subelements in our invocation of the `ejbjar` task. We added a `<dtd>` element, specifying a local copy of the EJB deployment descriptor DTD. This is helpful when you're developing offline; for example, as I write this I'm sitting on an airplane where I have little chance of making a URL connection to the Sun site where the original copies of the DTD lives.^[1] Using local DTDs can also speed up the parsing process that `ejbjar` performs since it avoids a network lookup.

^[1] Of course, this is helpful to me only because I'm paranoid enough to have copies of all the J2EE schemas and DTDs on my laptop.

The other child element is an `<include>` element that limits the scope of the `ejbjar` task to files in our configuration directory that end with `ejb-jar.xml`. This allows us to put other types of deployment descriptors and configuration files in our configuration directory, without risking a lot of confusion on the part of `ejbjar`.

As we already mentioned, this version of our target generates only a generic, standard EJB module archive. If the EJB container in your application server has additional requirements, such as vendor-specific deployment descriptors and support classes, you'll need to enhance this target to handle these additional server-specific requirements. The `ejbjar` task supports a number of common

application servers with subelements that can be used to manage vendor-specific details. For example, if we knew we were going to deploy our EJB components to a JBoss EJB container, we could use the JBoss-specific subelement of `ejbjar`, `<jboss>`, to create an EJB module that can be deployed directly to a JBoss EJB container. The JBoss-enabled version of our `create-ejbjars` target is shown in [Example 17-7](#).

Example 17-7. JBoss-enabled version of the EJB module target

```

. . .
<!-- Create any ejb modules required for this application -->
<target name="create-ejbjars-jboss"
    description="Generates an ejb-jar archive for this application"
    depends="compile">
    <!-- Invoke the ejbjar task, looking for deployment descriptors
        in the project config directory. We are using "directory"-based
        naming here, where the name of the directory containing the EJB
        deployment descriptor is used as the basis for the ejb-jar filename. -->
    <ejbjar descriptor=" ${config.dir}"
        srcdir=" ${java.classes.dir}"
        destdir=" ${basedir}"
        naming="directory"
        genericjarsuffix="-generic.jar"
        flatdestdir="true"
        dependency="super">
        <jboss destdir=" ${basedir}"/>
    <!-- Use local copies of descriptor DTDs, in case we're not online -->
    <dtd publicid=
        "-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans 2.0//EN"
        location=" ${jboss.home}/docs/dtd/ejb-jar_2_0.dtd"/>
    <!-- Parse only EJB descriptors -->
    <include name="**/*ejb-jar.xml"/>
    </ejbjar>
</target>
. . .

```

The inclusion of the `<jboss>` child element causes the `ejbjar` task to look for any JBoss EJB deployment files (named *jboss.xml*) wherever it finds a standard EJB deployment descriptor and to include the JBoss descriptor in the EJB module when it's assembled. Since JBoss automatically generates any required support classes when the module is deployed, there's nothing else that the `ejbjar` task needs to do to make the EJB module deployable in JBoss. But other application servers require you to pregenerate the support classes for the EJB modules and include them in the module archive when it's deployed. These tasks can be performed by `ejbjar` for the application servers that it supports. For EJB containers not supported by `ejbjar`, you'll need to handle this in other ways. Some EJB containers provide custom Ant tasks that you can import and use for processing EJB archives for deployment, for example.

It's important to note that putting environment-specific details into your buildfiles limits their portability. Our `create-ejbjars-jboss` target, for example, will work only in environments using JBoss:

application servers. If I give my project code to you and you want to deploy it in a WebLogic server, you would need to edit my buildfile to make it work in your environment. As much as possible, you want to avoid this situation so that the buildfile requires only configuration changes (e.g., to the *build.properties* file or any other configuration files that the buildfile references). Ideally, you want to be in a situation in which other developers can simply replace the default configuration files with ones that suit their environment, and the rest of the buildfile works as advertised.

"[Creating Portable Build Processes](#)" later in this chapter discusses some approaches to making your buildfiles as portable as possible, especially with regard to enterprise development tasks.

17.5.2. Assembling Applications

Creating J2EE application assemblies in Ant is very straightforward through the use of the built-in `ear` task. The `ear` task is a specialized version of the `jar` task, so it accepts all of the same child elements as the `jar` task. But the `ear` task automatically creates a jar file with the format of a J2EE application archive.

[Example 17-8](#) shows a `create-ear` target that uses our project properties and the Ant `ear` task to construct an application archive. We've made the `create-ear` target depend on the `create-wars` and `create-ejbjars` targets to ensure that our component archives are assembled before we assemble our application. The `ear` task has a number of attributes and child elements that can be used to control its behavior. In our example, we're using the `destfile` attribute to put the ear file in the root of the project directory, using the project name as the base of the filename. The deployment descriptor is set using the `appxml` attribute, and we're using the *application.xml* file from the configuration directory. Finally, we're including any war and jar files found in the root of the project directory in the application archive using the `includes` attribute, assuming that they are all web or EJB component archives built using the `create-wars` or `create-ejbjars` targets. If we wanted to be more selective (e.g., we may not want some jar files included in the application archive), we could use specific archive filenames in the `includes` attribute.

Example 17-8. Target for creating application archives

```

. . .
<!-- Create an application assembly from our web and EJB components -->
<target name="create-app"
    description="Make an application assembly from the component archives"
    depends="create-wars, create-ejbjars">
    <ear destfile="{proj.name}.ear"
        appxml="{config.dir}/application.xml"
        includes="*.war, *.jar">
    </ear>
</target>
. . .

```

It's important to note that any war or ejb-jar modules created by the `create-wars` or `create-ejbjars`

targets must be referenced in the *application.xml* descriptor using the same filenames created by those targets. Unlike the `ejbjar` task in Ant, the `ear` task does not automatically parse the application deployment descriptor and search for the files that must be included; it just pulls in any files referenced by the combination of the `includes` and `excludes` attributes on the `ear` task. If our project's `proj.name` property is set to `ShoppingCart`, for example, the `create-wars` target in [Example 17-8](#) will create a war file named *ShoppingCart.war*. If we also have an EJB deployment descriptor in a *CCardAccount* subdirectory of our configuration directory, the `create-ejbjars` target will create an EJB jar file named *CCardAccount.jar*. Since our `create-app` target is picking up both of these files as-is, the *application.xml* file must reference them like so:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE application PUBLIC
    "-//Sun Microsystems, Inc.//DTD J2EE Application 1.3//EN"
    'http://java.sun.com/dtd/application_1_3.dtd'>
<application>
  <display-name>JEnt Ant sample application</display-name>
  <!-- Declare the web component module -->
  <module>
    <web>
      <web-uri>ShoppingCart.war</web-uri>
      <!-- Give the web components a URL root -->
      <context-root>cart</context-root>
    </web>
  </module>
  <!-- Declare the EJB module -->
  <module>
    <ejb>CCardAccount.jar</ejb>
  </module>
</application>
```

As we saw with assembly of component archives, you may need to consider server-specific details when assembling application archives. Some application servers make use of their own application-level configuration files for various details not covered by the standard J2EE deployment descriptor, for example. If you need to include server-specific files in the ear file, you have to specify them explicitly using either the `includes` or `excludes` attribute or using child `<fileset>`, `<include>`, `<exclude>`, and `<patternset>` elements.

17.5.3. Deploying Components and Applications

Once you've compiled your code and assembled some component modules or an entire application archive, the next logical step is to deploy your code to an application server of some kind. As discussed in [Chapter 2](#), the specific steps that need to be taken to deploy components and applications to an application server are server-specific; it's done differently in Tomcat, JBoss, WebLogic, and WebSphere, for example. For some servers, like Tomcat and JBoss, deploying code is simply a matter of placing the archive file in a deployment directory that the server is configured to watch for new deployable code. In these cases, writing a target to deploy code is simply a matter of

using the Ant file-oriented tasks to move or copy the generated archive file to the right place. [Example 17-9](#) shows a target that deploys our application archive to a JBoss server. We've assumed here that a property named `jboss.home` has been set in the *build.properties* file to point to the root of the JBoss server installation. In the target, we just copy the application archive file (the target depends on the `create-app` target, so we can count on the archive existing) to the default deployment directory in JBoss, *server/default/deploy*.

Example 17-9. Target for deploying an application to JBoss

```

. . .
<!-- Deploy our application to a JBoss server -->
<target name="deploy-app-jboss"
    description="Deploy the application archive to a JBoss server"
    depends="create-app">
    <copy file="{proj.name}.ear"
        todir="{jboss.home}/server/default/deploy" />
</target>
. . .

```

Other vendors require you to use bundled deployment tools to deploy code to their application servers. BEA WebLogic, for example, provides a deployer tool that you use to deploy code to running WebLogic servers. One form of the tool is a Java class named `weblogic.Deployer` that can be invoked from the command line to deploy either a component or application archive. You could use this Java utility to create a WebLogic version of our deploy target, using the generic `java` task, as shown in [Example 17-10](#).

Example 17-10. Target for deploying an application to WebLogic

```

. . .
<!-- Define a classpath that includes the WebLogic library -->
<path id="weblogic.deploy.classpath">
    <path refid="java.compile.classpath" />
    <pathelement path="{weblogic.home}/server/lib/weblogic.jar" />
</path>
<!-- Deploy the application to a WebLogic server -->
<target name="deploy-app-weblogic"
    description="Deploy the application archive to a WebLogic server"
    depends="create-app">
    <java classname="weblogic.Deployer"
        classpathref="weblogic.deploy.classpath">
        <arg line="-adminurl http://{weblogic.http.host}:{weblogic.http.port}" />
        <arg line="-user {weblogic.user}" />
        <arg line="-password {weblogic.password}" />
        <arg line="-name {proj.name}" />
    </java>
</target>
. . .

```

```

        <arg line="-redeploy ${proj.name}.ear"/>
    </java>
</target>
. . .

```

In the target, we first set up a classpath to be used to run the WebLogic utility, which includes our compilation classpath with the necessary WebLogic libraries appended to it. Then we use the `java` task to run the utility, using the appropriate arguments required by the utility class. Note that we also need to define a few more new properties in our `build.properties` file to support this target, to indicate the details about the WebLogic server being used when we run the target:

```

# WebLogic-specific properties
weblogic.home=/usr/users/jfarley/servers/dev-81
weblogic.http.host=localhost
weblogic.http.port=8001
weblogic.user=system
weblogic.password=password

```

Some application servers provide custom Ant tasks that can be imported into your buildfile and used to drive the deployment process. WebLogic provides such an Ant task. Importing the custom task is done using the built-in `taskdef` task, specifying the appropriate classpath to locate the WebLogic classes:

```

. . .
<!-- Load the WebLogic Ant tasks from the server libraries -->
<taskdef name="wldeploy"
    classname="weblogic.ant.taskdefs.management.WLDeploy">
    <classpath>
        <path refid="weblogic.deploy.classpath"/>
    </classpath>
</taskdef>
. . .

```

Since the support classes for the Ant task are located in the same libraries as the `weblogic.Deploy` utility, we're just reusing the `weblogic.deploy.classpath` path that we defined earlier. Once the task is loaded, we can use it in our buildfile according to the server documentation. A simple example, roughly equivalent to the previous use of the `weblogic.Deployer` utility, is shown here:

```

. . .
<!-- Deploy the application to a WebLogic server -->
<target name="deploy-app-weblogic-ant"
    description="Deploy the application archive to a WebLogic server"
    depends="create-app">

```

```
<!-- Invoke the WebLogic wldeploy task to deploy our application -->
<wldeploy action="deploy"
  source="${proj.name}.ear"
  name="${proj.name}"
  user="${weblogic.user}"
  password="${weblogic.password}"
  adminurl="${weblogic.admin.url}"/>
</target>
. . .
```

If we wanted our buildfile to be used in additional application server environments (such as Apache Geronimo or WebSphere), we'd need to write new versions of our application deployment target and potentially add new server-specific properties to our properties file.

To some degree, we can try to address this portability issue using the Ant built-in task, `serverdeploy`. As of Ant 1.6, this task directly supports WebLogic and JOnAS application servers (through server-specific child elements that invoke the corresponding deployment tools provided by these servers). The task also includes a generic child element that can be used to invoke other server deployment tools, assuming that they are provided in the form of a Java class with a main method and that all of the arguments can be passed into the utility using the command line. The `serverdeploy` task isn't sufficient to cover all possible application servers, however (there's no way to use it to deploy an application to JBoss, for example).

A more general-purpose approach to the application server portability issue is described next.

17.6. Creating Portable Build Processes

As we've already seen, using Ant for enterprise development projects raises some potential portability issues with our buildfiles. Certain targets, such as deploying an application archive to an application server, are necessarily specific to a type of J2EE server and will need to be changed if another type of application server is used to run the application. These kinds of portability issues become even more complicated when you need to support multiple development and deployment environments concurrently and are multiplied again when your project makes use of a source code repository:

- Developer environments often vary in terms of the locations of support libraries, application server directories, and so on. This requires the editing of the build properties to suit the local development environment. If these build properties are stored in a file that's checked in with the project code in a code repository, these changes need to be made each time the project code is checked out and undone before the development changes are checked back in.
- In typical build/test/release development processes, the software being produced needs to be deployed and executed in one of several server environments, depending on the stage of the process. These environments are typically configured in different locations and may have different internal configurations. In addition, the application itself may need to be configured differently depending on the deployment environment locations of resource files, cache and other memory sizes, and the like may need to be set differently in different environments. Working around this issue involves the editing of build properties and possibly altering the application configuration files (such as J2EE deployment descriptors) to suit each environment.
- The same software may need to be deployed to different application server types from different vendors. This may require editing build properties, and possibly the structure of entire targets, in order to adjust to the different tools and deployment processes required by different servers.

In general, forcing developers and application deployers to edit project buildfiles and property files on a regular basis complicates development efforts and increases the chance of human error. In general it's preferable to have clean, modular ways to insert local configuration details into the build process so that developers, users, and deployers can simply take the project source files (in the form of a code repository checkout or a downloadable source archive), insert their local configuration details, and then use the Ant buildfile to build, test, deploy, and run the project code.

In this section, we offer some basic techniques for managing these issues when using Ant in order to make your project buildfiles more portable and, therefore, more useful. We've packaged these techniques into a template Ant buildfile that is included in the downloadable examples for this book (see <http://www.oreilly.com/catalog/javaentnut3>). This template buildfile demonstrates the concepts we've discussed here and elsewhere in this chapter, and it can be used to bootstrap your own project buildfile, if needed. Make note, however, that the approaches we discuss in this section about making buildfiles portable are just some of the many approaches to answering these same issues when using Ant. Our goal here is to highlight these portability issues and to demonstrate some potentially useful solutions, not to dictate a single "correct" approach.

17.6.1. Hierarchical Property Files

One of the first issues you tend to encounter in multideveloper projects is that developers manage their environments in very different ways. Software developers typically work on code on a personal workstation, which may be running Linux, Mac OS X, Windows, or a Unix variant of some kind. These development environments have libraries and server configurations that are potentially very heterogeneous. Meanwhile, your Ant buildfile will define various properties that specify where to find dependent libraries, where relevant application server directories can be found, and so on. These properties will be defined either directly in the buildfile or in a properties file referenced by the buildfile. If these properties are defined as default values in a singular location in the project structure, the average developer will need to edit these project files to suit her local development environment. If this is a product development project involving several developers across an organization using a source code repository, the developers will need to avoid checking these buildfile changes back into the repository when they check in their code changes. When the buildfile structure changes, they will need to manually merge the new buildfile with their local changes. In the case of an open source "product" when a person downloads the source distribution to build and deploy locally, a similar manual editing process will have to be done directly on the project files and redone in the future for new releases of the open source project.

One solution to this problem is to use hierarchical property files in your Ant buildfile. In this scheme, you take advantage of the precedence rules used by Ant when defining property values, as discussed in "[Properties](#)" earlier in this chapter. Properties are set once and are invariant once they are assigned a value. So far in our example buildfiles, we've referenced a single *build.properties* file that contains the definition of any required properties. The hierarchical scheme involves first attempting to reference a local, user-specific properties file and then referencing the project's default properties file. Our example buildfile could be extended to do this by simply adding another `<property>` entry before the entry that loads the *build.properties* file:

```
<project name="JEnt Ant portable" default="create-wars">
. . .
  <!-- Import any user-specific build properties for this project -->
  <property file="${user.name}-build.properties"/>
  <!-- Import our default build properties -->
  <property file="build.properties"/>
. . .
```

The first `<property>` entry looks for a properties file named `<username>-build.properties`, where *username* is the login name of the user. If the file is present, the properties it contains will be defined first, overriding any property definitions included in the default project *build.properties* file.

As an example of the usefulness of this scheme, our example *build.properties* file contains a `lib.dir` property that defines the base directory from which all dependent libraries are loaded, as well as several properties that specify the location of required jar files in this library directory. The project *build.properties* file defines default values for all of these properties:

```
. . .
# Directory containing libraries and other software required
```

```

# to build and/or run the project code
lib.dir = /usr/local/software
# Libraries required by the application code
xerces.home = ${lib.dir}/xerces-2_6_2
xerces.impl.jar = ${xerces.home}/xercesImpl.jar
xerces.jaxp.jar = ${xerces.home}/xml-apis.jar

# JUnit libraries
junit.jar = ${lib.dir}/junit-3.8.1/junit.jar
. . .

```

A developer might check out this project (or download the source distribution) with the intention of building the project, but they keep their class libraries in their home directory on their workstation. The developer can simply create a local properties file that overrides the `lib.dir` property:

```

# jsmith-build.properties: Local settings for the project Ant buildfile
lib.dir = ${user.home}/Software

```

If the user's login name were `jsmith`, he would put this into a file named *jsmith-build.properties* in the same directory that holds the project *build.properties* file. Now when he invokes the Ant buildfile, the libraries will be loaded from the *Software* directory in the user's home directory, instead of the default location in */usr/local/software*.

You can also do more fine-grained property overrides, if necessary. In our example, there might be another developer, `mjones`, who keeps most of the required libraries stored in the default location of */usr/local/software* but keeps the Xerces libraries in another location for some reason. Then `mjones` can just override the `xerces.home` property in her local *mjones-build.properties* file:

```

# mjones-build.properties: Local overrides
xerces.home = /home/packages/xerces-2_6_2

```

Various models can be applied for managing these local property files. In our case, we've assumed the property file will be named using the user's login name and will be located directly in the project directory structure. In this scheme, we can opt to include each user's property file with the project files themselves, assuming that usernames are unique among the developers working on the project. Alternatively, we could force the developers to maintain their local properties files themselves, copying them into the project directory structure when they check out or download the project source.

Another model is to reference the local properties file from a location outside of the project directory structure entirely. This prevents us from allowing developers to include their local properties files in the project structure directly but makes it easier for developers to manage their local properties files. They simply have to keep the file in the expected location in their local environment, and the project buildfile will pick it up automatically. The external location needs to be a directory that can be

referenced in a portable way by the buildfile the user's home directory is a perfect candidate since Ant has the standard `user.home` system property that references the user's home directory. Under this model, it doesn't make sense to include the user's login name in the local property filename since it's implied by the fact that the file is in the user's home directory. But if developers are working on multiple projects that use this same scheme for managing local properties files, they'll potentially need multiple local properties files for each project, so the filename should include the name of the project as well. To implement this model, we would need to alter our buildfile along these lines:

```
<project name="ProjectX" default="create-wars">
. . .
  <!-- Import any user-specific build properties for this project -->
  <property file="${user.home}/${ant.project.name}-build.properties"/>
  <!-- Import our default build properties -->
  <property file="build.properties"/>
. . .
```

In this case, if a user wanted to override any of the project properties, he would create a properties file named *ProjectX-build.properties* in his home directory and put any local property settings there.

17.6.2. Modular Target Definitions

In the section "Enterprise Tasks," we saw several examples of common J2EE development targets that needed to be specialized depending on the application server being used. The quick-and-dirty approach we took in our earlier examples was to create multiple versions of our targets, each one suited to a different environment (e.g., we created a generic `create-ejbjars` target and a JBoss-specific `create-ejbjars-jboss` version of the same target). This approach isn't very convenient for developers, since they have to manually invoke each version of each target according to the application server they're using. It also doesn't scale well if you need to support many different environments.

One approach to dealing with this issue is to use the `import` task to import properties and target definitions that are specific to the server platform being used. The `import` task is included with Ant as of Ant 1.6 and simply allows you to import a separate buildfile into another one. The properties, targets, and other entities defined in the root project element of the imported file are included in the buildfile as if they were inserted at the point where the `import` task is used. Using this scheme, we import the target definitions at the top of our buildfile, like so:

```
. . .
<!-- Import the default server configuration -->
<import file="${server.config.file}"/>
. . .
```

Then, our main buildfile targets can be rewritten to be general purpose, simply calling support target defined in the imported file. Any server-specific details about the target can be embedded in the

imported file, and if we want to use a different server platform, we simply replace the imported file with a different one that contains the same targets, implemented to suit the new platform.

Our `create-ejbjars` target, for example, can be generalized to be completely server-agnostic, as shown in [Example 17-11](#).

Example 17-11. Portable EJB archive target

```
. . .
<!-- Create any ejb modules required for this application -->
<target name="create-ejbjars"
    description="Generates an ejb-jar archive for this application"
    depends="compile">
    <!-- Use the as-create-ejb-jar target to create any EJB modules
        needed for this application. -->
    <antcall target="as-create-ejb-jar">
        <param name="as.ejb.name" value="ProfileEJB"/>
        <param name="as.ejb.desc.dir" value="${config.dir}"/>
        <param name="as.ejb.classes.dir" value="${java.classes.dir}"/>
        <param name="as.ejb.jar.dir" value="${basedir}"/>
    </antcall>
</target>
. . .
```

All we're doing here is deferring to an `as-create-ejb-jar` "helper" target, which we expect to be defined in the imported file. The helper target names are prefixed by `as` (short for "application server") to make it easy to distinguish them from the targets defined in the main buildfile itself. When we invoke the helper target, we pass in a set of property values using child `<param>` elements on the `antcall` task. These parameters are specific to the project itself and include details such as where we want the archive created, where the EJB component classes can be found, and so on.

With our main buildfile rewritten to invoke these helper targets, we can define the targets in the imported buildfile to suit the application server being used. In the JBoss "implementation" of the helper targets, for example, we would define the `as-create-ejb-jar` target as shown in [Example 17-12](#).

Example 17-12. JBoss version of the EJB archive "helper" target

```
. . .
<!-- Target: as-create-ejb-jar
    Creates an EJB archive deployable to the target application server. -->
<target name="as-create-ejb-jar"
    description="Assemble an EJB archive using the project properties">
    <echo message="Generating EJB archive '${as.ejb.name}'"/>
    <!-- Invoke the ejbjar task, looking for deployment descriptors
```

```

        in the ${as.ejb.desc.dir} directory. We are using "directory"-based
        naming here, where the name of the directory containing the EJB
        deployment descriptor is used as the basis for the ejb-jar filename. -->
<ejbjar descriptor=" ${as.ejb.desc.dir} "
    srcdir=" ${as.ejb.classes.dir} "
    destdir=" ${as.ejb.jar.dir} "
    naming="directory"
    genericjarsuffix="-generic.jar"
    flatdestdir="true"
    dependency="super">
<!-- Specify the JBoss variant of ejbjar, to ensure that the JBoss
    deployment descriptors are picked up -->
<jboss destdir=" ${as.ejb.jar.dir} "/>
<!-- Use local copies of descriptor DTDs provided by JBoss, in case we're
    not online -->
<dtd
    publicid="-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans 2.0//EN"
    location=" ${jboss.home} /docs/dtd/ejb-jar_2_0.dtd" />
<!-- Parse only EJB descriptors in files ending with "ejb-jar.xml" -->
<include name="**/*ejb-jar.xml" />
</ejbjar>
</target>
. . .

```

If someone wants to use our project buildfile in another server environment, all she needs to do is create a version of the import file with the required helper targets defined for that alternative platform and import the new set of target definitions into the main buildfile.

To make this approach even more portable, we can use a property to define the name of the imported file with the target definitions:

```

. . .
<import file=" ${server.config.file} "/>
. . .

```

This way, we can specify the default value for the filename in our *build.properties* file:

```
server.config.file = as-config.xml
```

Then, assuming we're using the hierarchical properties scheme described earlier, a user can specify an alternative set of target definitions for her local environment in her properties override file:

```
server.config.file = jfarley-as-config.xml
```

17.6.2.1. Importing targets in older Ant versions

As mentioned at the start of this section, the `import` task was introduced in Ant 1.6. If you're using an earlier version of Ant, you can still implement modular target definitions, using standard XML "imports."

An Ant buildfile is an XML file, so you can include entity definitions in the header of the file, per the XML standard. These are placed inside the `DOCTYPE` element for the document, which is placed before any actual content in the file. The following example shows an Ant buildfile defining an entity named `as-config` loaded from a local file named `as-config.xml` and then inserting the entity in the buildfile:

```
<!DOCTYPE project [
  <!ENTITY as-config SYSTEM "file:./as-config.xml">
]>

<project name=. . .>
. . .
  &as-config;
. . .
```

According to the syntax rules of XML, anything in the file referenced by the `ENTITY` element will be included verbatim at the point that the entity is referenced in the XML document. So we can specify our target definitions in a separate XML file and import it in a modular way into our buildfile. If we wanted to change the target definitions to suit the local environment, we could swap out our include file for the default one, or we could use multiple entities to allow users to override the default target definitions with their own:

```
<!DOCTYPE project [
  <!ENTITY as-config SYSTEM "file:./as-config.xml">
  <!ENTITY as-user-config SYSTEM "file:./as-user-config.xml">
]>

<project name=. . .>
. . .
  <!-- Include the user's target definitions first,-->
  <!--so they take precedence-->
  &as-user-config;
  <!--Include the default target definitions -->
  &as-config;
. . .
```

17.6.3. Putting It All Together

We've combined all of these portability schemes into a single Ant template, which you can find in the [Chapter 17-Ant/template](#) directory of the source bundle for this book (you can download it from <http://www.oreilly.com/catalog/javaentnut3>). The helper targets included in the import file cover the creation of web and EJB component modules and application archives and deploying applications to the application server, and we've included versions of the helper targets implemented for a few popular application servers to demonstrate the concept.

With all of these portability measures applied in a project buildfile, developers or deployers can work with the project without having to adjust the project buildfiles directly. They simply need to create a properties file with their custom set of project property values (library paths, filenames, etc.), and an import file containing target definitions tuned to their application server environment, and away they go.



Chapter 18. JUnit and Cactus

Testing software is arguably the most important success factor in software engineering. No matter how elegant your architecture, no matter how compelling the features and user interface, and no matter how well a system is marketed, untested or poorly tested software is failure waiting to happen. Software that's tested is software that works, and working software is a pleasure, or at least a pleasant surprise.

Software testing has many dimensions. You can perform black box functional testing to determine whether an application operates correctly from an external standpoint without concern for the internal workings of the software. You can perform white box functional testing to validate the internal workings of the system at various levels. You can test the performance of software to determine how quickly it executes tasks under typical usage levels. You can perform load testing to determine the load threshold at which the system fails (or executes so slowly that it becomes effectively useless). You can perform user-acceptance testing, which tests software at an aggregate level to verify that a system performs acceptably to the target users, with metrics ranging from functional correctness to speed to interface aesthetics.

Software quality assurance is a subject of study in itself, and we won't presume to summarize it broadly in a single chapter. Instead, what we cover here is the subset of software testing referred to as *unit testing*, which attempts to verify the correct functional behavior of software at various levels of granularity. A "unit" may be a method on a class, an entire class, a component made up of several classes, or an entire subsystem of an architecture containing several components or classes. Strictly speaking, unit testing is typically performed during development on independent units of software, prior to these units being integrated into a complete application and passed along to other testing processes, like user interface functional testing, performance testing, and user-acceptance testing.

Why do we limit our discussion to unit testing? Besides the fact that more complete coverage of testing practices would take several additional chapters, we felt that unit testing tools and techniques provide the broadest applicability and have a necessary coupling with the enterprise framework being used to build systems.

Unit testing has broad applicability because it can be leveraged to provide elements of several of the other flavors of testing we discussed earlier. You can unit test software at a number of levels in your architecture, providing assurance of not only functional correctness but also a certain amount of integration testing. If a component uses a number of subcomponents or classes to execute its functions, testing the functionality of the component ends up testing the integration of these subelements to some degree. Unit tests can also be fashioned to be either black box tests, purposely ignorant of the internal workings of the unit being tested, or they can serve as white box tests, intentionally exercising and validating the internal workings of a unit.

Unit testing is also the tool most likely to be used by the audience we're targeting with this book: software engineers. Unit testing typically involves writing tests in the same programming environment as the software units being tested as well as understanding the workings of the software units at this level (object and component interfaces, data structures, and so on). Software engineers are uniquely qualified to write these unit tests. In part due to this fact, unit testing is a standard element in a number of modern software engineering processes, such as extreme programming (XP)

and all of its many variations, feature-driven development (FDD), and the Unified Process (UP) defined in conjunction with the Unified Modeling Language (UML).

Finally, unit testing has a close coupling with the enterprise framework used to create applications, and since this book is focused on a particular enterprise framework (J2EE), it seemed appropriate to introduce unit testing tools aimed at this framework.

Please don't take our avoidance of performance testing, load testing, or UI functional testing as a statement that these forms of testing are less important than unit testing. These testing practices are equally important for creating successful software. But for the reasons given here, we felt that unit testing was most complementary to the rest of the material in this book and to the target readers of this book.



18.1. What's Covered Here?

This chapter demonstrates unit testing using both JUnit 3.8.1 and Cactus1.7, the latest releases of both of these tools at the time of this writing.

JUnit is a generic Java unit testing framework that can be used in any Java environment. You can download this tool from <http://www.junit.org>. We've written and executed the unit tests demonstrated in this chapter using a JDK 1.4.2 environment.

Cactus is a JUnit extension that allows you to write tests that execute inside a J2EE container. The Cactus framework supports unit testing of various J2EE components, like servlets and EJBs, and other server-side J2EE code, like JSP tags and servlet filters, that depend on a server-side container environment in order to function. You can download Cactus from <http://jakarta.apache.org/cactus>. The Cactus test examples in this chapter were run in JBoss 4.0 and WebLogic 9.0 server environments.

18.1.1. Other Tools to Consider

We chose to cover JUnit and Cactus in this tutorial because, as with the other tools we've chosen, these unit testing tools are in wide use and are a good place to start for your testing needs. Of course, other tools that you can use to supplement or replace the functionality provided by JUnit and Cactus are available. Some of these include:

MockObjects (<http://www.mockobjects.com>)

This site provides a set of frameworks for testing code using the mock objects approach, which is an alternative approach to the in-container testing supported by Cactus. Some of these mock objects frameworks can be used alone, with JUnit, and with Cactus.

StrutsTestCase (<http://strutstestcase.sourceforge.net>)

This JUnit extension supports the testing of Struts actions, mappings, and beans. It can be used with JUnit, with or without Cactus.

18.2. Unit Testing Concepts

Before we dive into the details of testing Java code using JUnit and testing J2EE components using Cactus, let's get acquainted with the concepts and issues underlying unit testing in general.

18.2.1. Units and Dependencies

The most obvious question to ask about unit testing is, "What's a unit?" The most conservative definition of a testable "unit" is an entirely independent module of code that can be tested autonomously, separate from any other code modules. The tangible interpretation of this idealistic definition depends on the programming environment being used. In function-based programming languages, such as C, Perl, and Fortran, a unit is typically defined as a single function. Similarly, a unit in object-oriented languages like C++, Java, and C# would normally be a class.

Of course, achieving this goal of a totally independent unit of code is not entirely realistic. Virtually any code module depends on other code modules to function properly. Functional decomposition encourages algorithms to be broken down into subfunctions that are invoked by higher-order functions to solve a problem. Object-oriented design has the analogous principle of delegation of responsibilities to subordinate classes used by higher-order classes to implement particular behaviors. These common divide-and-conquer approaches to software architecture naturally lead to interdependencies among code elements.

Given this fact, achieving the goal of unit testing (i.e., verifying the valid operation of each unit in the system) takes some forethought and planning in terms of defining your test suite. The dependencies between units need to be understood at a design level, and your tests need to be crafted to exercise low-level units first and then progress to higher-level units. This "reverse pyramid" approach to sequencing your tests will help to identify base issues early in a test suite and provide a basic sense of progress in terms of system stability (the farther you go without failing a test, the more stable the system, in a loose sense).

18.2.2. Unit Testing Versus Integration Testing

It's important to note at this point the distinction between unit testing and integration testing. While unit testing strives to verify the correct operation of individual units of code in isolation, integration testing attempts to verify the correct operation of multiple units working together in a cooperative fashion, integrated together as they would be in the real runtime environment. In the remainder of this tutorial chapter on unit testing, we will occasionally stray into the realm of integration testing with some of our examples. We'd ask the testing experts reading this to forgive us for this breach of orthodoxy, and we'd ask those who are new to unit testing to remember that the distinction between unit and integration testing is important and to keep it in mind when designing tests.

18.2.3. White Box Versus Black Box Testing

It's important to recognize what kind of testing you're attempting when defining a specific test or even a suite of tests. *Black box* testing, as mentioned earlier, refers to the testing of the functional specifications of a unit of code from the standpoint of an external interface specification. *White box* testing refers to validating the functionality of a unit from the standpoint of its internal implementation.

To demonstrate the difference, imagine we are testing a shopping cart module built into an online commerce system. The shopping cart has well-specified behavior that it must implement: items that are added stay in the cart until removed, total prices add up correctly, taxable and nontaxable items are reflected in the overall price, and so forth. In our implementation, we happen to know that the cart is implemented using an efficient list structure that allocates space for items in blocks of 25. Once 25 items are added to the cart, space for an additional 25 items is added to the list structure. When those are filled, space for an additional 25 items is added and so on.

If we were writing black box tests for our shopping cart, we would simply encode the behavioral specifications into tests that exercise that behavior and check for expected results. Our tests would do things like add an item and verify that it is recorded in the shopping cart, add various combinations of types of items, and check that the number and price of the items in the cart are reported correctly.

If we were engaging in white box testing, we would focus on the internal implementation details of the variable-sized list structure and verify that the shopping cart behaves correctly when the list's boundary conditions are encountered. Our white box tests would do things like add 25 items, then add the 26th item and verify that it is recorded correctly, then remove an item so that the item count falls below 25 again, and verify that the cart data is intact after the extra block of list items is removed internally.

In practical testing scenarios, you will find yourself creating a mix of black box and white box tests, sometimes referred to as *gray box* testing. An initial suite of tests may be strictly based on the written specifications for a unit and therefore contain only black box tests. Later, as issues are encountered with the implementation of the unit, they will be fixed and then tests will be added to the test suite to verify that these issues do not return. This is referred to as *regression testing*, ensuring that future changes to the code do not cause the system to regress and exhibit prior bad behavior. In the end, the test suite for a unit will typically contain both black box and white box tests. There's no problem with this as long as you're careful to track which white box tests are coupled with specific implementation details of the code module. If the implementation details change, you want to be able to easily review these tests to ensure that they are still relevant, to prune tests that are no longer relevant, and to add new white box tests specific to the new implementation, if needed.

18.2.4. Keep the Tests Simple

If you haven't written tests before and if you haven't divined it from the discussion so far you'll soon discover the main dilemma with testing: it involves more work. On top of sorting out the requirements, designing the architecture, and writing the code, there's the additional effort of writing tests for the code.^[1] The benefits of testing, as discussed earlier in this chapter, are conceptually obvious but sometimes difficult to remember when you're faced with a deadline.

^[1] Before the software engineering pundits cry foul, please note that there is no implied order to these steps in the process. I'm

not suggesting that testing must follow coding (to the ire of XP advocates everywhere) nor am I assuming a waterfall approach to software development (to the ire of nearly all software engineers).

We mention this here to underscore the value of testing (short-term additional effort leads to big long-term payoffs) but also to emphasize the importance of keeping your tests as simple as possible in terms of the structure of the code that drives the tests. You want to maximize the benefits of testing while minimizing the overhead involved in writing the tests. An overengineered test suite with an excess of support code will take time to write and more time to extend in the future. And each new engineer who comes along will need to take some time to learn how the whole test structure works, on top of figuring out how the actual application code works. At best, you'll be spending more time on overhead and less time writing application code, but an even worse possibility is that you and your team members will neglect the unit tests because of frustration or the squeeze of deadlines. Simple tests obviate these problems to a large degree.

Simple tests also help keep your test results clean and unambiguous. You want to avoid complicating the picture when it comes to evaluating the test results. If there's a lot of logic tied up in the code that drives the tests, you run the risk of bugs in the test code masking bugs in the application code being tested.

18.2.5. Test Coverage

There's an adage in software engineering that states, "If you didn't test it, it doesn't work." In essence, this implies that you should assume that any code that hasn't been tested will fail at some point. This principle leads you naturally to the issue of determining what code has been tested and what code hasn't. In a real-world application, even one of moderate size, it can be difficult to assess test coverage. There are tools available that can help determine test coverage and tools that insert themselves into the runtime environment, for example, and monitor what code is executed during a given test run and report on coverage in the source code. Good, understandable, modular architectures also help enormously in terms of managing the coverage of test suites. A good object-oriented design will have well-defined module interfaces, clean delegation of responsibilities among the modules, and obvious dependencies between the modules, all of which help when defining a test suite that exercises the full expected functionality of the code under test.

Having said all this, it's also important to be realistic about test coverage. It's unreasonable to expect to have 100% coverage of your code in your test suite. A software unit, whether it's a function, an object, a component, or a web service is made up of a set of code consisting of many possible logical pathways. The code takes input data, does work, and generates some output data. In order to exhaustively test a given code unit, you would need to provide it with every possible permutation of input data and invoke the unit so that every single possible logical pathway was traversed and, therefore, every possible output was generated. Not only is it unreasonable to attempt this for every unit in your system, it's also not terribly efficient, because in all the possible permutations of data inputs and code paths, only a subset of them (often a small subset) is relevant to the real usage of the unit in practice. Determining this critical subset is a context-dependent task, but it's an important element of effective unit testing.

Also in the category of test coverage is the management of evolving code. Once you accept the goal of a test suite that covers the critical functionality of your code, you then have to accept the reality that the test suite must evolve to match the code. If new functionality is added, tests for it need to be added as well. If functionality is changed or removed, tests need to be altered or removed to suit.

18.3. JUnit Overview

With this introduction to the concepts of unit testing behind us, let's dive into the technical details of using JUnit to create and execute unit tests on basic Java code. In a later section, we'll see how Cactus can be used to test code that must run in a J2EE container.

JUnit is a framework for writing unit tests that consists of a Java API that you use to write your tests and a set of basic tools that can be used to run your test suites and report on the results.

The JUnit API is provided as two packages: `junit.framework` contains the core classes that you'll typically use to define and execute your unit tests while `junit.extensions` contains some potentially useful extensions to the core framework classes.

In the JUnit model, tests themselves are represented as `TestCase` objects. A set of related tests is organized into a `TestSuite`. A `TestRunner` actually drives a suite of tests and reports on the results. All of these classes are in the `junit.framework` package. The JUnit framework has several other classes and interfaces, but these are the primary ones that you'll need to understand to get started. In keeping with the principle of simplicity we discussed earlier, the JUnit authors kept the class mode in the framework as simple as possible.

The JUnit runtime model is also fairly simple. In a nutshell, executing tests using JUnit typically involves the following steps:

1. You invoke a `TestRunner` on a `TestSuite`. Some `TestRunner` implementations allow you to specify an individual `TestCase` to be run, but in those cases, the `TestRunner` simply wraps the `TestCase` with a `TestSuite` and runs the suite. The `TestRunner` examines the suite of tests by iterating through the suite, finding each `TestCase` contained in the suite.
2. For each `TestCase` in the suite, the `TestRunner` determines which tests are to be run. When adding a `TestCase` to the suite, you can specify that all tests on the `TestCase` are to be run, in which case any method whose name starts with "test" is considered a test method. Alternatively, you can specify individual test methods on the `TestCase`, as described in a later section.
3. For each test method to be run, the `TestRunner` calls the `TestCase`'s `setUp()` method, then invokes its `runTest()` method to execute the test method itself. If the specified test method is not found or is inaccessible (nonpublic), a test failure is generated. You can override the `runTest()` method in your `TestCase` implementations, but this isn't normally necessary unless you're defining a specialized test harness of some kind. Once the test method completes (either successfully or with an error or exception of some kind), the `tearDown()` method on the `TestCase` is invoked, allowing you to clean up any resources set up for the test. The `tearDown()` method is called whether the test succeeds, fails, or generates an uncaught exception.
4. If a test method completes without an exception or error being thrown, the test is marked as passed. If the test method throws an `AssertionFailedError` (e.g., an assert method call in the

test method failed, or one of the `fail()` methods was called in the test method), the test is marked as failed. If an uncaught exception of any other type is thrown by the test method (e.g., a call in the test method generates a `NullPointerException` that is not caught and handled in the test method), the test is marked as an error, meaning that the test encountered an error that prevented a positive or negative result from being recorded.

5. Most `TestRunner` implementations report on the status of each test as it is run. The text-based `junit.textui.TestRunner`, for example, prints an `F` character on the console for each failed test, an `E` for each test that resulted in an error, and an unobtrusive dot (`.`) for each passed test. Once all of the test methods (in aggregate) have been run, the `TestRunner` reports on the total results, typically displaying statistics on the number of successful versus failed tests.

So from your perspective as a developer, unit testing with JUnit involves writing one or more `TestCase` subclasses with appropriate test methods on each, optionally organizing them into `TestSuites`, and then executing suites of tests through a `TestRunner`.

18.3.1. Defining Tests by Writing TestCases

A `TestCase` is meant to represent a set of tests targeted at a specific unit of code. The scope of the "unit of code" is up to you and how you decide to organize your tests overall, but typically a `TestCase` is aimed at a specific class in your overall model. You implement tests for your code unit by subclassing `TestCase` and defining test methods on the subclass. Each test method name must start with "test" and must have the signature:

```
public void testXXX( ) { . . . }
```

This method-naming convention allows the JUnit framework to find test methods through introspection when automatically invoking tests, as described in the previous section.

Traditionally, each `TestCase` corresponds to a class to be tested, but nothing in the JUnit framework enforces this practice. You can write multiple `TestCases` to test a single class in your design, or a single `TestCase` can include tests for multiple classes in your system. The traditional approach of creating one `TestCase` for each class is motivated by the need to assess test coverage and to isolate tests from each other (or at least understand their dependencies). Classes provide a natural organizing principle in both of these areas.

To demonstrate, [Example 18-1](#) shows a `TestCase` written for a `Person` Java bean. The `Person` class under test, as well as the rest of the application code used to generate examples in this chapter, are taken from the PeopleFinder sample application used in [Chapters 2](#) and [17](#).

Example 18-1. Simple TestCase for the Person class

```
import junit.framework.TestCase;
```

```

public class TestPerson extends TestCase {
    /** Default constructor */
    public TestPerson ( ) {
        super( );
    }

    /** Constructor with a name for the test case method */
    public TestPerson (String name) {
        super(name);
    }

    /** Test the name accessors on the Person bean */
    public void testSetName( ) {
        // Create a fresh person
        Person p = new Person( );
        // Set the first and last name fields
        String fName = "John";
        String lName = "Smith";
        p.setFirstName(fName);
        p.setLastName(lName);
        // Verify that the values were accepted and saved
        assertNotNull(p.getFirstName( ));
        assertEquals(p.getFirstName( ), fName);
        assertNotNull(p.getLastName( ));
        assertEquals(p.getLastName( ), lName);
    }

    /** Test the addition of email addresses */
    public void testAddEmail( ) {
        // Create a fresh person
        Person p = new Person( );
        // Set the email address
        String addr = "jsmith@anywhere.com";
        p.addEmailAddress(addr);
        // There should be only a single email address on the person
        assertTrue((p.getEmailAddresses( ).size( ) == 1));
        // The email address should be the value we just added
        String addrR = (String)p.getEmailAddresses().iterator( ).next( );
        assertNotNull(addrR);
        assertEquals(addr, addrR);
    }
}

```

The test class is called `TestPerson`, which follows a common convention used in JUnit with the `TestCase` class name derived by prepending "Test" to the name of the unit under test. Aside from two constructors that we include here to match those provided by the `TestCase` base class, the entire class consists of two test methods. The first, `testSetName()`, exercises and verifies the operation of the name accessors on the `Person` bean, and the other, `testAddEmail()`, tests the addition of email addresses to a person.^[2] In each of our tests, we first set up some test data by constructing a `Person` instance and setting some of its properties, then test for the expected results using the

`assertXXX()` methods available in any `TestCase`.

^[2] It's important to note here that some of our unit test examples are intentionally oversimplified. In the case of `testAddEmail()`, for example, it's unlikely that you will unit test every accessor in a real-world application. The number of unit tests would explode, and their usefulness would be suspect anyway since the implementation of setters and getters are typically extremely simple and not worthy of their own unit tests. But they do make for good examples to demonstrate basic JUnit usage.

18.3.2. Making Assertions in Test Methods

The assertion methods used in a `TestCase` are inherited from the `Assert` base class that `TestCase` extends. You invoke these assertion methods in the test methods on your `TestCases`. If the assertion passes, the method completes normally and the test method continues. If the assertion fails, an exception is thrown, the test method halts, and a failure is reported by the `TestRunner`.

Several flavors of assertions are provided for you to use in your test methods:

`assertEquals()`

These assertion methods allow you to assert that two items have equal values. Several versions are available, allowing you to compare objects and data items of various types (for example, `Strings`, `ints`, and `Objects`).

`assertFalse()/assertTrue()`

These methods can be used to test a predicate (represented as a `boolean` argument) for either truth or falsity.

`assertNull()/assertNotNull()`

These assertions involve checking that a given object is either null or not null.

`assertSame()/assertNotSame()`

These assertions check to see whether two objects refer to the same object or not.

In our `testSetName()` method, for example, we used the `assertNotNull()` method to assert that the first and last name attributes should not be null after they have been set. We also used the `assertEquals()` method to assert that the values should be equal to the intended values.

18.3.3. Failing Tests

A failed assertion causes a test to fail, but in some cases you may need to explicitly fail a test when you encounter a particular condition in your test method. For these situations, all `TestCases` also inherit the `fail()` methods from the `Assert` base class. Calling `fail()` causes the current test

method to fail immediately. You might need to fail a test explicitly if, for example, expected or unexpected exceptions are thrown in your test code. Adding the following test to our `TestPerson` test case, for example, tests the correct behavior of the `equals()` method on the `Person` bean:

```
public void testEquals( ) {
    // Set up a test person with some data
    mPerson.setFirstName("John");
    mPerson.setLastName("Jones");
    mPerson.addEmailAddress("jjones@somewhereelse.org");
    try {
        // The equals implementation on Person should
        // always return true when a Person is compared
        // to itself
        assertTrue(mPerson.equals(mPerson));
    }
    catch (Exception e) {
        // Fail if the equals method generates any exceptions
        fail("Unexpected exception: " + e.getMessage( ));
    }
}
```

In the test, we set up a `Person` object and use the `equals()` method to compare the `Person` with itself, asserting that this should always return true by using the `assertTrue()` method. We've wrapped the call to `equals()` with a `try/catch` block, and in the `catch` block, we fail the test with a descriptive message since the `equals()` method should never throw an exception under these test conditions.

18.3.4. Managing Test Fixtures

If you look at the tests on our `TestPerson` test case in [Example 18-1](#) (and our additional `testEquals()` test just shown), you'll notice that, in each test, we're setting up a `Person` object to serve as the subject of our test. The baseline set of data used for a unit test is sometimes called the *test fixture*. In our `TestPerson` tests, we always use a single `Person` object as the test fixture. To support the use of test fixtures in unit tests, JUnit provides `setUp()` and `tearDown()` methods on `TestCase`. The `setUp()` method is invoked before each test method on a `TestCase` is run, and the `tearDown()` method is called after each test method. You can implement these methods to perform any initialization of data or other resources needed across your test methods and then do any cleanup afterward.

[Example 18-2](#) shows an updated version of our `TestPerson` test case, using the `setUp()` and `tearDown()` methods to manage the fixture for the tests. Note that we've omitted some details here, like the constructors and some of the test methods. In this version of our test case, the test fixture is a `Person` object that we construct and initialize. In our tests, like the `testSetName()` method, we can now use the `Person` object as the subject of our tests. Notice that in our `testSetName()` method, we've opted to reset the first and last name attributes to local values. You might look on this as a waste of time (after all, the `setUp()` method has just initialized the `Person` object for us), but even with a test fixture in place, it might be useful to perform some redundant initialization in your test

methods, such as we've done here, in order to localize expected results in the test method and make your tests more readable.

Example 18-2. TestPerson with a test fixture

```
import junit.framework.TestCase;

public class TestPerson extends TestCase {
    private Person mPerson = null;

    . . .

    /** Set up the test fixture before each test */
    protected void setUp( ) {
        mPerson = new Person( );
        // Set the attributes to some defaults
        mPerson.setFirstName("John");
        mPerson.setLastName("Smith");
    }

    /** Clean up the test fixture after each test */
    protected void tearDown( ) {
        mPerson = null;
    }

    /** Test the name accessors */
    public void testSetName( ) {
        String fName = "Jane";
        String lName = "Doe";
        mPerson.setFirstName(fName);
        mPerson.setLastName(lName);
        assertNotNull(mPerson.getFirstName( ));
        assertTrue(mPerson.getFirstName( ).equals(fName));
        assertNotNull(mPerson.getLastName( ));
        assertTrue(mPerson.getLastName( ).equals(lName));
    }

    /** Test the addition of email addresses */
    public void testAddEmail( ) {
        String addr = "jsmith@anywhere.com";
        mPerson.addEmailAddress(addr);
        // There should be only a single email address on the person
        assertEquals(mPerson.getEmailAddresses( ).size( ), 1);
        // The email address should be the value we just added
        String addrR = (String)mPerson.getEmailAddresses().iterator( ).next( );
        assertNotNull(addrR);
        assertEquals(addr, addrR);
    }
}
```

```

    . . .
}

```

It's important to remember that `setUp()` and `tearDown()` are called for *every* test method on a `TestCase`. This ensures that each test operates on a clean version of the fixture, but it can also cause unexpected performance issues if you do a lot of heavy lifting in these methods. If you're testing a data-caching engine, for example, and load up a large amount of data into the cache before each test method, it may take a very long time for all of your tests to complete. If you want to initialize a set of test data once for an entire suite of tests, you can do that in various ways, such as a static initializer on the `TestCase` subclass.

Once you've written a `TestCase`, you'll want to run its tests periodically in order to verify the correct behavior of the target code. That's the whole point of unit testing, after all. You can run your JUnit test in a number of ways. The most explicit way is to simply invoke a `TestRunner` with the full name of your `TestCase` class, and it will automatically find and execute all test methods on your `TestCase` (using introspection). The following command will run our `TestPerson` tests using the text-based `TestRunner`, for example:

```

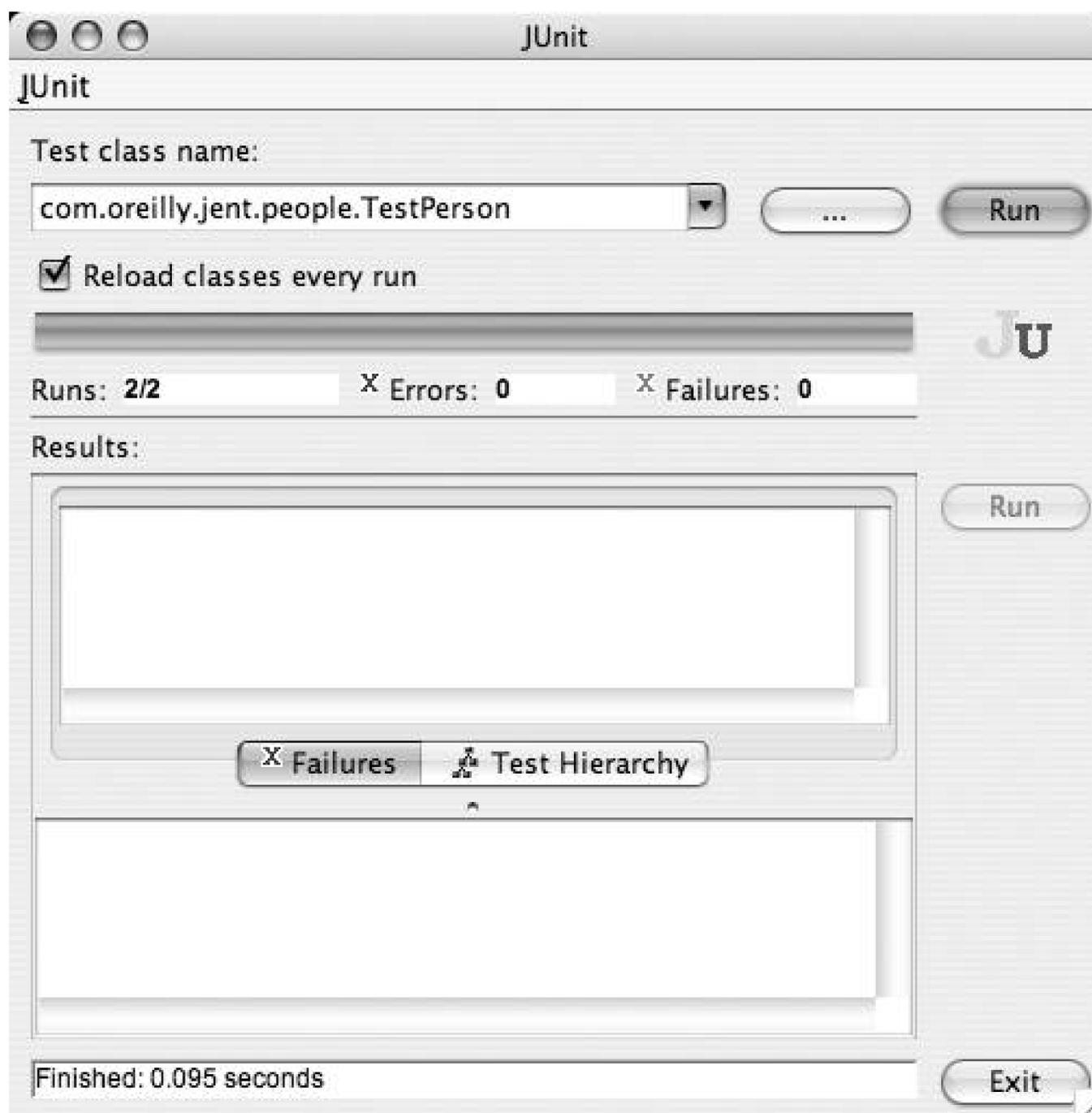
> java junit.textui.TestRunner com.oreilly.jent.people.TestPerson
...
Time: 0.021

OK (3 tests)

```

Notice that the text-based `TestRunner` prints a dot (.) for each successful test it runs and prints a summary of the results. If we ran the same `TestCase` using the Swing-based `TestRunner`, `junit.swingui.TestRunner`, we'd see the results shown in [Figure 18-1](#).

Figure 18-1. TestPerson executed through the Swing-based TestRunner



18.3.5. Organizing Tests Using TestSuites

Using a `TestRunner` to run each of your `TestCases` is not very practical in situations in which you are testing a system with many components and therefore many, many `TestCase` implementations to manage. Also, you may want to have better control over which specific test methods are executed and in what order these tests run. Further, you may want to organize groups of tests into categories that run separately and are managed by different subsets of your development team. There may be a collection of unit tests for each component of a large enterprise system, for example, and the teams responsible for each system component may write and manage the corresponding unit tests. Another set of unit tests might perform specific tests related to integrating these separate components and might run during integration testing prior to a full system release.

JUnit supports organization of tests in these ways through the `TestSuite` class. As mentioned earlier, a `TestRunner` uses a `TestSuite` to run all of its tests. If you don't provide one, it simply wraps the `TestCase` with a `TestSuite` and then operates on the new suite.

JUnit offers an abundance of flexibility in terms of how you can organize your tests into suites.

Starting at the most basic level, you can wrap any `TestCase` with a `TestSuite` by using the `TestCase` as a constructor argument. This example shows how we could create a `TestSuite` that includes all of the tests on our `TestPerson` class:

```
TestSuite suite = new TestSuite(TestPerson.class);
```

We can also specify that a `TestSuite` should include only a specific subset of the tests on a `TestCase`. Every `TestCase` has a constructor that takes a single `String` argument, interpreted as the name of a test method to be executed. If a `TestCase` is constructed with a method name this way and added to a `TestSuite`, only that test method will be run on the `TestCase` when the `TestSuite` is run through a `TestRunner`. If, for example, we wanted to run only our `testSetName()` test on `TestPerson`, we'd construct a `TestSuite` this way:

```
TestSuite suite = new TestSuite(new TestPerson("testSetName"));
```

You can also compose suites with a mix of individual tests and entire `TestCase` instances, using the `addTest()` and `addTestSuite()` methods on `TestSuite`:

```
TestSuite suite = new TestSuite( );
// Add the testSetName test from TestPerson
suite.addTest(new TestPerson("testSetName"));
// Add all the tests on the TestPersonDAO class
suite.addTestSuite(TestPersonDAO.class);
```

As mentioned earlier, a `TestRunner` always wants to operate on a `TestSuite`. When you pass a `TestCase` to a `TestRunner`, the runner first checks to see if the `TestCase` has a static `suite()` method on it. If it does, the runner invokes this method and uses the returned suite as the set of tests to run. If the `TestCase` has no `suite()` method, the runner constructs its own suite containing all of the tests found on the `TestCase`. If we wanted to limit the testing of `Person` to include only the `testSetName()` and `testAddEmail()` tests, for example, we could add the following `suite()` method to `TestPerson`:

```
static public Test suite( ) {
    TestSuite suite = new TestSuite( );
    suite.addTest(new TestPerson("testSetName"));
    suite.addTest(new TestPerson("testAddEmail"));
    return suite;
}
```

When any `TestRunner` is used with `TestPerson`, the runner calls this `suite()` method and gets back a

suite containing only these two test methods.

You can use the `suite()` method to define aggregate `TestCases` that include groups of tests for your system. For example, if we had two `TestCase` subclasses, `TestPerson` and `TestPersonDAO`, that tested the basic model and data access features of the PeopleFinder system, we could define a simple `TestCase` that included them to make it easier for us to drive all tests related to this subset of our system, as shown in [Example 18-3](#).

Example 18-3. Utility `TestCase` that aggregates other `TestCases` into a `TestSuite`

```
import junit.framework.Test;
import junit.framework.TestCase;
import junit.framework.TestSuite;

public class AllModelTests extends TestCase {
    public static Test suite( ) {
        TestSuite suite = new TestSuite("Tests for the PeopleFinder model");
        // First test the Person data bean
        suite.addTestSuite(TestPerson.class);
        // Test the Person DAO, which is dependent on the Person bean
        suite.addTestSuite(TestPersonDAO.class);
        return suite;
    }
}
```

We could then test all of our model-related unit tests by simply invoking this utility `TestCase` with a `TestRunner`, like so:

```
> java junit.textui.TestRunner com.oreilly.jent.people.AllModelTests
```

`TestSuites` can also be used to impose a certain order on the sequence of tests. In our previous example, we would likely want to run the tests on our `Person` bean first, followed by the tests on our `PersonDAO` since the `PersonDAO` has a functional dependency on the `Person` bean. If the `Person` bean tests fail, it is likely that the `PersonDAO` tests will fail as well (or, at a minimum, the results of the `PersonDAO` tests won't mean that much until we're sure that the `Person` bean works properly).

18.4. Using JUnit with Ant

The Ant project build tool includes a few built-in tasks that can be used to integrate JUnit tests into your build scripts. The JUnit support tasks discussed here are included among the optional tasks in the Ant distribution since, in addition to the core Ant libraries, they also require the JUnit libraries in order to run. That means that before using these tasks, you need to ensure that you have the optional Ant tasks as well as the JUnit library in the appropriate `CLASSPATH`. See [Chapter 17](#) for more details on general Ant usage.

It's possible, of course, to use the core Java support tasks to drive JUnit in your Ant buildfiles. For example, we could simply use the `java` task to invoke a `TestRunner` on a compiled `TestCase`, like our `AllModelTests` example:

```
<target name="run-tests"
  description="Run unit tests for the system"
  depends="compile-tests">
  <java classname="junit.textui.TestRunner"
    fork="true">
    <classpath>
      <path refid="java.compile.classpath"/>
      <path location="${java.classes.dir}"/>
      <path location="${junit.jar}"/>
      <path location="${java.test.classes.dir}"/>
    </classpath>
    <arg line="com.oreilly.jent.people.AllModelTests"/>
  </java>
</target>
```

This isn't too difficult—we invoke the `java` task with the name of the `TestRunner` class, and within the task we define the `CLASSPATH` to be used (making sure to include both our compiled application classes and the JUnit library), and we specify the test to be run using an `arg` child on the `java` task, mimicking the use of command-line arguments when invoking a `TestRunner` from the console. Notice that we are using the `fork` option on the `java` task, which runs JUnit in a separate JVM. We do this because the `TestRunner` we are using attempts to exit the JVM when it finishes running the tests, and naturally this raises a security exception when run within the same JVM as the Ant process.

But the built-in Ant tasks for driving JUnit make even this very simple use case a little bit more concise, as we see here where we've rewritten the `run-tests` target to use the built-in `junit` task:

```
<target name="run-tests-junit"
  description="Run unit tests for the system"
  depends="compile-tests">
```

```

    <junit printsummary="on">
      <classpath>
        <path refid="java.compile.classpath"/>
        <path location="${java.classes.dir}"/>
        <path location="${junit.jar}"/>
        <path location="${java.test.classes.dir}"/>
      </classpath>
      <test name="com.oreilly.jent.people.AllModelTests"/>
    </junit>
  </target>

```

For this very simple use of JUnit, the utility of the `junit` task is pretty minimal. By using the `junit` task instead of the general `java` task, we've avoided having to specify the `TestRunner` class (the `junit` task uses its own internal `TestRunner`). We also don't need to use the `fork` option as we did when using the general `java` task because the internal `TestRunner` used by the `junit` task is written to operate in the Ant context and doesn't attempt to exit the JVM when it's done. When using the `junit` task, however, we do need to turn on the `printsummary` option, as shown previously, at a minimum. This prints some basic feedback about the test results in the Ant output. You can set additional output options using the `<formatter>` subelement described in ["Formatting Test Results"](#) later in this chapter.

A few other features of the `junit` task are worth highlighting since they are most directly relevant to unit testing and the ones you're most likely to use.

18.4.1. Controlling the Build with Unit Tests

The example Ant targets shown earlier run the specified JUnit test case and report the results but have no impact on the rest of the Ant build process. If any failures or errors are encountered in the tests, they are reported, but the surrounding build process continues. It's likely that you will want to use the unit tests as conditions to control different parts of the build process. You want to use your JUnit tests to catch functional problems before deploying the application, for example. If the tests succeed, proceed with the deployment; otherwise, don't bother because there are known problems. This is roughly analogous to the conditional generation of class files by the Java compiler, just later in the build pipeline, checking functionality instead of syntax. In the case of the compiler, the rule is, if syntax or other compilation errors are encountered, don't generate a class file. In our buildfile, we might want to extend this conditional logic and say that if compilation succeeds, run the unit tests to verify the functional behavior of the code, but if any functional problems are encountered, don't deploy the compiled code.

Four attributes of the `junit` task can be used for this purpose: `haltonfailure`, `haltonerror`, `failureproperty`, and `errorproperty`. Setting `haltonfailure` or `haltonerror` to true causes the overall build process to stop if any failure or error (respectively) is encountered in the tests. The `failureproperty` attribute value is the name of a property to be set if a failure is encountered in the tests, and the `errorproperty` plays a similar role for errors.

To demonstrate the use of these attributes, let's take a modified version of the `deploy-app` target from the Ant template buildfile (details on the Ant template can be found in [Chapter 17](#)). The `deploy-app` target depends on the `create-app` target to create the deployable application file, which in turn

depends on the code compiling successfully and the component archives (if any) being constructed. In this modified version of the `deploy-app` target, we make the target depend on our `run-tests` target before we deploy the application using the `as-deploy` target:

```
<target name="deploy-app"
  description="Deploy application to application server"
  depends="create-app, run-tests">
  <!-- Deploy the application using the imported deploy target -->
  <antcall target="as-deploy">
    <param name="as.deploy.file"
      value="\${basedir}/\${ant.project.name}.ear"/>
    <param name="as.deploy.server.id" value="default"/>
  </antcall>
</target>
```

The `haltonfailure` and `haltonerror` attributes are the simplest approach to making the build conditional on the test results. We can just set these attributes to true when we invoke the `junit` task, as shown in this modification of the `run-tests` target:

```
<target name="run-tests"
  depends="compile-tests">
  <junit printsummary="on"
    haltonfailure="true">
    <classpath>
      . . .
    </classpath>
    <test name="com.oreilly.jent.people.AllModelTests"/>
  </junit>
</target>
```

With the `haltonfailure` attribute set to true, if any of the tests in our `AllModelTests` test case either fail or generate an error, the `junit` task halts the overall Ant build process. If you want the build to halt only if errors are generated by the tests but continue if any structured failures are encountered, you would use the `haltonerror` attribute. Since we've configured our `deploy-app` target to depend on the `run-tests` target, if the `junit` task aborts the build, the `deploy-app` target is aborted before it tries to deploy the application. And if there were other tasks to be run after the `deploy-app` target, those would be aborted as well.

We can get more flexibility in dealing with failed tests if we use the `failureproperty` and `errorproperty` attributes. With these attributes in use, instead of blindly halting the build process when a test fails or generates an error, the `junit` task simply sets the property named in the attribute value. We can then use the property as a condition in other parts of our buildfile. For example, let's change the `run-tests` target to use the `failureproperty` attribute instead of the `haltonfailure` attribute:

```

<target name="run-tests"
  depends="compile-tests">
  <junit printsummary="on"
    failureproperty="unit-tests-failed">
    <classpath>
      . . .
    </classpath>
    <test name="com.oreilly.jent.people.AllModelTests"/>
  </junit>
</target>

```

With this change, if any tests fail or generate an error, the `unit-tests-failed` property is set. We can use this condition however we'd like in the rest of the buildfile. In our case, we'd like the `deploy-app` target to be skipped if any unit tests fail. So we can set the `unless` attribute on the `deploy-app` target to `unit-tests-failed`, indicating that the target should be run unless this property is set:

```

<target name="deploy-app"
  depends="create-app,run-tests"
  unless="unit-tests-failed">
  . . .

```

Since the target depends on the `run-tests` target, `run-tests` is run first, setting the property if any tests fail. The `deploy-app` target's `unless` attribute is then checked, and the target is skipped if the property is set. But if any additional targets or tasks are queued to be run (e.g., the `deploy-app` target was invoked as part of another target), these subsequent tasks still execute unless they are also made conditional on the `test` property.

Note that the `<test>` subelement on the `junit` task also has `if` and `unless` attributes that you can use to conditionally run the tests themselves. We might have multiple `<test>` elements, for example, and we want to stop running tests once the first error or failure is encountered. If the surrounding `junit` task is using the `failureproperty` attribute as shown earlier, we can use the `unless` attribute on the `<test>` elements to do this, like so:

```

<target name="run-tests"
  depends="compile-tests">
  <junit printsummary="on"
    failureproperty="unit-tests-failed">
    <classpath>
      . . .
    </classpath>
    <test name="com.oreilly.jent.people.TestPerson"
      unless="unit-tests-failed"/>
    <test name="com.oreilly.jent.people.TestPersonDAO"
      unless="unit-tests-failed"/>
  </junit>
</target>

```

18.4.2. Running Batches of Tests

So far, we've used only the `<test>` subelement of the `junit` task to run single `TestCases`. The `junit` task also supports the running of multiple tests through the use of the `batchtest` subelement. The set of tests to be run is specified using a `<fileset>` element that contains either the test classes or their Java source files. So if we wanted to execute every test found in the directory where we compiled our test classes, we would do something like this:

```
<target name="run-tests"
  depends="compile-tests">
  <junit printsummary="on"
    failureproperty="unit-tests-failed">
    <classpath>
      . . .
    </classpath>
    <batchtest unless="unit-tests-failed">
      <fileset dir="${java.test.classes.dir}">
        <include name="**/*.class"/>
      </fileset>
    </batchtest>
  </junit>
</target>
```

Notice that we're also using the `unless` attribute here since `batchtest` supports both the `if` and `unless` attributes.

In the previous example, we used the test classes to specify the batch to run. We can also use the Java source files, like so:

```
<target name="run-tests"
  depends="compile-tests">
  <junit printsummary="on"
    failureproperty="unit-tests-failed">
    <classpath>
      . . .
    </classpath>
    <batchtest unless="unit-tests-failed">
      <fileset dir="${java.test.src.dir}">
        <include name="**/*.java"/>
      </fileset>
    </batchtest>
  </junit>
</target>
```

You can use multiple `<test>` and `<batchtest>` elements in your `junit` task. In our example, we might want to run our suite of model tests using a `<test>` element and selectively run some of our other tests using the `<fileset>` in a `<batchtest>` element:

```
<target name="run-tests"
  depends="compile-tests">
  <junit printsummary="on"
    failureproperty="unit-tests-failed">
    <classpath>
      . . .
    </classpath>
    <test name="com.oreilly.jent.people.AllModelTests"/>
    <batchtest>
      <fileset dir="${java.test.classes.dir}">
        <include name="**/TestDataUtils.class"/>
        <include name="**/TestDBUtils.class"/>
      </fileset>
    </batchtest>
  </junit>
</target>
```

18.4.3. Formatting Test Results

Running tests and detecting failures are useful in their own right within an Ant buildfile, but even more useful is collecting the results of the tests for viewing, reporting, and analysis. You might, for example, use a nightly build process that does a fresh build of all your system code followed by a run of your unit tests. In that situation, you'll want to collect the results and report them on a web page or through email to the members of the development team.

The format and destination of the test results can be controlled using the `<formatter>` subelement on the `junit` task. You can also use attributes on the `junit` task to generate test output: the `printsummary` attribute on the `junit` task prints a one-line summary after each `<test>` or `<batchtest>` runs (as we've done in all our previous examples), and the `showoutput` attribute prints the output from all tests. But the output generated using the `junit` attributes is written only to the Ant output console, which is useful for eyeballing the results as the tests are running, but not very useful for other reporting purposes. The `<formatter>` element lets you write the results to a file and format the results in various ways.

When you use the `<formatter>` element, you have to specify the type of the formatter. The simplest way to do this is using the `type` attribute. Three supported formatter types are `xml`, `plain`, and `brief`. The `plain` and `brief` types generate the test results in a plain-text format, while the `xml` type generates results in an XML format suitable for postprocessing.

The following example runs our model tests using the `plain` formatter:

```

<target name="run-tests"
  depends="compile-tests">
  <junit printsummary="on">
    <classpath>
      . . .
    </classpath>
    <formatter type="plain"/>
    <test name="com.oreilly.jent.people.AllModelTests"/>
  </junit>
</target>

```

If we don't adjust the destination of the results output, it will be written by the formatter to a file named *TEST-`<test case classname>.txt`*. In our case, the results are stored in the file *TEST-com.oreilly.jent.people.AllModelTests.txt*, and the contents look something like this:

```

Testsuite: com.oreilly.jent.people.AllModelTests
Tests run: 3, Failures: 0, Errors: 0, Time elapsed: 0.033 sec

Testcase: testSetName took 0.017 sec
Testcase: testAddEmail took 0.001 sec
Testcase: testEquals took 0 sec

```

Note that each individual test case that is run through a `<test>` or `<batchtest>` element generates its own separate results file, formatted according to the specified formatter. That means each `<test>` element generates a results file, and each test matched by the `<fileset>` of a `<batchtest>` element generates its own results file as well.

If we ran the same tests with the `xml` format type, the results would be placed in a file named *TEST-com.oreilly.jent.people.AllModelTests.xml* and would look like this:

```

<?xml version="1.0" encoding="UTF-8" ?>
<testsuite name="com.oreilly.jent.people.AllModelTests"
  tests="3" failures="0" errors="0" time="0.16">
  <properties>
    <property name="java.vendor" value="Apple Computer, Inc."></property>
    . . .
  </properties>
  <testcase name="testSetName"
    classname="com.oreilly.jent.people.TestPerson"
    time="0.014"></testcase>
  <testcase name="testAddEmail"
    classname="com.oreilly.jent.people.TestPerson"
    time="0.0010"></testcase>
  <testcase name="testEquals" classname="com.oreilly.jent.people.TestPerson"
    time="0.0"></testcase>
  <system-out><![CDATA[ ]></system-out>

```

```

    <system-err><![CDATA[ ]]></system-err>
</testsuite>

```

The XML format has a root `<testsuite>` element whose elements include the name of the test class that was run and summary data about the overall run. The first child element is a `<properties>` element that contains all of the Ant property values as they were set when the test was run. This information is useful for debugging purposes, but also helps you to ensure consistency across test runs. We've (mercifully) truncated the properties list in the sample results shown.

After the properties, the results of each test are output as `<testcase>` elements. Finally, any output that might have been printed to the system output or error streams is listed as `<system-out>` and `<system-err>` elements, respectively. In our case, the tests didn't write anything to output, so these elements are empty.

You can include a `<formatter>` element as a child of the `junit` task itself or specify `<formatter>` elements within individual `<test>` and `<batchtest>` elements. If a `<formatter>` is specified on a `<test>` or `<batchtest>` element, it overrides the main formatter for that test or set of tests.

If the standard text or XML formats aren't sufficient for your needs, you can write a custom formatter of your own by subclassing the `org.apache.tools.ant.taskdefs.optional.junit.JUnitResultFormatter` bundled with the Ant distribution. However, since the XML format provides all of the result data, you can also use an XSLT transformation to convert the results into whatever format you need (for example, text, HTML, PDF, or another XML format). Ant even provides another optional task, `junitreport`, which facilitates this. You can specify a set of XML test result files, and the `junitreport` task will merge the XML files into a single document and apply an XSLT stylesheet to the merged XML file to generate a final output report. The `junitreport` task has default stylesheets that generate an HTML-formatted report file, but you can provide your own as well. See the Ant documentation on the `junitreport` task for more details.

You can control (to a degree) the destination of the formatted results as well, using the `extension` attribute on the `<formatter>` element, and the `outfile` attribute on the `<test>` element. As we mentioned earlier, each built-in formatter (`xml`, `plain`, or `brief`) has a default file extension (`.xml` for the XML formatter and `.txt` for the plain and brief formatters). You can override these by specifying your own extension using the `extension` attribute on the `<formatter>` element, for example:

```

<junit . . .>
    . . .
    <formatter type="xml" extension=".fooxml"/>
    . . .
</junit>

```

Each `<test>` element can also have an `outfile` attribute, which overrides the base filename of the results file for that test. Note that one of the disadvantages of using the `<batchtest>` element is that you can't override the default filenames of the results files for the matching tests.

18.5. Testing Enterprise Components with Cactus

As we've seen, JUnit is a very capable tool for defining, running, and reporting on unit tests for basic Java code, such as the data access classes, model classes, and utility classes in your system. Enterprise components, however, present a unique challenge when it comes to unit testing. In order to effectively and accurately test enterprise components, you need to have a way to drive the components (i.e., provide them with the environment settings and inputs that they'll encounter at runtime) and capture the results so that they can be verified against expected results.

At a high level, the two general approaches to testing enterprise components are:

Mock objects model

In this scheme, the enterprise component is instantiated in a standard Java virtual machine, and the inputs and environment settings for the component are simulated using special implementations called mock objects. The advantage of the mock objects model is that it can be applied in any environment and does not require a full enterprise server environment. The disadvantage is that your components are not tested in a true container environment, which could lead to inaccurate or misleading test results in some cases.

In-container testing

In this scheme, the enterprise components run within an actual container environment, and the test environment drives the tests, either from within the server environment or externally through a special client. This approach has the advantage of testing the components in a more realistic environment using an actual J2EE container to process the inputs and exercise the container. This approach has the disadvantage of being more complicated to implement: a container-enabled server has to be available to test the components, and managing the test runs (driving the tests and collecting the results) is complicated by the fact that the tests are running within a server environment.

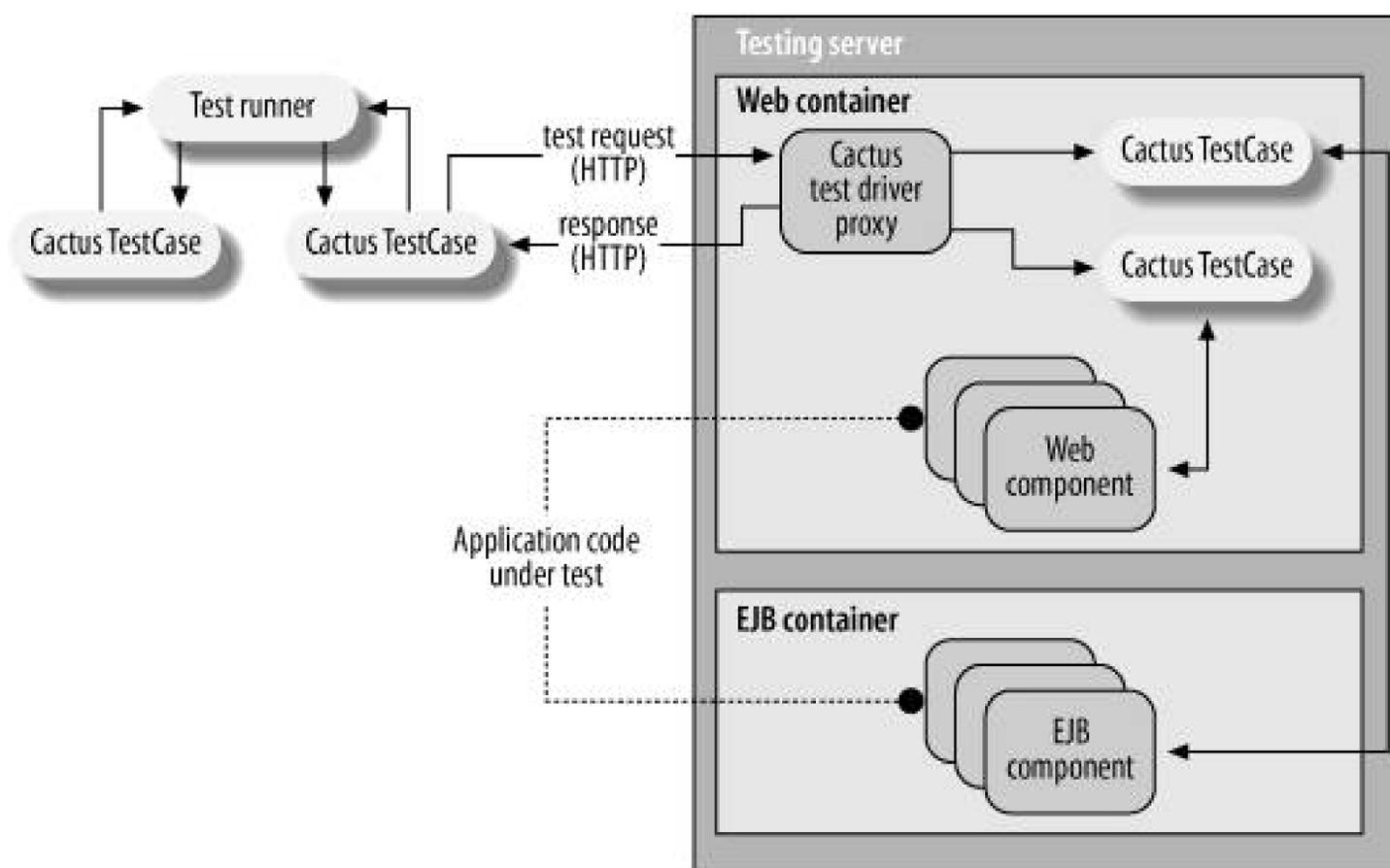
In this part of the chapter, we discuss Cactus, which is an extension of JUnit managed by the Apache Jakarta project. Cactus is an in-container enterprise testing framework. We are not covering mock objects approaches here because, for the most part, the principles we discussed in the earlier JUnit material can be used to manage mock objects tests. In terms of alternative in-container testing frameworks, we chose to discuss Cactus for the same reasons we chose other enterprise tools in this book: it is popular, it has an active development effort behind it, and it works well with other popular enterprise tools, such as Ant, various IDEs, and J2EE servers of various types.

18.5.1. The Cactus Model and Architecture

Cactus uses a proxy architecture to run server-side tests, as shown in [Figure 18-2](#). A JUnit `TestRunner` invokes a Cactus-based `TestCase`, and the `TestCase` invokes a Cactus proxy on a

designated server. The Cactus proxy runs inside the server's web container, taking requests from clients to run specific tests. The Cactus proxy runs the tests on behalf of the client, and the tests exercise various server-side components to test their behavior. The proxy collects the results (success, failure, or error) and returns them to the client to be reported. The `CactusTestCase` (the client in this model) accepts the results and passes them on to the `TestRunner`, and the `TestRunner` reports the results just as it does for "regular" JUnit tests.

Figure 18-2. Cactus runtime model



As the figure indicates, testing your server-side code with Cactus involves enhancing your application archive to include the necessary Cactus libraries as well as your Cactus-based unit tests, and then driving these tests through a Cactus proxy using a testing client of some sort. So to use Cactus to test your enterprise applications, you need to understand the process used by the Cactus proxies to drive your test, the different types of proxies that Cactus provides for running different flavors of test cases, the types of test clients provided by Cactus for invoking your server-side tests, how Cactus unit tests are actually written, and how to "Cactus-enable" your enterprise application archive to allow your unit tests to be run.

18.5.1.1. The Cactus test process

Let's look at the deeper details behind the Cactus test process shown in [Figure 18-2](#). These details are essentially the same regardless of what kind of test you're running and which Cactus proxy is handling the tests.

1. Starting a test run that includes Cactus tests is the same as any JUnit test run. You start a `TestRunner` and tell it which tests to run.
2. For each test method to be executed, the `TestRunner` invokes the `runTest()` method on the corresponding `TestCase` it's asked to run. Again, this is no different from any other JUnit tests.
3. If a given test is a Cactus test (i.e., it extends one of the `CactusTestCase` base classes as described in "[Writing Cactus Tests](#)" later in this chapter), its `runTest()` method checks for a `begin(WebRequest)` method on the `TestCase` and tries to execute it before running each test included in the `TestCase`. This method can be used to initialize any general resources or do any setup required by all tests. The `WebRequest` argument is a Cactus object that represents the simulated web request that will be passed to your test. You can use this object in the `begin()` method to add parameters such as required headers or URL arguments to the request. Again, `begin()` is called before each test method on the `TestCase`, so it should do only general-purpose setup applicable to all tests.
4. If the test method to be run is called `testXXX()`, `runTest()` looks for a `beginXXX(WebRequest)` method on the `TestCase` and executes it before running the test. This method serves the same general purpose as `begin()` (that is, adding URL arguments, HTTP headers, and the like to the request sent to the test proxy) but is specific to a particular test method.
5. At this point, the `runTest()` method on the `TestCase` connects to the appropriate Cactus proxy on the test server and asks it to run the target test method on this `TestCase`. Any parameters or header values set in the `WebRequest` object by the `begin()` or `beginXXX()` methods are passed to the server proxy. The proxy that is contacted depends on the type of `TestCase` being run. If the `TestCase` implements `ServletTestCase`, the `ServletRedirector` proxy is invoked. A `JspTestCase` is run through the `JspRedirector`, and a `FilterTestCase` is run through the `FilterRedirector`. There are system properties that Cactus uses to locate the test server and the proxies running there. They are detailed in "[Running Cactus Tests](#)," later in this chapter.
6. The Cactus proxy accepts the test run request and creates an instance of the target `TestCase` on the server. The proxy creates Cactus wrappers for all the necessary web context objects (i.e., servlet tests are given an `HttpServletRequest` and `HttpServletResponse`, `ServletConfig`, and `ServletContext`). These objects wrap the real context objects generated as part of the web call and ensure that the tests see realistic responses when context information is requested (e.g., the servlet URL available from an `HttpServletRequest` is representative of the web component being tested, not the Cactus proxy). These web context objects are set as member variables on the `TestCase` (through the Cactus base classes that they extend).
7. With the `TestCase` primed, the proxy now acts as the test driver and calls the `setUp()`, `testXXX()`, and `tearDown()` methods on the `TestCase`, in order. The `setUp()` and `tearDown()` methods can be used for server-side initialization and cleanup for each test to be run, as opposed to the `begin()` and `end()` methods, which perform initialization and cleanup on the client side of the Cactus proxy. The test method should exercise the server-side code it's supposed to test (invoke a servlet, JSP, JSP custom tag, servlet filter, EJB, or any other server-side code you want to test). Assertions and failures should be used just as with regular JUnit tests. A failure or uncaught exception generated by the test is picked up by the Cactus proxy and reported back to the caller.
8. Back on the client side of the Cactus proxy, the results of the test method are collected by the `TestRunner`. After each test method finishes, the corresponding `endXXX(WebResponse)` method on the `TestCase` is run (if it exists), letting you do any test-specific verification of the response

returned by the test call. Then the generic `end(WebResponse)` method is called, allowing you to perform any generic verification of the response. In either of these methods, you can use assertions and failures as you would in your test method. Note that both of these method calls (`endXXX()` and `end()`) are done on the `TestCase` instance on the client side of the proxy.

The three test proxies provided by Cactus alter this general test process in terms of the context objects that they provide to the `TestCase`, as described next.

18.5.1.2. Cactus test proxies

Cactus provides three types of proxies (called "redirectors" in the Cactus documentation) for driving your server-side tests. Each proxy is implemented as a different type of web component to test server-side code that needs to be run in various web contexts:

Servlet proxy

This test proxy is a servlet that exercises tests from within a servlet runtime context. This provides tests with access to servlet runtime objects, such as `HttpServletRequest`, `HttpServletResponse`, `ServletConfig`, and `ServletContext` objects. The servlet proxy is useful for testing servlets or for testing server-side code that may be invoked from a servlet, like EJBs or server-side utility code.

JSP proxy

This test proxy is a JSP that exercises tests from a JSP context. This provides tests with access to JSP runtime objects, such as the JSP `PageContext`. The JSP proxy is useful for testing JSPs, JSP custom tags, or other server-side code that may be invoked from a JSP.

Servlet filter proxy

This test proxy is a servlet filter that exercises tests from a servlet filter context. This provides tests with access to runtime objects available using a servlet filter, such as the `FilterConfig`. The servlet filter proxy is useful for testing servlet filters or other server-side code that may be invoked from a servlet filter.

In order to make use of these test proxies, they need to be added to the `web.xml`/deployment descriptor of a web module. To deploy the servlet proxy, your deployment descriptor will need the following `<servlet>` element and a `<servlet-mapping>` similar to the one shown in [Example 18-4](#).

Example 18-4. web.xml entries for Cactus servlet proxy

```

. . .
<servlet>
  <servlet-name>ServletRedirector</servlet-name>
  <servlet-class>

```

```

        org.apache.cactus.server.ServletTestRedirector
    </servlet-class>
</servlet>
. . .
<servlet-mapping>
    <servlet-name>ServletRedirector</servlet-name>
    <url-pattern>/ServletRedirector</url-pattern>
</servlet-mapping>
. . .

```

To deploy the JSP proxy, you'll need the entries shown in [Example 18-5](#).

Example 18-5. web.xml entries for Cactus JSP proxy

```

. . .
<servlet>
    <servlet-name>JspRedirector</servlet-name>
    <jsp-file>/jspRedirector.jsp</jsp-file>
</servlet>
. . .
<servlet-mapping>
    <servlet-name>JspRedirector</servlet-name>
    <url-pattern>/JspRedirector</url-pattern>
</servlet-mapping>
. . .

```

To deploy the servlet filter proxy, you'll need the entries shown in [Example 18-6](#).

Example 18-6. web.xml entries for Cactus servlet filter proxy

```

. . .
<filter>
    <filter-name>FilterRedirector</filter-name>
    <filter-class>
        org.apache.cactus.server.FilterTestRedirector
    </filter-class>
</filter>
. . .
<filter-mapping>
    <filter-name>FilterRedirector</filter-name>
    <url-pattern>/FilterRedirector</url-pattern>
</filter-mapping>
. . .

```

Note that the proxy implementations (the classes that implement the servlet and servlet filter proxies and the JSP for the JSP proxy) need to be available in your web archive as well. The classes are included in the main Cactus jar file, and the JSP proxy file is found in the Cactus distribution. The Cactus proxies also require a set of support libraries, which are included in the Cactus distribution. The whole point of having the proxies included in the web archive is to execute your unit tests, so the test classes, plus any application code that they test, also need to be included in the web archive.

You can make all of the Cactus-related enhancements to your web archive by hand, if you choose. If you are using Ant to build your project, Cactus also provides an Ant task called `cactifywar` that you can use to make these enhancements automatically. Before using the Cactus tasks, you'll need to load them into your buildfile. You can then use the tasks to augment a war file that you've already created:

```
<!-- Include the Cactus Ant tasks -->
<taskdef resource="cactus.tasks">
  <classpath>
    <pathelement location="${cactus.home}/lib/junit-3.8.1.jar"/>
    <pathelement location="${cactus.home}/lib/cactus-1.7.1.jar"/>
    <pathelement location="${cactus.home}/lib/cactus-ant-1.7.1.jar"/>
    <pathelement
      location="${cactus.home}/lib/commons-httpclient-2.0.jar"/>
    <pathelement
      location="${cactus.home}/lib/commons-logging-1.0.3.jar"/>
    <pathelement location="${cactus.home}/lib/aspectjrt-1.1.1.jar"/>
  </classpath>
</taskdef>
. . .
<!-- Create a Cactus-enhanced version of our war file -->
<cactifywar srcfile="myapp.war"
  destfile="myapp-test.war">
  <!-- Add our test classes to the war's classes dir -->
  <classes dir="test-classes" includes="**/*.class"/>
</cactifywar>
```

The `<taskdef>` shown here loads all of the Cactus-supplied Ant tasks from the Cactus jars. The `cactus.home` variable referenced in the example must be set to the directory where Cactus is installed.

The invocation of `cactifywar` in the example takes the web archive `myapp.war`, adds the Cactus proxy entries to the `web.xml` file, adds the Cactus libraries to the `WEB-INF/lib` directory of the archive, puts the JSP proxy `jspRedirector.jsp` into the root of the web archive, and adds our test classes to the `WEB-INF/classes` directory in the archive. The `cactifywar` task is a specialization of the built-in `war` task included with Ant, so it supports all of the `war` options. You can use the `<lib>` or `<classes>` elements, for example, to add additional classes required by your tests but not included in the original war file. In addition, the `cactifywar` task supports `<servletredirector>`, `<jspredirector>`, and `<filterredirector>` subelements that can be used to customize the URL mappings used for the proxies. If, for example, you want the JSP proxy to be accessible at the URL

`/runtests/jspproxy`, you could use the `<jspredirector>` element like so:

```
<!-- Create a Cactus-enhanced version of our war file -->
<cactifywar srcfile="myapp.war"
           destfile="myapp-test.war">
  <!-- Add our test classes to the war's classes dir -->
  <classes dir="test-classes" includes="**/*.class"/>
  <!-- Specify a custom mapping for the JSP proxy -->
  <jspredirector mapping="/runtests/jspproxy"/>
</cactifywar>
```

The `cactifywar` task adds the appropriate mapping entries in the generated `web.xml` file and, instead of the default mapping shown in [Example 18-5](#), the JSP proxy is given the following mapping as specified in the `<jspredirector>` element:

```
. . .
<servlet>
  <servlet-name>JspRedirector</servlet-name>
  <jsp-file>/jspRedirector.jsp</jsp-file>
</servlet>
. . .
<servlet-mapping>
  <servlet-name>JspRedirector</servlet-name>
  <url-pattern>/runtests/jspproxy</url-pattern>
</servlet-mapping>
. . .
```

18.5.2. Writing Cactus Tests

Luckily, all of the proxy-based test processing discussed in the previous sections is handled internally for you by Cactus. Writing a Cactus test is just a matter of extending one of the provided `TestCase` subclasses and implementing your test methods. Of course, we're talking about testing server-side code, so there's the added complexity of ensuring that the server code sees the same context information that it will in the production environment. But the Cactus framework does a lot to make this as simple as possible.

Cactus provides three subclasses of `TestCase` that you can use as the base of your own server tests: `ServletTestCase`, `JspTestCase`, and `FilterTestCase`, all within the `org.apache.cactus` package. Choosing between these `TestCase` variations is very simple. If your server tests need to run within the context of a servlet, use `ServletTestCase`; if they need to run in the context of a JSP, use `JspTestCase`; if they need to run in the context of a servlet `Filter`, use `FilterTestCase`. Each Cactus `TestCase` subclass makes context-specific objects available to the tests. A `ServletTestCase` has `HttpServletRequest` and `HttpServletResponse` objects, the `HttpSession` associated with the "client," and the `ServletConfig`. A `JspTestCase` has access to all of these objects plus the JSP `PageContext` and

the `JspWriter` output stream. A `FilterTestCase` has access to the `HttpServletRequest` and `HttpServletResponse` objects as well as the `FilterConfig` and `FilterChain` objects needed in a servlet `Filter` context. These context objects can be manipulated by your test code prior to invoking the code being tested, and they can be checked after the test runs to ensure that the expected results were generated.

18.5.2.1. Introduction to Cactus tests: Testing servlets

Let's start by writing tests for a servlet. If you're not already familiar with the servlet programming model and servlets' lifecycle within the web container, you should refer to [Chapter 3](#).

Specifically, we're going to write tests for a servlet that implements the search functionality in the person directory application from [Chapter 2](#). The servlet being tested is `PeopleFinderServlet` (found in the `com.oreilly.jent.people.servlet` package in the sample code bundle). It accepts URL arguments that specify the first name, last name, and email search terms, searches the underlying data source for people that match, and generates a web page showing the results. We won't actually list the code for the servlet before writing our tests (which should make extreme programming advocates happy). Instead, we'll write tests according to how we expect the servlet to behave (based on hypothetical functional specifications).

Since we're testing a component that requires a servlet context in order to be tested, we define our `TestCase` as an extension of `ServletTestCase`. We'll start with a single test on our `TestCase` implementation, which is shown in [Example 18-7](#).

Example 18-7. Servlet test example

```
// Imports omitted...

public class TestPeopleFinderServlet extends ServletTestCase {
    /** Common servlet instance, which serves as our test fixture */
    private PeopleFinderServlet mPFServlet = null;

    /** Constructors inherited from ServletTestCase */
    public TestPeopleFinderServlet( ) {
        super( );
    }
    public TestPeopleFinderServlet(String name) {
        super(name);
    }
    public TestPeopleFinderServlet(String name, Test t) {
        super(name, t);
    }

    /** Set up the test fixture. In this case, create the servlet instance */
    protected void setUp( ) {
        // Create the servlet instance. We also initialize the servlet
        // here, so that we know it's done only once. If any tests
```

```

// require an uninitialized servlet for some reason, the
// test will have to create its own servlet instance.
mPFServlet = new PeopleFinderServlet( );
try {
    mPFServlet.init(this.config);
}
catch (ServletException se) {
    fail("Unexpected servlet exception while setting up test case: "
        + se.getMessage( ));
}
}

/** Invoked on the "client side" (within the test runner) before
 * asking the proxy to run the validSearch test */
public void beginValidSearch(WebRequest request) {
    // Add a search parameter
    request.addParameter(PersonDAO.FIRST_NAME, "John");
}

/** Test method, run on the server side by the proxy. This test ensures
 * that a valid response is generated from a valid search request */
public void testValidSearch( ) {
    // Simply invoke the servlet's doGet( ) method, using
    // the request constructed in the begin method
    invokeGet( );
}

/** Invoked on the "client side" (within the test runner) after the
 * proxy runs the validSearch test and sends a response to
 * the test runner */
public void endValidSearch(WebResponse response) {
    // Since this was a valid search, there should be no "Error"
    // text in the response HTML, and a "Search Results" header
    // should be included
    assertTrue((response.getText( ).indexOf("Error:") < 0) &&
        (response.getText( ).indexOf("Search Results") >= 0));
}

/** Tear down the test fixture. Invoke the servlet's destroy method,
 * to ensure any servlet cleanup occurs */
protected void tearDown( ) {
    mPFServlet.destroy( );
    mPFServlet = null;
}

// Utility methods used by the tests
/** Invoke the get method on the servlet in our fixture. If any
 * exceptions are generated, fail the test.*/
private void invokeGet( ) {
    try {
        mPFServlet.doGet(this.request, this.response);
    }
}

```

```

        catch (IOException ioe) {
            fail("Unexpected IO exception: " + ioe.getMessage( ));
        }
        catch (ServletException se) {
            fail("Unexpected servlet exception: " + se.getMessage( ));
        }
    }
}

```

In our `TestPeopleFinderServlet` test case, we have a single test method, `testValidSearch()`, that exercises the basic functions of the servlet and tests for an appropriate response. To drive this test, we've written a `beginValidSearch()` method that adds a set of URL arguments to the web request. As depicted earlier in [Figure 18-2](#), the `beginValidSearch()` method is invoked on the "client" side of the test session, prior to making the HTTP request to the Cactus proxy. Our `beginValidSearch()` method has the required method signature as dictated by Cactus: the base of the method name (after the "begin") matches the base of the test method name (after the "test"), it is public, has a void return type, and takes a single `WebRequest` argument. The `WebRequest` encapsulates the details of the actual HTTP request that is generated and passed on to the Cactus proxy when it is asked to invoke the test method. In our `beginValidSearch()` method, we add a URL argument whose name is taken from the `PersonDAO.FIRST_NAME` constant (the name that the servlet uses to look up this argument) and with a value of "John". According to our understanding of the servlet being tested, this should be considered a valid search request.

The `testValidSearch()` method itself simply invokes the servlet being tested, using the `invokeGet()` utility method we've defined on our test case. In the `invokeGet()` method, we use a test fixture consisting of an instance of the `PeopleFinderServlet` class and simply invoke the servlet's `doGet()` method, passing along the `HttpServletRequest` and `HttpServletResponse` objects available in the `ServletTestCase` member variables. The request will include the URL arguments that we added in our `beginValidSearch()` method, so we just need to get the request processed by our servlet.

This approach to testing a servlet (directly constructing a servlet instance and exercising its methods) may seem a bit odd to those familiar with J2EE component development. Components are always constructed by their corresponding containers, and the component methods are invoked by containers while being managed at runtime. But handling servlets and other components directly is useful and sometimes necessary when writing Cactus tests. If you need to invoke utility methods on your components with specific input arguments, for example, you'll need direct access to a component instance, and you'll need to construct one yourself to make that happen since web containers do not directly expose servlets and JSP components for programmatic access. Another reason to use a direct servlet instance is to allow you to control the initialization of the component. You can adjust the contents of the `ServletConfig` object passed into the servlet's `init()` method before its request-handling methods are executed.

You can, however, choose to exercise a servlet that has been deployed in the standard way within a web container as long as it's the same web container in which the Cactus tests are being run. First, you'll need to ensure that the servlet being tested has been deployed in the application containing the Cactus proxy and the Cactus tests being run. This entails the usual servlet deployment details: the web deployment descriptor (*web.xml*) needs to have a servlet entry and at least one `servlet-mapping` entry for the servlet being tested. Then, in your Cactus test, during the execution of one of the `testXXX()` methods on the server side of the proxy, you can forward the HTTP request and response pair provided by the proxy to the servlet by using the `RequestDispatcher` available through the

`ServletContext`. If our `PeopleFinderServlet` has been deployed and mapped to the relative URL `/search-servlet`, for example, we can use this approach to invoke the servlet managed by the container:

```
try {
    RequestDispatcher dispatcher =
        config.getServletContext( ).getRequestDispatcher("/search-servlet");
    dispatcher.forward(request, response);
}
catch (IOException ioe) {
    fail("Unexpected IO exception: " + ioe.getMessage( ));
}
catch (ServletException se) {
    fail("Unexpected servlet exception: " + se.getMessage( ));
}
```

This approach offers you the advantage of fidelity: the servlet component is being tested while operating in the same runtime environment that it will experience when it is "live." But you can use this approach only for tests that can be executed using an HTTP request to the web component.

Taking a look at the initialization of our Cactus test fixture, we see that, just as with any JUnit test, we initialize our test fixture in the `setUp()` method and clean up the fixture in the `tearDown()` method. In `setUp()`, we initialize our servlet instance by constructing an instance and then calling its `init()` method, passing in the `ServletConfig` object member variable provided to all `ServletTestCase` instances. The Cactus proxy initializes this and other servlet context variables that our `TestPeopleFinderServlet` test case inherited from the Cactus `ServletTestCase` class. We invoke the servlet's `init()` method in our fixture setup routine to ensure that it is properly initialized before exercising its functionality in our tests. In a sense, we're (partially) taking on the role of the servlet container here and going through the expected lifecycle callbacks of the servlet component. In the `tearDown()` method, we invoke the servlet's `destroy()` method and then remove our reference to the instance so it can be garbage-collected.

Finally, in our `endValidSearch()` method, we check the response from the servlet against what we expected, given the request that we made of it. This method is executed on the "client" side of the Cactus proxy, after the test method has been run. Each `endXXX()` method takes a single `WebResponse` argument, which is a Cactus object that encapsulates the results returned to the "client" from the proxy on the server. In our case, we check to make sure that the returned HTML does not contain the text string `Error:`, which we know (from our hypothetical functional specifications for the servlet) should be included in any response to an invalid or failed search request. We also check to be sure that the response HTML includes a `Search Results` banner, as expected in a valid search result.

In this example test, we made use of both the `beginXXX()` and `endXXX()` methods to initialize the request parameters and check the response, respectively. In some cases, one or both of these will not be necessary. For example, our test servlet is supposed to treat a lack of search arguments as an invalid search. We can test for this behavior by adding the following test method to our

`TestPeopleFinderServlet`:

```
public void testNoSearchArguments( ) {
```

```

        // Invoke the doGet method on the servlet using the request
        // that was generated through the proxy. This request should
        // include no search arguments since we have not added any in
        // a begin method.
        invokeGet( );
    }

    public void endNoSearchArguments(WebResponse response) {
        // The response should contain the key phrase "Invalid search",
        // indicating a bad search request
        assertTrue(response.getText( ).indexOf("Invalid search") >= 0);
    }

```

Note that we do not have a `beginNoSearchArguments()` method defined here because we do not need to add anything to the servlet request before running the test. In fact, we explicitly avoid adding any URL arguments to trigger the functionality being tested (a bad search, lacking any search arguments).

In other cases, we may not need either the `beginXXX()` or `endXXX()` method for a particular test. A typical example is tests that exercise utility methods on a component. Our `PeopleFinderServlet`, for example, has a static utility method named `personToHTML()` that accepts a `Person` Java bean as an argument and returns an HTML snippet that represents that person's information. We don't need to prepare any request information, or test any response information, to perform this test. We simply need a standard test method that creates a sample `Person` object, passes it through the utility method, and checks the results:

```

/** Test the HTML conversion util on PeopleFinderServlet */
public void testPersonToHTML( ) {
    // Create a sample person
    String fname = "Andy";
    String lname = "Long";
    String eAddr = "andy.long@buffaloimports.org";
    Person samplePerson = new Person( );
    samplePerson.setFirstName(fname);
    samplePerson.setLastName(lname);
    samplePerson.addEmailAddress(eAddr);
    // Create the expected HTML output from the converter
    StringBuffer sampleHTML = new StringBuffer( );
    sampleHTML.append("<tr bgcolor=\"#EEEEEE\"><td>")
        .append(lname).append("</td><td>").append(fname)
        .append("</td><td>").append(eAddr).append("</td></tr>");
    // Now apply the method to our Person and validate the results
    String result = PeopleFinderServlet.personToHTML(samplePerson);
    assertEquals(result, sampleHTML.toString( ));
}

```

In this specific test, we don't really need the Cactus framework to allow us to exercise the componen

within a servlet container. We could run this test with a standard `JUnitTestCase` since we're not making use of any of the servlet context parameters. But since you typically want to test functionality that requires a servlet context (that includes, for example, request and response data), you may find yourself mixing simple tests like this with more complicated tests that require the full proxy framework of Cactus.

18.5.2.2. Testing JSPs

If you're not already familiar with programming JSPs and their lifecycle within the web container, you should refer to [Chapter 4](#).

Testing JSPs using Cactus is very similar to testing servlets. The same proxy scheme is used to set up context information, such as request parameters and cookies that are needed by the JSP to function properly. The JSP under test is invoked by forwarding the HTTP request to the page from within your test case, similar to our servlet test example in the previous section. To support JSP-specific context parameters, such as the various scopes of page variables (such as page, session, and request), Cactus provides a `JspTestCase` that can be used as the base class for JSP-related unit tests. The proxy that corresponds to this type of test operates as a JSP page and provides wrapped JSP context parameters (specifically, the `JSPWriter` and `PageContext`) to the test, in addition to the standard servlet context parameters.

The test shown in [Example 18-8](#) demonstrates a typical `JspTestCase`. In this case, the `TestSearchJSP` test case is testing the proper behavior of the `search.jsp` page from our `PeopleFinder` example application. The test itself is almost identical to the valid search test in our `TestPeopleFinderServlet` from [Example 18-7](#). This isn't surprising since both components are web components and they both function identically in this regard (i.e., they expect the same inputs and generate roughly the same output).

Example 18-8. Sample JSP test case

```
// Imports omitted...

public class TestSearchJSP extends JspTestCase {

    // Constructors omitted...

    /** Set up the request on the "client" side of the proxy */
    public void beginValidSearch(WebRequest request) {
        // Add a search parameter to the request
        request.addParameter(PersonDAO.FIRST_NAME, "John");
    }

    /** Pass the valid search request to the page */
    public void testValidSearch( ) {
        // Invoke the target page by forwarding the request
        invokePage( );
    }
}
```

```

/** Check for the appropriate response on the "client" side of the proxy */
public void endValidSearch(WebResponse response) {
    // Since this was a valid search, there should be no "Error"
    // text in the response HTML
    assertTrue((response.getText( ).indexOf("Error:") < 0) &&
        (response.getText( ).indexOf("Search Results") >= 0));
}

/** Utility method that forwards the request to the target page */
private void invokePage( ) {
    try {
        // Forward the request to the search JSP
        this.pageContext.forward("/search");
    }
    catch (ServletException se) {
        fail("Unexpected servlet exception: " + se.getMessage( ));
    }
    catch (IOException ioe) {
        fail("Unexpected I/O exception: " + ioe.getMessage( ));
    }
}
}

```

One noticeable difference is in how the JSP is invoked in our `invokePage()` utility method. Here, we're using the JSP `PageContext` object to forward the request from the Cactus proxy to the target JSP, using the mapped URL `/search` where we expect the JSP to be deployed. In our `TestPeopleFinderServlet` test case, we showed two ways we could pass a request to a target servlet. We could invoke the `forward()` method on the `RequestDispatcher` retrieved from the `ServletContext`, or we could manually construct a servlet instance, initialize it, and invoke its `doGet()` or `doPost()` method. The first approach (using `RequestDispatcher.forward()`) is roughly equivalent to the `PageContext.forward()` approach, and we could do the same thing here in our `JspTestCase`. When testing a JSP, however, it's not convenient for us to manually "construct" an instance of the target JSP and invoke it programmatically. The web container is responsible for parsing the JSP, converting it into a servlet instance, and sending the resulting web component the web requests. So the only viable approach for testing JSPs is to forward the test request directly to the deployed JSP through the web container.

If your tests require access to only the standard servlet context parameters (request, response, session data, and the `ServletConfig`), you can use a `ServletTestCase` to test your JSPs just as well as a `JspTestCase`. In fact, `JspTestCase` uses `ServletTestCase` as its base class. But if your tests need to modify the variables in the various scopes available to JSPs, you'll need access to the JSP `PageContext`, which is available only to `JspTestCase` instances.

18.5.2.3. Testing JSP custom tags

If you're not familiar with programming JSP custom tags and their lifecycle within the web container, you should refer to [Chapter 4](#).

JSP custom tags are a common part of J2EE web application architectures since they allow you to cleanly inject functionality into JSPs in the form of page tags that invoke Java code.

Several approaches to testing custom tags using Cactus are available. You can use a `JspTestCase`, such as those shown in the previous section, to exercise JSPs that contain the tag under test. In your `endXXX()` methods, you can test the response for specific content that the tag is expected to generate. The JSPs that you use can be either actual application pages or JSPs created specifically to test the target tags. Using specially constructed JSPs has its advantages: you can simplify the JSP to include only the elements necessary to test the tag, and you isolate your tests from the structure of your application's design.

You can also test a custom tag more directly, but in a slightly more artificial fashion, by programmatically constructing a tag instance within a `JspTestCase`. Continuing with our progressive PeopleFinder example, the search page uses the `SearchTag` custom JSP tag to actually perform the search. We can directly test our custom tag using a `JspTestCase`, as shown in the `TestSearchTag` test case shown in [Example 18-9](#).

Example 18-9. Test case for a custom JSP tag

```
// Imports omitted...

public class TestSearchTag extends JspTestCase {
    /** An instance of the SearchTag being tested */
    private SearchTag mTag = null;

    // Constructors omitted...

    /** Initialize our test fixture */
    public void setUp( ) {
        // Make a tag instance
        mTag = new SearchTag( );
        // Set its JSP context
        mTag.setPageContext(this.pageContext);
    }

    /** Clean up our test fixture */
    public void tearDown( ) {
        // Call the context popBody( ) method, which will ensure that
        // it emits its generated content from the tag, if any
        this.pageContext.popBody( );
        // Release the tag reference
        mTag = null;
    }

    public void testValidSearch( ) {
        // Set the name of the page variable where the results are to be
        // stored
        mTag.setVarName("people");
        // Set the first name search attribute on the tag
    }
}
```

```

    mTag.setFirstNamePattern("John");
    // Invoke the tag's doStartTag( ), performing the search
    mTag.doStartTag( );
    // If the tag performed as expected, there should be no
    // error message in "people-error", and a non-null collection
    // in the "people" page variable
    assertNull(this.pageContext.findAttribute("people-error"));
    assertNotNull(this.pageContext.findAttribute("people"));
  }
}

```

In our test, we're using a test fixture that includes an instance of a custom tag, stored in the member variable `mTag`. We set up the fixture in our `setUp()` method by constructing an instance of the tag and setting its JSP `PageContext` to the context provided to us by the JSP Cactus proxy. Again, this emulates (partially) the management of the tag performed by a JSP engine within a web container. In our `tearDown()` method, we clean up our fixture by invoking the `popBody()` method on the `PageContext`. This ensures that the content generated by our tag, if any, is flushed into the response, so that it can be checked for validity. In our case, the `SearchTag` doesn't generate any content, so the call to `popBody()` isn't necessary, but we include it here for completeness.

Our tag, `SearchTag`, performs a search against a set of person data using the search attributes defined on the tag. It stores the results as a collection of `Person` objects in the page variable specified by the `varName` attribute on the tag. If an error occurs, the tag generates an error string and stores it in a page variable named `<varName>-error`, where `<varName>` is the value of the `varName` attribute. Our `testValidSearch()` test method tests the behavior of the tag when it's given a valid search request. We set the value of the `varName` attribute, and then we set the `firstNamePattern` attribute to `"John"`. With the attributes set, we invoke the tag's `doStartTag()` method to exercise the tag's search functions. After the `doStartTag()` method returns, we test the results directly by checking the page variables through the `PageContext`. Since we're strictly testing the functionality of the tag and not any surrounding JSP, there's no need to set request arguments in `abeginValidSearch()` method. And since the results we need to test are JSP page variables that are available immediately after the tag is finished processing, we can check the results directly in our test method and have no need to check the HTTP response in the `endValidSearch()` method.

18.5.2.4. Testing servlet filters

If you're not already familiar with servlet filters and their role in the management of servlets and JSPs, you should refer to [Chapter 3](#).

Tests for servlet filters require their own Cactus proxy implementation and `CactusTestCase` base class since filter tests need access to the filter context parameters, similar to servlet tests and JSP tests and their respective context parameters. Filter tests implement the `FilterTestCase` base class, which provides the tests with wrappers for the `FilterConfig` and `FilterChain` objects associated with the filter request.

We're going to write a test for a `LoginFilter` that is used to authenticate users as they request web resources. The code for the filter isn't shown here (again, in part to save space, in part to cater to orthodox extreme programming advocates). But the expected functionality of the filter is fairly

simple. The filter maintains the authenticated ID of the user in a session variable, whose name is given by the value of the `LoginFilter.AUTHN_ID_VAR` constant. If a web request is picked up by the filter and the corresponding session does not have this variable, the filter checks for two request parameters, `LoginFilter.USER_VAR` and `LoginFilter.PASSWORD_VAR`, and these are compared against a set of username/password data in an attempt to authenticate the user. If either of these checks succeed (the session variable is present or the login parameters check out), the filter passes the request down the filter chain, allowing the user access to the target page or web component. If the session variable isn't present and the request parameters are not present in the request or if they don't match any known users, it's assumed the user still needs to be logged in, and he's redirected to a login page whose URL is given by the filter's `loginURI` property. This login page is expected to prompt the user for a username and password and redirect him back to the original resource that was requested.

Based on this description, the `TestLoginFilter` test in [Example 18-10](#) partially tests the functionality of the filter when a user makes a request without any login credentials. The test case uses an instance of the `LoginFilter` as its test fixture, constructing and initializing the filter in its `setUp()` method and destroying it in its `tearDown()` method. While initializing the filter, we invoke its `init()` method, passing in the wrapped `FilterConfig` inherited from `FilterTestCase` and initialized by the Cactus proxy.

Example 18-10. Test case for a servlet filter

```
// Imports omitted. . .

public class TestLoginFilter extends FilterTestCase {
    /** An instance of the filter under test */
    private LoginFilter mFilter = null;

    // Constructors omitted. . .

    /** Set up for tests */
    public void setUp( ) {
        // Construct and initialize the filter instance
        mFilter = new LoginFilter( );
        try {
            mFilter.init(this.config);
        }
        catch (ServletException se) {
            fail("Unexpected servlet exception: " + se.getMessage( ));
        }
    }

    public void tearDown( ) {
        mFilter.destroy( );
        mFilter = null;
    }

    /** Ensure that the filter performs a redirect when no authentication
     * credentials are present in the user session, and no login username and
     * password are provided as request parameters */
}
```

```

public void testLoginRedirect( ) {
    // Ensure that the user session is clear of our authentication
    // credentials
    HttpSession session = this.request.getSession( );
    if (session != null) {
        session.removeAttribute(LoginFilter.AUTHN_ID_VAR);
    }
    try {
        // Invoke the filter
        mFilter.doFilter(this.request, this.response, this.filterChain);
    }
    catch (IOException ioe) {
        fail("Unexpected I/O exception while invoking filter: "
            + ioe.getMessage( ));
    }
    catch (ServletException se) {
        fail("Unexpected servlet exception while invoking filter: "
            + se.getMessage( ));
    }
}

public void endLoginRedirect(WebResponse response) {
    // The response should contain an HTTP "MOVED TEMPORARILY"
    // status with a redirect
    assertEquals("Unexpected status code seen in response",
        response.getStatusCode( ),
        HttpServletResponse.SC_MOVED_TEMPORARILY);
}
}

```

In our single test, `testLoginRedirect()`, we ensure that the filter is given a clean request by clearing the session of the authentication variable, if it's present from a previous user request (e.g., another earlier test method may have set the session variable). Then we pass the request and response objects to the filter's `doFilter()` method, along with the wrapped `FilterChain` inherited from `FilterTestCase`. If the filter operates as expected, this should generate a redirect status in the response, which we check in the `endLoginRedirect()` method. Note that we do not need to define a `beginLoginRedirect()` method in this case because there are no request parameters to initialize. In fact, for the purposes of this test, we need the request to have none of the login request parameters in order to force the redirect we're testing.

In our example test, we're programmatically creating an instance of the filter being tested and exercising its methods directly. Just as with servlets and JSPs, we also have the option of testing a live filter that has been deployed in the web container. The advantages and disadvantages of the two approaches are similar to the servlet case, and there are times when one or the other approach will be needed, depending on the test. Using a direct filter instance allows you to exercise any methods on the filter, including utility methods that are not directly called by the web container. Exercising a live filter, on the other hand, gives you more fidelity in terms of seeing the true behavior of the component in its natural context. But there are some complications to consider, such as configuring the filter so that it can be tested cleanly, without interfering with the testing of other components in the application. In our `PeopleFinder` application, for example, our `LoginFilter` would normally be

mapped to "cover" the entire application to ensure that only authorized users are using the functions in the system. But for the purposes of testing, this would complicate our testing of the web components (servlets and JSPs). With the `LoginFilter` in place, we'd have to ensure that all of our servlet and JSP tests attach valid authentication credentials to the test requests by initializing the session variable checked by the `LoginFilter`. This complicates our other tests, forcing them to include details about another part of the system. So for testing purposes, we would likely set up a special test configuration for the application, where the various web components being tested are deployed without the `LoginFilter` protecting them.

Another point to consider about servlet filters and in-container testing: take care that your application's servlet filters are not applied to the Cactus test proxies. If, for example, you define a filter mapping in your `web.xml` file that covers all URLs in the application, like so:

```
<filter-mapping>
  <filter-name>LoginFilter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

You will also be inserting this filter in front of the Cactus test proxies that will be added to your application when it's deployed for testing. Your filters may interfere with the proper functioning of the Cactus proxies, so you should be careful to control your mappings so application filters are applied only to your application components. As discussed in ["Cactus test proxies"](#) earlier in this chapter, you can control the way Cactus proxies are mapped to URLs in your application, so you should be able to find a way to manage the various mappings appropriately.

18.5.2.5. Testing EJBs and component types

We've seen how various J2EE web components (servlets, filters, JSPs, JSP custom tags) can be tested using the Cactus framework. Cactus can also be useful for testing other J2EE components since it provides a way to run a test from inside a container environment. This ability to test from within a container can be critical in many situations. There may be functionality on the components that can be exercised only from within a container, for example. The most extreme example of this is EJBs that expose only a local client interface. It's not possible to test this interface from a standalone JVM running standard JUnit tests because a remote JVM will need a remote interface. You could create remote interfaces for the purposes of testing, but then you wouldn't really be testing the component as it will be used in the production environment. You'll also be adding new, unnecessary code to your system that is simply there to support testing.

When writing a Cactus test for other J2EE components and for general server-side utility classes, you'll need to decide which Cactus base class to use for your test. In most cases, the choice isn't critical. Generally, when testing an EJB component, for example, you can extend your test from `ServletTestCase`, `FilterTestCase`, or `JspTestCase` equally well. Any of these will allow your tests to be driven by a server-side Cactus proxy. If your components or other server code need to be driven from a particular type of web component, though, you'll need to use the corresponding type of test case as the base classes for your tests so that you can properly initialize and exercise the component. A real-world example of this is an authorization service that is invoked from within a servlet filter in the system. The authorization service could be designed to check the user credentials

and, based on their assigned roles and permissions, route the user to different areas in the application based on settings provided in the filter's initialization parameters. Any tests you wrote for this service would be best written as an extension of `FilterTestCase` so that all the necessary context parameters are available to pass to the service.

As a demonstration of testing other J2EE code with Cactus, [Example 18-11](#) shows `TestPeopleFinderEJB`, which tests the EJB from the `PeopleFinder` sample application.

Example 18-11. Sample Cactus test for an EJB component

```
// Imports omitted...

public class TestPeopleFinderEJB extends ServletTestCase {
    private PeopleFinder mFinderBean = null;
    private static Logger sLog =
        Logger.getLogger(TestPeopleFinderEJB.class.getName( ));

    // Constructors omitted. . .

    /** Create an instance of the EJB component under test */
    protected void setUp( ) {
        // Initialize an EJB reference
        String beanName = "java:comp/env/ejb/PeopleFinder";
        try {
            InitialContext ctx = new InitialContext( );
            PeopleFinderHome home =
                (PeopleFinderHome)PortableRemoteObject.narrow(
                    ctx.lookup(beanName), PeopleFinderHome.class);
            this.mFinderBean = home.create( );
        }
        catch (NamingException ne) {
            fail("Unable to lookup EJB component using name '" +
                beanName + "': " + ne.getMessage( ));
        }
        catch (CreateException ce) {
            fail("Unable to create PeopleFinder bean: " + ce.getMessage( ));
        }
    }

    /** Clean up our EJB instance */
    protected void tearDown( ) {
        try {
            mFinderBean.remove( );
            mFinderBean = null;
        }
        catch (RemoveException re) {
            fail("Removal of EJB in fixture failed: " + re.getMessage( ));
        }
    }
}
```

```

// Verify that the EJB correctly rejects a query with no arguments
public void testNoSearchArguments() {
    // Try to pass in an empty parameter set
    try {
        mFinderBean.findPeople(new SearchArg[0]);
        fail("PeopleFinder allowed an empty search");
    }
    catch (InvalidSearchException ise) {
        sLog.info("PeopleFinder correctly rejected an empty search");
    }
    catch (PersistenceException pe) {
        fail("Unexpected persistence exception: " + pe.getMessage());
    }
}
}
}

```

Our test case extends `ServletTestCase` not for any particular reason, just to allow our test to be driven by a server-side Cactus proxy. We could just as easily have extended our test from `JspTestCase` or `FilterTestCase`.

Our test maintains a handle on a `PeopleFinder` EJB component in its fixture. The `setUp()` method looks up the EJB home interface through the server's JNDI context and then creates an EJB reference. Our `tearDown()` method invokes the `remove()` method on the reference and then sets the fixture variable to null. Our only test method, `testNoSearchArguments()`, verifies that the EJB correctly rejects a search request that has no search parameters included. We have no `beginNoSearchArguments()` and `endNoSearchArguments()` methods because the functionality we're testing doesn't require any particular web context information like request parameters or session data.

18.5.2.6. Testing asynchronous code

Even with the Cactus proxy architecture and its ability to test J2EE code running in a container environment, some distributed and asynchronous architectures are difficult to test programmatically using a unit test framework. Typical examples in the J2EE realm are JMS message handlers, message-driven EJBs, and message-oriented SOAP services implemented using JAX-RPC or SAAJ. In these situations, a message or request of some kind is generated and sent to the target handler. The handler processes the message or request and generates a response or some side effects. In a typical test scenario, you'll want to generate specific inputs and verify the results of the handler's processing of those inputs. It's problematic in these cases because of the asynchronous and distributed nature of the handlers. A JMS message handler, for example, will take an indeterminate amount of time to be notified of the message by the JMS service. Once it has been notified, it may generate side effects on the server or generate a response message. If the only result generated by the JMS message handler is a response message, we can simply poll for the response message within the Cactus test and verify its structure:

```

. . .
public void testJMSResponse( ) {

```

```

QueueReceiver recvr = . . . ;
QueueSender sender = . . . ;
Message testMsg = . . . ;
// Send the test message
sender.send(testMsg);
// Poll for the expected response
Message response = recvr.receive( );
// Validate response
. . .
}
. . .

```

But even this simple case is complicated a bit by the fact that the handler may fail to generate a message, meaning that we need to give up waiting for a message after some period of time and then fail the test. But how long do we wait? Is 30 seconds too long? Two minutes? Sometimes these questions can be answered by using maximum delay thresholds that have been defined as requirements on the system. By definition, if a response is not received in this time, the handler has failed. We're mixing functional testing and performance testing in this situation, but at least we can define our test in a concrete way and validate the results.

If the handler generates side effects only on its side of the asynchronous interaction, your test must have direct access to the side effects and be able to verify them. In some cases, this is possible for example, when the JMS handler makes modifications to data stored in an RDBMS that we can check from the test code. But there's still the issue of knowing when to check the results. How do we know when the handler has received the message and finished processing it?

There are a number of approaches to dealing with this. The right answer for you depends on how and what you need to test with regard to your asynchronous, distributed subsystems. A couple of possible approaches are:

Use mock objects.

In the JMS case, for example, rather than creating a real JMS message and delivering it to your handler through a real JMS service, you can create a mock `Message` and invoke the `onMessage()` callback on the handler directly. This will unit test the handler code, which is usually the goal. And the removal of the JMS service from the picture converts the test scenario from an asynchronous situation to a synchronous one (when the `onMessage()` method completes, the handler is finished and all of its side effects should be committed).

Use white box testing instead of black box testing.

Rather than testing the functioning of the handler as a black box, pull apart the handler logic and test its individual elements as a set of white box tests. If a SOAP service operation, for example, invokes objects A, B, and C to respond to a particular request, write a series of unit tests that exercises these same server-side objects in the same way as the SOAP service. This also eliminates the asynchronous communications from the mix and tests the equivalent functionality, but at the cost of managing your test suite against the implementation details to ensure that your white box tests remain consistent with the unit's implementation.

18.5.3. Running Cactus Tests

Cactus tests are run using a JUnit `TestRunner`, just like standard JUnit tests. The difference is that the `TestRunner` needs to be run in such a way that it "knows" where to find the Cactus server-side proxies so that the Cactus tests can pass their test requests along to be run in the server container(s).

Cactus uses a set of Java properties to define the network location of its proxies. These properties are set in a `cactus.properties` file that's read from the classpath used to run the `TestRunner`, or they can be specified using command-line arguments to the JVM. The properties that Cactus supports are shown in [Table 18-1](#).

Table 18-1. Cactus runtime properties

Property name	Description
<code>cactus.contextURL</code>	The base URL for the various Cactus web proxies. This property is required.
<code>cactus.servletRedirectorName</code>	The name to which the Cactus servlet proxy is mapped in the test application's <code>web.xml</code> /deployment descriptor. The default value for this is <code>"ServletRedirector"</code> , which is the name used by default by the <code>cactifywar</code> Ant target when it adds the Cactus proxies to your <code>web.xml</code> /file.
<code>cactus.jspRedirectorName</code>	Same as above, but this specifies the name for the JSP proxy. The default value is <code>"JspRedirector"</code> .
<code>cactus.filterRedirectorName</code>	Same as above, but this specifies the name of the servlet filter proxy. The default value is <code>"FilterRedirector"</code> .

When using a standalone JUnit `TestRunner` (such as the text-based `TestRunner` we used in our JUnit examples or a `TestRunner` integrated with an IDE like Eclipse), we can set these properties in a `cactus.properties` file and ensure that this properties file is in the classpath when we invoke the `TestRunner`. The `cactus.contextURL` property is the only one that is strictly required the others are needed only if you used custom mappings for the Cactus proxies when you added them to your test application. So a minimal `cactus.properties` file would be:

```
# Set the base URL for the Cactus proxies
cactus.contextURL = http://localhost:8080/jent-peopleFinder-test
```

If any Cactus tests are run with this properties file, the `ServletTestCase` tests will attempt to connect to the servlet proxy at the URL <http://localhost:8080/jent-peopleFinder-test/ServletRedirector>, the `JspTestCase` tests will talk to the JSP proxy at <http://localhost:8080/jent-peopleFinder->

[test/JspRedirector](#), and so on.

With this properties file in our classpath somewhere, we can invoke Cactus tests from the command line just as we did with regular JUnit tests:

```
> java junit.textui.TestRunner
com.oreilly.jent.people.servlet.TestPeopleFinderServlet
...
Time: 5.763

OK (3 tests)
```

Of course, we still need to ensure that the application server running our test application (and the Cactus proxies) is up and running before we do this.

This same approach can be used to run Cactus tests using JUnit facilities built into IDEs like Eclipse. Simply configure the JUnit tool to include the proper *cactus.properties* file in its runtime classpath, and it will be able to drive your server-side Cactus tests along with standard JUnit tests.

18.5.3.1. Running Cactus tests with Ant

We can also use the standard `junit` Ant task to drive our Cactus tests if we want. Just as with the other `TestRunner` examples, we need to make sure that the appropriate *cactus.properties* file is in the classpath of the `junit` task when we invoke it:

```
<target name="run-server-tests-junit">
  <junit printsummary="on"
        showoutput="true">
    <classpath>
      <!-- Add the application classes and the test classes -->
      <path refid="java.test.compile.classpath"/>
      <path location="${java.test.classes.dir}"/>
      <!-- Add the directory containing the cactus.properties -->
      <path location="${config.dir}"/>
    </classpath>
    <test name="com.oreilly.jent.people.servlet.TestPeopleFinderServlet"/>
  </junit>
</target>
```

Notice that we're including all of the application classes (specified elsewhere as the value of the `java.test.compile.classpath` path), and the test classes themselves (specified using the `java.test.classes.dir` directory, where the test classes are compiled). Again, before invoking the `junit` task, we'll need to ensure that the test application server is up and running. If we have Ant targets defined to start the test server, we could add those as dependencies on this testing target.

As an alternative to driving server-side Cactus tests using Ant, Cactus includes the `cactus` Ant task. This task essentially plays the same role as the `junit` task (in fact, `cactus` is implemented as an extension of the `junit` task). The difference is that `cactus` also starts up the test application server before running the Cactus tests you specify and shuts down the application server after it runs the tests. This task is especially useful when you're using one or more dedicated application servers for testing, and when the application servers are running locally, allowing the `cactus` task to start and stop them.

Of course, in order to start and stop the application server running the test application, the `cactus` task needs to be told about the server configuration (such as what type of server it is and where it is installed). The `cactus` task supports several popular server types, including JBoss, WebLogic, and Tomcat. It does this with specific subelements that you use to specify the server parameters. As an example, the following target uses the `cactus` task to drive a set of tests run in a local JBoss server:

```
<target name="run-server-tests-cactus">
  <cactus earfile="${basedir}/${ant.project.name}-test.ear"
    printsummary="yes">
    <classpath>
      <path refid="java.test.compile.classpath"/>
      <pathelement location="${java.test.classes.dir}"/>
    </classpath>
    <containerset>
      <jboss3x dir="/tools/servers/jboss-3.2.5"
        config="default"
        port="8080"/>
    </containerset>
    <test name="com.oreilly.jent.people.AllServerTests"/>
  </cactus>
</target>
```

In the main `<cactus>` element, we specify the test application archive file to be deployed for testing using the `earfile` attribute. The `cactus` task also supports the other attributes used with the `junit` task to toggle summary output, forking options, and the like. Within the `cactus` task, we specify the classpath needed to run the tests. The `cactus` task needs this information in order to run the specified tests in an internal `TestRunner`. Notice that we do *not* have to provide a `cactus.properties` file in the classpath when using the `cactus` task. We are specifying the test application archive to run, and the `cactus` task can extract the mappings for the Cactus proxies from this. The base URL for the proxies will be constructed from the server parameters given in the server-specific child elements.

Next, we specify one or more application servers to be used to run the test application and the tests, using the `<containerset>` child element. Each server used to run the tests is specified as a child element of `<containerset>`. In this example, we've specified a single JBoss 3.x server, installed in the `/tools/servers/jboss-3.2.5` directory. We're using the default server from this JBoss installation, running on port 8080. If you are using a server not directly supported by the `cactus` task with its own child element, you can use the `<generic>` element within the `<containerset>`. This element lets you specify other Ant targets that should be used to start and stop the application server you're using as well as the HTTP port for the server's web container. You can then implement the start and stop targets in Ant to suit the particular application server in use.

18.5.3.2. Running Cactus tests using ServletTestRunner

Possibly the simplest way to run your Cactus tests is to use the `ServletTestRunner` provided with Cactus. This `TestRunner` is actually a servlet that you can invoke from any web browser. You deploy the `ServletTestRunner` servlet with your test application, mapping it to a particular URL in your application's web context by adding entries like the following to your test application's `web.xml` file:

```
<servlet>
  <servlet-name>ServletTestRunner</servlet-name>
  <servlet-class>
    org.apache.cactus.server.runner.ServletTestRunner
  </servlet-class>
</servlet>

. . .

<servlet-mapping>
  <servlet-name>ServletTestRunner</servlet-name>
  <url-pattern>/ServletTestRunner</url-pattern>
</servlet-mapping>
```

This servlet requires the Cactus libraries in order to run, but these will already be included in your test application to support your Cactus tests.

Once the `ServletTestRunner` is deployed, you can invoke specific `TestCase` or `TestSuite` classes using the `suite` URL argument to the servlet. Assuming that our server is running locally on port 8080, that the context root for the test application is `jent-peopleFinder-test`, and that the mapping shown here is used for the `ServletTestRunner`, then to run our `TestPeopleFinderServlet` test case, we'd go to our browser and navigate to the URL:

<http://localhost:8080/jent-peopleFinder-test/ServletTestRunner?suite=com.oreilly.jent.people.servlet.TestPeopleFinderServlet>

If all goes well, the results will be returned to the browser as an XML file, as shown in [Figure 18-3](#).

Figure 18-3. Results of TestPeopleFinderServlet



This output format is pretty raw, useful only in browsers that render XML directly (like Mozilla Firefox or Internet Explorer) or when you want to take the XML output and postprocess it yourself to generate a report. If you'd prefer to have the results provided in a different output format, the `ServletTestRunner` allows you to specify an XSL transform file to process the output of the test run. Just include the XSL transform file in your test application's web archive and tell the `ServletTestRunner` where to find it by setting a servlet init parameter named `xsl-stylesheet`. Cactus provides a default stylesheet called `cactus-report.xsl` that transforms the XML output into HTML. If we put this file in the `styles` directory of our web archive, we could optionally apply it to the test output by adding the `<init-param>` element to the servlet `web.xml` entry as shown here:

```

<servlet>
  <servlet-name>ServletTestRunner</servlet-name>
  <servlet-class>
    org.apache.cactus.server.runner.ServletTestRunner
  </servlet-class>
  <init-param>
    <param-name>xsl-stylesheet</param-name>
    <param-value>styles/cactus-report.xsl</param-value>
  </init-param>
</servlet>

```

We can then choose to apply this transform to the test output by including a "transform" URL argument (with any value) to the URL used to invoke the `ServletTestRunner`. The result will look like [Figure 18-4](#).

Figure 18-4. Transforming output from TestPeopleFinderServlet with XSL

Unit Test Results

Designed for use with [Cactus](#).

Summary

Tests	Failures	Errors	Success rate	Time
3	0	0	100.00%	0.536

Note: failures are anticipated and checked for with assertions while errors are unanticipated.

TestCase com.oreilly.jent.people.servlet.TestPeopleFinderServlet

Name	Status	Type	Time(s)
testValidSearch	Success		0.121
testNoSearchArguments	Success		0.199
testPeopleToHTML	Success		0.211

[Back to top](#)

Done

← PREV

Chapter 19. Struts

If you browse to the home page of the Apache Struts project (<http://struts.apache.org>), you will discover in the first sentence or two what Struts is: Struts is an open source framework for building Java web applications. Each of the key words in this brief definition of Struts encapsulates a crucial dimension of its overall scope and requires some expansion.

← PREV

19.1. The Scope of Struts

Struts is open source. We wouldn't normally belabor the open source nature of Struts, except that Struts was one of the first widely adopted open source Java projects. Unlike many of the other APIs presented in this book, Struts did not emerge from Sun Microsystems or from the Java Community Process, and there is no Java Specification Request for it. The original version came from the keyboard of a single individual, Craig McClanahan, who had contributed to Apache JServ and Tomcat and who is now the architect of the Sun Java Studio Creator. In part due to this single point of origin and the need to "scratch an itch" (the desire for a robust web application framework that could support internationalization), Struts has its idiosyncrasies and some irregular naming conventions. Indeed, JSF ([Chapter 5](#)) may be seen in some ways as a more refined response to Struts, and McClanahan was a cospecification lead for JSF.

As of January 2005, Struts was considered finished, and the Struts team now refers to it as Struts Classic. Their new offering will be Struts Shale, a JSF-based framework.

As an open source project, you get the Struts code under a generous license. While Struts is mature and stable, it's good to have the source so that you can adapt it as necessary. (Indeed, we urge you to download the source from <http://struts.apache.org> and study it as you read this chapter; we will be making occasional references to the code.) In open source fashion, you can add fixes of your own and feed them back into the code base, thus giving back to the Struts community. Even if you don't contribute a bug fix yourself, there is some possibility that a bug you discover in a current release and report to the Struts project will be fixed for a later release.

Struts is a framework. Struts is designed to take some of the tedium out of web development. At its best, it provides techniques for dealing with common problems according to the best known practices; at the very least, it provides techniques that work fairly well. Struts provides a large number of features for designing sophisticated web applications: there are classes to support internationalization, [\[1\]](#) to enable a number of different data access strategies, and to render your information using several different presentation layers. In short, Struts provides the foundation for a web-based application so that you can get on with your domain-specific job of coding.

^[1] Internationalization was one of the original motivators for Struts; see Craig McClanahan's blog entry on the subject at <http://blogs.sun.com/roller/page/craigmcc/20040927>.

Struts is for *building* web applications. It is not a web application, and it does not build the application for you. Because it is a tool, and only a tool, it can be used badly.

The Struts home page also tells us that the framework encourages application architectures based on the model-view-controller (MVC) pattern. Struts implements what the Sun Blueprints [\[2\]](#) call the Model 2 architecture, in which a servlet (or its delegate) manages client interaction and business logic execution, while presentation is located primarily in JSP pages. The MVC pattern allows for a clean separation of the *model*—broadly speaking, the code that deals with your business problem—and the *view*, or presentation. Orchestrating these two areas of responsibility is the *controller*, which provides for communication between the model and the view. Users interact with views, and the controller mediates between the views and the model, passing state changes from the model back to the view and converting user activities from the view into operations that can be performed on the model. The

separation of view and model has a few big benefits. First, by decoupling the view from the model, it is much more likely that you'll be able to redesign the "skin" of an application without needing to touch the code that deals with business rules, data access, or communication with other services. Second, the division of responsibilities in code between model and view frequently matches the differences in skills on a development team. By splitting the concerns of the view and the model into different components, your staff can specialize in the areas they know best: the HTML and JSP developers can work with the view, and the backend specialists can work with the model. Finally, separating user interface code from business code can make both parts easier to test.

[2] See *Designing Enterprise Applications with the J2EE Platform*(Addison-Wesley).

Struts is not the only MVC framework for web applications out there. Other worthy projects include WebWork and Maverick. Struts, however, was among the first web application frameworks that provided definite value to developers. It has a large mind share, making it a less risky choice for architects concerned about finding qualified developers to maintain their applications in the future.

And then there's JSF. While Struts and JSF can coexist, the choice between them is a matter of orientation. With Struts, you are essentially modeling and mapping the request/response structure of HTTP. The events to which you respond are fundamentally those of the client browser, while with JSF the events are those of your user interface components. JSF does more for you, and you will likely find that you are writing less code; this is true for the JSF example in this book. Typically with JSF, you will yield more control to the framework. JSF was also designed to work well with tools; your IDE may be able to generate much of the code for a JSF application, in the same way that Swing or AWT components may be developed on the fly. As the Cockney proverb has it, "You pays your money and you takes your choice," except that, given the generous licensing of both Struts and JSF, you will pay for your choice in time, not money.

This chapter focuses on the architecture of Struts and the key classes that make up its version of the MVC pattern. The chapter also presents some of the Struts features that make it a useful framework for a basic application. You may find that Struts is more than you need for a simple application. Most applications grow, however, and it is often the case that choosing an appropriate framework such as Struts in the early stages can make refactoring easier later on.

Before we get started, here's a caveat. Struts is a large project, and we will not discuss certain important features of Struts, such as its ability to manage more than one "module," its support for internationalization, and its provisions for non-HTML page representations (e.g., WML) and non-JSP-rendering APIs (e.g., Velocity). We will not provide exhaustive accounts of all of the tags and attributes that may be defined in the Struts configuration file, the Struts validator configuration files, and the Struts Tags API. For discussion of all of these features and a complete reference to Struts, see O'Reilly's *Programming Jakarta Struts* by Chuck Cavaness. In this chapter, we will stick to the most common case: HTML pages rendered by JSP.

19.2. The Sample Application

To illustrate some of the most important features of Struts, we provide an application called Library-Struts. Library-Struts allows you to log in as a user (a Borrower or a Librarian) and add books, check out books, and list all of the books. This sample application has the same functionality as the example in the JSF chapter (in fact, the JSF application was developed first, and the Struts version is a port).^[3] For a high-level description of the purpose of this application, see [Chapter 5](#). For readers attempting to make a choice between Struts and JSF, a comparison of these chapters and the sample applications may be illuminating.

^[3] Perhaps the only known port from JSF to Struts?

To run the sample Struts application, you will need Version 1.2.4 of Struts (<http://struts.apache.org>) and a Servlet 2.3 (or later) compatible container such as JBoss Version 4 (<http://www.jboss.org>). The downloadable package of examples for this book, available at <http://www.oreilly.com/catalog/javaentnut3>, includes an Ant script for building the application and deploying it to JBoss.

19.3. The Development Process with Struts

Prefatory to creating a Struts application, you must do three basic things. First, determine what views will be displayed, especially the forms in the application, the data they will collect, and the actions that will be linked to the data collected. Second, record those decisions in an XML configuration file for the Struts controller (*struts-config.xml*), representing the forms and actions. Third, write the classes representing the forms and actions. It is crucial to note that different developers may tackle these tasks in a different order. If you are working from existing requirements, you would likely move straight to the XML configuration file. If application flow and data may be defined piecemeal, you might start working on some form and action classes and integrate them gradually into your configuration file. If you are a new Struts developer, you will likely find that you are constantly moving back and forth between the configuration file, your views, and your Java code; you will also find that, while reading this chapter, you will need to move back and forth between the section discussing the Struts configuration file and the other aspects of a Struts project. This tension between what belongs in the configuration file and what belongs in the other resources has motivated a number of Struts tools such as IDE plug-ins, XDoclet templates, and so forth; but, as with many frameworks, until you have experienced the pain, you won't obtain much gain from using them.

Here is a sample to-do list for getting your Struts application off the ground:

Steps 1-4: Determine views, actions, and flow

1. Enumerate all of the views that will be sent to the browser. Typically these are JSP pages.
2. For each view with HTML form data, itemize the groups of parameters that actions will process. You will implement these as Struts `ActionForm` beans. At this point, you might also consider any validations that are required on these parameters.
3. Define all of the "actions" that your application must perform. Typically these are behaviors that respond to parameterized data submitted by an HTML form using the HTTP POST method. You will implement these as Struts `Action` classes.
4. Determine the flow of your application. To do this, you will need to decide on the possible outcomes of each page and to what other action or page control should be directed, according to each outcome.

Step 5: Configure

1. Write the Struts configuration file that defines for Struts the relationships between views (JSP), actions (`Action` classes), and groups of parameters (`ActionForm` classes).

Steps 6-7: Write code

1. Write your JSPs, `Action` classes, and `ActionForm` classes.
2. Compile your code and build your application for deployment.

Let's take a look at the sample application with this to-do list in mind and consider, in order, a view (item 1), its form data (item 2), the action that will respond to form data (item 3), and to what page the user should be directed after the form data has been processed (item 4). Items 5 and 6 will be addressed through the rest of the chapter. For item 7, compiling and building your application, see the file layout and Ant script with the code bundle.

After the librarian logs in (username: librarian; password: librarian), one option is to follow the Book Administration link and change the status of individual books from checked out to checked in or vice versa. [Figure 19-1](#) shows what the page looks like in my browser.

Figure 19-1. Library administration page



Following our to-do list (and at the risk of being repetitive):

1. Clearly this is a dynamic view of some data the books that are in the Library. One book is currently checked out: Crawford and Kaplan's *J2EE Design Patterns* (it must be a good one!). The view is represented with a JSP file, `/view/adminbooks.jsp`.
2. The form data for this view is the list of checkboxes in the third column. (Typically a form has a submit button; throughout the application, we detect changes in form inputs with JavaScript and submit the form immediately; this lends the application a more "event-driven" feel.) The form data is represented by a Java class, `AdminBooksActionForm`, extending the Struts `ActionForm` class.

3. When a checkbox's state is changed, Java code will change the state of an object that holds all the data pertinent to the Library. This action is represented by another Java class, `AdminBooksAction`, extending the Struts `Action` class.
4. Once the checkbox state changes have been registered in the Library, we return the user to this very page. In other words, the "forward" upon successful completion of the action is to `/view/adminbooks.jsp`. The relationship between the view, the form, the action, and the forward is represented by an `ActionMapping`, an object that is created by Struts when it reads its configuration file, `struts-config.xml`. If the user wants to go do something else, she clicks the Return Home link.

 PREY

19.4. The Struts Controller

In a moment we will discuss the Struts approach to the MVC controller and then provide detail on the building-block classes, `org.apache.struts.action.Action` and `org.apache.struts.action.ActionForm`. But first we should take a step back and look at the nail for which Struts is attempting to be the hammer. In many web applications, the basic HTTP interaction is not especially complicated. An HTTP request includes at least the name of a resource and zero or more parameters. Based on the name of the resource and the parameters, control must be handed off to specialized code that performs an action. The action determines what response is appropriate; this HTTP response is then encoded and returned to the requester, typically with HTML as the message body. So far, so good.

But in ad hoc web programming, it is all too easy to handle each of these tasks in an idiosyncratic way. For example, a developer might write a servlet to handle a request and might create his own methods and conventions for validating parameters (ensuring that certain parameters are syntactically valid phone numbers, perhaps). Meanwhile, another coder, working primarily on GET requests, writes her own method for validating parameters. Clearly, there are organizational and maintenance issues here with the existence of two modules doing similar tasks, maintained by two different people. But the larger issue is that there are well-known patterns for handling such minutiae and if handled once, and well, those patterns become amenable to all of the good features we associate with object-oriented programming: encapsulation, separation of responsibilities, inheritance where appropriate, and the rest. In fact, validation is small potatoes compared to the main course of organizing the management of actions in a systematic way. When it comes to the more elaborate requirement of making it easy to rearrange the relationships between requests, actions, and responses, we need a framework such as Struts to provide an object representing each piece in request/response processing.

Struts provides an object encapsulating each of the features in our earlier summary of request processing. It's worth noting at this point a significant difference between Struts and JSF. In Struts, the developer is exposed more directly to the underlying request processing, whereas in JSF, much more of the detail of request processing is handled by the framework. This distinction also affects our discussion of these two frameworks. For Struts, you will find that we frequently talk about classes and methods, whereas in JSF, we talk about XML declarations that define the "wiring" of the relationships between beans. Let's look at request processing again, but from the point of view of Struts.

The starting point is the connection between the servlet container and Struts. Struts typically monopolizes all requests in your web application with its own implementation of `javax.servlet.Servlet` (`org.apache.struts.action.ActionServlet`). Control is passed to this `Servlet` by the container. It is this `ActionServlet` that chooses which class should handle any particular request. The choice is based on settings in an XML file so that you can change the control flow without recompiling.

An `Action` class handles the request and extends `org.apache.struts.action.Action`. This `Action` class has an `execute` method that you will almost always want to override. It is passed the request and response instances that the `ActionServlet` receives as well as references to the form that contains the parameter data and a "mapping" that defines the control flow.

An `ActionForm` represents the parameter data. Once again, you extend a Struts class: `org.apache.struts.action.ActionForm`.

Now for some more detail on each of these fundamental classes.

19.4.1. The ActionServlet

The fundamental feature of Struts is a servlet, `org.apache.struts.action.ActionServlet`, which delegates incoming HTTP GET and POST requests to a request processor (`org.apache.struts.action.RequestProcessor`) and from there to special-purpose handlers instances of the action classes.

Specify the Struts `ActionServlet` and a mapping for it in the `web.xml` file for the application:

```
<servlet>
  <servlet-name>Struts Action Servlet</servlet-name>
  <servlet-class>org.apache.struts.action.ActionServlet</servlet-class>
  <init-param>
    <param-name>config</param-name>
    <param-value>/WEB-INF/struts-config.xml</param-value>
  </init-param>
  <load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
  <servlet-name>Struts Action Servlet</servlet-name>
  <url-pattern>*.do</url-pattern>
</servlet-mapping>
```

The `servlet-name` value can be anything you like, although it should be descriptive. You should also specify a parameter named `config` whose value defines the location of a Struts-specific XML configuration file. The default value for the location of this configuration file is `/WEB-INF/struts-config.xml` (for more details, see "[The Struts Configuration File](#)" later in this chapter). It is conventional to specify this file even when you are using the default location so that it is obvious in your `web.xml` where the Struts-specific configuration lives. Struts must perform its configuration work before any requests are served, so the `<load-on-startup>` element is essential to ensure that Struts initializes first (note: if you have a singleton class that should be initialized when Struts starts, see "[Struts Plug-ins](#)" later in this chapter). All requests to URL patterns matching `*.do` will be handled by the Struts `ActionServlet`. Mapping Struts actions to the `.do` extension is conventional. Some Struts developers map a path to the Struts `ActionServlet`, which can be handy if you want to provide J2EE security for the path that is mapped to Struts actions (see [Chapter 10](#)) or put a servlet filter in front of your Struts actions (see [Chapter 3](#)). Note that if you're using Struts modules, you must use extension mapping.

When `ActionServlet` is initialized by the servlet container, it performs a number of tasks. It initializes Struts "modules," it sets up Struts-managed `DataSources`, it initializes Struts plug-ins, and it reads in and processes the Struts configuration file.

Once initialized, the `ActionServlet` will satisfy requests that are passed to it by the container. When `doGet()` or `doPost()` is called, `ActionServlet` immediately calls a protected method that then delegates to an instance of a `RequestProcessor`.

19.4.2. The RequestProcessor

If you've downloaded the source to Struts, open the source for `org.apache.struts.action.RequestProcessor` and follow along. Understanding how this class uses resources is the key to becoming an adept Struts developer. By the way, here as elsewhere with Struts, the source reveals a number of tidbits that can be quite helpful; as a tiny but useful example, you will see where Struts provides debug messages to the console.

When an HTTP GET or POST request is passed to the `process` method of `RequestProcessor`, the following tasks are performed in this order:

1. If the request is a multipart request, it is handled by a special processor (`org.apache.struts.upload.MultipartRequestWrapper`). Multipart requests are typically file uploads.^[4]

^[4] To learn more about uploading and Struts, see recipe 7.10 in Bill Siggelow's *Jakarta Struts Cookbook* (O'Reilly).

2. The path is extracted from the request. This path will be used as the basis for selecting a "mapping" (an instance of `ActionMapping`) in step 7, which in turn will determine the selection of the `ActionForm` (step 9) and `Action` (step 12) for handling the request (step 13). (The relationship between the mapping, the `Action`, and the `ActionForm` is defined in the `struts-config.xml`/file see "[The Struts Configuration File](#)" later in this chapter.)
3. If the controller has been configured to select the locale automatically, the user's locale is set. This locale will be the basis for the choice of a resource bundle with defining properties for that locale. (For more information, see the internationalization section of the current Struts User Guide at <http://struts.apache.org> or [Chapter 12](#) of O'Reilly's *Programming Jakarta Struts*.)
4. The controller sets content and caching headers, if so configured.
5. The method `processPreprocess()` is called. If you extend the class `RequestProcessor` and override this method, you may conduct some preprocessing for instance, you might add special attributes to the request or you might manipulate the session data. (We provide an example of a custom `RequestProcessor` later.)
6. Messages that were "cached" for display in the prior request are removed in particular, error messages managed by the `org.apache.struts.action.ActionErrors` class and the `<html:errors>` tag are removed.
7. Based on the path extracted in step 2, the `ActionMapping` for the current request is looked up.
8. Roles are checked for the request and mapping; if the user is not allowed to proceed, the processor doesn't continue.

9. The `ActionForm`, if any, is determined based on the `ActionMapping`. If an `ActionForm` is found, it is prepopulated. If the `ActionForm` has been configured to require validation, it is validated.
10. If a forward has been specified for this mapping, the request is forwarded. (For discussion of redirects and forwards, see the sections on "[Servlet Responses](#)" and "[Request Dispatching](#)" in [Chapter 3](#).)
11. If an include has been specified for this mapping, the inclusion takes place.
12. The `Action` specified for the mapping is looked up or created.
13. The action is processed, returning an `ActionForward` that specifies what is to be done next.
14. The returned `ActionForward`'s path is processed, meaning that either a servlet-style forward is performed via a `RequestDispatcher` or a redirect response is returned.

The key steps in this process are the ones that select the `ActionMapping` based on the path, populate and validate the `ActionForm`, acquire the `Action`, process the `Action`, and then return the `ActionForward`.



19.5. The Action Class

The `Action` class is an adapter between servlet requests and your business logic. In other words, it is appropriate for your `Action` class to handle issues such as gathering data from the user interface, controlling flow through the application, and shaping the data for presentation. But it is inappropriate for the `Action` class itself to handle business functions. For example, if your `Action` class is updating a database itself, you have probably inappropriately put business functions into the web tier.

You will need to extend `org.apache.struts.action.Action` for each `Action` you want to define. The most important method to override is `execute()`. This example is from `AdminBooksAction`:

```
public ActionForward execute(ActionMapping mapping, ActionForm form,
    HttpServletRequest request, HttpServletResponse response)
    throws Exception {
    AdminBooksActionForm adminBooksForm = (AdminBooksActionForm) form;

    Map books = getLibrary().getBooks();

    // First set all books to checked in
    List bookList = getLibrary().getBookList();
    Iterator i = bookList.iterator();
    while (i.hasNext()) {
        Book book = (Book) i.next();
        if (book != null) {
            book.setCheckedOut(false);
        }
    }

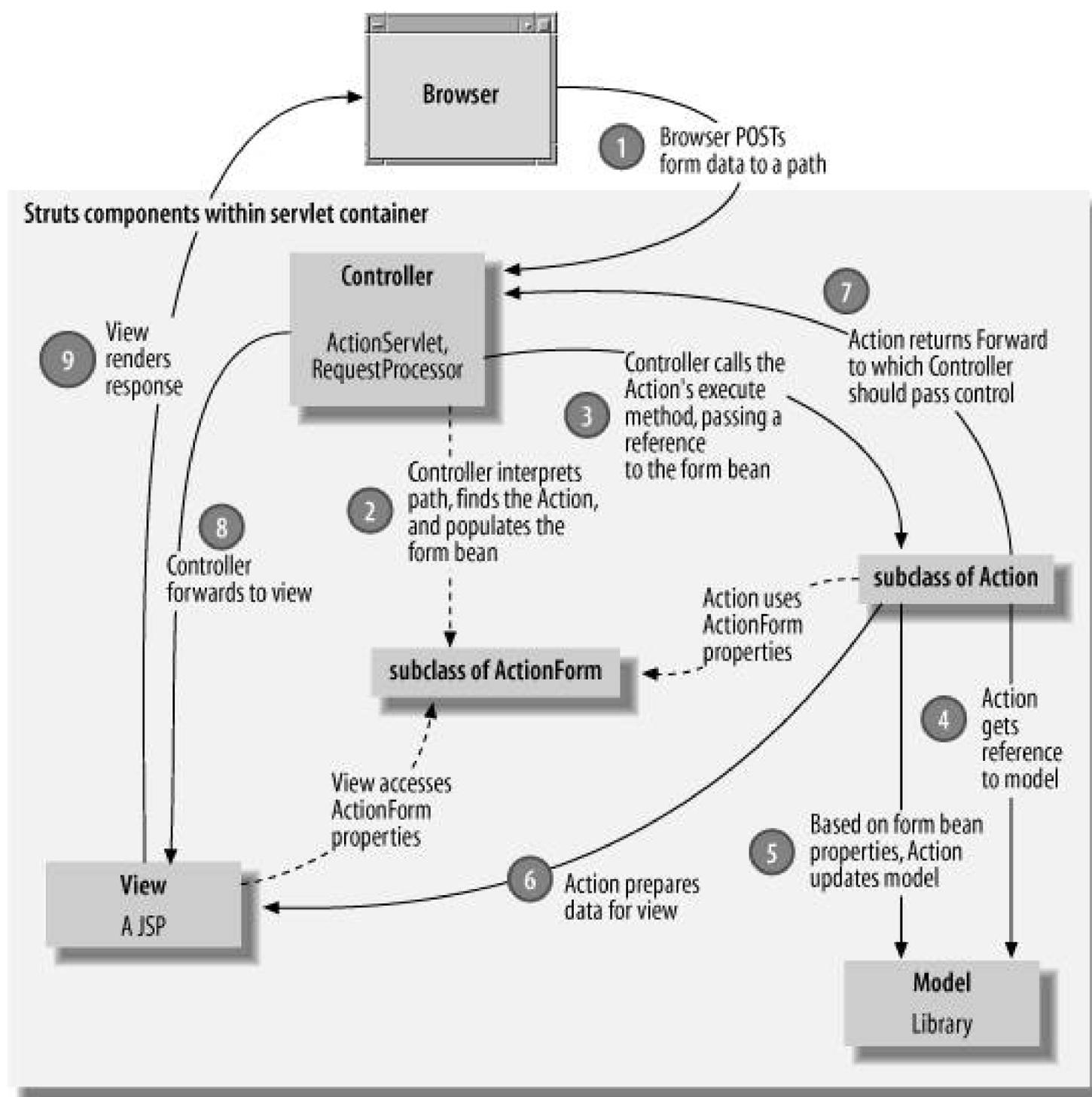
    // Now based on the checks in the form, set checked books
    // to "checked out"
    String[] picks = adminBooksForm.getPicks();
    if (picks != null) {
        for (int j = 0; j < picks.length; j++) {
            ((Book) books.get(picks[j])).setCheckedOut(true);
        }
    }
    return mapping.findForward("adminbooks");
}
```

This `Action` handles the form submission from `/view/adminbooks.jsp`. The pattern here is classic and is simply a version of the request processor's earlier tasks list but with the focus on the `Action`:

1. The browser submits form data via an HTTP POST.
2. The controller (`RequestProcessor`) interprets the path requested by the HTTP POST (`/adminBooks.do`) and uses the path to find the appropriate `Action` (`AdminBooksAction`). Then the controller populates the form bean (`AdminBooksActionForm`).
3. The controller calls the action's execute method, passing a reference to the form bean.
4. `AdminBooksAction` obtains a reference to a component of the model with `getLibrary().getBooks()`. (`getLibrary()` is a superclass method in `LibraryBaseAction`.) In a more complicated application, the model might be accessed via a database connection or a message queue. The model might be anything else that should be manipulated based on user activity. We might also obtain a reference to a remote business object (say a stateless session EJB) whose state will be changed. In this example, the `Map` of books is simply a reference to data within the `Library` singleton.
5. `AdminBooksAction` checks the properties from the form bean. Based on information submitted via the form bean, changes are made in the model.
6. If need be, the `Action` adds, removes, or changes request and session scope attributes. (In the specific case of `AdminBooksAction`, no such changes are made.)
7. `AdminBooksAction` uses the `ActionMapping` passed in via the execute method to look up an `ActionForward` to which Struts should pass control. In this case, we simply return an `ActionForward` named `adminbooks` (defined in the `global-forwards` section of `struts-config.xml`) that is aimed back at the path for a Struts `Action` that sets up the checkboxes for the `/view/adminbooks.jsp` page.
8. The controller forwards control to the appropriate view, specified by the `ActionForward`.
9. The view renders the page for the browser.

We have now covered enough information to look at this in schematic form in [Figure 19-2](#) follow the numbers from the browser at the top.

Figure 19-2. Struts components satisfying a request



Of course, your `Action` is playing its part in the larger context of Struts, so for more of the story, we must zoom out to the `RequestProcessor`.

19.5.1. RequestProcessor and Actions

The Struts `RequestProcessor` will instantiate only one instance of your `Action` class; multiple threads will use the class at the same time. This means the class must be threadsafe: your instance and static variables must not store data for any specific request. Basically, your `Actions` are essentially delegates of the Struts `RequestProcessor`.

Before control is handed over to your action, the `RequestProcessor` calls the `processPreprocess()` method. If you extend `RequestProcessor` and implement your own `processPreprocess()`, you will have a convenient means to perform some common tasks that run before all calls to the execute methods of `Actions`. In the example application, a `UserSession` object is used to aggregate all

information associated with a user; a custom `RequestProcessor` ensures that the `UserSession` object is always available:

```
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;

import org.apache.struts.action.RequestProcessor;

public class LibraryRequestProcessor extends RequestProcessor {

    protected boolean processPreprocess(HttpServletRequest request,
        HttpServletResponse response) {
        UserSession userSession = null;
        HttpSession session = request.getSession( );
        if (session != null) {
            userSession = (UserSession) session
                .getAttribute(WebGlobals.USER_SESSION_KEY);
            if (userSession == null) {
                userSession = new UserSession( );
                session.setAttribute(WebGlobals.USER_SESSION_KEY, userSession);
            }
        }
        return super.processPreprocess(request, response);
    }
}
```

Your custom `RequestProcessor` must be specified in the Struts configuration file via the `<controller>` element:

```
<controller
    processorClass=
        "com.oreilly.jent.struts.library.ui.LibraryRequestProcessor"/>
```

Developers often provide convenience methods in a "base" `Action` class. In the example, `LibraryBaseAction` provides a protected method for retrieving a reference to a `UserSession` (which is available because of the work done in the `LibraryRequestProcessor.processPreprocess()` method). Here it is:

```
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpSession;

import org.apache.struts.action.Action;

public abstract class LibraryBaseAction extends Action {
```

```

protected Library getLibrary( ) {
    return (Library) servlet.getServletContext( ).getAttribute(
        WebGlobals.LIBRARY_KEY);
}

/**
 * Return the UserSession from the session.
 *
 * @param request
 * @return the UserSession
 * @throws RuntimeException
 */
protected UserSession getUserSession(HttpServletRequest request) {
    HttpSession session = (HttpSession) request.getSession( );
    if (session == null) {
        throw new RuntimeException("Couldn't get a reference to the Session");
    }
    UserSession userSession =
        (UserSession) session.getAttribute(WebGlobals.USER_SESSION_KEY);
    if (userSession == null) {
        throw new RuntimeException
            ("Couldn't retrieve UserSession attribute from Session");
    }
    return userSession;
}
}

```

The combination of overriding `RequestProcessor.processPreprocess()` to ensure that resources are available for an `Action`'s execute method, coupled with convenience methods in a "base" `Action` that access such resources, can be a very helpful way to organize your code and avoid bloated `Action` classes and copy-and-paste-style code copying.

19.5.2. The Struts Configuration File

Much of the power of Struts derives from its configuration file. Little is hardwired in the Struts management of your application's logic since the names for specific components, as well as the relationships between cooperating components, are specified in the `struts-config.xml` file. Because of this deployment-time level of abstraction, it is frequently possible to tweak a Struts setting, or substitute an implementation of some functionality, without recompiling any of your code. The `struts-config.xml` file provides control over the display name and description of your application, data sources that will be managed by Struts, form beans, action mappings, the specific implementation of the controller class, message resources, and Struts plug-ins, as well as exceptions and forwards that are available to actions globally. Typically you will find that you will define most of these elements.

As mentioned earlier, the location of the Struts configuration file is defined in the `web.xml` file as a parameter ("config") on the definition of the `ActionServlet`. The conventional value is `WEB-`

INF/struts-config.xml. The *struts-config.xml* file for the sample application starts and ends like this:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>

<!DOCTYPE struts-config PUBLIC
    "-//Apache Software Foundation//DTD Struts Configuration 1.2//EN"
    "http://jakarta.apache.org/struts/dtds/struts-config_1_2.dtd">

<struts-config>

    <!-- Omitted -->

</struts-config>
```

Below the root element (`<struts-config>`), we define a number of elements that are found in almost all Struts applications. The DTD for the *struts-config.xml* file specifies that they must appear in this order; none are required (for a complete rundown of the elements, you will need to study the online Struts documentation and the DTD for the configuration file (http://jakarta.apache.org/struts/dtds/struts-config_1_2.dtd). [Table 19-1](#) lists elements that are frequently used in Struts applications.

Table 19-1. Typical configuration elements

Element name	Purpose
<code><form-beans></code>	Defines the form beans (<code>ActionForms</code>).
<code><global-forwards></code>	Defines names for "forwards," paths to which you can pass control. By specifying a name here, you can decouple the name used for a specific resource. Thus, instead of referring to <i>login.jsp</i> , you can refer to the forward named <code>login</code> should it happen that you need to introduce a new login JSP (perhaps <i>login2.jsp</i>), you can change the forward path in the configuration file without having to change any of your references elsewhere.
<code><action-mappings></code>	Defines the mappings (the relationship between a path, an <code>Action</code> , and an <code>ActionForm</code>).
<code><controller></code>	The controller to use. If not specified, the default, <code>org.apache.struts.action.RequestProcessor</code> , will be used.
<code><message-resources></code>	The name of a properties file that defines name/value pairs for messaging.
<code><plug-in></code>	A plug-in is a class that implements the interface <code>org.apache.struts.action.PlugIn</code> .

The *struts-config.xml* file for the sample application ([Example 19-1](#)) uses all of these elements (for the sake of this illustration, we have removed some internal comments regarding aspects of the application you might want to change).

Example 19-1. struts-config.xml

```
<?xml version="1.0" encoding="ISO-8859-1" ?>

<!DOCTYPE struts-config PUBLIC
    "-//Apache Software Foundation//DTD Struts Configuration 1.2//EN"
    "http://jakarta.apache.org/struts/dtds/struts-config_1_2.dtd">

<struts-config>
  <form-beans>
    <form-bean name="LoginForm"
      type="com.oreilly.jent.struts.library.ui.LoginActionForm"/>

    <form-bean name="BookDisplayForm"
      type="com.oreilly.jent.struts.library.ui.BookDisplayActionForm"/>
    <form-bean name="AddBookForm"
      type="org.apache.struts.validator.DynaValidatorForm">
      <form-property name="title" type="java.lang.String"/>
      <form-property name="author" type="java.lang.String"/>
      <form-property name="isbn" type="java.lang.String"/>
      <form-property name="addBookAction" type="java.lang.String"/>
    </form-bean>
    <form-bean name="AdminBooksForm"
      type="com.oreilly.jent.struts.library.ui.AdminBooksActionForm"/>
  </form-beans>

  <global-forwards>
    <forward name="home" path="/home.do"/>
    <forward name="login" path="/login/index.jsp"/>
    <forward name="logout" path="/logout.do"/>
    <forward name="viewbooks" path="/view/listbooks.jsp"/>
    <forward name="addbook" path="/view/addbook.jsp"/>
    <forward name="adminbooks" path="/adminBooksSetup.do"/>
  </global-forwards>

  <action-mappings>
    <action path="/home" parameter="/home.jsp"
      type="org.apache.struts.actions.ForwardAction" />
    <action path="/login" name="LoginForm"
      scope="request" validate="true" input="/login/index.jsp"
      type="com.oreilly.jent.struts.library.ui.LoginAction">
      <forward name="success" path="/login/success.jsp" redirect="false"/>
      <forward name="failure" path="/login/index.jsp" redirect="false"/>
    </action>
    <action path="/logout"
```

```

        type="com.oreilly.jent.struts.library.ui.LogoutAction"/>
<action path="/bookDisplay" name="BookDisplayForm" scope="session"
        type="com.oreilly.jent.struts.library.ui.BookDisplayAction"/>
<action path="/addBook" name="AddBookForm" scope="request"
        validate="true" input="/view/addbook.jsp"
        type="com.oreilly.jent.struts.library.ui.AddBookAction"/>
<action path="/adminBooksSetup" name="AdminBooksForm" scope="request"
        type="com.oreilly.jent.struts.library.ui.AdminBooksSetupAction">
    <forward name="adminBooksView" path="/view/adminbooks.jsp"
        redirect="false"/>
</action>
<action path="/adminBooks" name="AdminBooksForm" scope="request"
        type="com.oreilly.jent.struts.library.ui.AdminBooksAction"/>
</action-mappings>

<controller processorClass=
    "com.oreilly.jent.struts.library.ui.LibraryRequestProcessor"/>

<message-resources parameter="application"/>

<plug-in className="org.apache.struts.validator.ValidatorPlugIn">
    <set-property
        property="pathnames"
        value="/WEB-INF/validator-rules.xml,/WEB-INF/validation.xml"/>
</plug-in>

<plug-in className="com.oreilly.jent.struts.library.ui.LibraryPlugIn"/>

</struts-config>

```

First we define our form beans. The form bean `LoginForm` is bound to a class we wrote, `LoginActionForm`. Another alternative is to define a form bean dynamically, using `DynaActionForm` or a subclass. For dynamic form beans, the `<form-property>` element provides a means to define a property, giving it a name and type. An additional attribute not shown here is `initial`, allowing you to provide a value that will be injected into the form bean when it is created.

Global forwards are extremely useful for providing common names for places to which control should be directed. Notice that the path can be an ordinary JSP or a path into the Struts `ActionServlet`; to forward to an action, you must make sure that you are observing the servlet's URL pattern specified in *web.xml* (in the example, we append *.do* to the name of the action's path).

The `<action-mappings>` element comprises a list of `<action>` elements, which define `ActionMappings`. Yes, that's right; even though what is being defined is an `ActionMapping`, the name is `<action>`. The `<action>` element may be the single most significant element in the entire Struts configuration file, so let's pause to describe its attributes in [Table 19-2](#) (note that the definitions in the Purpose column are quotations or paraphrases from the documentation in the DTD).

Table 19-2. Attributes for struts-config.xml `<action>` element

Attribute name	Purpose
<code>path</code>	The relative path of the submitted request, starting with a <code>/</code> character, and without the filename extension if extension mapping is used.
<code>name</code>	Name of the form bean, if any, that is associated with this action mapping. That is, the name must correspond to the name of a <code>form-bean</code> .
<code>type</code>	Fully qualified Java class name of the <code>Action</code> subclass that will process requests for this action mapping.
<code>scope</code>	The scope (<code>request</code> or <code>session</code>) that is used to access our <code>ActionForm</code> bean, if any. Optional if <code>name</code> is specified, otherwise not valid. The default scope is <code>session</code> .
<code>validate</code>	Set to <code>true</code> if the <code>validate</code> method of the <code>ActionForm</code> bean should be called prior to calling the <code>Action</code> object for this action mapping or set to <code>false</code> if you do not want the <code>validate</code> method called. The default value is <code>true</code> .
<code>input</code>	The relative path of the action or other resource to which control should be returned if a validation error is encountered. Valid only when <code>name</code> is specified. Required if <code>name</code> is specified and the input bean returns validation errors. Optional if <code>name</code> is specified and the input bean does not return validation errors.

Look at this action mapping:

```
<action path="/adminBooks" name="AdminBooksForm" scope="request"
  type="com.oreilly.jent.struts.library.ui.AdminBooksAction"/>
```

It says, when a request is made to the relative path within the application context, */adminBooks.do*, create in request scope an instance of the `ActionForm` type bound to the form bean named `AdminBooksForm` if it has not already been created. The `Action` to which execution should be delegated is `AdminBooksAction`. The form will be validated (since the default value of the `validate` attribute is `true` and we haven't specified it). By the way, a hidden assumption here is that the validation will never fail (no `input` attribute is specified). This seems a safe bet since only checkbox information is being submitted. To be sure, an evildoer might write a program to post a naughty value into the form; to barricade yourself against this possibility, specify an `input` attribute.

The `<controller>` element defines the `ControllerConfig` bean, whose details are beyond the scope of this discussion. When left out, the standard Struts request processor (`org.apache.struts.action.RequestProcessor`) will be used. For reasons described earlier, our application uses a custom request processor, which is specified with the `processorClass` attribute.

The `<message-resources>` element defines the message resource bundle for localization. The default class uses the `parameter` attribute to provide the base name of the resource bundle properties file. So in this case, the file will be *application.properties*. Note that message templates for error messages are found here.

The `<plug-in>` element defines any Struts plug-ins; the `<className>` element provides the fully qualified Java class name of the plug-in. The subordinate `<set-property>` element gives you the

means to pass initialization data to the plug-in. The idiom you see here for setting up the Struts validator is typical.



19.6. Views in Struts

Views present information from the model. The typical view architecture used with Struts is JSP, but in fact, any servlet that is, any path mapped by the container to which Struts can forward control can serve as a view. We will stick to JSP because it is so common, but increasingly developers are becoming interested in view frameworks with simpler syntax or other niceties, such as Velocity. Thankfully, Struts can still be used if you decide to pursue these alternatives.

The interesting question regarding views is how they communicate with `Actions` and the request processor. Views need to transfer data in two ways. First, they must be able to pick up data that is provided to them by `Actions`. This part is easy: an `Action` puts objects into request or session scope, just as any servlet would; references to these objects may then be retrieved by the JSP from request or session scope directly through Java scriptlets referencing the `HttpServletRequest` or `HttpSession` objects or through tags, such as those provided by JSTL (JSP Standard Tag Library) or the Struts Tag Libraries (more on this topic later in this chapter).

The second requirement for data transfer in the view is to obtain values that should be displayed in a form. Enter the `ActionForm`.

19.6.1. The ActionForm Class

As we discuss the `ActionForm` class, please refer to [Figure 19-2](#). The key to understanding `ActionForms` is to notice that they are situated at a number of places in the request/response lifecycle. In particular, form data gets to an `Action` via an `ActionForm` parameter on the `execute` method, and an `ActionForm` bean can be tapped by Struts tags to ensure that form data is populated correctly in a JSP.

A class extending `ActionForm` represents the parameters submitted by an HTTP POST request. Your `ActionForm` must have JavaBeans-style getters and setters. In addition, you may override utility methods from the base `ActionForm`. The `reset` method is called by the framework to return the instance to its original state, and the `validate` method is called after population to give your `ActionForm` a chance to verify that its values are acceptable. [Example 19-2](#) shows an `ActionForm` that represents the username and password fields for a login form.

Example 19-2. Login ActionForm

```
import javax.servlet.http.HttpServletRequest;

import org.apache.struts.action.ActionErrors;
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionMapping;
import org.apache.struts.action.ActionMessage;
```

```

import org.apache.struts.action.ActionMessages;

public class LoginActionForm extends ActionForm {

    private String username;
    private String password;

    public String getUsername( ) {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    public String getPassword( ) {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }

    public ActionErrors validate(ActionMapping mapping,
        HttpServletRequest request) {
        ActionErrors actionErrors = new ActionErrors( );
        if (password == null || password.trim( ).length( ) == 0) {
            actionErrors.add(ActionMessages.GLOBAL_MESSAGE, new ActionMessage(
                "errors.required", "Password"));
        }
        if (username == null || username.trim( ).length( ) == 0) {
            actionErrors.add(ActionMessages.GLOBAL_MESSAGE, new ActionMessage(
                "errors.required", "Username"));
        }
        return actionErrors;
    }
}

```

19.6.1.1. Validation

After the Struts controller populates the bean properties, it calls the `validate` method to give your `ActionForm` an opportunity to return to Struts any errors that are encountered. In `LoginActionForm` (see [Example 19-2](#)), we require a value for the username and password properties, so when those properties are null or contain only whitespace, we generate an `ActionMessage` object representing the error message that should be conveyed back to the view. We return an instance of `ActionErrors` that includes a list of all `ActionMessages` encountered. (Before Struts 1.2, `ActionErrors` was used to represent an error; that class is deprecated in favor of `ActionMessage`.)

The first parameter on the `ActionErrors.add()` method is `ActionMessages.GLOBAL_MESSAGE`. This key

is used elsewhere in Struts to look up the list of `ActionErrors` from the scope (request or session) where the `ActionForm` is stored. `ActionMessage` has a number of constructors. Here we use the constructor that accepts two `String` parameters. The first is the name of a property (`errors.required`) defined in `application.properties`. This property gives the error message template that should be used to display the error to the user. The second parameter is the name of the field for which we are noting an error; this field name will be plugged into the template. Here are the error formatting properties and values from `application.properties`:

```
errors.required={0} is required.
errors.minlength={0} must be at least {1} characters.
errors.credentials.login=Login Error Unable to log you in - username and password
combination not found.
errors.isbn.length=ISBN must be 10 characters.
```

Thus in our example, if the username is missing, the error message template `{0} is required.` will be obtained via the key `errors.required`, and the `{0}` will be filled by the word `Username`.

19.6.1.2. Scope

Remember that a specific `ActionForm` class is bound to a name in the `struts-config.xml` file. That name is referenced in an `<action>` element:

```
<action path="/bookDisplay" name="BookDisplayForm" scope="session"
    type="com.oreilly.jent.struts.library.ui.BookDisplayAction"/>
```

This definition can also include an attribute defining the scope in which the `ActionForm` instance will live. If the `ActionForm` instance is needed for only one request/response sequence, specify request scope. You may also specify session scope if the `ActionForm` should be preserved from `Action` to `Action`. For instance, if you have actions that must "look back" to forms that have been submitted in earlier request/response interactions (say, for a wizard), session scope provides you with a means to keep that data available. In the example application, there is little need to use session scope because each Struts `Action` puts its response into request scope. There is one exception, however. In our `BookDisplayForm`, we want the "safe mode" (which suppresses the Check In/Check Out buttons) to be "sticky," so that it stays on no matter where the user navigates in the application. By putting the form into session scope, the very same form bean is available (and reutilized) every time the page is displayed. Thus once we turn on safe mode, the mode is persistent, even if you move off of the page.^[5] Indeed, it is sticky even if we log out this is just one reason why logout operations typically invalidate their sessions.

[5] Also notice that, since the same form is reused, the same book will be selected if we click Return Home and then go back to View Books. To experiment with this, change the scope to request scope, redeploy, and observe how safe mode is forgotten after clicking the Return Home link. This is because a form bean in request scope is created afresh every time it is needed.

19.6.1.3. The ActionForm and the JSP

Let's look at how this `ActionForm` is utilized in the JSP page that makes up a login page.

To create a page with a form and its associated `ActionForm` and `Action`, the Struts developer must typically manipulate five or six files:

- The Struts configuration file (*struts-config.xml*).
- The JSP page that includes the HTML form. This JSP page will include Struts tags to reference an `Action` and tags from the JSTL.
- The Java code for the `ActionForm` that defines the parameters for the HTML form's POST action.
- The Java code for the `Action` that defines what should be done with the parameters.
- A Struts property file (typically *application.properties*).

Additionally, if you are using the Struts validator, validation definitions will be in a file such as *validation.xml*.

Here's a JSP that uses our `LoginActionForm`:

```
<%@ include file="/includes/setup.inc" %>
<html>
  <head>
    <title>
      Struts Library Login
    </title>
    <link rel="stylesheet" type="text/css"
          href='<c:url value="/css/app.css"/>'>
  </head>
  <body>
    <h1>
      Struts Library Login
    </h1>
    <hr>
    In order to borrow or add books you must log into the Library
    application.
    <html:errors/>
    <html:form action="/login" focus="username">
      <table>
        <tr>
          <td>
            Username:
          </td>
          <td>
            <html:text property="username"/>
          </td>
        </tr>
      </table>
    </html:form>
  </body>
</html>
```

```

    <tr>
      <td>
        Password:
      </td>
      <td>
        <html:password property="password" redisplay="false" />
      </td>
    </tr>
  </table>
  <p>
    <html:submit />
  </p>
</html:form>
</body>
</html>

```

The first line of the JSP is a static include of *setup.inc* that we use in all of the JSPs in the sample application. This include file defines the prefixes used in the application for the Struts tags and JSTL tags. Additionally, a reference to the current list of books is copied into request scope from a member of an application-scoped reference to the singleton instance of `com.oreilly.jent.struts.library.model.Library` (see *LibraryPlugIn.java*). To learn about the JSTL `<c:set>` and `<c:url>` tags, see [Chapter 4](#).

Moving down the page, the first Struts-specific item is the `<html:errors/>` tag. Any errors found during the form validation phase will be displayed here. Notice that this tag is XML, not HTML. Therefore, you must always provide a close tag. We might have written `<html:errors></html:errors>`, but we are using the XML shorthand syntax for an "empty" tag `<html:errors/>`.

The form is defined with the `<html:form>`, `<html:text>`, and `<html:password>` tags. Struts tags are discussed in more detail later in this chapter; for now, note that these tags are rendered to HTML tags such as `<form>`, `<input type="text">`, and `<input type="password">`. In the `<html:form>` tag, the value (`/login`) of the required `action` attribute defines the action mapping in *struts-config.xml* to which the Struts `ActionServlet` will delegate the POST request.

If you are running the application, a useful exercise at this point is to look at the HTML that is produced by this assortment of Struts and JSTL tags:

```

<form name="LoginForm" method="post"
  action="/jent-struts/login.do">
  <table>
    <tr>
      <td>

        Username:
      </td>
      <td>
        <input type="text" name="username" value="">
      </td>
    </tr>
  </table>
</form>

```

```

</tr>
<tr>
  <td>
    Password:
  </td>

  <td>
    <input type="password" name="password" value="">
  </td>
</tr>
</table>
<p>
  <input type="submit" value="Submit">
</p>
</form>

```

As you can see from the generated HTML, Struts has made a number of substitutions. The most important one is that the `action` attribute on the form has the value `/jent-struts/login.do`. Struts has included the context path (`/jent-struts`) as well as appended the `.do` extension (specified in the servlet mapping in the `web.xml` file), marking this as a Struts target that will be intercepted by the Struts servlet. [\[6\]](#)

^[6] Perhaps you are wondering, as I did, how Struts figures out that you have mapped the `ActionServlet` to `*.do` since the developer is not required to define this mapping in any Struts configuration file. This information is not available through any of the standard classes such as `ServletConfig`. If you study the Struts source, you will discover that the Struts `ActionServlet` itself parses your `web.xml` file to dig up the mapping.

Now let's look briefly at the login action mapping in `struts-config.xml`.

```

<action path="/login" name="LoginForm" scope="request"
  validate="true" input="/login/index.jsp"
  type="com.oreilly.jent.struts.library.ui.LoginAction">
  <forward name="success" path="/login/success.jsp"
    redirect="false" />
  <forward name="failure" path="/login/index.jsp"
    redirect="false" />
</action>

```

Notice that the name of the form (`LoginForm`) is defined in `struts-config.xml`, the form name is not referenced directly in the JSP. Because `LoginForm` is mapped to the `LoginActionForm` class in the `struts-config.xml` file, you can control which specific class defines the form properties without changing your JSP.

19.7. Struts Tags

Struts was one of the first frameworks to provide a library of custom JSP tags. The Struts tags provide an amazing array of utilities useful in JSP; also, they provide access to Struts objects so that you don't have to include Java snippets in your pages. Since the introduction of JSTL, the most common practice is to prefer a JSTL tag when its functionality is similar to an equivalent Struts tag. Struts tags are still very handy for integrating the rendering of pages to Struts data structures such as `Actions` and `ActionForms`. In general, JSTL and Struts tags cohabitate quite well, with one exception: internationalization. For internationalization, you are advised to stick to JSTL or to Struts, because their strategies for bundling up the internationalized versions are different.

There are too many Struts tags to describe in this chapter; we will focus here on the tags and tag attributes that are used in the code example. If you have never used a tag library, read the discussion of JSTL in [Chapter 4](#) first.

In the next section, we'll look at the core tags for HTML rendering, obtaining bean properties, and forwarding requests to Struts `Actions`. These tags are in the Struts tag libraries named `html`, `bean`, and `logic`. For each library, you must include a directive at the top of each page that references the particular library, associating it with a prefix of your choosing. (The prefix names `html`, `bean`, and `logic` are conventional, but you may change them to avoid namespace collisions with other tag libraries you might be using.) It is typical to put these directives into an include file that is then included in all of your pages. For instance, in the example code, the list of directives for including Struts tags looks like this (in `includes/setup.inc`):

```
<%@ taglib prefix="html"
      uri="http://jakarta.apache.org/struts/tags-html" %>
<%@ taglib prefix="bean"
      uri="http://jakarta.apache.org/struts/tags-bean" %>
<%@ taglib prefix="logic"
      uri="http://jakarta.apache.org/struts/tags-logic" %>
```

And the inclusion is just this single line:

```
<%@ include file="/includes/setup.inc" %>
```

19.7.1. Form Handling and Form Tags

What we are going to describe now is the magic by which Struts tags orchestrate the mapping of form fields to bean properties, get those properties validated, and in the case of errors, redisplay

those form fields to the user for correction. First the nuts and bolts regarding syntax and tying the various files together.

19.7.1.1. <html:form>

You use the `<html:form>` tag in your JSP to render an HTML form tag. The layout of your Struts tags for a typical form follows the same structure as an HTML form. Here's a compressed version of the login form from [Example 19-2](#):

```
<html:form action="/login" focus="username">
  Username: <html:text property="username"/>
  Password: <html:password property="password" redisplay="false"/>
  <html:submit/>
</html:form>
```

`<html:text>`, `<html:password>`, and `<html:submit>` map to the obvious HTML tags; more about that in a moment.

Notice the `action` attribute. For the `action` attribute, specify a path that is defined for an action mapping in your *struts-config.xml* file. So in this case, the form data will be posted to the Struts action with the path */login*. Here's the action mapping defined in *struts-config.xml* for the path */login*.

```
<action path="/login" name="LoginForm" scope="request"
  validate="true" input="/login/index.jsp"
  type="com.oreilly.jent.struts.library.ui.LoginAction">
  <forward name="success" path="/login/success.jsp" redirect="false"/>
  <forward name="failure" path="/login/index.jsp" redirect="false"/>
</action>
```

Notice that the Struts `ActionForm` to be used is looked up from the *struts-config.xml* file based on the action mapping. So in this case, the name of the `ActionForm` to be used is `LoginForm`, which is defined in *struts-config.xml* as follows:

```
<form-bean name="LoginForm"
  type="com.oreilly.jent.struts.library.ui.LoginActionForm"/>
```

This level of indirection means that you can change parameters associated with form handling (e.g., the scope of a form) without changing your JSP. Or you might swap in a different `ActionForm` that provides more sophisticated validation.

The resulting HTML is:

```

<form name="LoginForm" method="post"
  action="/jent-struts/login.do">
  Username: <input type="text" name="username" value="">
  Password: <input type="password" name="password" value="">
  <input type="submit" value="Submit">
</form>

```

The most significant transformation is the rendering of the HTML action attribute with the appropriate context for your application, along with the addition of the file type (specified by you in the Struts servlet parameters in *web.xml*) that the container will recognize as a request for the Struts servlet.

Now for the magic. The first thing an `<html:form>` tag does is find the `ActionForm` object (the form bean) associated with the `ActionMapping`. If the form bean doesn't exist, it will create it. When the form is submitted, form data will be loaded into the bean and then the `validate` method will be called (generating an `ActionErrors` object). If the validation fails, Struts will forward to the page specified for the `ActionMapping`'s "input" attribute. Typically, this is the same page that displayed the original form (though it needn't be). And with the `<html:errors>` tag, this page can display the errors that were encountered. Meanwhile, since the very same form bean is used, the form will be prepopulated with the invalid values so that the user may correct them.

19.7.1.2. `<html:text>`, `<html:password>`, and `<html:submit>`

Individual HTML form elements can be specified through a matching Struts tag. In the previous example, for instance, the Struts `<html:text>` tag will be rendered as an HTML input tag with the `type` attribute set to `text`. Similarly, the `<html:password>` tag is rendered as an HTML input tag with `type` set to `password`. The value of the property tag ties the particular Struts tag to the property on the `form-bean`. You can also easily associate a property value with an `<html:submit>` tag. By this means, you can provide two submit buttons with the same property name but different values, and then, in your `Action`, differentiate between them. For instance, in the form on */view/addbook.jsp*, we define two submit buttons like so:

```

<html:submit property="addBookAction" value="Add Book"/>
<html:submit property="addBookAction" value="Verify"/>

```

On the backend in `AddBookAction`, we skip adding the book if only the Verify button were pressed:

```

public ActionForward execute(ActionMapping mapping, ActionForm form,
  HttpServletRequest request, HttpServletResponse response)
  throws Exception {
  DynaActionForm addBookForm = (DynaActionForm) form;
  String addBookAction = addBookForm.getString("addBookAction");
  if (!"Verify".equals(addBookAction)) {
    String title = addBookForm.getString("title");
    String author = addBookForm.getString("author");

```

```

        String isbn = addBookForm.getString("isbn");
        Book book = new Book(title, isbn, author);
        getLibrary( ).addBook(book);
        return mapping.findForward("home");
    }
    return mapping.findForward("addbook");
}

```

19.7.1.3. <html:select>, <html:options>, and <html:checkbox>

Struts provides tags that can render all of the standard HTML input widgets. These tags can be enormously convenient because they provide help with iterating over the elements in a `Collection` to generate the complete HTML. For example, observe the code using `<html:select>` for generating an HTML `<select>` tag:

```

<html:select property="trackingID" size="10" onchange="submit( )">
    <html:options collection="bookList"
        property="trackingID" labelProperty="title" />
</html:select>

```

The surrounding form is bound to the action with path `/bookDisplay`, which uses the `form-bean` `BookDisplayForm`, which is bound to the `BookDisplayActionForm`, which contains the property `TRackingId`. (That's a lot of interaction, but you will find that you are conducting this kind of code tracing on a regular basis with Struts.) Based on the user's selection, `trackingId` will be populated after the form is submitted. To enumerate the books associated with each `trackingId`, we use the page-scoped variable `bookList`, which is bound to a `Collection` of `Books`. The Struts tag `<html:options>` takes the reference to the collection, iterates through it, and renders an HTML `<option>` tag with the value coming from `Book.trackingId` and the display text coming from `Book.title`.

Our use of the `<html:checkbox>` tag is quite simple: when the box is checked, we submit immediately thanks to the JavaScript embedded in the `onchange` attribute:

```

<html:checkbox property="safeModeChange" onchange="submit( )" />

```

(Tying the checkbox to the form submission provides for some cheap "event handling"; you can also use an `<html:checkbox>` to represent an ordinary checkbox input.) Then the `Action` handling the form checks the value of the property associated with the checkbox.

19.7.1.4. <html:multibox>

If you need to render a checkbox whose value is determined by the loop in which the checkbox is

enclosed, use an `<html:multibox>`. Here's an example (simplified version from *adminbooks.jsp* in the example):

```
<c:forEach var="book" items="${library.bookList}">
  <html:multibox property="picks" onclick="submit( )">
    <c:out value="${book.trackingID}" />
  </html:multibox>
</c:forEach>
```

In this code fragment, the JSTL `<c:forEach>` tag is used to iterate through the individual books. The `<html:multibox>` tag defines the `form-bean` property that will be populated when the checkbox is checked. The actual value of each generated checkbox is pulled from the `trackingID` property of each book in the list. The JSTL `<c:out>` tag renders the property value of the book bean's `trackingID` property.

19.7.2. Additional Tags

The additional tags we discuss handle a variety of important functions, including links, errors, and forwards.

19.7.2.1. `<html:link>`

The `<html:link>` tag renders an HTML `<a>` tag. You specify the value of the link using one of the following required attributes: `forward`, `action`, `href`, or `page`. The `forward` and `action` attributes take the names of Struts forwards and actions as their values. Using one of these attributes allows you to update your *struts-config.xml* file to change the destinations of links without having to change your JSP files. If you use the `href` attribute, its value is used directly as the value of the generated `href` attribute. The `page` attribute provides support for server-relative URIs: the context path and module prefix (if you are using Struts modules) will be automatically added to the value given for this attribute.

Other attributes on the `<html:link>` tag allow you to specify URL parameters, as well as mappings for all of the standard attributes for the HTML `<a>` tag.

Here's a simple example of the `<html:link>` tag, using the `forward` attribute to specify the target of the link:

```
<html:link forward="login">
  Log In
</html:link>
```

If we now browse to this page and view the generated HTML source, we will see this substitution based on information from the *struts-config.xml* file (assuming you're using the sample config file

provided with the example code for this chapter):

```
<a href="/jent-struts
/login/index.jsp;jsessionid=7A505454F6FC979CE94F5AEA76775BA3">Log
In</a>
```

The `<html:link>` tag conveniently utilizes `HttpServletResponse.encodeURL()` from the Servlet API to rewrite the URL, adding the context name (*/Jent-Struts-example*), the path for the forward (*/login/index.jsp*), and the standard session ID token and value.

19.7.2.2. `<html:errors>`

We discussed error handling in our earlier section on the `ActionForm` class. To recap, validation is conducted in an `ActionForm`'s `validate` method; when a validation fails, your code adds an `ActionMessage` to an `ActionErrors` object. The `validate` method returns a reference to the `ActionErrors` object, which Struts then puts into request or session scope (by default, under the key `Globals.ERROR_KEY`) where you can pick it up in a JSP and display it to your users.

Instead of working your way through the `ActionErrors` with your own code, you can use the `<html:errors>` tag to display whatever `ActionErrors` are available. The display of the errors can be controlled by defining message keys in your message resources file (typically *application.properties*). The key names and their functions are shown in [Table 19-3](#) (the descriptions are quoted verbatim from the Struts API documentation).

Table 19-3. Key names for `<html:errors>`

Key	Description
<code>errors.header</code>	Text that will be rendered before the error messages list. Typically, this message text will end with <code></code> to start the error messages list.
<code>errors.footer</code>	Text that will be rendered after the error messages list. Typically, this message text will begin with <code></code> to end the error messages list.
<code>errors.prefix</code>	Text that will be rendered before each individual error in the list.
<code>errors.suffix</code>	Text that will be rendered after each individual error in the list.

The Struts API documentation suggests extremely plain markup for error formatting, which is a good bet for getting something that looks decent in most contexts. But you can also reference custom styles or invoke any other HTML that will make your errors look pretty. Here are the definitions of these attributes for the sample application:

```
errors.header=<div class="validationWarning">
errors.footer=</div><br>
errors.prefix=
errors.suffix=<br>
```

Each of these values may be overridden by an attribute on the tag (`header`, `footer`, `prefix`, or `suffix`), but the typical scenario is to define this markup once in the message resources file and then invoke the tag with no attributes. In the sample application, we display errors on the pages that display the login form and the form for adding books the invocation of `<html:errors>` is simply:

```
<html:errors/>
```

Note that Struts clears out the errors from the previous page (see step 6 in ["The RequestProcessor"](#) earlier in this chapter).

19.7.2.3. <logic:forward>

At times, you want a non-Struts JSP to take the user to a page managed by Struts. The classic use for this is to take a request to the application's welcome page (defined in `web.xml`) and have it handled by Struts. For example, our `index.jsp` forwards the request to the global forward `home`:

```
<%@ include file="/includes/setup.inc" %>
<logic:forward name="home" />
```

The global forward `home` is bound to the path `/home.do` which returns `home.jsp`.

19.7.3. Using the JSTL Expression Language in Struts Tags

Everything you learned in [Chapter 4](#) regarding the JSTL EL can be used in the attributes of Struts tags. In your Struts download in the `contrib/struts-el/lib` directory, you will find a `struts-el.jar` file that may be used in place of `struts.jar`. In your `taglibs` directives, now define your HTML tags like this:

```
<%@ taglib prefix="html-el"
    uri="http://jakarta.apache.org/struts/tags-html-el" %>
```

Now you can use dynamic expressions as the values of attributes on Struts tags. For example:

```
<html-el:link forward="{home}">Return to Home Page</html-el:link>
```

In this example, we make the choice of the `forward` attribute dynamic, based on the value of the `home` page attribute.



19.8. Struts Plug-ins

Struts plug-ins are used for defining operations that should occur when Struts is starting up and shutting down.

The `PlugIn` interface defines `init()` and `destroy()` methods that are called at the start and end of the application lifecycle. The most common plug-in is the Struts validator. But the plug-in infrastructure is a great way to get Struts to call your own initialization routines. Because the `init()` method is passed a reference to the Struts `ModuleConfig`, it is particularly helpful for initializations that require some knowledge of what's going on within Struts. If your initialization routines are generic, you should implement them simply as a `javax.servlet.ServletContextListener`. In the example application, it is helpful to put a reference to the `Library` model into the application context. Here's the complete `LibraryPlugIn`:

```
import javax.servlet.ServletException;

import org.apache.struts.action.ActionServlet;
import org.apache.struts.action.PlugIn;
import org.apache.struts.config.ModuleConfig;

public class LibraryPlugIn implements PlugIn {

    public void destroy( ) {
    }

    public void init(ActionServlet actionServlet, ModuleConfig moduleConfig)
        throws ServletException {
        actionServlet.getServletContext( ).setAttribute(WebGlobals.LIBRARY_KEY,
            new Library( ));
    }
}
```

Now any Struts `Action` or JSP page can easily retrieve a reference to the singleton `Library` by looking it up from the application context. For example, in `view/adminbooks.jsp`, we can use the reference to the library to loop through the books:

```
<c:forEach var="book" items="${library.bookList}">
```

This is a trivial example, but as you can see, implementing a Struts `PlugIn` is a convenient technique for any kind of initialization routine.



19.9. DynaActionForms and the Struts Validator

After the early versions of Struts were released, it became clear that there were certain `ActionForm` idioms that were so conventional, it would be helpful to introduce ways to declare such `ActionForms` without requiring Java code to be written. For example, most `ActionForms` are nothing more than ordinary JavaBeans with getters and setters for very common property types: `Strings`, `ints`, `Booleans`, and so forth. Similarly, validations observed certain conventions: some fields were required, others were required and needed to be sequences of digits forming integers, and so forth.

Enter `DynaActionForms` and the Struts validator.

`org.apache.struts.action.DynaActionForm` is a subclass of `ActionForm` that creates properties on the fly through the Java Reflection API. In your Struts config file, you simply declare that your form is bound to a `DynaActionForm`, and then write XML to define the properties. Instead of accessing the properties through explicit method calls (e.g., `getUsername()`), you call a generic `get` method with the name of the property (for example, `get("Username")`).

We will focus on a subclass of `DynaActionForm` that provides for writing even less code (or more XML!) called `org.apache.struts.validator.DynaValidatorForm`. With `DynaValidatorForm`, we can not only declare our properties in XML, we can also define standard validations that will be exercised by the Struts validator (a Struts `PlugIn`).

The example application provides two versions of the `LoginForm`. One version is implemented through Java code; the other is implemented through declarations in the Struts configuration file, binding the form to an instance of `org.apache.struts.validator.DynaValidatorForm`. To observe the difference between the Java implementation of the `LoginForm` and the dynamic version, you will need to look at the comments in `struts-config.xml`, `LoginAction.java`, and `index.jsp`. In each of these three files, there are lines to be commented in or out depending on whether you want to use the Java implementation of `LoginForm` or the dynamic version. Additionally, the dynamic version depends on three more files: `validator-rules.xml`, `validation.xml`, and `application.properties`.

For the `DynaValidatorForm` version, the `LoginForm` is defined in `struts-config.xml`.

```
<form-bean name="LoginForm"
  type="org.apache.struts.validator.DynaValidatorForm">
  <form-property name="username" type="java.lang.String"/>
  <form-property name="password" type="java.lang.String"/>
</form-bean>
```

Voila! You have just saved yourself about 30 lines of Java code.^[7] `LoginAction.java` requires a bit of tweaking, because now the username and password values are retrieved through the generic `get` method on `DynaValidatorForm`:

[7] For another example of using a `DynaValidatorForm`, see the comments in `struts-config.xml` regarding the form bean

`BookDisplayForm`. For this example, we have to extend `DynaValidatorForm` to handle checkbox values.

```
DynaActionForm loginForm = (DynaActionForm) form;
String username = (String) loginForm.get("username");
String password = (String) loginForm.get("password");
```

The final components you will need to look at are the two configuration files *validation.xml* and *validator-rules.xml*, which work together, and *index.jsp*. The *validator-rules.xml* file is a standard file that comes with Struts. You add it to the web archive under */WEB-INF*; you will not have to change this file unless you are introducing validations of your own. The *validation.xml* file contains your requirements for validation; it also goes under */WEB-INF*. References to both of these files must be present in the declaration of the Struts validator plug-in in *struts-config.xml*.

```
<plug-in className="org.apache.struts.validator.ValidatorPlugIn">
  <set-property
    property="pathnames"
    value="/WEB-INF/validator-rules.xml,/WEB-INF/validation.xml"/>
</plug-in>
```

The *validation.xml* file is where you define the mappings between your form fields and the validations that are required for them. Here's an extract of our *validation.xml* that defines the validations for the `LoginForm` (for the full story on the structure of this file and the elements that comprise a validation declaration, see [Chapter 11](#) in O'Reilly's *Programming Jakarta Struts*):

```
<form name="LoginForm">
  <field property="username" depends="required">
    <msg name="required" key="errors.required"/>
    <arg0 key="login.username"/>
  </field>
  <field property="password" depends="required">
    <msg name="required" key="errors.required"/>
    <arg0 key="login.password"/>
  </field>
</form>
```

In this extract from *validation.xml*, we specify the validations for the fields `username` and `password` of the form `LoginForm`. The `depends` attribute of the `<field>` element specifies the names of the validations that must be executed when the request processor calls the `validator()` method on the `ActionForm` (i.e., the `DynaValidatorForm`). The `<msg>` element defines the name of the key in *application.properties* that should be used as an error message template when the validation fails. Thus, in *application.properties*, we have defined the property `errors.required` as `{0} is required`. The `<arg0>` element provides the key for the name of the field to fill into the `{0}` in the error message template.

We have left the last bit of magic for last. The Struts validator plug-in provides for client-side validation (through JavaScript) as well as server-side validation. To enable this, you must provide some additional code in the JSP. To the `<html:form>` tag, you must add an `onsubmit` attribute to ensure that the appropriate dynamically generated JavaScript is called for the particular form. The revised `<html:form>` tag looks like this:

```
<html:form action="/login" focus="username"
  onsubmit="return validateLoginForm(this);">
```

And to generate the JavaScript itself, we add, after the `</html:form>` close tag, an `<html:javascript>` tag:

```
<html:javascript formName="LoginForm"/>
```

In "The ActionForm and the JSP" earlier in this chapter, we discuss routine validation through the `validate()` method that may be overridden when implementing an `ActionForm`. In the default implementation, if you enter nothing for your credentials during login, you will see error messages in the page. These messages are the results of server-side validation conducted by the Java implementation.

Now when you attempt to log in with empty credentials, you should see validation messages in the form of JavaScript pop-ups. Thanks to the Struts validator plug-in, you have just saved yourself the pain of writing and debugging JavaScript.

Chapter 20. Hibernate

Object relational mapping (ORM) frameworks provide application developers with the ability to focus on the high-level language of the domain instead of the lower-level language of a database. Hibernate is a framework that allows you to interact with your relational data in an object-centric fashion. Hibernate's mapping strategy allows you to convert your object-oriented schema into a database schema and vice versa. This means that as a Java programmer, you interact only with Java to get your database work done. Hibernate takes care of bridging the gap between your code and the database.

Just as important, Hibernate's API is pluggable. Though it comes with a variety of services out of the box, whenever you don't like one of them, you can replace it. For example, Hibernate provides a local transaction manager for single connections to a single datasource. If you need cross-database transactions, you can simply plug in the JTA instead. This pluggability allows Hibernate to be used in a broad array of situations since it plays well with its surroundings.

We wrote this chapter and the accompanying code with the beta versions of Hibernate 3.0. Some of the behavior and interfaces in the framework may change slightly by the time the book is published. Before we get started, here's a caveat: Hibernate is a big subject, and several books are available. We recommend *Better, Faster, Lighter Java* by Bruce Tate and Justin Gehtland and *Hibernate: A Developer's Notebook* by James Elliott, both from O'Reilly.



20.1. The Sample Application

In order to focus on the features and requirements of Hibernate, we are using a relatively simple application as an example. The application will track the students, professors, classes, and departments of a university. Through this framework, we'll be able to demonstrate the configuration, mapping, and manipulation of persistent domain objects.

The application will consist of five major domain classes:

Department

Will contain multiple classes and professors but no students

UniversityClass

Will have a single professor and multiple students, will belong to a single department, and will have a single syllabus

Syllabus

Will belong to a `UniversityClass`; every `Syllabus` will be unique to a specific `UniversityClass`

Professors

Will teach one or more classes but have a single primary department

Students

Will attend multiple classes but have no formal association with either a `Professor` or a `Department`

In addition, we'll introduce the `Person` interface to serve as a common ancestor between `Professor` and `Student`. Last, we need to add a layer to manage persistent domain objects. For other applications, you might decide to add a Data Access Object (DAO) with a façade layer on top. We decided that a DAO layer was overkill for such a small application, so we omitted it.

20.2. Principles of Hibernate

Often, programmers tend to skip introductory material and move right to the code. Sometimes we skip the code examples too and just pull up a seat at the keyboard and start hacking. Tempting as it may be to do so, plowing ahead without understanding the principles that Hibernate was built on is a dangerous thing to do. While persistence frameworks are meant to make life easier, they can be your worst nightmare if you don't understand the principles that drive the framework.

20.2.1. Transparency

First and foremost, Hibernate strives to be a transparent service provider. When authoring an application, your customer derives value not from the choice of tools you use to create an implementation but from the domain model you create to solve his real-world problems. That domain model should be as testable, portable, and deployable as possible. Hibernate goes to great lengths to ensure that your application is as minimally dependent on it, its libraries, and its own dependencies as possible.

This means that the domain objects themselves are oblivious to the persistence mechanism you employ. They implement no interfaces, extend no base classes, refer to no APIs, nor even make any design concession to the Hibernate framework. The persistence logic and APIs are encapsulated away in the façade layer, allowing your domain objects to be truly domain-specific.

20.2.2. Reflection

Any ORM framework must provide a way to dynamically generate SQL CRUD (create, read, update, delete) statements to bridge the chasm between the running application and the persistence store. When things change in the domain, the framework has to know about them to persist them to the data store and vice versa. This means that the ORM framework has to have knowledge of the specifics of not only the class definitions but also the runtime instances of those classes.

That kind of dynamic knowledge can come from one (or more) of three sources:

Reflection

The framework can combine mapping files with runtime reflection to generate CRUD statements based on current values.

Bytecode injection

The framework can instrument your compiled domain classes, injecting new code paths that

automatically generate CRUD statements or related metadata.

Code generation

The framework can, through a more detailed mapping scheme, allow you to decorate your domain with metadata that will become Java code when run through a preprocessing step.

When most Java programmers are asked for their first thoughts about the Reflection API, the word that pops from their mouths is "slow." While it's true that calls made through reflection are necessarily slower than native calls with direct manipulation of Java classes, the difference between the two has been closing with each release of the JVM. The Hibernate team explicitly chose reflection because it has minimal impact on the build process for an application and because the impact of having a few reflective calls will be hidden by any actual access to the database, which is where most applications incur most of their operational cost.

The other two choices, bytecode injection and code generation, change the build process of an application and result in code being inserted into your domain classes. While not inherently bad, the Hibernate team chose instead to keep your domain classes unencumbered with extraneous service code by choosing reflection.

20.2.3. Java, JDBC, and RDBMS

Hibernate is a framework that specifically provides for transparent persistence of Java objects over JDBC^[1] to a relational database management system. This specificity allows Hibernate to provide performance optimizations and general ease-of-use features that would otherwise be more difficult.

[1] For more information on JDBC, see [Chapter 8](#).

20.2.4. Pluggability

Hibernate provides a variety of services to an application: transaction management, caching, optimizations, and so on. These services might be adequate for a broad spectrum of applications, but sometimes application requirements call for specific service providers or features from the service that the Hibernate version cannot support. Hibernate allows for almost every service in its libraries to be replaced by a third-party implementation, usually through nothing more than a change to the configuration files.

20.2.5. Performance

Hibernate is designed to provide high-performance access to a relational database. In the most basic sense, the Hibernate API is designed to favor patterns and programming decisions that allow for high concurrency by not holding open JDBC connections for longer than absolutely necessary. Additionally, Hibernate caches SQL statements to be executed as late as possible in large batches and holds prepared statements for faster execution. Through optimization techniques like lazy loading and proxy initialization, Hibernate gives developers still more control over the performance of their

applications. Last, through a series of debugging and monitoring features, developers can learn detailed information about how well their application performs both at development and deployment time.



20.3. Configuration and Mapping

Hibernate applications need two separate sets of configuration information. The first is global metadata that connects the Hibernate framework to your specific persistence store and provides the framework with any information about finding, connecting to, and manipulating that store. The second is the mappings that connect your domain objects to the tables where they will be persisted, also defining the relationships between your classes and how those relationships will manifest in the database.

20.3.1. Global Configuration

Hibernate needs to be told where to find the database that your application will be using and how to connect to it. Additionally, you can configure a variety of services to be provided to your application from the Hibernate framework. You can do this programmatically using the `Configuration` and `Hibernate` classes provided in the Hibernate APIs (see "The Hibernate API" later in this chapter) or you can do it declaratively.

The declarative settings can be placed in one of two files:

- *hibernate.properties*
- *hibernate.cfg.xml*

Both formats offer the same features but with a different syntax. Creating either file and placing it within your application's classpath will automatically configure Hibernate with the settings therein. If both files present on the classpath, both are read in, and the *hibernate.cfg.xml* settings are used to override the settings in the *hibernate.properties* file. And, of course, you can also override any of these settings programmatically.

The XML format has become popular among developers for Hibernate settings because of the familiarity and expressiveness of XML as well as the common use of XML configuration formats in a variety of contexts. J2EE deployment descriptors, application server configuration files, and the Hibernate mapping files all use XML file formats, for example. For the sake of simplicity, however, we use the properties file format for our examples here. Translating these to the XML format is fairly straightforward.

At a bare minimum, the configuration should specify the location and type of the database you are using as well as the default access credentials to use to establish a connection (see Example 20-1).

Example 20-1. Sample hibernate.properties file

```
hibernate.dialect org.hibernate.dialect.MySQLDialect
hibernate.connection.driver_class com.mysql.jdbc.Driver
hibernate.connection.url jdbc:mysql://localhost/registration
hibernate.connection.username registration
hibernate.connection.password registration
```

The first line in the preceding configuration is especially important; it tells Hibernate what flavor of database you are using so that the framework can generate acceptable SQL queries, using the correct types. See the Hibernate documentation for a full list of available dialects; the framework supports all major database vendors.

After the basics, Hibernate provides a wide array of further configuration choices that you can add to *hibernate.properties* or *hibernate.cfg.xml*. Following are several options, broken down by category. The list is not exhaustive, but we find these settings to be particularly helpful. Again, we present these for the properties file format only; equivalent options are available in the XML format as well.

20.3.1.1. Debugging

`hibernate.show_sql`

Set this to `true` to have Hibernate dump each SQL statement it generates to stdout. This is invaluable if you are trying to find problems in the data being transferred or performance bottlenecks, but be aware that turning it on causes a performance degradation.

`hibernate.generate_statistics` (new in 3.0)

Set this to `True` to have Hibernate create and track a series of metrics about your application. You can access these metrics individually using the `getStatistics()` method of `SessionFactory`, but the easiest way to use this data is to call `SessionFactory.getStatistics().logSummary()`, which outputs a summary of statistics for the application that looks like Example 20-2.

Example 20-2. Hibernate statistics output

```
2005-03-16 17:07:33,950 INFO [org.hibernate.stat.StatisticsImpl] - Logging statistics...
2005-03-16 17:07:33,953 INFO [org.hibernate.stat.StatisticsImpl] - start time: i
1111010849088
2005-03-16 17:07:33,958 INFO [org.hibernate.stat.StatisticsImpl] - sessions opened: 1
2005-03-16 17:07:33,961 INFO [org.hibernate.stat.StatisticsImpl] - sessions closed: 1
2005-03-16 17:07:33,964 INFO [org.hibernate.stat.StatisticsImpl] - flushes: 0
2005-03-16 17:07:33,967 INFO [org.hibernate.stat.StatisticsImpl]
- connections obtained: 1
2005-03-16 17:07:33,980 INFO [org.hibernate.stat.StatisticsImpl] - second level cache
puts: 0
2005-03-16 17:07:34,052 INFO [org.hibernate.stat.StatisticsImpl] -
second level cache hits: 0
2005-03-16 17:07:34,069 INFO [org.hibernate.stat.StatisticsImpl] -
second level cache misses: 0
2005-03-16 17:07:34,073 INFO [org.hibernate.stat.StatisticsImpl] - entities loaded: 2
2005-03-16 17:07:34,077 INFO [org.hibernate.stat.StatisticsImpl] - entities updated: 0
2005-03-16 17:07:34,083 INFO [org.hibernate.stat.StatisticsImpl] - entities inserted: 0
2005-03-16 17:07:34,086 INFO [org.hibernate.stat.StatisticsImpl] - entities deleted: 0
```

```

2005-03-16 17:07:34,089 INFO [org.hibernate.stat.StatisticsImpl] - entities fetched
(minimize this): 0
2005-03-16 17:07:34,092 INFO [org.hibernate.stat.StatisticsImpl] - collections loaded: (
2005-03-16 17:07:34,099 INFO [org.hibernate.stat.StatisticsImpl] - collections updated:
2005-03-16 17:07:34,137 INFO [org.hibernate.stat.StatisticsImpl] - collections removed:
2005-03-16 17:07:34,141 INFO [org.hibernate.stat.StatisticsImpl] - collections
recreated: 0
2005-03-16 17:07:34,145 INFO [org.hibernate.stat.StatisticsImpl] - collections fetched
(minimize this): 0
2005-03-16 17:07:34,149 INFO [org.hibernate.stat.StatisticsImpl] - queries executed to
database: 1
2005-03-16 17:07:34,152 INFO [org.hibernate.stat.StatisticsImpl] - query cache puts: 0
2005-03-16 17:07:34,156 INFO [org.hibernate.stat.StatisticsImpl] - query cache hits: 0
2005-03-16 17:07:34,171 INFO [org.hibernate.stat.StatisticsImpl] - query cache misses: (
2005-03-16 17:07:34,175 INFO [org.hibernate.stat.StatisticsImpl] - max query time:
635ms

```

You can use the statistics to track down bottlenecks in your code and understand the effects of different configuration and API usage strategies on your system's performance.

20.3.1.2. Other JDBC properties

```
hibernate.connection.pool_size
```

Set the size of the connection pool, if using a poolable provider.

```
hibernate.connection.isolation
```

The default isolation level for transactions. The allowed values are those specified in the `java.sql.Connection` interface. See Chapter 16 for more details on transaction isolation levels.

20.3.1.3. Transaction management

```
hibernate.transaction.factory_class
jta.UserTransaction, hibernate.transaction.manager_lookup_class
```

Set these to replace Hibernate's provided local transaction manager with the JTA implementation from your container.

20.3.1.4. Optimization

```
hibernate.jdbc.fetch_size
hibernate.jdbc.use_scrollable_resultset
```

Optimize Hibernate's use of JDBC and its connection-level properties.

```
hibernate.cglib.use_reflection_optimizer
```

Replaces as much use of reflection with bytecode injection as possible.

20.3.1.5. JNDI

```
hibernate.connection.datasource
hibernate.jndi.url
hibernate.jndi.class
```

Configure Hibernate to use JNDI to find a container-managed datasource.

20.3.1.6. SQL

```
hibernate.query.substitutions
```

A collection of name/value pairs specifying token replacement options when creating Hibernate Query Language (HQL) queries. You can use this to replace SQL tokens with more convenient replacements (e.g., `true=1` , `false=0`).

20.3.2. The Hibernate Type System

When you begin writing mapping files, you can use either Java native types or Hibernate types to match your Java properties to their database field counterparts. It is usually preferable to use the Hibernate types since they are there to provide the bridge between the Java type system and your database vendor type system. The Hibernate types provide mappings for every major Java native type, such as `int` , `double` , `string` , `date` , `time` , `boolean` , and so on.

If the types provided are not sufficient to suit your needs, you can extend (or even replace) the existing definitions by writing your own conversions. To add your own conversion type, you have to write a class that implements `org.hibernate.UserType` (if it is a single Java value mapping to a single database column) or `org.hibernate.CompositeUserType` (if it is a combination of one or more properties and/or one or more columns). You can then add this new conversion class to the type mappings available and use it for your domain objects. If you need even more specialized mappings, Hibernate provides a bevy of alternate implementations for you, like `UserCollectionType` and `ParameterizedType` .

20.3.3. Class Mappings

Once you configure Hibernate to locate your database, next you have to map your domain model onto the database schema. Bear in mind that Hibernate is Java-centric, not data-centric, meaning that you provide mappings for your classes, not your tables. Each persistable class in your model has to have a mapping (or portion of a collected mapping file). Each class must map to one root table. If your class is represented in the data store by multiple tables, you will have to either decompose the class into multiple, smaller classes or create a database view that spans the physical tables. Hibernate supports database views transparently; some databases, however, do not provide this feature.

Class mappings are stored in files that fit the pattern `<classname>.hbm.xml`. Each Hibernate mapping file should specify the appropriate Hibernate DTD version that it is based on. The mapping must then specify the class name being mapped, including the full package name. You can use a convenience attribute on the `<hibernate-mapping>` element called `package` to set the default package name for the mapping so that you don't have to keep typing it everywhere, as shown in Example 20-3.

Example 20-3. Sample class mapping (Professor.hbm.xml)

```
<hibernate-mapping package="com.oreilly.jent">
  <class name="Professor" table="professors">
    <!-- ... -->
  </class>
</hibernate-mapping>
```

Mapping files are often placed in the same directory as the classes they map. However, you can place them anywhere on the classpath. If you prefer a separate mapping folder, by all means configure your application that way. Just make sure that Hibernate has access to the folder where the mapping files reside.

20.3.3.1. Unique IDs

Every persistable class must have one property or a combination of properties that can be used to uniquely identify the row in the database where the instance lives. The mapping file uses the `<id>` element to specify this property or combination and tell Hibernate how to manage its value. The ID value can either be controlled by your application or you can let the database handle it.

Example 20-4 shows a simple class from our application, `Professor`.

Example 20-4. Professor.java

```
public class Professor {
    private Long id;
    private String firstName;
    private String lastName;
    // getters/setters elided for brevity...
}
```

Example 20-5 is the mapping file for it, showing the application-assigned ID strategy.

Example 20-5. Class mapping for Professor (Professor.hbm.xml)

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping package="com.oreilly.jent.hibernate">

<class name="Professor" table="professors">
  <id name="id" column="professor_id"
    type="long">
    <generator class="assigned"/>
  </id>
</class>
```

When using application-assigned IDs, you are in charge of the logic that generates those IDs and manages their uniqueness. Attempting to persist an object whose ID is either unpopulated or a duplicate of an existing record causes Hibernate to throw exceptions.

If you would rather allow the database to manage IDs, you can choose from a variety of methods, depending on what your database supports. You can choose sequence tables (for example, if you are using Oracle), HiLo algorithms, guides and uuids, or plain old autoincrementing fields. You can specify the particular strategy you want, or you can let Hibernate choose for you by specifying the "native" generator class. Our demo app uses autoincrementing fields in a MySQL database, so our ID mapping in *Professor.hbm.xml* actually looks like this:

```
...
<id name="id" column="professor_id" type="long">
  <generator class="increment"/>
</id>
...
```

20.3.3.2. Properties

The mapping file then provides metadata for each persistable property of your class. Not every property needs to be persistable; if the property doesn't appear in your mapping file, Hibernate ignores its existence entirely. This can be useful for volatile or calculated properties or properties whose values must be looked up from a nonrelational data store (like an XML file).

When mapping persistent properties, at a bare minimum, you have to specify the name of the property and the name of the column it maps to. Optionally, you can specify the type, but Hibernate can also discern the type via reflection on the class itself, so unless the standard Hibernate mapping is insufficient for your needs, you can just omit the type from your mapping. The demo application supplies types everywhere for clarity. Extending our `Professor` mapping to add the `firstName` and `lastName` properties, the full mapping file now becomes:

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping package="com.oreilly.jent.hibernate">

<class name="Professor" table="professors">
    <id name="id" column="professor_id"
        type="long">
        <generator class="increment"/>
    </id>
    <property name="firstName" column="first_name"
        type="string"/>
    <property name="lastName" column="last_name"
        type="string"/>
</class>
```

20.3.4. Collection Types

So far, we've been working with what Hibernate calls "first-class" objects. Any instance of `Professor` is unique, and Hibernate manages the identity constraints of each instance of `Professor` to ensure that you access only a single instance per record at any given time. Quite often, though, your domain classes will have a collection property of simple value types that needs to be persisted. For example, every `Professor` might have one or more email addresses, which are just represented as simple `String`s. You don't want to create a first-class persistent object from them, but you do need to map them to a table in the database.

The schema is shown in Example 20-6.

Example 20-6. SQL for creating the Professor database

```
CREATE TABLE 'professors' (
    'professor_id' int(11) NOT NULL auto_increment,
    'first_name' varchar(25) NOT NULL,
    'last_name' varchar(50) NOT NULL,
    PRIMARY KEY ('professor_id')
)
CREATE TABLE 'emails' (
    'professor_id' int(11) NOT NULL,
```

```

        'email' varchar(255) NOT NULL
    )

```

`Professor` s might have 0, 1, or many email addresses in the corresponding table. The domain will represent the collection of email addresses as a `java.util.Set` :

```

public class Professor {
    private Long id;
    private String firstName;
    private String lastName;
    private Set emails;
    // getters/setters elided for brevity...
}

```

The last step is the mapping. When creating the mapping, you have to tell Hibernate which property is the collection of values, what table to go to when looking up the values, and what the foreign key relationship is. In our case, the property is called `emails` , the table is also called `emails` , and we've mimicked the primary key field from our `professors` table in the new `emails` table (`professor_id`). Here's the relevant excerpt from `Professor.hbm.xml` :

```

...
<set name="emails" table="emails">
    <key column="professor_id"/>
    <element column="email" type="string"/>
</set>
...

```

When mapping collections, you can choose `Set` , `Map` , `List` , `Bag` , or `Array` as the collection type. Each has its own distinct syntax for the mapping file.

20.3.5. Relationship Mappings

The biggest challenge for an ORM framework is mapping the relationships between persistent classes and defining the rules for loading and saving those relationships. Hibernate provides mapping tools for even the most complex domain models, with relationship mappings for one-to-one , one-to-many, and many-to-many relationships both unidirectionally and bidirectionally. In addition, Hibernate allows for lazy associations between objects, which means Hibernate postpones loading the related classes until they are needed.

20.3.5.1. One-to-one

In our `University` application, each `UniversityClass` has a single `Syllabus` , and each `Syllabus` is unique

to that class. Example 20-7 and Example 20-8 show a partial listing of the domain classes.

Example 20-7. UniversityClass.java

```
public class UniversityClass {
    private Long id;
    private String name;
    private Syllabus syllabus;
    //getters/setters elided for clarity...
}
```

Example 20-8. Syllabus.java

```
public class Syllabus {
    private Long id;
    private String abstract;
    private String fulltext;
    private UniversityClass uclass;
    //getters/setters elided for clarity...
}
```

In our example, `UniversityClass` is the primary object in the relationship, and we'll consider it the parent. `Syllabus` is the dependent, or child, object. The relational schema we'll generate will look like Example 29.

Example 20-9. SQL for creating the University database

```
CREATE TABLE 'university_classes' (
    'university_class_id' int(11) NOT NULL auto-increment,
    'name' varchar(255) NOT NULL,
    'syllabus_id' int(11),
    PRIMARY KEY ('university_class_id')
)
CREATE TABLE 'syllabus' (
    'syllabus_id' int(11) NOT NULL auto-increment,
    'abstract' varchar(255) NOT NULL,
    'fulltext' text,
    PRIMARY KEY ('syllabus_id')
)
```

Notice the foreign key relationship: each record in `university_classes` points to a record in `syllabus`.

According to our domain classes shown earlier, the relationship is bidirectional (each endpoint has a pub property exposing the other endpoint). Therefore, the mapping has to exist in both class's mapping files. The mapping of the class whose primary key is the foreign key to the other table (in this case, `Syllabus`) is fairly straightforward. We'd simply add this entry to the `Syllabus.hbm.xml` mapping file:

```
...
<one-to-one name="uClass" property-ref="syllabus" />
...
```

This entry tells Hibernate that the `Syllabus` class has a property called `uClass` that is at one end of a one-to-one relationship and that the other end can be found in a property called `syllabus` on whatever type the `uClass` property is. The mapping for `UniversityClass` is slightly more complex, as shown in this excerpt from the `UniversityClass.hbm.xml` file:

```
...
<many-to-one name="syllabus" column="syllabus_id"
  unique="true" />
...
```

Right away, you'll notice the `many-to-one` designation, even though we are mapping a one-to-one relationship. That's because the physical representation of the one-to-one looks just like the physical relationship of a many-to-one (a foreign key lookup in the database), but in this case we know we should find only a single match, hence the `unique=true` attribute. Hibernate will now manage this relationship, throwing exceptions if you ever try to create two records in one table that map to a single record in the other. The `column` attribute tells Hibernate which field in the record maps to the primary key column of the associated table. We've used the same name (`syllabus_id`) on both sides, but that is not a requirement.

20.3.5.2. One-to-many and many-to-one

One-to-one relationships are interesting, but the most common form of relationship mapping in most object domains is the bidirectional one-to-many. In this relationship type, one object is the parent, and has multiple children. The children are bound to their parent and only their parent; if they are moved to another parent, the association with the original parent is severed. In our demo application, there is a one-to-many relationship between several classes. Let's look at `Professor` and `UniversityClass`.

Each `Professor` teaches one or more classes, but each class is taught by a single `Professor`. If you switch `Professor`s for a class, the original `Professor` no longer has to show up on Tuesdays at 9:30, so the original relationship must be severed. Here's the updated SQL for the relevant tables:

```
CREATE TABLE 'professors' (
  'professor_id' int(11) NOT NULL auto_increment,
  'first_name' varchar(25) NOT NULL,
  'last_name' varchar(50) NOT NULL,
  PRIMARY KEY ('professor_id')
)
```

```
CREATE TABLE 'university_classes' (
  'university_class_id' int(11) NOT NULL auto-increment,
  'name' varchar(255) NOT NULL,
  'syllabus_id' int(11),
  'professor_id' int(11) NOT NULL,
  PRIMARY KEY ('university_class_id')
)
```

The `professors` table is the same as in the earlier example, but the `university_classes` table now has a column to hold the ID of the professor teaching it.

The changes to the domain classes, `UniversityClass` and `Professor`, are simple. Here's `UniversityClass`

```
public class UniversityClass {
  private Long id;
  private String name;
  private Syllabus syllabus;
  private Professor professor;
}
```

And here is `Professor` :

```
public class Professor {
  private Long id;
  private String firstName;
  private String lastName;
  private Set emails;
  private Set classes;
}
```

Once again, you have to provide mapping support in the files for both domain classes. The `Professor` mapping has to specify the collection type while the `UniversityClass` has to specify merely the foreign key mapping. Here's the change to the mapping in `Professor.hbm.xml`:

```
...
<set name="classes">
  <key column="professor_id"/>
  <one-to-many class="UniversityClass"/>
</set>
...
```

And here's the change to *UniversityClass.hbm.xml*:

```
...
<many-to-one name="professor" column="professor_id" not-null="true"/>
...
```

Whenever a `Professor` instance is loaded, Hibernate will now know how to find and instantiate all the `UniversityClass` es that the `Professor` teaches. Likewise, loading any `UniversityClass` prompts Hibernate to find the associated `Professor` .

20.3.5.3. Many-to-many

Many-to-many relationships are more difficult to represent in a database. These types of relationships require a *join table*, an intermediate entity that has a foreign key relationship to each of the two primary tables. You could map such a relationship by creating a first-class object that maps to the join table and then creating one-to-many relationships from each primary class to the join class, or you just perform the many-to-many mapping directly without adding the third domain model. Though the Hibernate team recommends the former approach, experience shows that many people choose the latter.

In our application, the relationship between `Students` and `UniversityClasses` is a classic many-to-many relationship. `Students` are supposed to enroll in (and attend) multiple classes per semester, and classes are supposed to have multiple students attending. The SQL to create the relevant tables is listed next. The `university_classes` table is the same as in our earlier example, but the new `students` and `students_classes` tables hold basic student information and links each student to one or more classes:

```
CREATE TABLE 'university_classes' (
    'university_class_id' int(11) NOT NULL auto-increment,
    'name' varchar(255) NOT NULL,
    'syllabus_id' int(11),
    'professor_id' int(11) NOT NULL,
    PRIMARY KEY ('university_class_id')
)
CREATE TABLE 'students' (
    'student_id' int(11) NOT NULL auto-increment,
    'firstName' varchar(25) NOT NULL,
    'lastName' varchar(50) NOT NULL,
    PRIMARY KEY ('student_id')
)
CREATE TABLE 'students_classes' (
    'student_id' int(11) NOT NULL,
    'university_class_id' int(11) NOT NULL
)
```

Each end of the domain relationship now has to have a property that is a collection of the other end of the relationship. Here's the updated `UniversityClass` class:

```
public class UniversityClass {
    private Long id;
    private String name;
    private Syllabus syllabus;
    private Professor professor;
    private Set students;
}
```

Here is the new `Student` class to represent student data:

```
public class Student {
    private Long id;
    private String firstName;
    private String lastName;
    private Set classes;
}
```

Finally, each class's mapping has to specify the collection properties and join table specifics to create the full, bidirectional relationship. We'd have the following mapping in the `Student.hbm.xml` file:

```
...
<set name="classes" table="students_classes">
    <key column="student_id"/>
    <many-to-many column="university_class_id" class="UniversityClass"/>
</set>
...
```

The following mapping entry would go in the `UniversityClass.hbm.xml` file:

```
...
<set name="students" table="students_classes" inverse="true">
    <key column="university_class_id"/>
    <many-to-many column="student_id" class="Student"/>
</set>
...
```

The relationship in `UniversityClass` is marked `inverse=true`. This makes the `Student` the "parent" in the relationship, and Hibernate will not try to insert or modify the properties of the relationship if the changes are made only from the `UniversityClass` side.

20.3.5.4. Proxies

All of the relationship mapping that we have described up until now is very powerful but has one large downside. Without due care and consideration, a single load of any single record in the database could cause a cascading load effect that eventually loads the entire database. This is an unacceptable scenario any but the tiniest applications; not only would the load time be staggering, but the resource consumption would be unmanageable.

Instead, using lazy loading strategies, you should carefully manage what gets loaded and when. There are two different kinds of lazy initialization: proxies, for one-to-one mappings, and lazy collections, for one-to-many and many-to-many.

Declaring that a class can be proxied means that whenever that class appears on the "to-one" end of a relationship, instead of returning the object itself, Hibernate creates and returns an impersonator. This impersonator implements an interface or derives from a concrete class, as specified in the mapping file. However, it does not read any data from the database. Instead, whenever any of its properties are accessed for the first time, it asks Hibernate to load the values from the database at that moment. This delays the resource consumption and network round-trips until absolutely necessary, and, if that proxy is never utilized, the read never happens.

To declare a class to be "proxy-able," you merely append the `proxy` attribute to the class mapping definition. For example, the `Professor` class will be proxied by a dynamically derived subclass of `Professor`. The updated `Professor.hbm.xml` file would look like the following:

```
<class name="Professor" proxy="Professor">
  <!-- other mapping information -->
</class>
```

Later in this chapter, we'll introduce polymorphic mapping strategies. At that time, we'll formally introduce a new member of the domain, the `Person` interface. If you use such a domain model, you can choose to have the proxy be an implementation of any interface your domain class implements, so the mapping could become:

```
<class name="Professor" proxy="Person">
  <!-- other mapping information -->
</class>
```

With this mapping in place, the following scenario could play out:

1. The application loads an instance of `UniversityClass`.
2. Hibernate finds and loads the record for that `UniversityClass` instance.
3. Hibernate traverses the many-to-many relationship between `UniversityClass` and `Student`, loading each `Student` and placing her in the `students` set of `UniversityClass`.

4. Hibernate sees the relationship between `UniversityClass` and `Professor` , but does not load the associated `Professor` , instead creating a dynamic instance of a class that implements `Person` (as seen in the earlier mapping).
5. Hibernate returns the `UniversityClass` instance to the application.
6. The application accesses the `students` collection of the `UniversityClass` ; no further database access is required since the collection was already loaded.
7. The application accesses the `Professor` property of the `UniversityClass` ; Hibernate performs a read on the database and populates all the underlying property data.

You should be aware of several caveats when using proxies. First, if you are using classes that exist in a complex inheritance hierarchy, the proxy you receive from Hibernate cannot be cast to other members of that hierarchy. Second, if a class designated with the `proxy` attribute appears on the "to-many" side of a relationship, the collection will be populated by actual instances of the persistent class, not instances of proxy. Proxies are returned only when on the "to-one" side of an association.

20.3.5.5. Lazy loading

When you want to perform lazy loading on the "to-many" side of a relationship, the mapping strategy is very different. The proxy designation is an attribute of the class itself; whenever that class appears as a "to-one" association, the proxy will be used. When declaring lazily loaded collections, the mapping is an attribute of the relationship itself. So, if you want to declare that the `students` property of `UniversityClass` be lazily initialized, you would add `lazy="true"` to the relationship mapping in `UniversityClass.hbm.xml`:

```
...
<set name="students" table="students_classes" lazy="true">
  <key column="university_class_id"/>
  <many-to-many column="student_id" class="Student" inverse="true"/>
</set>
...
```

Instead of returning proxies of the objects in the collection, Hibernate returns a Hibernate-specific implementation of the collection interface itself. If you are using a `Set` , Hibernate returns an instance of `org.hibernate.collection.PersistentSet` . If you are using a `Map` , you'll get an `org.hibernate.collection.PersistentMap` and so forth.

If your application never makes use of the collection, the values of the objects in the collection are never loaded from the database. However, any access to the collection causes the entire collection to be loaded (barring advanced optimizations like scrollable result sets). This means any access of any individual element of the collection or any summary information (like the size of the collection). To determine the size of a collection, Hibernate would have to perform the actual select in order to determine the full result set of the join; if Hibernate must execute the full select anyway, it might as well return the results to the collection to prevent executing it again later.

20.3.5.6. Special note about lazy initialization

As we have not, as yet, covered the Hibernate API and usage patterns, we cannot speak adequately about the problem of disconnected objects, lazy loading, and the `LazyInitializationException`. For now, it is enough to say that anytime Hibernate needs to access the database, there must be an open JDBC connection to the database and the object for whom access is being made must be associated with that connection. Any attempt to access the database to populate lazily loaded associations without an open connection will result in a `LazyInitializationException`. We'll learn how to avoid this problem in "The Hibernate API" later in this chapter.

20.3.6. Polymorphic Mappings

Hibernate is built to supply relational mappings for Java, an advanced object-oriented programming language. Java provides full object-oriented design capabilities, which includes polymorphic inheritance. Hibernate must therefore support this feature. The lack of high-quality support for polymorphism was a major drawback in the EJB 1.x and 2.x specifications.

When creating an inheritance hierarchy, you can choose between concrete inheritance (base class/derived class) or interface inheritance (interface/implementation). Hibernate supports both methods with the same tools and techniques. In our demo application, we'll be using interface inheritance to model the persons at our university. Both the `Professor` and `Student` classes represent people and they share some common properties, namely an ID, first name, and last name, and a `Set` of `UniversityClasses`. Our `Person` domain object is modified to become:

```
public interface Person {
    Long id;
    String firstName;
    String lastName;
}
```

Let's also add a couple of properties to `Professor` and `Student` that are specific to each:

```
public class Professor implements Person {
    private Long id;
    private String firstName;
    private String lastName;
    private Set classes;
    private Set emails;
    private String degree;
}
public class Student implements Person {
    private Long id;
    private String firstName;
    private String lastName;
    private Set classes;
    private String ssn;
}
```

Hibernate provides three strategies for mapping such an inheritance chain, which are discussed next. The simplest version (table per class hierarchy) is easiest to map and set up in the database, but it is also the least data-normalized schema.

20.3.6.1. One table per class hierarchy

In this strategy, you create a single table that holds all instances of all classes in a single inheritance chain. This necessarily means that you will have some fields on some rows with null values. It also means that the table has to have a special field (called the discriminator field) to determine which class each row is supposed to map to.

Example 20-10 shows the schema for our combined `person` table.

Example 20-10. SQL for the combined person table

```
CREATE TABLE 'person' (
    'person_id' int(11) NOT NULL auto-increment,
    'person_type' varchar(15) NOT NULL,
    'first_name' varchar(25) NOT NULL,
    'last_name' varchar(50) NOT NULL,
    'ssn' varchar(11) NULL,
    'degree' varchar(11) NULL,
    PRIMARY KEY ('person_id')
)
```

The `person_type` field is our discriminator value. It is not mapped as a property on any of our domain classes but rather is used as metadata for the mapping itself. The `ssn` and `degree` fields are members of our derived types, and since there will be rows in the table that map to the other type (and hence these fields will not have values), the fields themselves must be marked so that they can contain `NULL` values.

When mapping any of the inheritance hierarchies, we move away from having a single mapping file per class to creating one mapping file for the entire hierarchy. The `<class>` element maps to the root of the hierarchy and then you use special subclass elements to represent the children. To map a tree that goes deeper than two levels, the subclasses can have their own subclasses. We'll create a mapping file starting with our `Person` interface, as shown in Example 20-11.

Example 20-11. Mapping for the Person interface

```
<hibernate-mapping package="com.oreilly.jent.hibernate">
<class name="Person" table="person" polymorphism="explicit">
    <id name="id" column="id" type="long">
        <generator class="native"/>
    </id>
    <!-- Tells Hibernate which column is used to determine the type of a given
row -->
    <discriminator column="person_type" type="string"/>
</class>
</hibernate-mapping>
```

```

<property name="firstName" column="first_name" type="string"/>
<property name="lastName" column="last_name" type="string"/>
<!-- Defines a specific implementation of the interface -->
<subclass name="Professor"
  discriminator-value="professor">
  <property name="degree" column="degree"
    type="string"/>
  <set name="classes" lazy="true">
    <key column="person_id"/>
    <one-to-many class="UniversityClass"/>
  </set>
</subclass>
<!-- Defines a second implementation of the interface -->
<subclass name="Student"
  discriminator-value="student">
  <property name="ssn" column="identifier"
    type="string"/>
  <set name="classes" table="students_classes" lazy="true">
    <key column="person_id"/>
    <many-to-many class="UniversityClass"
      column="university_class_id"/>
  </set>
</subclass>
</class>
</hibernate-mapping>

```

With this mapping, Hibernate can populate all properties of any instance loaded from the `Person` table. Hibernate also provides full type fidelity for any loaded instance; when you load an instance of a `Student` even if you ask for it as a `Person`, you can cast the result down to a `Student`.

20.3.6.2. One table per class

With this strategy, you create a separate table for every class in the hierarchy (whether or not it is a concrete class). This provides the highest degree of data normalization but makes the actual mapping somewhat more complex. Our schema would become:

```

CREATE TABLE 'person' (
  'person_id' int(11) NOT NULL auto-increment,
  'first_name' varchar(25) NOT NULL,
  'last_name' varchar(50) NOT NULL,
  PRIMARY KEY ('person_id')
)
CREATE TABLE 'students' (
  'person_id' int(11) NOT NULL,
  'ssn' varchar(11) NOT NULL,
  PRIMARY KEY ('person_id')
)
CREATE TABLE 'professors' (

```

```

    'person_id' int(11) NOT NULL,
    'degree' varchar(25) NOT NULL,
    PRIMARY KEY ('person_id')
)

```

The `person` table is now the primary table where the autoincrementing ID is created. The dependent tables, `students` and `professors`, utilize the `person_id` as the primary key but do not mark it as autoincrementing since the value will always be supplied by the root table. Notice also that the `ssn` and `degree` fields no longer have to accept `NULL` values since rows will be added to the dependent tables only for actual instances of that class.

As you'll see in Example 20-12, there are not many changes to the mapping except that you no longer need the discriminator column, the subclasses become joined subclasses, and you have to specify the table they live in.

Example 20-12. Updated Person mapping file using a table per class

```

<hibernate-mapping package="com.oreilly.jent.hibernate">
<class name="Person" table="person" polymorphism="explicit">
  <id name="id" column="id" type="long">
    <generator class="native"/>
  </id>
  <property name="firstName" column="first_name" type="string"/>
  <property name="lastName" column="last_name" type="string"/>
  <!-- Defines a specific implementation of the interface -->
  <joined-subclass name="Professor" table="professors">
    <key column="person_id"/>
    <property name="degree" column="degree"
      type="string"/>
    <set name="classes" lazy="true">
      <key column="person_id"/>
      <one-to-many class="UniversityClass"/>
    </set>
  </joined-subclass>
  <!-- Defines a second implementation of the interface -->
  <joined-subclass name="Student" table="students">
    <key column="person_id"/>
    <property name="ssn" column="identifier"
      type="string"/>
    <set name="classes" table="students_classes" lazy="true">
      <key column="person_id"/>
      <many-to-many class="UniversityClass"
        column="university_class_id"/>
    </set>
  </joined-subclass>
</class>
</hibernate-mapping>

```

Hibernate can now determine the actual type of an instance by whether or not it matches a row in the specific subclass table.

20.3.6.3. One table per concrete class

This mapping would require only two tables in our application, one each for `Professor` and `Student`. This method is also not optimally normalized since you have repeated columns defined in the two tables (each has `first_name` and `last_name`, for example). Furthermore, those columns have to have identical names so that Hibernate can map them correctly. Otherwise, this strategy looks very similar to the one-table-per-class strategy.

20.3.7. Mapping Tools

Creating appropriate mappings is the heart and soul of a good Hibernate application. Making sure that your relationships are efficiently modeled, that lazy initialization is established when necessary, and that the right type mappings are used is a time-consuming process. Beyond that, you then have to actually create the domain classes and the data schema.

The Hibernate team ships a variety of tools that you can use to ease the burden on your development team. There are command-line tools that can generate one or more of the artifacts (Java code, mapping files, or database schema) from the others. You can use existing tools like XDoclet or Middlegen to turn code decoration into mapping files (see Chapter 21 for general details on XDoclet). There's a set of plugins available for Eclipse, IntelliJ Idea, and most other IDEs. Finally, there's a standalone product available from the Hibernate team called Hibernate Console (formerly Hibern8IDE) that allows you to put your mappings to the test.

20.4. The Hibernate API

Now that we've covered how to map your domain onto a database, we'll examine using the Hibernate API to actually perform database operations. The various portions of the API are treated separately. A Hibernate application will usually make use of most of what you see here, though `Interceptors` and `Events` are somewhat less common than the others.

20.4.1. Configuration and Hibernate

Hibernate provides a class, `org.hibernate.cfg.Configuration`, that serves as the entry point to your application's interaction with the database. You use the `Configuration` class to load mapping files, specify global options, and retrieve metadata about your schema and objects.

If you are using the `hibernate.properties` or `hibernate.cfg.xml` files and they are in the classpath of your application, the easiest way to configure Hibernate is to create an instance of `Configuration` and call the `configure` method:

```
Configuration cfg = new Configuration( ).configure( );
```

The `configure` method can take no arguments, as shown, in which case Hibernate will search for the first matching configuration file. Optionally, you can pass a `String`, `url`, `File`, or `Document` to the method to load the configuration file from an alternate location.

This step needs to be taken only once per application lifetime. Parsing the configuration properties and establishing the global settings for Hibernate is a relatively lengthy and expensive process. The application should execute the step only once and cache the resulting instance of `Configuration` somewhere from which it can be retrieved later.

20.4.2. SessionFactory

After establishing the configuration settings, you use the configuration instance to create an instance of `org.hibernate.SessionFactory`. The `SessionFactory` generates actual connections to the data store. Usually, you should create only one `SessionFactory` per application lifetime. Keep a reference to that `SessionFactory` for use throughout the application.

```
SessionFactory factory = cfg.buildSessionFactory( );
```

In special cases, such as when you have to connect to multiple data stores, you will need multiple `SessionFactory`s. If your application must communicate with multiple databases, you must provide multiple sets of configuration properties and multiple connection wrappers. In this scenario, you would create multiple properties files (*datastore1.hibernate.properties* and *datastore2.hibernate.properties*) and load them each manually:

```
Configuration cfg1 =
    new Configuration.configure("datastore1.hibernate.properties");
Configuration cfg2 =
    new Configuration.configure("datastore2.hibernate.properties");
```

From each instance of `configuration`, you would build a separate `SessionFactory`. For example:

```
SessionFactory factory1 = cfg1.buildSessionFactory( );
```

20.4.3. Session

The `Session` object is the real meat of the Hibernate API. `Session` instances wrap logical JDBC connections. We say "logical" because the `Session` does not necessarily have to have an open JDBC connection; the connection could be in a closed state or have been released to the connection pool. Regardless, the `Session` instance is the only way to access the actual database connection in Hibernate, and it is the primary vehicle through which persistence operations are performed.

You retrieve an instance of `Session` by calling `SessionFactory.openSession()`. Hibernate creates an instance of `Session` but does not retrieve an open JDBC connection from the pool. That act is delayed until you use the `Session` in such a way as to require a connection. From that moment on, the `Session` has an open JDBC connection to use until you release the connection, by calling either `Session.disconnect()` or `Session.close()` (more on this in a minute).

Once the physical connection is established, you should use the `Session` to do your work as quickly as possible and then release the connection. The longer your application waits to release the connection the higher the likelihood of a concurrency conflict on the databases. Remember always that JDBC connections are rare, valuable resources and should be used as quickly as possible and released for others to use.

Whenever you use the `Session`, you should use appropriate error-handling blocks. The standard usage pattern is shown in [Example 20-13](#).

Example 20-13. Standard Session try/catch/finally usage pattern

```
Session session = null;
```

```

try {
    session = factory.openSession( );
    // Use the Session
} catch (Exception ex) {
    // Handle exceptions
} finally {
    if(null != session) session.close( );
}

```

20.4.3.1. Loading objects

The primary uses of the `Session` are to load and save first-class Hibernate objects. To load an instance of a persistent class, you have to tell Hibernate what kind of object you are looking for as well as the unique ID that allows Hibernate to find the specific instance:

```

// Load a UniversityClass
UniversityClass class =
    (UniversityClass)session.load(UniversityClass.class, new Long(1));

```

If you use the `Session` to load instances from an inheritance hierarchy, you specify the specific subclass you want to load or the base class if you don't care. Hibernate will not load the wrong type in response to such a request; if you ask for an instance of `Professor`, but provide the ID of a `Student`, Hibernate will return an `ObjectNotFoundException`.

```

// Load a Professor
Professor prof =
    (Professor)session.load(Professor.class, new Long(2));
// Load a Person
Person person =
    (Person)session.load(Person.class, new Long(1));

```

When any of these load commands is issued, the lazy initialization rules you established in your mapping files are applied. Loading the `UniversityClass` will cause all its `Students` to be loaded at the same time unless the `students` collection is designated as a lazy collection.

20.4.3.2. Saving new objects

Adding new objects is just as straightforward as loading them. First, create an instance of the given persistent class. Then, use the `Session` to save the object. If there are conflicts when you attempt the save, Hibernate will throw detailed exceptions to explain the problem.

```

// Create a new Professor
Professor prof = new Professor( );
prof.setFirstName("Jim");
prof.setLastName("Jacobs");
Session session = null;
try {
    session = factory.openSession( );
    session.save(prof);
} catch (Exception ex) {
    // Handle the exception
} finally {
    if(null != session) session.close( );
}

```

Earlier, we established that our persistent class IDs were going to be managed by the database, which means we don't have to set our new `Professor`'s ID property. Hibernate will recognize that the ID property is in an uninitialized state and fill in the value for us during the save procedure. After the successful execution of the save, the value is populated.

We also know that our `Professor` can have a collection of email addresses. Adding them is simple:

```

Professor prof = new Professor( );
prof.setFirstName("Jim");
prof.setLastName("Jacobs");
prof.getEmails.add("jjacobs@directional.edu");
prof.getEmails.add("jim_j@other.obscure.journal.com");
// Then save as before

```

The `emails` table will now have two new entries, each with a foreign key relationship back to the new record of the `person` table where our new `Professor` resides.

We also know that the `Professor` can teach one or more `UniversityClasses`. We can create them all at once using a single `Session`:

```

Professor prof = new Professor( );
prof.setClasses(new HashSet( )); // empty initial collection
// Set prof properties ...
UniversityClass english = new UniversityClass( );
english.setName("English 101");
UniversityClass lit = new UniversityClass( );
lit.setName("Literature 105");
session.save(english);
session.save(lit);
prof.getClasses.add(english);
prof.getClasses.add(lit);
session.save(prof);

```

In this example, we could have called `session.save()` on the three new objects in any order we desired; it is easier to keep track of them if you call `save` on the children first, then the parent.

20.4.3.3. Updating existing objects

The simplest scenario for managing updates is to perform modifications to a loaded instance while the loading `Session` is still open. Modifications will be made to the underlying record when the session is flushed or the transaction is committed (we'll discuss the details later in this chapter). There is no need to tell Hibernate to perform the update; it will notice if the state of the persistent object has changed and push the update automatically.

```
Session session = null;
try {
    session = factory.openSession( );
    Professor prof = (Professor)session.load(Professor.class, new Long(3));
    // Just got her doctorate
    prof.setDegree("PhD");
    session.flush( );
} catch (Exception ex) {
    // Handle exception
} finally {
    if(null != session) session.close( );
}
```

20.4.3.4. Retrieving multiple objects

Sometimes, loading a single object at a time is not sufficient. You need to load an entire collection of objects, whether that is the entirety of a table or just a selected portion. The `Session` object exposes a set of methods that allows you to execute a query against the database (for more details, see "[Query](#)" and "[HQL \(Hibernate Query Language\)](#)," later in this chapter). In its simplest format, the query API allows you to specify a persistent class and some criteria for loading from its table(s). Using this query, you can retrieve a `List` of the results, an iterator to the list, or a scrollable result set.

```
List professors = session.createQuery("from Professor").list( );
Iterator studentIterator = session.createQuery("from Student").iterate( );
```

What if you wanted to find a filtered set of results? For example, all `Professors` with a PhD? You could iterate through the list of all `Professor` objects, query the property using `getDegree()`, and just keep the matches, or you could use the `createQuery()` method.

```
List professors =
    session.createQuery("from Professor where degree = 'PhD'").list( );
```

This is much faster than iterating over the objects in Java because the database is optimized to provide this kind of filtering, the number of rows returned is minimized, and the number of objects created is likewise kept to a minimum. For more details about the kinds of queries you can execute, see "[Query](#) and "[HQL \(Hibernate Query Language\)](#)," later in this chapter.

20.4.3.5. Connection management

While you are using a `Session`, you can perform three distinct operations to cause the state of the `Session` to be written to the database. They are:

- `Session.flush()`
- `Session.connection().commit()`
- `Session.close()`

Flushing the `Session` causes the state of the objects associated with the `Session` to be synchronized with the state of the database. This is critical to understand: saves, updates, and deletes do not get persisted to the data store until the `Session` is flushed. Flushing can occur multiple times during the lifetime of a single `Session`. Updates between flushes are cached in the `Session` and executed as a batch when a flush is triggered (either explicitly, by calling `Session.flush()`, or implicitly, as part of another call, like `Session.close()`).

Using a previous example, the three saves are all cached until the flush happens at the end of the method:

```
Professor prof = new Professor( );
prof.setClasses(new HashSet( )); // empty initial collection
// Set prof properties ...
UniversityClass english = new UniversityClass( );
english.setName("English 101");
UniversityClass lit = new UniversityClass( );
lit.setName("Literature 105");
session.save(english);
session.save(lit);
prof.getClasses.add(english);
prof.getClasses.add(lit);
session.save(prof);
session.flush( );
```

Conversely, several calls to `Session.flush()` could be interspersed throughout the method to keep the state of the `Session` synchronized with the database during long-running methods.

If you attempt to flush a `Session` and the underlying state of the data has changed, say because another user has modified the record you are working with after you originally loaded it, Hibernate will throw a `StaleStateException`. In order to recover from this, you must refresh the state of the object you were persisting by calling `Session.refresh(staleobject)`. This will update the object in memory to match the current values in the database. Then, you must reapply your changes and attempt the flush again.

When you are through using the `Session`, it is imperative that you close it. That's why, in previous code examples, the `Session.close()` appears in a finally block, ensuring that it gets called. Closing the `Session` not only released the JDBC connection back to the pool but also released references to persistent objects that were associated with the session, allowing the garbage collector to collect them (as long as your application code has released its references as well). Any call to `Session.close()` automatically invokes `Session.flush()` as well.

If you would like to be able to release connections to the connection pool but not terminate the `Session` (and therefore orphan all the persistent objects associated with the session), you can use the lighter-weight `Session.disconnect()/Session.reconnect()` pair of methods. Disconnect simply drops the JDBC connection back into the pool. Persistent objects are still associated with the `Session`, and you can still manipulate their properties, but nothing will be flushed back to the database. Calling `Session.flush()` while the session is disconnected will result in a `HibernateException`. Likewise, manipulating lazy associations that haven't been initialized while disconnected will raise a `LazyInitializationException`. To get a new JDBC connection from the pool, simply call `Session.reconnect` and all your functionality will be returned.

20.4.3.6. Disconnected objects

It is perfectly acceptable to load objects from the database and then close the `Session` that was used to load them. Objects in this state are considered disconnected. You may pass them around, access and modify their properties, and call methods on them in this state. The only exception is that you may not access lazily loaded associations that have not been initialized yet since there is no open `Session` to use to retrieve the lazy data.

If you need to then persist the changes to the disconnected object, you can associate it with a new `Session`. You have several choices for doing so. First, if the state of the object is unchanged since the original `Session` was closed, you can use the `Session.lock()` method to associate the object with the new session. You must specify the lock mode you wish to use (none, read, upgrade, or write):

```
Session session1 = factory.openSession();
Professor prof = (Professor)session.load(Professor.class, new Long(2));
session1.close();
// Do other work
Session session2 = factory.openSession();
session2.lock(prof, LockMode.WRITE);
prof.setDegree("PhD");
session2.close(); // Causes a flush
```

If the state of the object has changed, or has the possibility of having changed, since the original `Session` was closed, but the new `Session` has not loaded its own version of that instance, you can use

the `Session.update()` method to associate the object with the new `Session`. If that `Session` turns out to have its own loaded copy of that instance, Hibernate will throw a `HibernateException` explaining that there is an identity conflict in the new `Session`.

```
Session session1 = factory.openSession( );
Professor prof = (Professor)session.load(Professor.class, new Long(2));
session1.close( );
// Do other work
Session session2 = factory.openSession( );
session2.update(prof);
prof.setDegree("PhD");
session2.close( ); // Causes a flush
```

Finally, if the new `Session` is known to have its own version of the disconnected instance, you can use `Session.merge` to merge the values from the disconnected copy into the new `Session`'s connected copy.

20.4.4. Query

Hibernate's `Query` class gives you an object-oriented way to manage repeatable, parameterized queries to the database. For full details about what is possible using HQL, see the HQL section later in this chapter. To create a `Query`, use the `Session.createQuery()` method we used earlier:

```
Query getprofs = session.createQuery("from Professor");
```

While such a query might be useful (it can be cached and executed multiple times), it does not provide much benefit over directly executing it. What would be more useful is if the query required one or more parameters. Creating a cached, parameterized query is convenient programmatically (it is easy to set the parameters and reexecute the query), more efficient since the statement has been prepared already, and more secure, because parameter values are never treated as potentially executable code (thus they mitigate the risk of SQL injection attacks).

```
Query getprofsbydegree =
    session.createQuery("from Professor where degree = :degree");
getprofsbydegree.setString("degree", "PhD");
List results = getprofsbydegree.list( );
```

The named-parameter strategy is the most convenient, as you can specify the parameter values in any order. If you prefer, you can use standard JDBC-style numbered parameters, but bear in mind that the first parameter in a Hibernate query is parameter 0, not 1.

20.4.5. Interceptors

It seems these days that if a framework doesn't offer some kind of aspect-oriented programming technique, then it can't be considered "modern" or "complete." Therefore, Hibernate offers the `Interceptor` interface and a strategy for connecting the interceptor to a session. Bear in mind, interceptors are configured either per-session or once globally, and each session can have only a single interceptor associated with it. Writing good, general interceptors can then become an exercise in cramming a lot of code into a single class.

The interceptor interface is large and complex. It allows you to trap a variety of different events in the `Session`: for example, `onLoad`, `onDelete`, `onFlushDirty`, and `beforeTransactionCompletion`. The interceptor gets the first crack at these events; if you need to cancel the rest of the event from the interceptor, you have to throw a `CallbackException`, which will unwind the stack before the `Session` can complete the action.

Let's just look at the `onSave()` method of the interface:

```
public boolean onSave(Object o, Serializable serializable,
                    Object[] objects, String[] strings,
                    Type[] types) throws CallbackException {
    return false;
}
```

The method returns a `boolean` value, which is to notify Hibernate if you have modified any of the values passed into the method (the object or any of its properties). The parameters themselves seem complex but are in fact quite straightforward:

`Object o`

The persistent object being operated on

`Serializable serializable`

The ID of the object

`Object[] objects`

An array of (only) the persistent property values of the object

`String[] strings`

An array of (only) the persistent property names of the object

`Type[] types`

An array of (only) the persistent property types of the object

Interceptors are often used for security or logging purposes. A common security interceptor looks like this:

```
public boolean onSave(Object o, Serializable serializable,
    Object[] objects, String[] strings, Type[] types)
    throws CallbackException {

    // A custom Authorize class that implements
    // your business security rules
    if(!Authorize.isAuthorized(o))
        throw new CallbackException
            ("Current user not authorized to save this object.");
    return false;
}
```

To associate the interceptor with a specific `Session`, you specify it during the call to `SessionFactory.openSession()`:

```
Session session = factory.openSession(new SecurityInterceptor( ));
```

Conversely, you can associate it with every `Session` by setting it on the `Configuration` object:

```
Configuration cfg = new Configuration( ).configure( );
cfg.setInterceptor(new SecurityInterceptor( ));
```

20.4.6. Events

Brand new in Hibernate 3.0, `Events` replace the old `Lifecycle` interface for providing an alternate event-listening strategy. The `Lifecycle` interface was implemented by your domain objects to provide event callbacks on individual persistent instances. This made your domain objects decidedly nontransparent; they had direct references to the Hibernate libraries and were now cluttered with extraneous, non-domain-related code.

Instead, Hibernate now supplies a series of interfaces, one for each event published by the `Session`. To create a listener, you write a class that implements the chosen event interface or derives from one of the supplied default implementations of those interfaces. Then, you configure Hibernate to recognize your event.

Each event interface provides a single method that you implement to handle the inbound call. The methods all have slightly different signatures based on the activity they represent, but they all start with a parameter called `event` that allows you access to the entity being acted on and any context data associated with the call. [Example 20-14](#) shows our security interceptor, rewritten as a `SaveOrUpdate` event listener.

Example 20-14. SecureUpdateListener.java

```
public class SecureUpdateListener
    extends org.hibernate.event.DefaultUpdateEventListener {
    public Serializable onSaveOrUpdate(SaveOrUpdateEvent event)
        throws HibernateException {
        if(!Authorize.isAuthorized(event.getEntity( )))
            throw new HibernateException("Not authorized.");
        return super.onSaveOrUpdate(event);
    }
}
```

To configure Hibernate to utilize the listener, you can set it programmatically:

```
Configuration cfg = new Configuration( ).configure( );
cfg.setListener("saveOrUpdate", new SecureUpdateListener( ));
```

Or declaratively (say, in *hibernate.cfg.xml*):

```
<hibernate-configuration>
  <session-factory>
    ...
    <listener type="saveOrUpdate"
      class="SecureUpdateListener"/>
  </session-factory>
</hibernate-configuration>
```

You can use the same class to listen for multiple events. Just implement all the appropriate interfaces on that class, implement the methods on those interfaces, and use the same class name for each event that you are listening on. Hibernate creates a singleton instance of each listener to use, so the same instance will be used across the different event calls. Bear in mind that, as with interceptors, you can configure only a single listener per event type in your application. Setting the listener again replaces the original listener with the new one and does not append it to the call chain.

20.5. HQL (Hibernate Query Language)

Database applications, no matter how object-oriented, need to have direct access to the underlying database from time to time. We've already seen some reasons why (filtering data by parameterized query). The need for queries breaks down into the following categories:

- Retrieve a collection of objects from the database all at once
- Retrieve a sorted collection of objects from the database all at once
- Retrieve nonobject values from the database
- Retrieve summary information from the database

All four of these categories involve either collecting persistent objects or avoiding them altogether, making the direct object/relational mapping strategy an obstacle.

Hibernate provides Hibernate Query Language (HQL) and the Query interface to allow you to execute statements that meet these goals. Hibernate Query Language looks an awful lot like SQL. HQL queries can have a **SELECT** clause, a **FROM** clause, and a **WHERE** clause. You can, if you choose, author direct SQL statements as HQL queries; as long as the database can perform the query, the query will execute. On the other hand, you can use object-oriented notation instead of data-oriented notation in your HQL queries. This means, instead of using table and field names, you can use class and property names, or any mix of the two.

20.5.1. FROM

The **FROM** clause of an HQL query tells Hibernate where to look for the requested values. You can use either the name of one or more persistent classes, the name of one or more tables in the database, or any combination thereof. When all you want is for Hibernate to return one or more first-class persistent objects, you can issue an HQL query consisting of nothing but the **FROM** clause:

```
List professors = session.createQuery("FROM Professor").list( );
```

Hibernate doesn't require a **SELECT** clause because, by default, results will include every field in the table that is needed to create a `Professor` instance. Without a **WHERE** clause, the query returns every row in the table.

20.5.2. SELECT

You can use the `SELECT` clause to provide aliases to the first-class objects being returned or to return values other than first-class persistent types. For example, if you wanted to return just the last names of all professors, you could use the following query:

```
List profLastNames =
    session.createQuery("SELECT lastName from Professor").list( );
```

The result is a `List` of `String` values, one for each `Professor`, containing just the last-name field. Asking for multiple values returns a list of arrays containing the values in order:

```
Iterator profNames =
    session.createQuery("SELECT firstName, lastName from Professor").list( ).
    iterator( );
while(profNames.hasNext( )) {
    Object[] names = profNames.next( );
    // firstname == names[0]
    // lastname == names[1]
}
```

20.5.3. WHERE

The `WHERE` clause can contain any specific kind of filtering or grouping criteria that your database vendor supports. If you use a database that allows subqueries, HQL will happily allow you to use subqueries. Standard equality, comparison, and join notations are all available. If you attempt to use a SQL feature that is unavailable on your database platform, Hibernate will throw a SQL exception when it tries to execute the command on the database.

20.6. Hibernate Services

Hibernate provides a variety of services to you, whether through native Hibernate libraries or third-party libraries that you can download along with the core framework. For the purposes of this chapter, we'll examine just three: transaction management, caching support, and security. Many others are available, and you can learn about them in Hibernate's excellent online documentation.

20.6.1. Transactions

Most database applications require transactions in order to provide some level of assurance around updates to the tables. Hibernate provides a fully ACID-compliant local transaction manager for use with applications talking to a single database.^[2] You can make use of the transaction provider with minimal effort, but it is often useful to be explicit about your use of transactions in your data code. This means making use of Hibernate's `Transaction` object.

^[2] For more information on transactions, see [Chapter 16](#).

When beginning transactional work against a single database, you can explicitly begin a transaction within an open `Session`:

```
Session session = factory.openSession( );
Transaction tx = session.beginTransaction( );
```

Remember that the `Session` object does not actually grab a JDBC connection from the pool until you make a call that requires data access. The call to `beginTransaction` forces the `Session` to retrieve a connection because a transaction is meaningless without a connection as context.

Once a transaction has been established, you can do your data work. Any usage of the `Session` to access the database between the call to `beginTransaction` and a call to `transaction.commit()` will be considered part of the local transaction; there is no need to associate each specific data operation with the transaction. When you have finished all the transactional work, call `transaction.commit()`. If exceptions are thrown during the course of the method, you must explicitly roll back the transaction, close the `Session`, and handle your errors. The overall pattern looks like this:

```
Session session = null;
Transaction tx = null;
try {
    session = factory.openSession( );
    tx = session.beginTransaction( );
    Professor p = session.load(Professor.class, new Long(1));
    UniversityClass class = new UniversityClass( );
    class.setName("English 333");
```

```

    p.getClasses.add(class);
    Student s = session.load(Student.class, new Long(4));
    s.setSss("111-111-1111");
    tx.commit( );
} catch (Exception ex) {
    if(null != tx) tx.rollback( );
    // report error, etc.
} finally {
    if(null != session) session.close( );
}

```

Local atomic transactions like this are scoped to a single connection. You should always be explicit about manually committing your transactions; never assume that calling `Session.close()` will result in the appropriate behavior, because whether or not a transaction closed in this way commits or rolls back is a function of your database and your connection provider.

Because the transaction is scoped to a single connection, we know that it cannot survive a call to `Session.close()`. Likewise, transactions do not survive calls to `Session.disconnect()`, either. Since `disconnect` drops the physical connection back into the pool and the connection is the containing context of a transaction, disconnecting the `Session` has the exact same consequences as closing the session out from under the transaction. Once again, you should explicitly commit or roll back the transaction before calling either `disconnect` or `close`.

The explicit transaction strategy is great for local atomic transactions, but useless in the face of a distributed transaction for several reasons:

- A distributed transaction, by definition, spans multiple databases. A single `Session` object (wrapping a JDBC connection) connects only to a single data source. The explicit `transaction` object lives within a `Session`. Therefore, it cannot be used to talk to more than one database.
- Distributed, managed transactions do not like it if you attempt to commit them prematurely. The container and the JTA decide when a transaction is to be committed, and calling `tx.commit()` would result in an exception being thrown.
- Likewise, distributed transactions are not happy when explicitly rolled back. Again, the container and the JTA manage these decisions.

To use the JTA to take part in a distributed, managed transaction, you have to first replace the Hibernate local transaction manager with the JTA. This is accomplished by adding these configuration settings to *hibernate.properties*:

```

hibernate.connection.datasource = java:/your/datasource
hibernate.transaction.factory_class =
    org.hibernate.transaction.JTATransactionFactory
hibernate.transaction.manager_lookup_class =
    org.hibernate.transaction.<Vendor>TransactionManagerLookup

```

In the preceding snippet, you would use the transaction lookup manager appropriate to your container by replacing `<Vendor>` with JBoss, WebLogic, or one of the other manager implementations provided with Hibernate.

Once the configuration is set, Hibernate relies on the container to manage transactions. Whenever Hibernate is called, it attempts to associate with the current global JTA transaction in effect. If there isn't one, it creates one for the length of the current call. As long as the container is around, your application can safely ignore all Transaction API calls. The JTA transaction either already exists or is created directly by Hibernate, so you do not have to call `Session.beginTransaction()`. The `transaction` is committed by the JTA and container based entirely upon rules in the deployment descriptor, so you never have to call `transaction.commit()`. To force a rollback of the transaction, you simply raise an exception that makes it back to the container. The container will roll back the transaction automatically, so you don't have to call `tx.rollback()`. All you have to do is write code that accesses the database.

20.6.2. Caching

Beware of premature optimization. Too many times, development teams implement performance improvement strategies before ever measuring actual performance, or even finding out what an acceptable performance range might look like. If it weren't for the fact that most performance gains come at the expense of other, equally important concerns (simplicity, ease of maintenance, ease of testing), it wouldn't be a big problem.

Caching is an optimization technique; the idea behind caching is to prevent unnecessary trips to the database. If data is unlikely to change much, or is only read but never written or written but not read, you can save time and processing power by short-circuiting the round-trip before it gets started. Since database applications are almost always performance-bound to the network and I/O costs of hitting the database server, this is an appropriate first stop on the optimization path.

Hibernate provides two main kinds of caching: a `Query` cache and the more general second-level cache. The `Query` cache is built into the Hibernate framework, while the second-level cache is implemented in a third-party library, and there are several choices of implementation. Both strategies should be used carefully.

The problem with caching is that the application has to maintain a duplicate, local version of data in the central data store. Reads (or writes) happen to the local cache, which prevents the round-trip and the overhead of invoking the database APIs. The downside is that managing the lifecycle of the cache is complicated; what happens when the database changes? How should the cache be updated? What should happen if changes to the cache can't be pushed back onto the database? The extra layer of indirection is useful, but comes at the price of complexity.

20.6.2.1. The second-level cache

It turns out that, all along now, Hibernate has been busily caching data for you, without your knowledge. The `Session` itself is a cache of data. The state of each object, during the course of a transaction, is maintained in the `Session`. This makes sense. When you load an object in a `Session`, you wouldn't want to make a round-trip back to the database each time you access a property on the object. The `Session` caches those values for you. When the transaction is committed and the `Session`

is closed, the cache is destroyed.

Sometimes you need a cache that spans the transaction boundary. When accessing data that is either infrequently or never updated, you may find it faster to load the data once, place it in the local cache, and access it from there for future needs. You'll want that data to stick around between `Sessions` and be accessible from any `Session`. Because of these requirements, you need something beyond the `Session` class itself to manage the cache.

The second-level cache is implemented in the EHCACHE library. You can replace it with any library that implements the Hibernate interface for cache management. Currently, appropriate libraries are EHCACHE, OSCACHE, SWARMCACHE, the JBoss TreeCache, Java Caching System or JCS (though it is deprecated now and will disappear altogether in future versions), and even a plain old hashtable, though this is frowned upon for anything but simple testing.

EHCACHE allows you to use an in-memory or disk-based cache. When you configure it, you can establish your own rules for what objects are added to the cache and when they are removed from it. The in-memory store uses the `LinkedHashMap` for actual storage if you are using JDK 1.4 or 5.0; otherwise, it uses the Apache Commons `LRUMap`. The disk storage version uses several well-known locations for the data to be persisted.

To enable EHCACHE, you have to set the global cache property, normally in the global configuration file, *hibernate.properties*:

```
hibernate.cache.provider_class = net.sf.ehcache.hibernate.Provider
```

After that, it's just a matter of configuring which objects, queries, and collections you want to cache. This sounds simple; remember, second-level caches are not notified by the database when changes have been made to the underlying store. The caches need to be expired intelligently if there are likely to be concurrency conflicts. Caches should be used for data that is seldom or never written.

To enable the cache for a given persistent class, add the appropriate caching directive to the mapping file. For example, you might imagine that the classes in the various departments use the same textbooks year in and year out (the pace of change in textbooks is glacially slow, after all), and this data could be considered relatively static. The `UniversityClass` would be extended to include a new `textbooks` property:

```
private Set textbooks;  
// Accessor methods elided
```

And a new class, `TextBook`, would be added:

```
public class TextBook {  
    private String title;  
    private String author;  
    private String isbn;
```

```

    // Accessor methods elided
}

```

The mapping for `TextBook` would mark it as a cacheable class; since the data is unlikely to change much, we could have a generous expiration limit on the cache and prevent lots of round-trips to the database:

```

<hibernate-mapping package="com.oreilly.jent.hibernate">
  <class name="TextBook">
    <cache usage="read-only"/>

    <!-- rest of mapping elided -->
  </class>
</hibernate-mapping>

```

When specifying the caching strategy, you can use `read-only`, which means that the underlying data has no chance of being written to while the cache is in effect. Other strategies are `read-write`, for data that may change out from underneath the cache, and `nonstrict-read-write`.

The `read-write` and `read-only` cache strategies both use synchronized access to the cache, meaning that the cache is thread safe and cache consumers are guaranteed to get the most recent version of an object available in the cache. `nonstrict-read-write` is still threadsafe but does not guarantee that each consumer will get the most recent version of any given object. `nonstrict-read-write` is the most efficient cache strategy, but it requires more thought and planning for the developer who must anticipate retrieving stale objects from the cache.

20.6.2.2. The Query cache

The Query cache utilizes the second-level cache. Its job is to retain the results of a frequently run query. The full results are not cached, however. The Query cache maintains two separate physical caches: one for the `ids` of the returned results, and one for the timestamp of each `id`. When queries whose results have been cached are executed again, Hibernate retrieves the `id` cache and looks up the values in the second-level cache. This prevents wasted resources by preventing double storage of items, but it also means that you cannot just implement the Query cache without the second-level cache being enabled.

20.6.3. Security

Hibernate 3.0 introduces a new declarative security model for securing access to your persistent classes. Based on the Java Authorization Contract for Containers (JACC) and the Java Authentication and Authorization System (JAAS),^[3] Hibernate now allows you to map user roles to actions you can perform on persistent classes:

^[3] For more information on JACC and JAAS, see [Chapter 10](#).

```
<listener type="pre-delete"
  class="org.hibernate.secure.JACCPreDeleteEventListener"/>
<listener type="pre-update"
  class="org.hibernate.secure.JACCPreUpdateEventListener"/>
<listener type="pre-insert"
  class="org.hibernate.secure.JACCPreInsertEventListener"/>
<listener type="pre-load"
  class="org.hibernate.secure.JACCPreLoadEventListener"/>
```

Remember, once these are installed, you can't add any more listeners for these events. Now all that remains is to configure your access permissions. Permissions are established globally in the configuration file.

```
<grant role="depthhead" entity-name="Professor" actions="*" />
<grant role="professor" entity-name="Student" actions="read" />
```

In order for all this to work, your application must already support a JAAS login, and the roles you use in the configuration must map to the roles provisioned by the JAAS login module.

[← PREV](#)

20.7. Conclusion

Hibernate has changed the landscape for persistence management. Its popularity, and the popularity of similar ORM packages for Java, has all but eliminated the idea that developers should manage JDBC on their own. Likewise, Hibernate has shown that a high-performance, robust persistence framework can remain lightweight and transparent to your domain model. Hibernate has had such an influence on the persistence landscape that the next version of the EJB specification is going to look remarkably like what you've seen here today. Learning Hibernate isn't just a good idea now it is a form of future proofing as well.

[← PREV](#)

← PREV

Chapter 21. Annotations with XDoclet and J2SE Metadata

Code annotations can simplify enterprise development tasks. They allow you to place metadata about your code directly into your code and then use this metadata in various ways, such as automatically generating artifacts related to the code (documentation, deployment descriptors, and configuration files), running reports of various kinds over the code, and even generating supplementary code.

There are currently a number of annotation-related tools and libraries available through open source projects as well as tools available for specific Java IDEs and application server environments. In addition, a standard code metadata facility was added to the language as of J2SE 5.0.

← PREV

[← PREV](#)

21.1. What's Covered Here?

In this chapter, we will be discussing two complementary annotation systems: XDoclet and J2SE annotations. XDoclet is arguably the most popular open source annotation tool in the Java community. We'll examine various ways that you can use the annotations available in XDoclet to make enterprise development tasks easier. The examples shown in this chapter were written using XDoclet 1.2.3.

We also provide an overview of the J2SE 5.0 annotation facility, which is now part of the Java language and is the basis for various proposed annotation extensions to key Java enterprise APIs like EJB and JAX-RPC. The annotation facility built into J2SE is based on the JSR-175 specification. The J2SE annotation examples shown in this chapter were developed using J2SE 5.0 update 2.

[← PREV](#)

21.2. What Are Code Annotations?

If you write Java code and document your code (one doesn't necessarily imply the other, sadly), you are already familiar with a particular kind of code annotation: Javadoc tags. With Javadoc, you put metadata about the code (like the author, references to other related classes, and descriptions of the meanings of methods) into the code itself, in the form of annotations. You then run a tool (in this case, the Javadoc generator) over the code to generate HTML documentation from these annotations.

Generalized annotation schemes follow this same basic model. The difference is that you can do much more than just generate documentation (as useful as that is). Code annotations can be used to generate deployment descriptors, configuration files, and even Java code, as shown in [Figure 21-1](#).

Figure 21-1. General model for code annotations

As depicted in the figure, annotations are inserted as markup directly in your code. As an example, here's an XDoclet annotation specifying some deployment settings for a web component:

```
/**
 * @web.servlet
 *     name="ShoppingCart"
 *     display-name="The Shopping Cart Servlet"
 */
public class ShoppingCartServlet extends HttpServlet {
    . . .
}
```

When you're dealing with enterprise development scenarios, annotating your code in a structured way can benefit you in a number of ways:

Reduce development effort

An annotation system allows you to mark up your code directly and then use this markup information to automatically generate all the trappings required for the context at hand. A deployment descriptor, for example, can be generated using some basic settings provided in a set of code annotations. The annotation processor can generate the proper XML structure and insert default values for any unspecified but required deployment settings.

Simplify code management

Annotations are embedded in the code, which makes tracking, updating, and finding key metadata about the code much easier. Similar to the principle of Javadoc annotations for documentation, keeping the metadata with the code makes it much easier to keep the information in line with the current state of the code and vice versa.

Make your code more flexible

If you use annotations consistently across your project code, they can be a significant aid when it comes time to refactor your code in certain ways. If, for example, you consistently generate deployment descriptors using annotations, you can very easily deploy your code to various types and versions of J2EE application servers simply by altering the parameters used by the annotation processor.

[← PREV](#)

21.3. Annotation Tools

A number of annotation facilities are available for the Java environment. We already mentioned Javadoc, which is built into Java and is aimed at autogeneration of HTML documentation for Java code. Javadoc is somewhat extensible in itself. You can write your own custom Javadoc processor to take code with Javadoc comments and generate your own documentation formats. You can also define your own custom Javadoc tags and handle them in your custom Javadoc handlers.

Javadoc is, however, aimed principally at documentation-oriented code annotations and is used almost exclusively for that purpose. Since this book is about enterprise application development, we discuss XDoclet and J2SE annotations that are particularly relevant to enterprise contexts. XDoclet includes a wide array of predefined annotations for enterprise development, and J2SE annotations are being used to define standard annotations in the next versions of key enterprise Java APIs.

[← PREV](#)

21.4. XDoclet Tutorial

XDoclet is a toolset built on top of the Javadoc annotation mechanism. The XDoclet toolset is an open source project, currently managed on SourceForge. Essentially, XDoclet provides a suite of predefined custom Javadoc tags designed for use in various development scenarios, such as developing web and EJB components, implementing web services, integrating Hibernate persistence management, deploying components to specific application servers, and so on. Annotating your code is done using these Javadoc tags within code comments. XDoclet includes a set of Ant tasks that are used in your buildfiles to process the annotations and generate the appropriate outputs.

21.4.1. Annotating Your Code

XDoclet annotations can be applied to specific entities in your Java code:

- Classes or interfaces
- Methods and constructors
- Member variables

These annotations look a lot like standard Javadoc comments. The general syntax for XDoclet annotations is:

```
@tag-group.tag-name param1="value1" param2="value2"
```

The tags are organized into groups, indicated by the `tag-group` label, and each tag has a unique name within its group, indicated by the `tag-name`. A tag can optionally accept one or more parameters, which are provided in the annotation using the syntax shown here. You can also put parameters on lines following the tag line. The XDoclet processor will consider all subsequent parameter lines to belong to the initial tag you specify. So the previous example could also be put into your code as:

```
@tag-group.tag-name  
    param1="value1"  
    param2="value2"
```

XDoclet annotations always reside within a Javadoc comment, and the location of the comment (class, method, or member variable) determines the level of the annotation. Here is a class-level

XDoclet annotation:

```
/**
 * Class: MyNeatServlet
 * @author Jim Farley
 *
 * @web.servlet
 *     name="neat-servlet"
 */
public class MyNeatServlet extends HttpServlet {
    . . .
}
```

In this case, the annotation tag group is `web`, and the tag name is `servlet`. The annotation has a single parameter, `name`, with a value of `neat-servlet`. The `web.servlet` annotation applies at the class level because it sits within a class-level Javadoc comment.

A method annotation would look like this:

```
public class MyDataObject {
    . . .
    /**
     * Description property
     *
     * @hibernate.property
     *     column="DESC"
     */
    public String getDescription( ) { . . . }
    . . .
}
```

The `hibernate.property` annotation sits within a Javadoc comment for the `getDescription()` method, so it applies to that method.

A member-level (also called a field-level) annotation is not as common in XDoclet but still useful in some situations. A member annotation sits in a member-level Javadoc comment, as shown in this example:

```
public class MyEJB extends EntityBean {
    . . .
    /**
     * @ejb.resource-ref res-name="jdbc/MyDBPool"
     */
    protected DataSource mDBPool = null;
    . . .
}
```

21.4.2. Processing XDoclet Annotations

The only way to process XDoclet annotations right now is to use the provided XDoclet Ant tasks in an Ant buildfile. Each XDoclet Ant task is intended for a particular development context and processes a targeted subset of the available annotations. The `webdoclet` Ant task, for example, is aimed at developing various types of web components (servlets, JSPs, custom JSP tags) and processes annotations from the `web` group and a few related annotation groups. The `ejbdoclet` task, on the other hand, is for EJB component development and can process annotations in the `ejb` and related annotation groups.

More complete details on the general topic of writing Ant buildfiles can be found in [Chapter 17](#). Here, we'll just look at the basics of how to import the XDoclet tasks into your buildfile. In the following sections, we'll take a look at a few key scenarios to demonstrate how the XDoclet Ant tasks are used to process some of the commonly used annotations.

Adding the XDoclet tasks to your buildfile is the same as adding any other custom Ant tasks. You use the `taskdef` task to define a new task that is implemented by classes in the XDoclet libraries. The dependent classes for the task are specified using a nested `<classpath>` element. In [Example 21-1](#), we "import" the `webdoclet` task from the XDoclet libraries.

Example 21-1. Importing the webdoclet task into an Ant buildfile

```
<property name="xdoclet.home" value="/usr/local/xdoclet-1.2.3"/>
<!-- Include the XDoclet webdoclet Ant task -->
<taskdef name="webdoclet"
  classname="xdoclet.modules.web.WebDocletTask">
  <classpath>
    <path>
      <fileset dir="{xdoclet.home}/lib">
        <include name="*.jar"/>
      </fileset>
    </path>
    <path refid="java.compile.classpath"/>
  </classpath>
</taskdef>
```

We're using an `xdoclet.home` property to point to the root of the XDoclet installation, just as a convenience. In the `taskdef` task, we specify that we're defining a task named `webdoclet` and that it is implemented by the class `xdoclet.modules.web.WebDocletTask`. This class will be located in the `<classpath>` we define within the `taskdef`. In our `<classpath>`, we're including two sets of `<path>` elements. First, we're adding all of the XDoclet jars using a `fileset` that includes any file ending in `.jar` in the XDoclet `lib` directory. Following this, we add the `<classpath>` named `java.compile.classpath` (whose definition is not shown here). The `<classpath>` that's specified in the `taskdef` will be used whenever the task is used in the buildfile. In the case of the XDoclet `webdoclet`

task, the `<classpath>` has to include all of the web component libraries—the servlet and JSP libraries at a minimum, possibly others depending on the `webdoclet` options you use. The `java.compile.classpath` path used here must include the jars for the servlet and JSP packages, pulled from whatever application server we're using. Note that you can also set the classpath for the task at the point in the buildfile where you actually use the task. Setting the classpath here in the `taskdef` can be convenient when the imported tasks will be used several times in the buildfile with the same base classpath.

To add other XDoclet tasks to our buildfile, we would repeat these previous steps above, specifying different task names and implementation classes. The `ejbdoclet` task definition would be imported as shown in [Example 21-2](#). In this case, since the `ejbdoclet` task will need to have access to the EJB libraries (`javax.ejb`, etc.), we'll need to ensure that these classes are included in the `java.compile.classpath <path>` element.

Example 21-2. Importing the ejbdoclet task into an Ant buildfile

```
<!-- Include the XDoclet ejbdoclet Ant task -->
<taskdef name="ejbdoclet"
  classname="xdoclet.modules.ejb.EjbDocletTask">
  <classpath>
    <path>
      <fileset dir="{xdoclet.home}/lib">
        <include name="*.jar"/>
      </fileset>
    </path>
    <path refid="java.compile.classpath"/>
  </classpath>
</taskdef>
```

Once the tasks have been defined and imported into the buildfile, you can use them like other Ant tasks, nested within your build targets. In the following example, we use the `webdoclet` task inside of a `create-war` target to generate a `web.xml`/deployment descriptor from the source code before assembling the web archive (war) file:

```
<target name="create-war">
  <!-- Generate a web.xml file using annotations on the
  web components -->
  <webdoclet destDir="{web.dir}">
    <fileset dir="{src.dir}">
      <include="**/web/*Servlet.java"/>
    </fileset>
    <deploymentdescriptor
      servletspec="2.4"
      destinationFile="web.xml"/>
  </webdoclet>

  <!-- Assemble the classes, deployment descriptor, etc.
```

```

        into a web archive, using the Ant built-in war task -->
        <war warfile="myapp.war" webxml="${web.dir}/web.xml">
            . . .
        </war>
</target>

```

Aside from the general details of invoking the `webdoclet` Ant task, you should also note here that the `war` task that follows the `webdoclet` task is reading in the `web.xml`/deployment descriptor being generated by the `webdoclet` task. The deployment descriptor is being generated directly from annotations found in the servlet source code. This ability to extract key code artifacts directly from source code annotations is an important benefit provided by XDoclet and other annotation-driven facilities.

21.4.3. Generating Deployment Descriptors

We just saw an example of generating a web deployment descriptor using XDoclet, but now let's look in more detail at using XDoclet to assist in the generation of deployment descriptors for your J2EE components.

21.4.3.1. "Pure" annotation-driven deployment descriptors

The most straightforward approach is to have the entire deployment descriptor generated entirely from code annotations. In this case, the code annotations include all of the information to be placed in the deployment descriptor. If we wanted to generate a `web.xml`/deployment descriptor for our application's web components, for example, we would use annotations in the `web` group in XDoclet and place them as needed in the web component code.

As a simple example, suppose we had a `PeopleFinderServlet` that we wanted to deploy using a single URL mapping. First, we'd need to let XDoclet know that our class is a servlet, using the `web.servlet` annotation, and we would use the `web.servlet-mapping` annotation to indicate how to map the deployed web component to a URL. [Example 21-3](#) shows the annotated source code for the `PeopleFinderServlet` class.

Example 21-3. Annotated servlet code

```

/**
 * Class: PeopleFinderServlet
 *
 * @author <a href="mailto:jim@jimfarley.org">Jim Farley</a>
 *
 * XDoclet tags:
 *
 * @web.servlet
 *     name="PeopleFinderServlet"

```

```

*      display-name="PeopleFinder Servlet"
*
* @web.servlet-mapping
*      url-pattern="/pf"
*
*/
public class PeopleFinderServlet extends HttpServlet {
    // Servlet implementation details omitted
    . . .
}

```

In the `web.servlet` annotation, we're indicating some minimal information about the web component's component name and display name to be used in the deployment descriptor. In the `web.servlet-mapping` annotation, we indicate that we want the component mapped to the `/pf` URL within the application's overall web context.

We would process this annotated source in our Ant buildfile using the `webdoclet` task, as shown here:

```

<target name="create-war">
  <!-- Generate a web.xml file using annotations on the
       web components -->
  <webdoclet destDir="${web.dir}">
    <fileset dir="${src.dir}">
      <include="**/PeopleFinderServlet.java"/>
    </fileset>
    <deploymentdescriptor
      servletspec="2.4"
      destinationFile="web.xml"/>
  </webdoclet>
  . . .
</target>

```

This example is very similar to our previous Ant example, except that we're specifically processing the `PeopleFinderServlet` source file, rather than all source files whose class names end in `Servlet`. We are trying to generate only a `web.xml` deployment descriptor, so we use the `<deploymentdescriptor>` child element of the `webdoclet` task. In the `<deploymentdescriptor>` element, we use the `servletspec` attribute to indicate the version of the servlet specification that our application server supports. This will influence the DTD/schema that XDoclet references in the generated deployment descriptor and in some cases the structure of the generated XML itself (though not in this simple example). The `destinationFile` attribute indicates where we want the deployment descriptor file deposited.

If we ran this task on the source code shown in [Example 21-3](#), XDoclet would generate the deployment descriptor shown in [Example 21-4](#). This is a complete and valid deployment descriptor that we could use to deploy the `PeopleFinderServlet` to a Servlet 2.4 web container.

Example 21-4. XDoclet-generated web deployment descriptor

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

<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
                             http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
         version="2.4">
  <distributable/>

  <!-- XDoclet comments omitted... -->

  <servlet>
    <display-name>PeopleFinder Servlet</display-name>
    <servlet-name>PeopleFinderServlet</servlet-name>
    <servlet-class>
      com.oreilly.jent.people.annotation.PeopleFinderServlet
    </servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>PeopleFinderServlet</servlet-name>
    <url-pattern>/pf</url-pattern>
  </servlet-mapping>

</web-app>
```

The generated *web.xml* file properly references the XML schema for the Servlet 2.4 deployment descriptor, and all of the deployment information that we placed in our code annotations has been placed into its proper locations in the *web.xml* file. And if we change the source code, we have the annotations right there in the code and we can keep them up-to-date as the code changes. The *web.xml* file is automatically regenerated each time we run the appropriate task in our Ant buildfile.

Many application servers require a supplementary deployment descriptor or other configuration file in order to specify server-specific deployment details (see [Chapter 2](#) for more information about deploying applications). XDoclet can generate these descriptors as well for a set of supported application servers. If we're using JBoss 4.0 to run our web components, for example, we can ask XDoclet to generate a JBoss web deployment descriptor by adding the `<jbosswebxml>` element to our `webdoclet` task:

```
<target name="create-war">
  <!-- Generate a web.xml file using annotations on the
       web components -->
  <webdoclet destDir="${web.dir}">
    <fileset dir="${src.dir}">
      <include="**/PeopleFinderServlet.java"/>
    </fileset>
    <deploymentdescriptor
```

```

        servletspec="2.4"
        destinationFile="web.xml"/>
    <jbosswebxml
        version="4.0"
        destinationFile="jboss-web.xml"/>
</webdoclet>
    . . .
</target>

```

In the `<jbosswebxml>` subelement, we're specifying that we want a deployment file for a JBoss 4.0 server and that we want the file saved to the file *jboss-web.xml*. If we ran this modified version of our `create-war` target on the source code in [Example 21-3](#), the result would be the following JBoss web deployment file:

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE jboss-web
    PUBLIC "-//JBoss//DTD Web Application 2.4//EN"
        "http://www.jboss.org/j2ee/dtd/jboss-web_4_0.dtd">
<jboss-web>
    <!-- XDoclet comments omitted -->
</jboss-web>

```

There's nothing interesting here because there are no details in our code annotations that need to be reflected in a JBoss deployment file. XDoclet has a set of server-specific annotations that can be used to generate these deployment files as well. If, for example, our web component required a resource reference to a `DataSource`, we could add annotations to our servlet class to indicate the resource reference needed in the *web.xml*/file as well as the entry to be placed in the JBoss web deployment file. The servlet code with the additional annotations is shown here:

```

/**
 * @web.servlet
 *     name="PeopleFinderServlet"
 *     display-name="PeopleFinder Servlet"
 *
 * @web.servlet-mapping
 *     url-pattern="/pf"
 *
 * @web.resource-ref
 *     name="jdbc/MyDataSource"
 *     type="javax.sql.DataSource"
 *     auth="Container"
 *
 * @jboss.resource-ref
 *     res-ref-name="jdbc/MyDataSource"
 *     jndi-name="jdbc/DataSourceX"
 *

```

```

*/
public class PeopleFinderServlet extends HttpServlet {
    . . .
}

```

The `web.resource-ref` annotation is used to specify the resource reference required by the web component. In this case, we're declaring that a `DataSource` is needed at `jdbc/MyDataSource` in the JNDI namespace. The `jboss.resource-ref` annotation indicates how this resource reference should be handled in the JBoss web deployment file. In this case, we're declaring that the `jdbc/MyDataSource` JNDI reference should be linked to the `jdbc/DataSourceX` JNDI resource on the JBoss server.

If we run our earlier Ant target again using these updated annotations, our `web.xml` file will now look like this:

```

<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
    http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
  version="2.4">
  <distributable/>
  <servlet>
    <display-name>PeopleFinder Servlet</display-name>
    <servlet-name>PeopleFinderServlet</servlet-name>
    <servlet-class>
      >com.oreilly.jent.annotation.web.PeopleFinderServlet
    </servlet-class>
  </servlet>

  <servlet-mapping>
    <servlet-name>PeopleFinderServlet</servlet-name>
    <url-pattern>/pf</url-pattern>
  </servlet-mapping>

  <resource-ref>
    <res-ref-name>jdbc/MyDataSource</res-ref-name>
    <res-type>javax.sql.DataSource</res-type>
    <res-auth>Container</res-auth>
  </resource-ref>
</web-app>

```

The new `resource-ref` entry in `web.xml` was generated because of the new `web.resource-ref` annotation we added to the source file. Similarly, the new `jboss.resource-ref` annotation we added to the source file will be reflected in the new JBoss web deployment file:

```

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

```

```

<!DOCTYPE jboss-web
  PUBLIC "-//JBoss//DTD Web Application 2.4//EN"
    "http://www.jboss.org/j2ee/dtd/jboss-web_4_0.dtd">
<jboss-web>
  <resource-ref>
    <res-ref-name>jdbc/MyDataSource</res-ref-name>
    <jndi-name>jdbc/DataSourceX</jndi-name>
  </resource-ref>
</jboss-web>

```

21.4.3.2. Mixed models: Deployment using annotations and external data

The preceding annotation examples work very well when we are deploying a set of components in a specific environment, where the deployment details do not vary. But suppose we handed this system over to another developer working in a different environment. The other environment might require different deployment details: modified URL mappings for the web components, different resource names on the application server, and perhaps even a different application server altogether with its own extended deployment files.

In a situation like this, it would be bad practice to have to change the source code of the application in order to deploy it. Even though you are modifying "only" annotations and those annotations are inside of comments that shouldn't affect compiled code behavior, some basic principles still apply: source code should encode the behavior of the system globally, and anything that needs to be localized to a particular environment should happen in externalized configuration files and build scripts that are distinctly separate from the source code.

Thankfully, XDoclet recognizes the need, in some cases, for having some deployment data reside in external sources (external to the source code, that is). It supports this practice in two ways. Deployment data sitting in external files can be merged with the deployment data generated from annotations, using the `mergeDir` options on the XDoclet Ant tasks. In addition, XDoclet annotations can use Ant property names to parameterize the values of their properties. This allows you to put deployment parameters into the Ant buildfile (or in separate properties files loaded into the Ant buildfile), and they will be automatically applied when the parameterized annotation is processed.

As an example of using the merged deployment descriptor approach, suppose we have a situation in which the URL mapping of particular web components needs to be set on a per-environment basis. Perhaps other web applications are being deployed in these environments and in some cases the default URL mappings are conflicting. We could alter our servlet example to remove the URL mapping annotation and instead use a separate XML file to provide the URL mappings for the web components. The `<deploymentdescriptor>` subelement on the `webdoclet` task supports the use of a `mergeDir` attribute. When annotated files are being processed for deployment-related annotations, the `webdoclet` processor will check this directory for specific XML files containing segments of the deployment descriptor to be merged together with the generated deployment data to produce the final `web.xml` file. In our case, we want to provide servlet mappings in an external file, and `webdoclet` expects these to be in a file named `servlet-mappings.xml`. So we would make a `servlet-mappings.xml` file in our `mergeDir` that looks like this:

```

<servlet-mapping>
  <servlet-name>PeopleFinderServlet</servlet-name>

```

```

    <url-mapping>/my-pf-context</url-mapping>
</servlet-mapping>

```

Assuming that this file sits in a directory called *config/xdoclet-merge* in our project area, then we would modify our call to the `webdoclet` task in our Ant file to look like this:

```

<webdoclet destDir="${web.dir}">
  <fileset dir="${src.dir}">
    <include name="**/PeopleFinderServlet.java"/>
  </fileset>
  <deploymentdescriptor
    servletspec="2.4"
    mergeDir="config/xdoclet-merge"
    destinationFile="web.xml"/>
</webdoclet>

```

If we ran this task on the annotated code shown in [Example 21-3](#), the resulting *web.xml* file would look like the following:

```

<?xml version="1.0" encoding="UTF-8"?>
<web-app
  xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
    http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
  version="2.4">
  <distributable/>
  <servlet>
    <display-name>PeopleFinder Servlet</display-name>
    <servlet-name>PeopleFinderServlet</servlet-name>
    <servlet-class
      >com.oreilly.jent.annotation.web.PeopleFinderServlet</servlet-class>
  </servlet>

  <servlet-mapping>
    <servlet-name>PeopleFinderServlet</servlet-name>
    <url-mapping>/my-pf-context</url-mapping>
  </servlet-mapping>

  <servlet-mapping>
    <servlet-name>PeopleFinderServlet</servlet-name>
    <url-pattern>/pf</url-pattern>
  </servlet-mapping>
</web-app>

```

The highlighted servlet mapping came from the *servlet-mappings.xml* file while the rest of the *web.xml* file came from processing the annotations in the source code. The `webdoclet` task supports a specific XML merge file for each section of a web deployment descriptor: servlets, filters, listeners, resource references, and so on. You can decide in your context what deployment information is truly universal and belongs in the source code and what needs to be localized to the environment and therefore placed in one of these externalized XML fragments.

XDoclet also allows you to use Ant properties to parameterize your annotations. This applies to any XDoclet annotations, not just the web-related ones we're showing here. In our case, rather than using an external, merged XML file to specify a localized servlet mapping, we could place the URL mapping value in our Ant buildfile, as the value of a property called `pf.servlet.context`:

```
<property name="pf.servlet.context" value="/pf-ant-param"/>
<webdoclet destDir="${generated.web.dir}">
  <fileset dir="${java.src.dir}">
    <include name="**/PeopleFinderServlet.java"/>
  </fileset>
  <deploymentdescriptor
    servletspec="2.4"
    destinationFile="web.xml"/>
</webdoclet>
```

We could then reference this Ant parameter in our code annotation:

```
/**
 * @web.servlet
 *   name="PeopleFinderServlet"
 *   display-name="PeopleFinder Servlet"
 *
 * @web.servlet-mapping
 *   url-pattern="${pf.servlet.context}"
 */
public class PeopleFinderServletParam extends HttpServlet {
    . . .
}
```

Our *web.xml* file will now be generated with the value of the `pf.servlet.context` Ant property injected. This value will be read at the time the XDoclet Ant task is run, so in our case the *web.xml* file will end up looking like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app
  xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```

    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
                        http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
    version="2.4">
<distributable/>
<servlet>
    <display-name>PeopleFinder Servlet</display-name>
    <servlet-name>PeopleFinderServlet</servlet-name>
    <servlet-class>PeopleFinderServlet</servlet-class>
</servlet>

<servlet-mapping>
    <servlet-name>PeopleFinderServlet</servlet-name>
    <url-pattern>/pf-param</url-pattern>
</servlet-mapping>
</web-app>

```

21.4.4. Generating Source Code

Generating documentation, deployment files, and general configuration files from code annotations can be a powerful tool, as we've just seen. But some code annotations provided with XDoclet actually generate other Java source code. We'll take a look at two specific examples: generating support interfaces for SOAP web services and generating client and home interfaces for EJB components.

21.4.4.1. Web services

As we discussed in our web services coverage in [Chapter 12](#), a SOAP service can be implemented in JAX-RPC using a simple RMI programming model. We won't review all the details of writing JAX-RPC web services here (refer to [Chapter 12](#) for those details). We'll just stick to the basics here to demonstrate how annotations can help with code generation.

With JAX-RPC, a `Remote` interface is used to declare the methods that will be mapped into operations on the SOAP service, and a concrete Java class provides the implementations of these methods. There is a direct relationship between the implementation methods and the declared methods on the `Remote` interface, so it can be useful to have the `Remote` interface automatically generated from the implementation class. Otherwise, each time you adjust the implementation class (change the signature of a web service method or add a new service method), you need to manually update the corresponding `Remote` interface to keep the two in sync.

XDoclet supports this with the `web` group of annotations in combination with the `service-endpoint` subtask of the `webdoclet` Ant task. [Example 21-5](#) shows a modified version of one of the JAX-RPC SOAP service implementations we looked at in [Chapter 12](#), `PeopleFinderServiceImpl`. Notice that we've added a `web.servlet` annotation at the class level and a `web.interface-method` annotation on each method that we want exposed in the SOAP service interface.

Example 21-5. Annotated JAX-RPC service implementation

```

/**
 * PeopleFinderImpl: Implementation of our remote PeopleFinder interface
 *
 * XDoclet tags:
 *
 * @web.servlet
 *     name="PeopleFinderService"
 */
public class PeopleFinderServiceImpl {
    public PeopleFinderImpl( ) throws RemoteException {}

    /**
     * @web.interface-method
     */
    public Person[] findPeople(SearchArg[] args)
        throws InvalidSearchException, PersistenceException,
            RemoteException {
        . . .
    }

    /**
     * @web.interface-method
     */
    public Person findPerson(String id)
        throws InvalidSearchException, PersistenceException,
            RemoteException {
        . . .
    }

    /** Utility method for retrieving a database connection */
    private Connection getConnection( ) {
        . . .
    }
}

```

To process these annotations, we still use the same `webdoclet` Ant task we used earlier to generate web deployment descriptors. In this case, though, we want to generate source code related to our web service implementation. So we use a different subtask to the `webdoclet` task, `service-endpoint`, as shown in [Example 21-6](#). The `service-endpoint` subtask is used to generate JAX-RPC service interfaces for any classes annotated as web components with the `web.servlet` tag that also have one or more methods annotated as web service operations using the `web.service-interface` tag.

Example 21-6. Using the webdoclet task to generate JAX-RPC interfaces

```

<target name="create-ws">
    <webdoclet destDir="${ws.src.dir}">

```

```

        <fileset dir="${src.dir}">
            <include name="**/PeopleFinderServiceImpl.java"/>
        </fileset>
        <service-endpoint/>
    </webdoclet>
</target>

```

If we run this Ant target on the source code in [Example 21-5](#), the service interface shown in [Example 21-7](#) will be generated into the *ws.src.dir* directory.

Example 21-7. Generated web service interface

```

/*
 * Generated by XDoclet -- Do not edit!
 */

/**
 * service_endpoint_interface_for arguments: [Ljava.lang.String;@35b92e
 */
public interface PeopleFinderService
    extends java.rmi.Remote{
    public Person[] findPeople( SearchArg[] args )
        throws InvalidSearchException, PersistenceException,
            java.rmi.RemoteException;

    public Person findPerson( java.lang.String id )
        throws InvalidSearchException, PersistenceException,
            java.rmi.RemoteException;
}

```

Notice that the class name for the generated service interface was taken from the `name` parameter provided on the `web.servlet` annotation. The two methods on the interface are the two marked with the `service-interface` tag in the source. Notice as well that the `getConnection()` method on the service implementation was not mapped to the generated interface because it was not annotated.

21.4.4.2. EJBs

Many developers are put off by the complexity of writing EJBs because of the many code artifacts that have to be generated. As described in [Chapter 6](#), an EJB component can include one or more home interfaces, one or more client interfaces, a bean implementation class, and a custom primary key class (for certain entity bean situations) in addition to the deployment descriptor. Since all of these artifacts have specific relationships with one another (defined by the EJB specification), this seems like an ideal context for code annotations to help out. In fact, XDoclet began as a tool to assist in the development of EJBs and has since grown and been generalized to many other contexts, some

of which we've already explored.

The `ejb` tag group in XDoclet contains a set of EJB-oriented annotations. Many of these are meant to generate deployment descriptor information, similar to the `web` annotations we saw in the previous section. But here we will take a look at using annotations to generate some of the actual source code that makes up an EJB component.

[Example 21-8](#) shows an annotated version of the implementation class for the `ProfileManager` EJB discussed in [Chapter 6](#). In the class-level comments, we've inserted an `ejb.bean` annotation indicating that the name of the EJB should be `ProfileManager` and that it is a stateless session bean. This EJB implementation has a single create method, so we've annotated that method with an `ejb.create-method` tag. Finally, the bean has a single business method, `getProfile()`, and we've annotated that with an `ejb.interface-method` tag. On this tag, we've set the `view-type` property to `both` to indicate that we want this business method to be exposed in both the remote and local client interfaces for this EJB, if either or both are generated.

Example 21-8. Annotated EJB implementation class

```
/**
 * @ejb.bean
 *     name="ProfileManager"
 *     type="Stateless"
 */

// Imports omitted...

public class ProfileManagerBean implements SessionBean {
    // Store session context
    private SessionContext mContext = null;

    public ProfileManagerBean( ) {}

    /** Activate notification method */
    public void ejbActivate( ) {}

    /** Remove notification method */
    public void ejbRemove( ) {}

    /** Passivation notification method */
    public void ejbPassivate( ) {}

    /** Container session context accessor */
    public void setSessionContext(SessionContext context) {
        mContext = context;
    }

    /**
     * No-argument create( ) -- nothing to initialize in this case.
     *
     * @ejb.create-method
     */
}
```

```

    */
    public void ejbCreate( ) {}

    /**
     * Get a profile for a named person
     *
     * @ejb.interface-method
     *     view-type=both
     */
    public ProfileBean getProfile(String name)
        throws NoSuchPersonException {
        ProfileBean profile = new ProfileBean(name);
        return profile;
    }
}

```

We use the same general approach to process these annotations as with any other XDoclet annotations: we run the appropriate XDoclet Ant task over the source code. In this case, we're dealing with EJBs and tags from the `ejb` group, so we use the `ejbdoclet` task rather than the `webdoclet` task we've been using so far. A sample target that makes use of the `ejbdoclet` task to process our `ProfileManagerBean` source code is shown in [Example 21-9](#).

Example 21-9. Ant target for processing EJB annotations

```

<target name="gen-ejb-code">
  <!-- Generate EJB home and client interfaces from an
       annotated bean implementation class -->
  <ejbdoclet destDir="${generated.ejb.dir}"
            ejbSpec="2.1">
    <fileset dir="${java.src.dir}">
      <include name="**/ProfileManagerBean.java"/>
    </fileset>
    <remoteinterface/>
    <localinterface/>
    <homeinterface/>
    <localhomeinterface/>
  </ejbdoclet>
</target>

```

Since our goal here is to generate support source code, we've used only subtasks of the `ejbdoclet` task related to generating code. The `<remoteinterface>`, `<localinterface>`, `<homeinterface>`, and `<localhomeinterface>` elements are included to indicate that we want these respective pieces of code generated for us from the annotations found in the source files included in the `<fileset>`.

If we run this target against the annotated source code in [Example 21-8](#), XDoclet will generate

remote and local client interfaces for our EJB (named `ProfileManager` and `ProfileManagerLocal`) and remote and local home interfaces (named `ProfileManagerHome` and `ProfileManagerLocalHome`). The names of these interfaces are generated automatically, based on the `name` parameter provided on the `ejb.bean` annotation. We could have specified the names of these interfaces using additional annotations in the source, but, for this example, the automatically generated interface names are fine.

The generated remote client interface, `ProfileManager`, is shown in [Example 21-10](#), and the remote home interface, `ProfileManagerHome` is shown in [Example 21-11](#). The local versions are nearly identical, so we won't show them here. Notice that the `ProfileManager` client interface contains a declaration for the `getProfile()` method that was annotated with an `ejb.interface-method` tag in the source for the implementation class. Also notice that the `ProfileManagerHome` home interface contains a declaration for a single no-argument `create()` method because we annotated the corresponding `ejbCreate()` method in the bean class with an `ejb.create-method` tag.

Example 21-10. Generated remote client interface

```

/*
 * Generated by XDoclet -- Do not edit!
 */

/**
 * Remote interface for ProfileManager.
 */
public interface ProfileManager
    extends javax.ejb.EJBObject{
    public ProfileBean getProfile( java.lang.String name )
        throws NoSuchPersonException, java.rmi.RemoteException;
}

```

Example 21-11. Generated remote home interface

```

/*
 * Generated by XDoclet -- Do not edit!
 */

/**
 * Home interface for ProfileManager
 */
public interface ProfileManagerHome
    extends javax.ejb.EJBHome{
    public static final String COMP_NAME="java:comp/env/ejb/ProfileManager";
    public static final String JNDI_NAME="ProfileManager";

    public ProfileManager create( )
        throws javax.ejb.CreateException, java.rmi.RemoteException;
}

```

```
}
```

You'll also notice a few static `String` members defined in the remote home interface. These variables were generated from the `name` parameter on the `ejb.bean` annotation and represent default JNDI mappings that XDoclet will generate for the EJB in any application server configuration files we ask it to generate. So these static variables are a convenience for any client code that makes use of this EJB component.

Keep in mind that we are showing you only a small part of the options available in these XDoclet annotations and Ant tasks. You can, for example, generate a deployment descriptor for the EJB component as well by just adding a `deploymentdescriptor` subtask to the `ejbdoclet` task:

```
<target name="gen-ebb-code">
  <ejbdoclet destDir="${ejb.dir}"
            ejbSpec="2.1">
    <fileset dir="${src.dir}">
      <include name="**/ProfileManagerBean.java"/>
    </fileset>
    <homeinterface/>
    <localhomeinterface/>
    <remoteinterface/>
    <localinterface/>
    <deploymentdescriptor/>
  </ejbdoclet>
</target>
```

Running this version of the `gen-ebb-code` target will generate the same client and home interfaces as before and also the EJB deployment descriptor shown in [Example 21-12](#).

Example 21-12. Generated EJB deployment descriptor

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

<ejb-jar 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/ebb-jar_2_1.xsd"
         version="2.1">
  <description><![CDATA[No Description.]]></description>
  <display-name>Generated by XDoclet</display-name>

  <enterprise-beans>

    <!-- Session beans -->
    <session >
```

```

<description><![CDATA[ ]]></description>

<ejb-name>ProfileManager</ejb-name>

<home>ProfileManagerHome</home>
<remote>ProfileManager</remote>
<local-home>ProfileManagerLocalHome</local-home>
<local>ProfileManagerLocal</local>
<ejb-class>ProfileManagerBean</ejb-class>
<session-type>Stateless</session-type>
<transaction-type>Container</transaction-type>

</session>
<!-- Various XDoclet comments omitted... -->
</enterprise-beans>
<!-- Various XDoclet comments omitted... -->
</ejb-jar>

```

21.4.5. Other Annotations

XDoclet has many other uses that we haven't seen in our examples. Just in the `web` and `ejb` tag groups are many other annotations and parameters on these annotations that you can use to generate all the other parts of their respective deployment descriptors as well as configuration files for other application servers and other code artifacts.

But there are a number of other tag groups you may find useful, depending on what kinds of enterprise development you are doing. We mention a few key ones here:

`@wsee`

This tag group contains annotations that can be used to generate JAX-RPC namespace mapping files for web services. This can be as useful as generating deployment descriptors for J2EE components, allowing you to keep your mapping file in sync with the source code as it evolves. There is a `wseedoclet` Ant task that can be used to process these annotations.

`@hibernate`

This set of annotations can be used to mark up data objects, allowing you to generate the Hibernate mapping files and configuration files for a set of JavaBeans that you want to persist using Hibernate. The `hibernatedoclet` Ant task is used to process these annotations.

`@jsf`

These annotations can be used to mark up JavaServer Faces code, such as managed JavaBeans, validators, and converters, and use these annotations to generate JSF configuration files. These annotations are processed using the `webdoclet` task by including the

`facesconfigxml` subtask.

We'd encourage you to explore the XDoclet documentation to learn about some of the other capabilities that might be useful to you.

21.4.6. Custom XDoclet Annotations

It is possible to write your own XDoclet annotations, if you need to do that. Some reasons you may feel the need to define your own annotations might include:

- Custom configuration files that you've defined in your application
- Design patterns that you've adopted that require support code that can be generated from base classes plus some annotations
- Third-party libraries that you are using to implement key service interfaces in your system, where these implementation classes can be autogenerated from annotations placed in the interfaces

The details of writing custom XDoclet tags and processors is beyond the scope of this chapter, but it essentially boils down to writing a template file to specify the structure of the output you want generated from your code annotations, or writing a custom tag handler class that will be invoked at runtime when one of your tags is encountered. It's also possible to define your own custom Ant task for processing your tags, although in many cases, it's possible to just use the generic `doclet` Ant task provided with XDoclet to process a set of code using a template file you provide. For more details, consult the XDoclet developer's documentation.

21.4.7. A Word of Caution

As we mentioned earlier in the chapter, you need to temper your use of code annotations with some practical considerations. Deployment descriptors, configuration files, and other code artifacts are often separate from the source code for a good reason. Some configuration parameters need to be localized to the target deployment environment, for example, and if you have several environments to deal with (development, test, and production; developers A, B, and C; or clients X, Y, and Z), you don't want to be forced to edit the source code to deploy your system across these environments. Make use of Ant properties to parameterize your annotations, or make use of externalized configuration file fragments, as we demonstrated in our web deployment descriptor examples. These practices, in conjunction with the general Ant localization tactics discussed in [Chapter 17](#), should give you all the power you need to both take advantage of code annotations yet maintain the ability to easily deploy your applications in various environments.

21.5. J2SE Annotations Tutorial

J2SE 5.0 introduced a standard, generalized annotation model to the Java language itself. J2SE annotations are an actual part of the Java language syntax, with annotations inserted directly into your code (not just within comments, as you do with Javadoc tags). These annotations can be processed using a standard annotation processing tool provided with J2SE 5.0 environments, or you can write your own Java code to access and process annotations.

J2SE annotations differ from XDoclet annotations in two key ways. First, J2SE annotations are part of the actual language syntax. These annotations are supported directly by Java compilers, and new annotations are defined much like new classes and interfaces, using Java interface-like declarations.

The other difference is a temporary one. At this writing, the general infrastructure for defining and using J2SE annotations has been introduced in Java 5.0, and a few standard annotations are part of the core Java API. Annotations and processing tools based on J2SE annotations aimed at enterprise development (e.g., generating deployment descriptors or code artifacts for components) are still in development, as part of the next round of specifications in the web, EJB, and web services spaces, among others.

In the meantime, and even after these standard enterprise annotations are available, XDoclet and J2SE annotations are complementary tools that you can use together to ease the development and management of enterprise systems.

21.5.1. General Programming Model

Before we launch into some examples of J2SE annotations, you should note that the core annotations, and the support interfaces and enumerations we reference here, all reside in the `java.lang` and `java.lang.annotation` packages in the core Java API. We've omitted the imports of various `java.lang.annotation` elements in some of our examples for brevity.

J2SE annotations use the `@` character to denote the start of code annotations, keeping with the convention set by Javadoc and XDoclet annotations. As with these other annotations, J2SE annotations can be used to assign metadata at various levels in your Java classes (class, method, and member variable). More formally, a J2SE annotation can be used anywhere that standard Java modifiers (like `public`, `private`, `protected`, or `static`) can be used.

[Example 21-13](#) demonstrates the general syntax of J2SE annotations. The example shows a simple Java file with a class-level and method-level J2SE annotation. The name of the annotation immediately follows the `@` character, and an annotation can optionally include a comma-delimited set of parameter values that are enclosed in parentheses immediately after the annotation name. These parameter values typically take the form of `name=value`, as in the `Test` annotation in the example. If the annotation accepts only a single parameter, as in the `SuppressWarnings` annotation, the parameter value can be given without providing the name. If an annotation takes no parameters, the annotation has no parameters or parentheses following it in the source, as in the `Deprecated` annotation.

Example 21-13. Sample code using J2SE annotations

```
@SuppressWarnings("serial")
public class Sample implements Serializable {
    @Deprecated
    public String doThis(String target) { return target + " done"; }

    @Test(success="Success", failure="doThat call failed")
    public int doThat( ) { return this.hashCode( ); }
}
```

In the following sections, we'll first look at the set of standard annotations that is built into the J2SE environment. Then we'll look at defining your own custom annotations.

21.5.2. Core Annotations

J2SE 5.0 provides a handful of standard annotations as part of the core Java API. These annotations are directly embedded in the Java environment. Currently, there are two categories of these core annotations. The first is *compiler-related annotations*. These annotations are general-purpose ones, used in source code to influence the checks performed at compile time. The `@Override`, `@SuppressWarnings`, and `@Deprecated` annotations fall into this category. We give a quick introduction to these annotations in the remainder of this section.

The other core annotations are used in the definition of other annotations, and they are sometimes called *meta-annotations*. The `@Target`, `@Retention`, `@Inherited`, and `@Documented` annotations fall into this category. We'll see the `@Target` and `@Retention` meta-annotations in action in the next section. To read about the `@Inherited` and `@Documented` annotations, see the more detailed coverage of annotations in *Java in a Nutshell* (O'Reilly).

21.5.2.1. @Override

This annotation is used to mark methods that are intended to override methods from their parent classes. You use this annotation to avoid the common error of intending to override a parent method but getting the signature slightly wrong. In most cases, the compiler will simply treat the method as an overloaded version of the parent's method and let it pass with no warning. The `@Override` annotation tells the compiler to fail outright if a method doesn't override a method in the parent class.

Here's a sample base class, `Base`, with a default implementation of a `getIdentifier()` method:

```
public class Base {
    /** Get the ID of this thing */
    public String getIdentifier( ) {
        return this.getClass( ).getName( );
    }
}
```

```

    }
}

```

Now let's make a subclass of `Base`, named `Child`, with which we attempt to override the `getIdentifier()` method. Unfortunately, the signature is incorrect, and we've actually overloaded `getIdentifier()` rather than overridden it.

```

public class Child extends Base {
    /** Get the ID of this thing */
    public String getIdentifier(String type) {
        return type + ":" + this.getClass( ).hashCode( );
    }
}

```

The `Child` class doesn't break any syntax rules, though, so the compiler will happily compile it with no errors. We can explicitly state that the `getIdentifier()` method in `Child` is meant to override a parent implementation by using the `@Override` annotation:

```

public class Child extends Base {
    /** Get the id of this thing */
    @Override
    public String getIdentifier(String type) {
        return type + this.getClass( ).hashCode( );
    }
}

```

If we compile this annotated code, the compiler will now generate an error:

```

wunderkatzen[tcsh]> javac Base.java Child.java
Child.java:3: method does not override a method from its superclass
    @Override
    ^
1 error

```

If you've ever struggled to find a subtle bug like this in a large collection of classes, the ability to annotate your overrides like this is a real boon. Now the compiler can warn you, for example, if you refactor your code and unintentionally break the inheritance chain.

21.5.2.2. @SuppressWarnings

The `@SuppressWarnings` annotation is used to filter out warnings that you do not want to see in the compiler output. Suppose, for example, we've implemented a `Serializable` data object in our system but would rather not specify a serial version ID, opting to allow the `java.io` runtime to generate a default one for us. Some compilers will complain about `Serializable` classes that do not provide their own serial version ID, but we can suppress these warnings using the `@SuppressWarnings` annotation:

```
@SuppressWarnings("serial")
public class MySerializable implements Serializable {
    public MySerializable( ) {
        super( );
    }
}
```

Note that each compiler can choose its own naming convention for warning types for the `@SuppressWarnings` annotation. We assume here that the compiler is using `"serial"` to designate the serializable warnings we want to filter out, which is the designation used by the Sun J2SE 5.0 implementation. In addition, general support for the `@SuppressWarnings` annotation is optional as of this writing, so some Java 5.0 compilers may not even honor it. But support for it should firm up in future revisions of compilers, and the naming convention for compiler warnings should become more consistent as well.

21.5.2.3. @Deprecated

The `@Deprecated` annotation plays essentially the same role as the existing `@Deprecated` tag in Javadoc. You can use the `@Deprecated` annotation to mark classes and methods that may be removed in future versions of the code. The compiler will generate warnings when code uses these marked classes or methods to encourage developers to remove them from their code.

21.5.3. Creating and Using Custom Annotations

To demonstrate how to create and use your own J2SE annotations, we're going to use the context of a Java programming course. As the teacher of this course, you naturally create a number of code examples to demonstrate key principles discussed in class. You find yourself writing up detailed guides for the source code, highlighting key areas in the source code that participants should consult for examples relevant to various points discussed in class. Wouldn't it be nice, you think to yourself, I could put these instructional highlights directly into the code and then generate the code guides directly from the annotated source code? J2SE annotations to the rescue.

[Example 21-14](#) shows a definition of a simple annotation called `@Learn`. This annotation is a simple marker annotation because it takes no arguments. It simply marks the annotated item, similarly to the `@Deprecated` built-in annotation. Note that the syntax for defining an annotation is very similar to writing a Java interface. The annotation type uses the `@interface` modifier to distinguish it from a regular interface.

Example 21-14. Custom marker annotation

```
import java.lang.annotation.*;

@Target({ ElementType.TYPE,
          ElementType.METHOD,
          ElementType.CONSTRUCTOR,
          ElementType.FIELD })
@Retention(RetentionPolicy.RUNTIME)
public @interface Learn {}
```

In our example, we're making use of some of the built-in meta-annotations that are used to modify annotation definitions themselves. The first one in our example, `@Target`, is used to specify the code elements that are relevant for this annotation. Values from the `ElementType` enumeration are used to specify the allowable code locations. In our case, we want to use the `@Learn` annotation on classes and interfaces (`TYPE`), on methods (`METHOD`), on constructors (`CONSTRUCTOR`), and on member variables (`FIELD`). We could also specify `LOCAL_VARIABLE` for local variables in methods, `PACKAGE` for package-level annotations, and `PARAMETER` for annotations on method arguments. If there is no `@Target` annotation on an annotation definition, the annotation can be used anywhere. In our case, we're going to write a processor for these annotations and we don't want to insert annotations that won't be handled, so we explicitly name the cases we support with the `@Target` annotation. If we try to use our `@Learn` annotation on any code elements not specified in the `@Target` meta-annotation, we'll receive an error from the compiler.

The other meta-annotation we've used is `@Retention`. This annotation controls when and how we can access the annotation and its parameters, if any. This meta-annotation takes a single parameter, one of the three values in the `RetentionPolicy` enumeration: `SOURCE`, `CLASS`, or `RUNTIME`. The `SOURCE` value indicates that the annotation should be retained only in the source, so the compiler can discard it while processing the source code. The `CLASS` value is the default, and it indicates that the annotation should be retained in the compiled class file but not loaded into the Java runtime. The `RUNTIME` value indicates that we want the annotation to be available at runtime for processing. We're using the `RUNTIME` setting because we want to write our own Java program to process the `@Learn` annotations in our source.

Once we compile our annotation, we can use it in our source code to annotate elements that are interesting from an instructional standpoint, as in this example:

```
// Imports omitted...
@Learn
public class CourseExample extends HttpServlet {
    @Learn
    private Context mContext = null;

    @Learn
    protected CourseExample( ) {
        super( );
    }

    @Learn
```

```

    private String doSomethingInteresting( ) {
        . . .
    }

    private void thisMethodHasNoInstructionalValue( ) {}

    @Learn
    public void doPost(HttpServletRequest request,
                       HttpServletResponse response) {
        . . .
    }
}

```

Note that, if we define our custom annotation with a package, we need to import the annotation from that package in order to use it in source code. So in our example, if `@Learn` were defined in the package `com.oreilly.jent.annotations`, the `CourseExample` class would need to import it in the same way that external classes and interfaces need to be imported:

```

import com.oreilly.jent.annotations.Learn;
. . .

```

This version of our `@Learn` annotation is a decent start, but we will quickly realize that we need more metadata in order to make this useful. First of all, we need to be able to put some descriptive text with the `@Learn` annotations to tell the students why this is valuable from an instructional perspective. We can extend our `@Learn` annotation to include a single text parameter, as shown in [Example 21-15](#).

Example 21-15. `@Learn` annotation with a single parameter

```

import java.lang.annotation.*;

@Target({ ElementType.TYPE,
          ElementType.METHOD,
          ElementType.CONSTRUCTOR,
          ElementType.FIELD })
@Retention(RetentionPolicy.RUNTIME)
public @interface Learn {
    String value( );
}

```

Annotation parameters are defined using no-argument methods on the annotation interface, and the return type of the method indicates the type of the parameter. Annotation parameters are limited in terms of supported types. They can be primitive types (`int`, `float`, etc.), `Strings`, `Classes`,

enumerated types, other annotations, or arrays of any of these.

Normally, you specify the value of an annotation parameter using the following syntax, providing the name of the parameter with its value:

```
@Learn(value="You can learn a lot from this class (no pun intended)")
public class Example { . . . }
```

If an annotation has a single parameter named `value`, however, you can eliminate the parameter name from the annotation when you use it, like so:

```
@Learn("You can learn a lot from this class (no pun intended)")
public class Example { . . . }
```

With this extended `@Learn` annotation, we can beef up our code markup quite a bit:

```
// Imports omitted...
@Learn("This servlet demonstrates the use of the XYZ design pattern.")
public class CourseExample extends HttpServlet {
    @Learn("As discussed in lesson 4, the use of the Context member " +
        " is meant to simplify the rest of the servlet code")
    private Context mCtx = null;

    @Learn("This no-argument constructor will become interesting " +
        "in module 3")
    protected CourseExample( ) {
        super( );
    }

    @Learn("Examine this method to see the \"keep it simple\" principles " +
        + "discussed in module 2.")
    private String doSomethingInteresting( ) {
        . . .
    }

    private void thisMethodHasNoInstructionalValue( ) {}

    @Learn("Notice that we have a doPost( ), but no doGet( )")
    public void doPost(HttpServletRequest request,
        HttpServletResponse response) {
        . . .
    }
}
```

We might even decide to extend our `@Learn` annotation even more, with another `String` parameter to indicate the relevant course module and a `boolean` flag to indicate whether the text of the annotation should be included only in the guide for its relevant module or in all module guides:

```
import java.lang.annotation.*;

@Target({ ElementType.TYPE,
         ElementType.METHOD,
         ElementType.CONSTRUCTOR,
         ElementType.FIELD })
@Retention(RetentionPolicy.RUNTIME)
public @interface Learn {
    String value( );
    String moduleId( ) default "";
    boolean moduleOnly( ) default false;
}
```

Notice that we've specified a default value for our `moduleId` and `moduleOnly` parameters, which allows us to leave those parameters unspecified when we use the annotation in code. This feature of annotations is useful in its own right, but in our example, it can help us avoid a lot of backward-compatibility issues with existing annotations in our code. Our existing `@Learn` annotations have only a single `String` parameter, and they will all still be valid with this new version of our `@Learn` annotation since the new parameters all have defaults. But, we can also make use of the new parameters where they are useful, like so:

```
// Imports omitted...
@Learn("This servlet demonstrates the use of the XYZ design pattern.")
public class CourseExample extends HttpServlet {
    @Learn(value="As discussed in lesson 4, the use of the Context member" +
           " is meant to simplify the rest of the servlet code",
          moduleId="1")
    private Context mCtx = null;

    @Learn(value="This no-argument constructor is used in our Cactus tests",
           moduleId="3",
           moduleOnly=true)
    protected CourseExample( ) {
        super( );
    }

    @Learn(value="Examine this method to see the \"keep it simple\"
               + \" principles discussed in module 2.\"
           moduleId="2")
    private String doSomethingInteresting( ) {
        . . .
    }

    private void thisMethodHasNoInstructionalValue( ) {}
}
```

```

@Learn("Notice that we have a doPost( ), but no doGet( )")
public void doPost(HttpServletRequest request,
                  HttpServletResponse response) {
    . . .
}
}

```

21.5.4. Processing Annotations at Runtime

Defining and using custom annotations is all well and good, but they're really interesting only when we process the annotations to do something useful. Any annotations that are available at runtime (i.e., in their definitions, they've been marked with an `@Retention` annotation with a value of `RetentionPolicy.RUNTIME`), you can access these annotations at runtime using the extensions to the Reflection API introduced in Java 5.0.

In our example, we want to process our `@Learn` annotations and generate HTML guides to the source code for the participants in our course. To accomplish this, we could write a simple Java class like `LearnProcessor`, shown in [Example 21-16](#).

Example 21-16. Utility class for processing `@Learn` annotations

```

// Imports omitted...
public class LearnProcessor {
    public static void main(String[] args) {
        if (args.length < 2) {
            System.err.println("USAGE: . . .");
            System.exit(1);
        }
        String targetClassname = args[0];
        try {
            // Look up the class provided in the arguments
            Class targetClass = Class.forName(targetClassname);
            // Output the page header and start the table structure.
            // If the class is annotated with a Learn tag, include the
            // notation in the table as well.
            if (targetClass.isAnnotationPresent(Learn.class)) {
                Learn annot =
                    (Learn)targetClass.getAnnotation(Learn.class);
                outputHeader(targetClass, annot.value( ));
            }
            else {
                outputHeader(targetClass, null);
            }
            // Check each member variable on the class, and output
            // any notations found
            Field[] fields = targetClass.getDeclaredFields( );

```

```

    for (int i = 0; i < fields.length; i++) {
        if (fields[i].isAnnotationPresent(Learn.class)) {
            Learn annot =
                (Learn)fields[i].getAnnotation(Learn.class);
            outputFieldNotation(fields[i], annot.value( ));
        }
    }
    // Check each constructor on the class, and output a notation
    // for any that have a Learn annotation
    Constructor[] cons = targetClass.getConstructors( );
    for (int i = 0; i < cons.length; i++) {
        if (cons[i].isAnnotationPresent(Learn.class)) {
            Learn annot =
                (Learn)cons[i].getAnnotation(Learn.class);
            outputConstructorNotation(cons[i], annot.value( ));
        }
    }
    // Check each method on the class, and output a notation
    // for any methods that have a Learn annotation
    Method[] methods = targetClass.getMethods( );
    for (int i = 0; i < methods.length; i++) {
        if (methods[i].isAnnotationPresent(Learn.class)) {
            Learn annot =
                (Learn)methods[i].getAnnotation(Learn.class);
            outputMethodNotation(methods[i], annot.value( ));
        }
    }
    // Print the page footer
    outputFooter(targetClass);
}
catch (ClassNotFoundException cnfe) {
    System.err.println("Class \"" + targetClassname
        + "\" not found");
}
}
// Output methods omitted...
. . .
}

```

This is a simple utility class that's meant to be run from the command line, passing in the full name of the annotated class to be processed. We first attempt to load the class from the `CLASSPATH` using `Class.forName()` and then we use reflection to check for annotations on the class. First, we check for annotations at the class level itself, using the `isAnnotationPresent()` method, and if there is an `@Learn` annotation on the class, we call our `outputHeader()` method with the value parameter from the annotation:

```

if (targetClass.isAnnotationPresent(Learn.class)) {
    Learn annot =
        (Learn)targetClass.getAnnotation(Learn.class);
}

```

```
        outputHeader(targetClass, annot.value( ));  
    }
```

We then retrieve the fields declared on the class using the `getFields()` method and iterate over them, checking for `@Learn` annotations and calling the appropriate output method for each one found. We repeat the basic processing for the class constructors and methods.

We've omitted the details of the output methods here, but if we ran our `LearnProcessor` over the `CourseExample` class shown earlier, like so:

```
> java LearnProcessor CourseExample
```

it would not be hard to imagine generating HTML as shown in [Figure 21-2](#).

Figure 21-2. HTML guide generated from `@Learn` annotations

21.5.5. The Future of J2SE Annotations

If you compare XDoclet annotations to J2SE annotations, you'll quickly note that XDoclet is largely a set of predefined annotations that are handy for enterprise development, while J2SE is a tool for creating annotations that doesn't (yet) include any predefined annotations for things like generating deployment descriptors or web service interfaces.

Fortunately, work is afoot to define standard annotations, based on the J2SE annotation syntax, for a variety of types of enterprise development. The JAX-RPC 2.0 specification (in draft at this moment) includes standard annotations that allow us to annotate JAX-RPC service implementation classes in much the same way that the XDoclet `wsee` tag group does. A JAX-RPC service implementation will be

annotated something like this in JAX-RPC 2.0:

```
@WebService
public class PeopleFinderImpl {
    . . .
    @WebMethod
    public Person[] findPeople(SearchArg[] args)
        throws InvalidSearchException, PersistenceException {
        . . .
    }
}
```

Standard annotations are being defined in the EJB 3.0 specification (also in draft) that will allow you to indicate EJB types (stateful, stateless, entity), annotate business methods, specify persistence mappings, automatically generate remote and local interfaces, and more. See the end of [Chapter 6](#) for a look at these annotations.

In the meantime, J2SE is useful in its own right as a general annotation facility, and XDoclet provides a wealth of enterprise-oriented annotations that you can use right now.

Part III: Appendixes

This part of the book contains summary and reference information. It consists of the following appendixes:

- [Appendix A](#), *J2EE Deployment Descriptor Reference*
- [Appendix B](#), *JavaServer Faces Tag Libraries*
- [Appendix C](#), *Enterprise JavaBeans Query Language Syntax*
- [Appendix D](#), *SQL Reference*
- [Appendix E](#), *JMS Message Selector Syntax*
- [Appendix F](#), *RMI Tools*
- [Appendix G](#), *IDL Reference*
- [Appendix H](#), *Java IDL Tools*

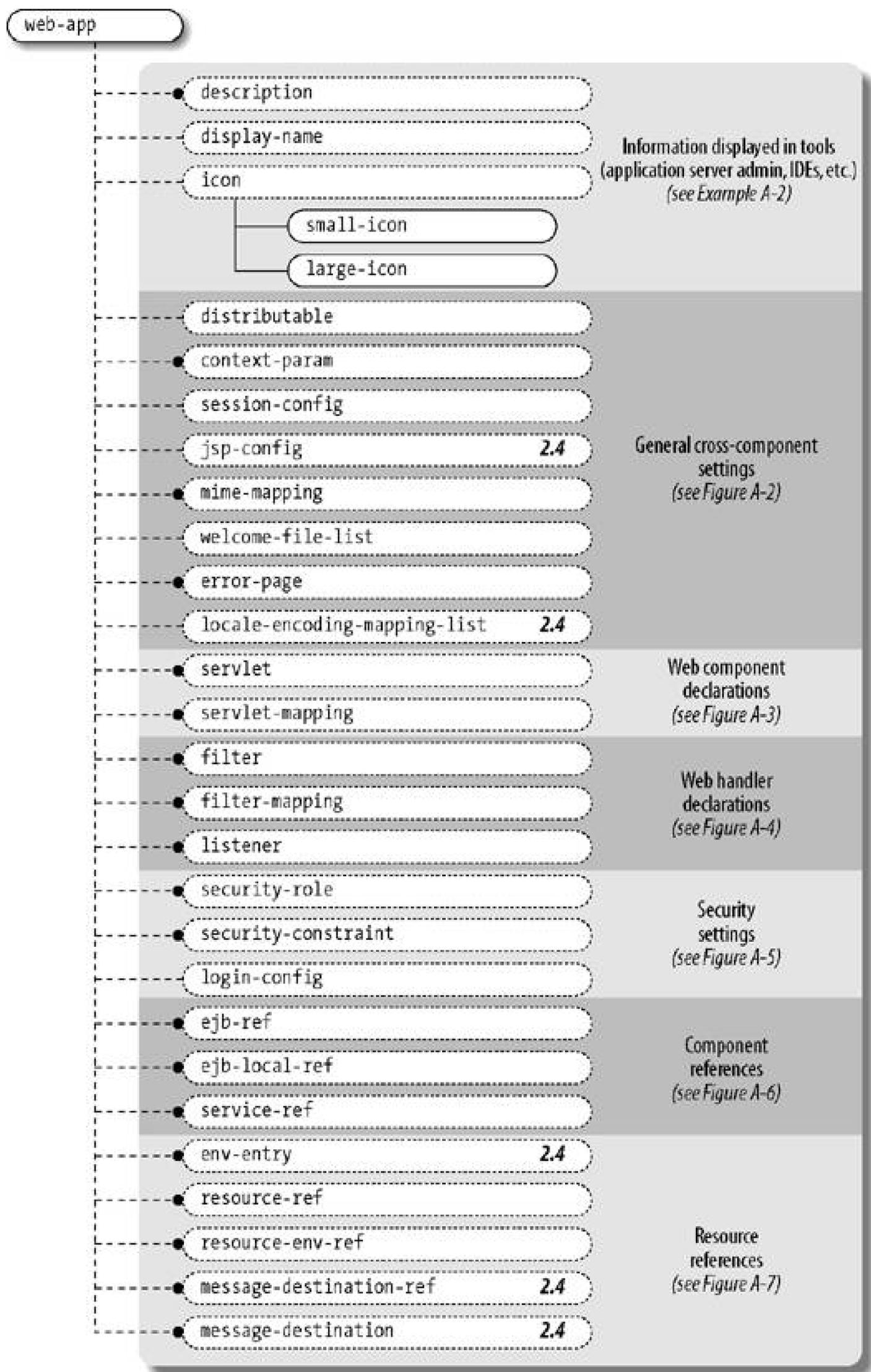
Appendix A. J2EE Deployment Descriptor Reference

This appendix provides a short reference to the J2EE deployment descriptors for components and resources. We describe the J2EE 1.4 descriptor formats here. Deployment descriptors for earlier versions of the J2EE specifications will vary slightlywe highlight discrepancies in key areas. The full deployment model and the mechanics of assembling and deploying J2EE components are described in [Chapter 2](#).

The sections that follow contain annotated deployment descriptors for the various components available in J2EE environments (web, EJB, web service). These annotated descriptors are an attempt to complement the standard XML schemas that are available from <http://java.sun.com/xml/ns/j2ee>.

Each section provides details on a specific deployment descriptor schema. At the beginning of the section is a high-level summary diagram that shows the first level of XML elements available in the descriptor, and if needed, subsections of the XML structure are described by more detailed diagrams that follow. These XML structure diagrams follow the format described in the Preface.

Figure A-1. Web deployment descriptor overview





A.1. Web Components (web.xml)

Web components are deployed using a *web.xml*/deployment descriptor. The high-level descriptor layout for web deployment descriptors is shown in [Figure A-1](#). In all the diagrams in this section, deployment elements that were introduced in the Servlet 2.4 specification are labeled with a "2.4" indicator. All other elements were present in the 2.3 (and possibly earlier) version of the Servlet specification.

The format for an empty *web.xml*/file is shown in [Example A-1](#).

Example A-1. Web deployment descriptor header

```
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
                             http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
         version="2.4">
    . . .
</web-app>
```

The example shows the XML schema for the Servlet 2.4 deployment descriptor being referenced from the default location on the java.sun.com server. To reference the Servlet 2.3 DTD instead (e.g., if you are using a J2EE 1.3 application server like JBoss 3.x or a Servlet 2.3 container like Tomcat 4.x), you should start your deployment descriptor with the following `DOCTYPE` element and no schema reference:

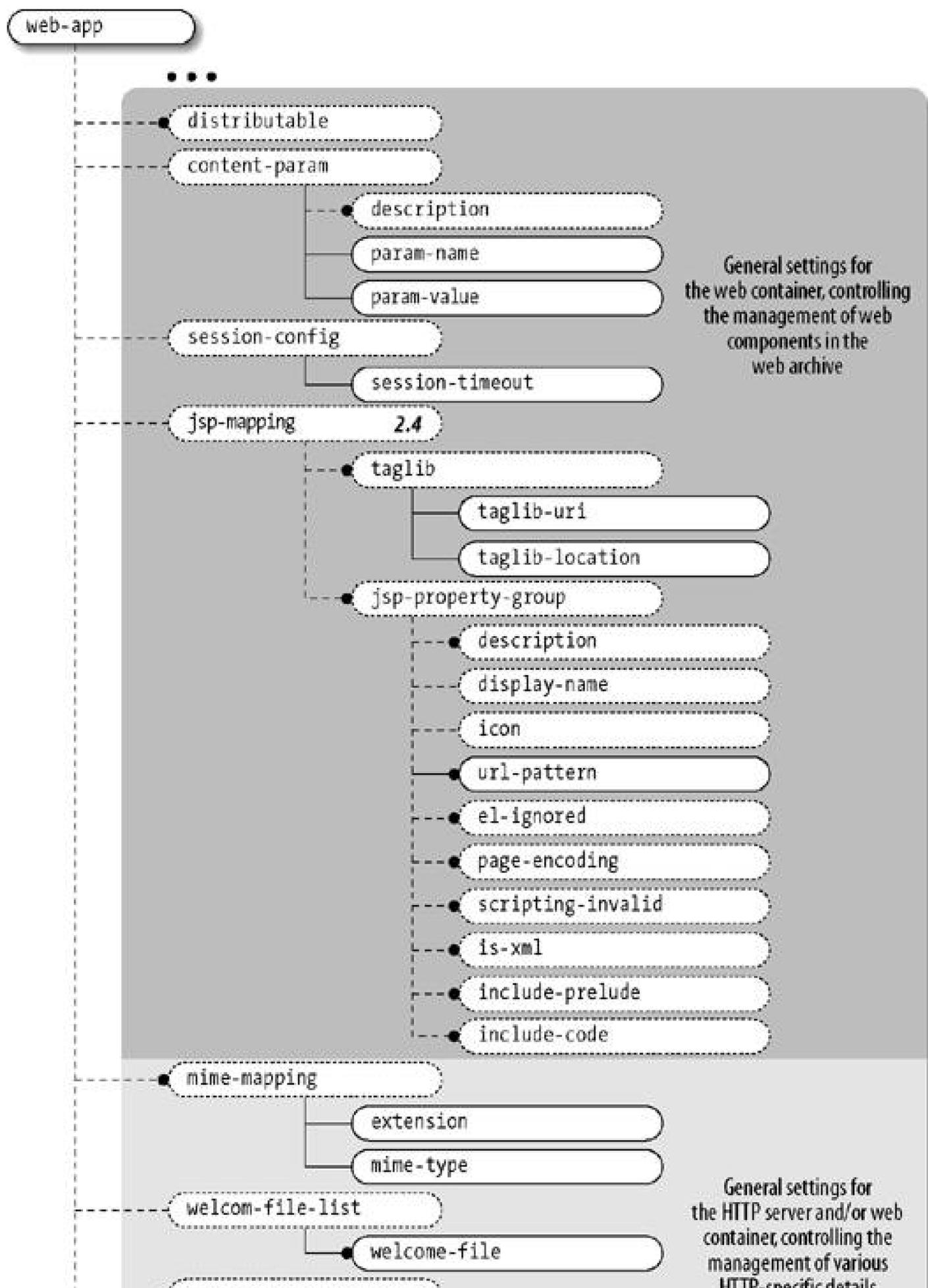
```
<!DOCTYPE web-app PUBLIC
    "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
    "http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
    . . .
</web-app>
```

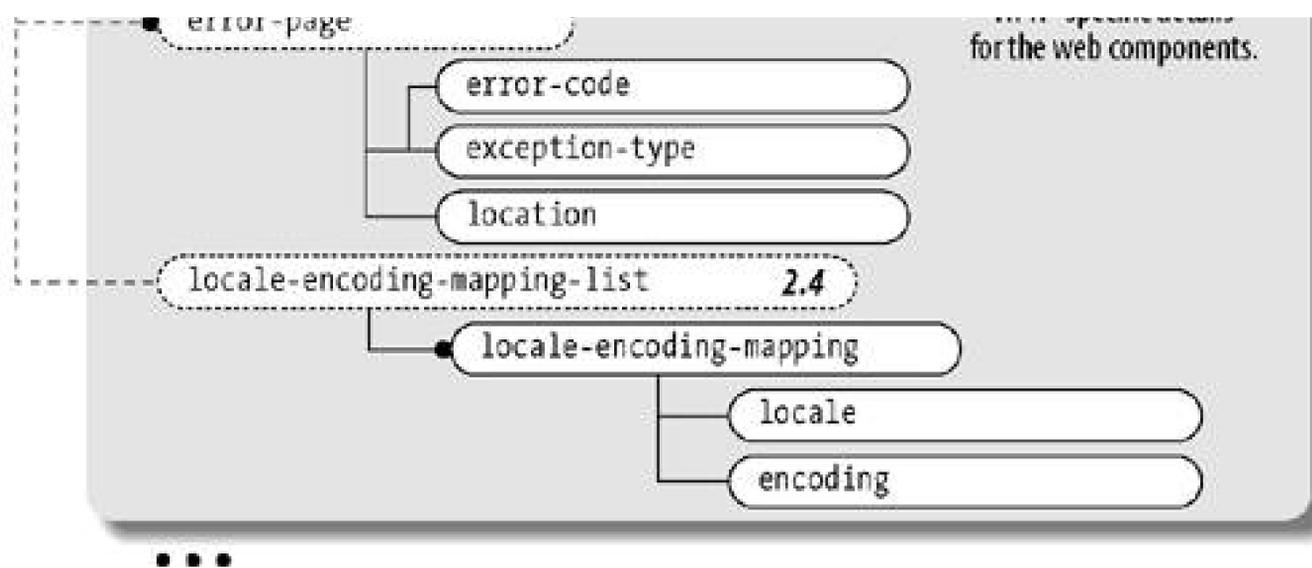
A.1.1. General Cross-Component Settings

The elements shown in [Figure A-2](#) are used for deployment settings that apply across the web components in the web archive.

The elements in this section of the web deployment descriptor are demonstrated in the annotated descriptor segment shown in [Example A-2](#).

Figure A-2. General web archive deployment settings





Example A-2. Annotated general deployment settings

```

<web-app . . . >
  <!-- Display name for the web application. -->
  <display-name>name</display-name>
  <!-- Description of the web application. -->
  <description>desc</description>
  <!-- Icon for representing the web application visually in tools,
        such as IDEs and app management tools. -->
  <icon>
    <small-icon>/icons/my-small-icon.jpg</small-icon>
    <large-icon>/icons/my-large-icon.jpg</large-icon>
  </icon>
  <!-- The presence of this element indicates to the web container
        that this web application can be safely distributed in clustered
        servlet containers. It may also trigger additional runtime
        checks by the container, for things like Serializable data
        in Session parameters. -->
  <distributable/>
  <!-- These elements declare context parameters that
        are made available to all servlets in this module, using the
        getInitParameter( ) method on their ServletContexts. -->
  <context-param>
    <!-- Description of the context parameter. -->
    <description>The default color for the sky</description>
    <!-- Name of the init parameter. -->
    <param-name>sky-color</param-name>
    <!-- Value of the init parameter. The value is always
        retrieved by the servlet as a String (the return type of the
        getInitParameter( ) method). -->
    <param-value>blue</param-value>
  </context-param>
  <!-- Defines the session parameters for all components in this
        module. The only config parameter allowed as of Servlet 2.4 is the
        session timeout period. -->

```

```

<session-config>
  <!-- Defines the length of time a session will live without
    any activity by the user. The time is given in minutes. -->
  <session-timeout>30</session-timeout>
</session-config>
<jsp-config>
  <!-- Each taglib element specifies the mapping of a custom tag library
    URI (used in a taglib directive in a JSP) to a Tag Library Description
    TLD file within
    the web archive that should be used to define the tags. -->
  <taglib>
    <!-- The URI serves as an identifier used in a JSP
      file to load a given tag library. The URI in the JSP
      taglib directive is matched to this taglib-uri element,
      and the corresponding TLD file is loaded for the JSP. -->
    <taglib-uri>http://my.jsp.app/taglibs/commerce</taglib-uri>
    <!-- The location, relative to the root of the web application
      context, of the TLD file for
      the tag library. -->
    <taglib-location>WEB-INF/taglibs/commerce.tld</taglib-location>
  </taglib>
  <!-- This element defines groups of JSP pages within the web archive,
    and various parameters for handling of the JSPs in the group. -->
  <jsp-property-group>
    <!-- The URL pattern that defines the JSP group. -->
    <url-pattern>/admin</url-pattern>
    <!-- If set to true, any JSP EL expressions in the
      JSPs will not be parsed. -->
    <el-ignored>>false</el-ignored>
    <!-- Specifies the character encoding of the pages.
      This value must match any pageEncoding directives used
      in the JSP pages themselves. Valid character encoding
      values can be found at www.iana.org. -->
    <page-encoding>US-ASCII</page-encoding>
    <!-- If true, any page scripting elements in the
      JSPs in this group will be rejected as invalid. -->
    <scripting-invalid>>false</scripting-invalid>
    <!-- Indicates that all pages in the URL pattern
      are to be interpreted as XML-based JSP pages, rather
      than depending on other factors, like file suffixes. -->
    <is-xml>>true</is-xml>
    <!-- If present, the named page element (HTML or
      JSP) is included, as an include directive, in the
      beginning of every JSP in the group. -->
    <include-prelude>/my/company-header.jsp</include-prelude>
    <include-prelude>/my/admin-header.jsp</include-prelude>
    <!-- If present, the named page element
      (HTML or JSP) is included, as an include directive, at
      the end of each JSP in the group. -->
    <include-coda>/my/admin-footer.jsp</include-coda>
    <include-coda>/my/company-footer.jsp</include-coda>
  </jsp-property-group>

```

```

</jsp-config>
<!-- These elements are used to define a mapping from a file extension
to a MIME type. This can be used for special data types that the
web components will generate for the user or to override the
settings of the underlying HTTP server. -->
<mime-mapping>
  <!-- The file extension (suffix) to be mapped. Examples are
  "gif", "txt", "xls". -->
  <extension>txt</extension>
  <!-- The MIME type to be broadcast to web clients for files with
  the above extension. -->
  <mime-type>text/plain</mime-type>
</mime-mapping>
<!-- An ordered list of welcome files, used to automatically
resolve client requests for URLs that resolve to directories rather
than specific JSP pages or servlets. The files listed here will be
searched for in order within the target directory, and if one is found
in the directory, it will be served to the user. -->
<welcome-file-list>
  <!-- Each welcome-file element contains a single filename. -->
  <welcome-file>index.jsp</welcome-file>
  <welcome-file>index.html</welcome-file>
</welcome-file-list>
<!-- Each error-page element contains a mapping
from an HTTP error code or Java exception type, to the URL for an error
page. When the error or exception is encountered, the client will be
redirected to the error page. -->
<error-page>
  <!-- Each error-page element must contain either an error-code
  specification that contains a valid HTTP error code (e.g., "500"), or
  an exception-type element that contains the fully qualified class name
  for a Java exception (e.g., "java.lang.NumberFormatException"). -->
  <error-code>404</error-code>
  <!-- The URL of the error page the user should be sent to if the
  HTTP error code or Java exception is encountered. The location has to
  be a relative path within the web archive file and starts
  with a '/'. -->
  <location>/my/error/page.html</location>
</error-page>
<error-page>
  <exception-type>my.app.exception</exception-type>
  <location>/my/error/page2.html</location>
</error-page>
<!-- A list of mappings of locales to character encodings.
Requests that are associated with a specific locale will
have the responses encoded using the character sets
specified below. -->
<locale-encoding-mapping-list>
  <!-- Specify that requests tagged with the China/Tiawan
  locale are encoded using the Big5 character set. -->
  <locale-encoding-mapping>
    <locale>zh_TW</locale>

```

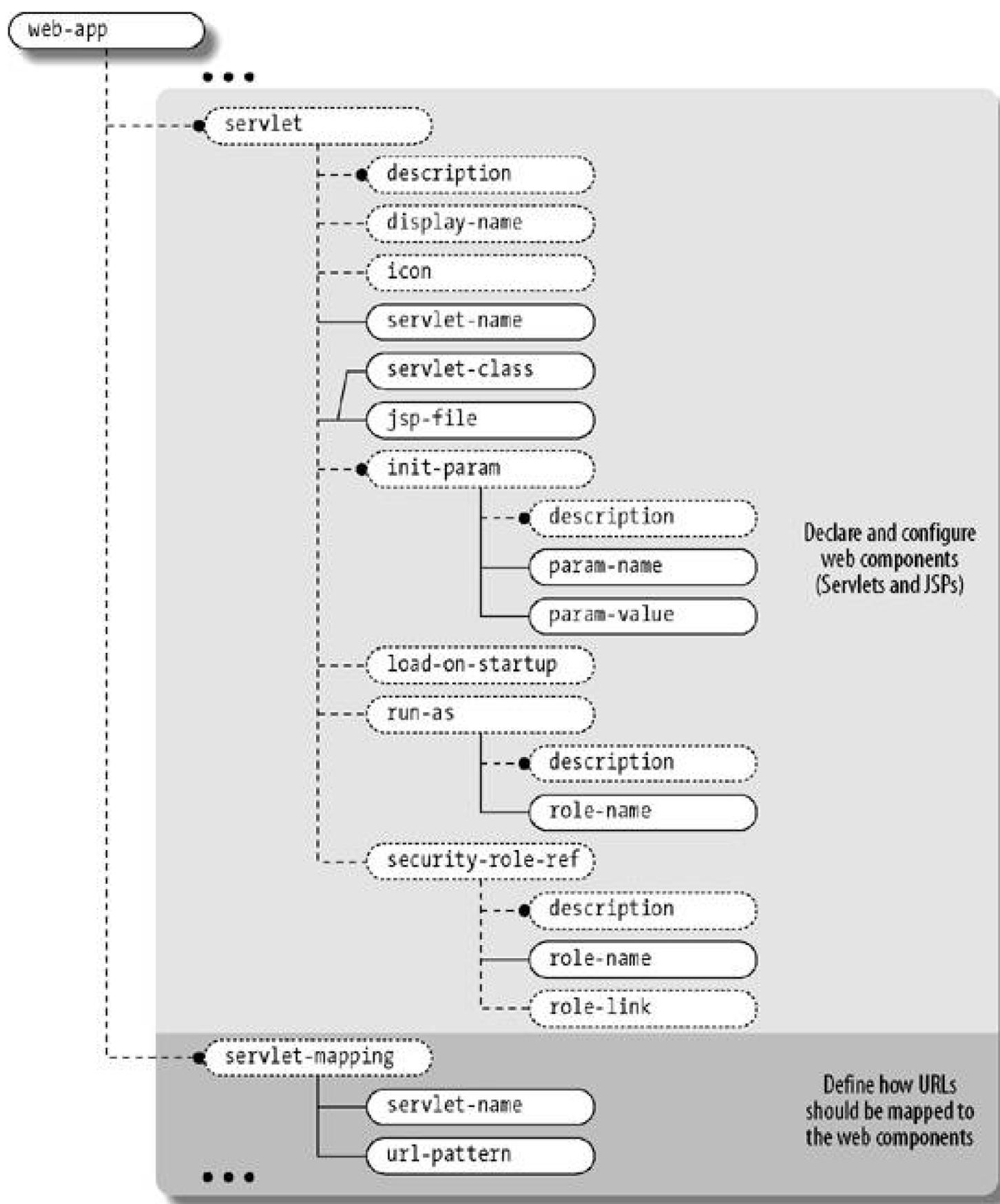
```
        <encoding>Big5</encoding>
    </locale-encoding-mapping>
    <!-- Specify that any English locales (US, UK, etc.) are
        encoded using UTF-8 -->
    <locale-encoding-mapping>
        <locale>en_US</locale>
        <encoding>UTF-8</encoding>
    </locale-encoding-mapping>
</locale-encoding-mapping-list>
. . .
</web-app>
```

A.1.2. Web Component Declarations

Web components (servlets and JSPs) are declared and configured in this section of the deployment descriptor. [Figure A-3](#) shows an overview of the elements present in this segment of the descriptor.

[Example A-3](#) shows sample web component declarations, annotated with descriptive comments.

Figure A-3. Web component declarations



Example A-3. Annotated web component declarations

```
<web-app . . . >
. . .
<!-- Each web component in this module is declared with a servlet
element. This element is used for both servlets and JSP files.
JSP files do not strictly require a servlet element, since they
have a default URL mapping (their relative path within the
```

web archive). But they do need an entry if any other configuration is needed for the JSP. -->

```

<servlet>
  <!-- Description of component. -->
  <description>desc</description>
  <!-- Display name for the component, used in tools. -->
  <display-name>name</display-name>
  <!-- Icon used to represent the web component in tools. -->
  <icon>
    <small-icon>/icons/my-small-servlet-icon.jpg</small-icon>
    <large-icon>/icons/my-large-servlet-icon.jpg</large-icon>
  </icon>
  <!-- Name to be used to refer to this component elsewhere in
  this deployment descriptor. -->
  <servlet-name>MyServlet</servlet-name>
  <!-- A declaration of either the fully qualified
  servlet class name (servlet-class) or the absolute path to the
  JSP file within the web archive (jsp-file). -->
  <!-- role="strong"><jsp-file>/my/jsp/file.jsp</jsp-file> -->
  <servlet-class>my.servlet.classname</servlet-class>
  <!-- Declare initialization parameters that will be
  accessible to this component through the getInitParameter( ) method
  on its ServletConfig object. -->
  <init-param>
    <!-- Description of the init parameter. -->
    <description>desc</description>
    <!-- Name of the init parameter. -->
    <param-name>name</param-name>
    <!-- Value of the init parameter. The value is always
    retrieved by the servlet as a String (the return type of the
    getInitParameter( ) method). -->
    <param-value>value</param-value>
  </init-param>
  <!-- Used to indicate whether and when this web component should
  be loaded when the enclosing web application is started. The value
  of this element is a single integer that is used to determine the
  order in which components will be loaded. A negative integer allows
  the container to load the component whenever it wants during startup.
  A non-negative integer puts the component in a load order, lower-
  numbered components are loaded first. Components with the same value
  are loaded in an undetermined order by the container. -->
  <load-on-startup>42</load-on-startup>
  <!-- Indicates a user role that should be used to run this
  web component. If this element is not provided, the role(s) of the
  client will be applied while the request is processed. -->
  <run-as>
    <!-- Description of the run-as role. -->
    <description>desc</description>
    <!-- The role this component should always be run as. The
    role name should match the name of a security-role element in
    this deployment descriptor. -->
    <role-name>name</role-name>

```

```

</run-as>
<!-- Declare security roles referenced in
this web component. These roles are made available at runtime
through the HttpServletRequest.isUserInRole( ) method. -->
<security-role-ref>
  <!-- A text description of the role. -->
  <description>desc</description>
  <!-- The role name as used in the web component code. -->
  <role-name>Administrator</role-name>
  <!-- Normally, the role-name must have a corresponding
security-role defined in the web.xml file. Alternatively,
you can link the role-name to another security-role entry
using the role-link element. -->
  <role-link>admin</role-link>
</security-role-ref>
</servlet>
<!-- These elements are used to define mappings from
URLs to servlet components defined above. For example, to map a
filter to everything in the "images" directory: "/images/*"; to map
a filter to all PDF files: "*.pdf"; to map to a specific document or
component, use the full URL, e.g., "/docs/intro.html".-->
<servlet-mapping>
  <!-- The name of the servlet/JSP component. This name must match
the servlet-name of a servlet element defined above. -->
  <servlet-name>MyServlet</servlet-name>
  <!-- The URL pattern that is to be mapped to the servlet/JSP. The
pattern uses the same syntax as that used for filter mappings. For
example, to map a servlet to everything in the "tools" directory:
"/tools/*", to map a servlet to all PDF files: "*.pdf"; to map to a
specific URL, use the full URL, e.g., "/tools/login". -->
  <url-pattern>/admin/login</url-pattern>
</servlet-mapping>
. . .
</web-app>

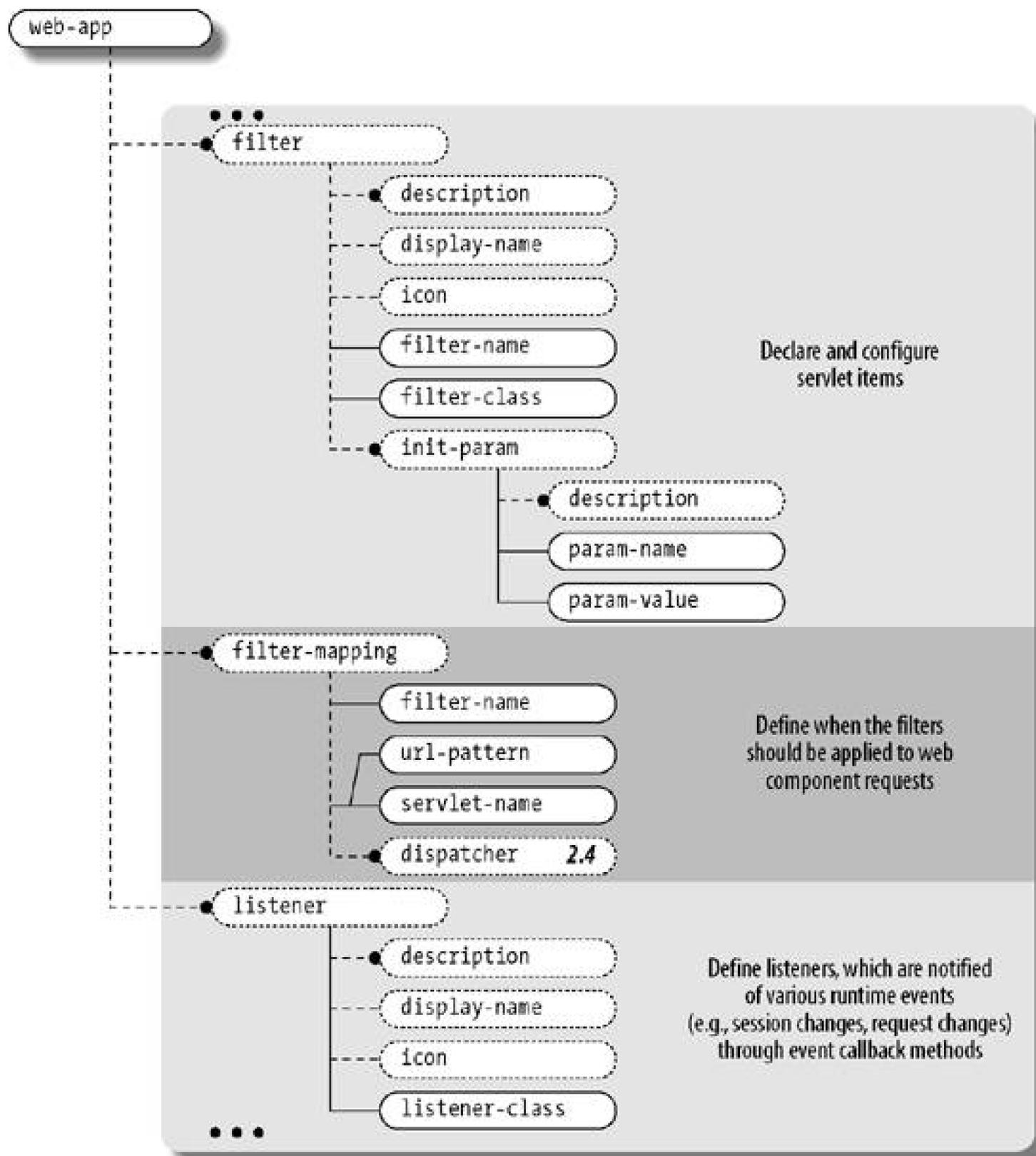
```

A.1.3. Web Handler Declarations

You can define two kinds of web handlers in the deployment descriptor, *filters* and *listeners*. Web filters are executed prior to the delivery of requests to specific web components or documents. Listeners are notified of events during the runtime management of the components, such as the creation and deletion of user sessions. The deployment elements relevant to web handlers are shown in [Figure A-4](#).

[Example A-4](#) shows sample web handler declarations annotated with descriptive comments.

Figure A-4. Web handler deployment elements



Example A-4. Annotated web handler declarations

```

<web-app . . . >
    . . .
    <!-- Declare a filter that can be applied to incoming requests for
         servlets or static URLs. Filters are mapped to servlets and URLs
         using filter-mapping elements, below. -->
    <filter>
        <!-- Description of filter. -->
        <description>Description</description>

```

```

<!-- Display name, to represent the filter in tools. -->
<display-name>Filter display name</display-name>
<!--Icon for representing the filter in tools. -->
<icon>
    <small-icon>/icons/my-small-filter-icon.jpg</small-icon>
    <large-icon>/icons/my-large-filter-icon.jpg</large-icon>
</icon>
<!-- Name for the filter, to be used in filter-mappings. -->
<filter-name>MyFilter</filter-name>
<!-- The fully qualified class name of the filter. The class
    must implement the javax.servlet.Filter interface. -->
<filter-class>my.filter.classname</filter-class>
<!-- Define initialization parameters. These parameters
    are accessed by the filter using the getInitParameter( ) method on its
    FilterConfig. -->
<init-param>
    <!-- Description of the init parameter. -->
    <description>desc</description>
    <!-- Name of the init parameter. -->
    <param-name>param-name</param-name>
    <!-- Value of the init parameter. The value is always
        retrieved by the filter as a String (the return type of the
        getInitParameter( ) method). -->
    <param-value>param-value</param-value>
</init-param>
</filter>
<!-- These elements are used to
    determine which filters to apply to client requests for given
    URLs. These filter mappings are checked in order to determine
    all of the filters applicable to a given resource. All of the
    matching filters make up an ordered chain of filters that are called in
    sequence before the actual servlet or URL resource is accessed. -->
<filter-mapping>
    <!-- Name of the filter to be applied. The name must match the
        filter-name for a filter element above. -->
    <filter-name>MyFilter</filter-name>
    <!-- Specify the servlet/JSP name or URL(s) the filter should be applied
        to. Either a url-pattern or servlet-name must be provided. The
        value of the url-pattern follows the rules of servlet mappings. For
        example, to map a filter to everything in the "images" directory:
        "/images/*"; to map a filter to all PDF files: "*.pdf"; to map to a
        specific URL, use the full URL, e.g., "/docs/intro.html". If a
        servlet-name is used, it has to match the name in a servlet-mapping
        element, above. -->
    <url-pattern>/filterurl</url-pattern>
    <!-- <servlet-name>name</servlet-name> -->
</filter-mapping>
<!-- These elements declare listeners that should be
    applied to the web application. Listeners must implement one of the
    EventListener interfaces defined in the javax.servlet and
    javax.servlet.http packages. -->
<listener>

```

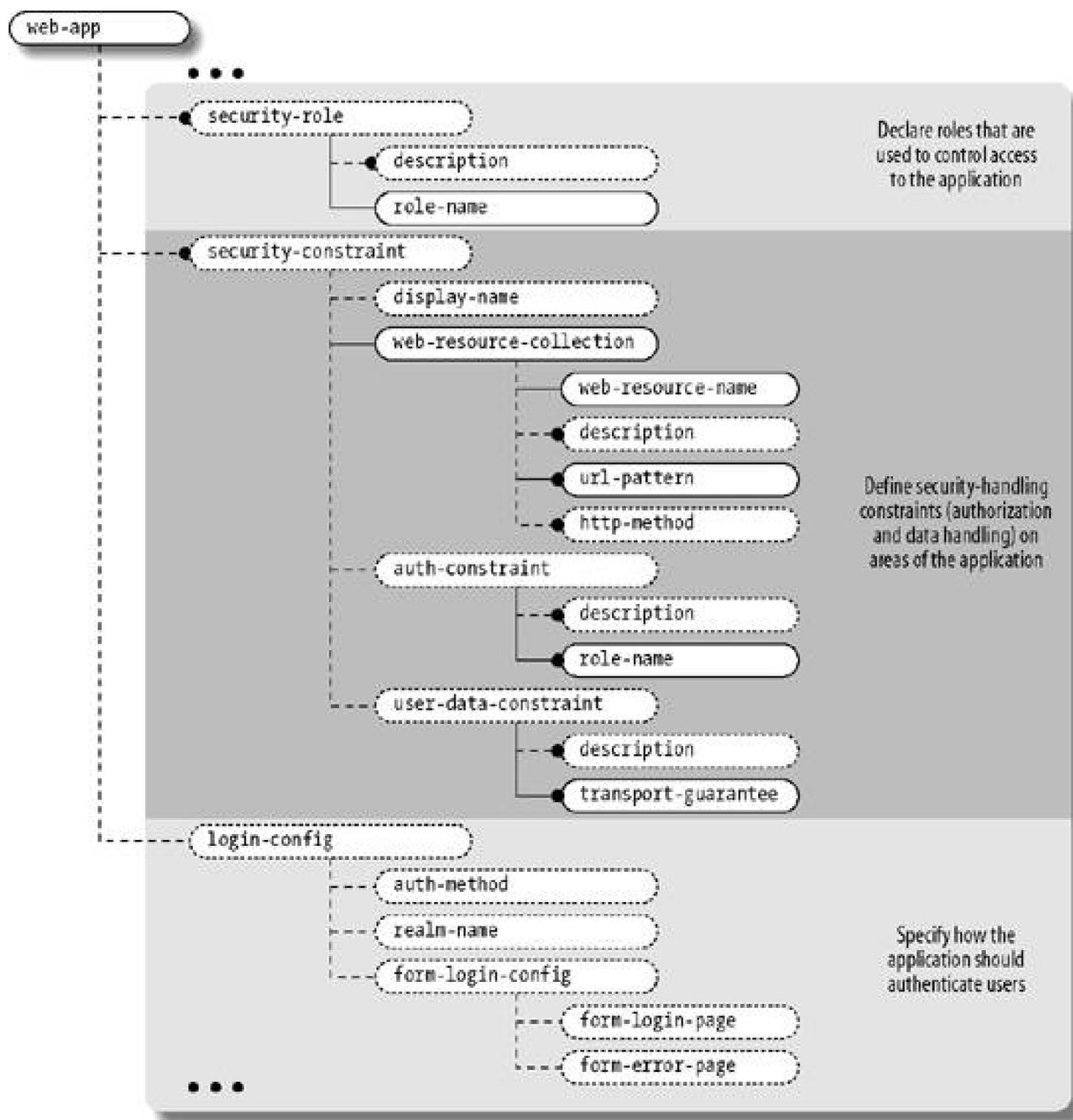
```
        <!-- The fully qualified class name of the listener. -->
        <listener-class>my.listener.classname</listener-class>
    </listener>
    . . .
</web-app>
```

A.1.4. Security Configuration

The deployment information includes the configuration of security constraints on the application. These include the declaration of abstract authorization roles, specifications of security levels (access and data protection) on subsets of the web application, and the manner in which users should be authenticated. [Figure A-5](#) shows the elements in this section of the deployment descriptor.

[Example A-5](#) shows sample security constraints annotated with descriptive comments.

Figure A-5. Security constraint elements



Example A-5. Annotated security constraints

```

<web-app . . . >
    . . .
    <!-- Each security-role element declares a security
         role used by web components in this archive. Each named role must be
         mapped to a physical group/identity within the runtime security
         environment of the web container/J2EE server.-->
    <security-role>
        <!-- A description of the role. -->
        <description>My admin role, used for super-users</description>
        <!-- Name of the role, as used in the rest of this descriptor. -->
        <role-name>admin</role-name>
    
```

```

</security-role>
<!-- Define security constraints on subsets of the application. The
application subset is defined by the web-resource-collection
element, and the constraints are defined in the auth-constraint
and user-data-constraint elements. -->
<security-constraint>
  <!-- Display name for the constraint. -->
  <display-name>Security setting for admin area</display-name>
  <!-- Define the domain of this constraint, using one or more
URL patterns, plus optional HTTP method filters. -->
  <web-resource-collection>
    <!-- Name for the resource. -->
    <web-resource-name>Admin-pages</web-resource-name>
    <!-- Description of the subset of the application. -->
    <description>The admin area of the application</description>
    <!-- One or more URL patterns that define the set of the
application being constrained. -->
    <url-pattern>/admin/*</url-pattern>
    <url-pattern>/admin-docs/*</url-pattern>
    <!-- One or more HTTP method filters, that further refine
the constraint to apply to only specific types of
requests. -->
    <http-method>GET</http-method>
    <http-method>POST</http-method>
  </web-resource-collection>
  <!-- Define the role(s) that are allowed to access this subset
of the application. -->
  <auth-constraint>
    <!-- Role name, must match an entry in the
security-role section -->
    <role-name>admin</role-name>
  </auth-constraint>
  <!-- Declare the degree to which data should be protected as
it is transmitted from the server to the client. -->
  <user-data-constraint>
    <!-- Description of the constraint. -->
    <description>Ensure admin ops are encrypted</description>
    <!-- Set the transport guarantee to one of NONE (no special
handling required), INTEGRAL (data must not be changed in
transit), and CONFIDENTIAL (data must not be observable
by outside parties during transit). -->
    <transport-guarantee>CONFIDENTIAL</transport-guarantee>
  </user-data-constraint>
</security-constraint>
<!-- This element is used to configure how authentication should be
handled by the web application. -->
<login-config>
  <!-- The type of authentication the web application
should use. Allowed values are "BASIC" (for HTTP Basic
authentication using the web server), "FORM" (to use your own
login forms), "DIGEST" (more secure but less-supported web
server login), and "CLIENT-CERT" (to use public-key

```

```

        authentication between the client and the server). -->
<auth-method>FORM</auth-method>
<!-- When BASIC authentication is chosen, this element
       indicates which realm, as defined in the web server and/or
       application server, to use for authentication. -->
<realm-name>app-users</realm-name>
<!-- If FORM authentication is chosen, this element
       provides the information on how form logins will be handled. -->
<form-login-config>
  <!-- The absolute location, as a path within the
       war file, of the page to be used for form-based login
       authentication. -->
  <form-login-page>/admin/login</form-login-page>
  <!-- The error page, where users will be sent
       when authentication fails. -->
  <form-error-page>/auth/error.jsp</form-error-page>
</form-login-config>
</login-config>
. . .
</web-app>

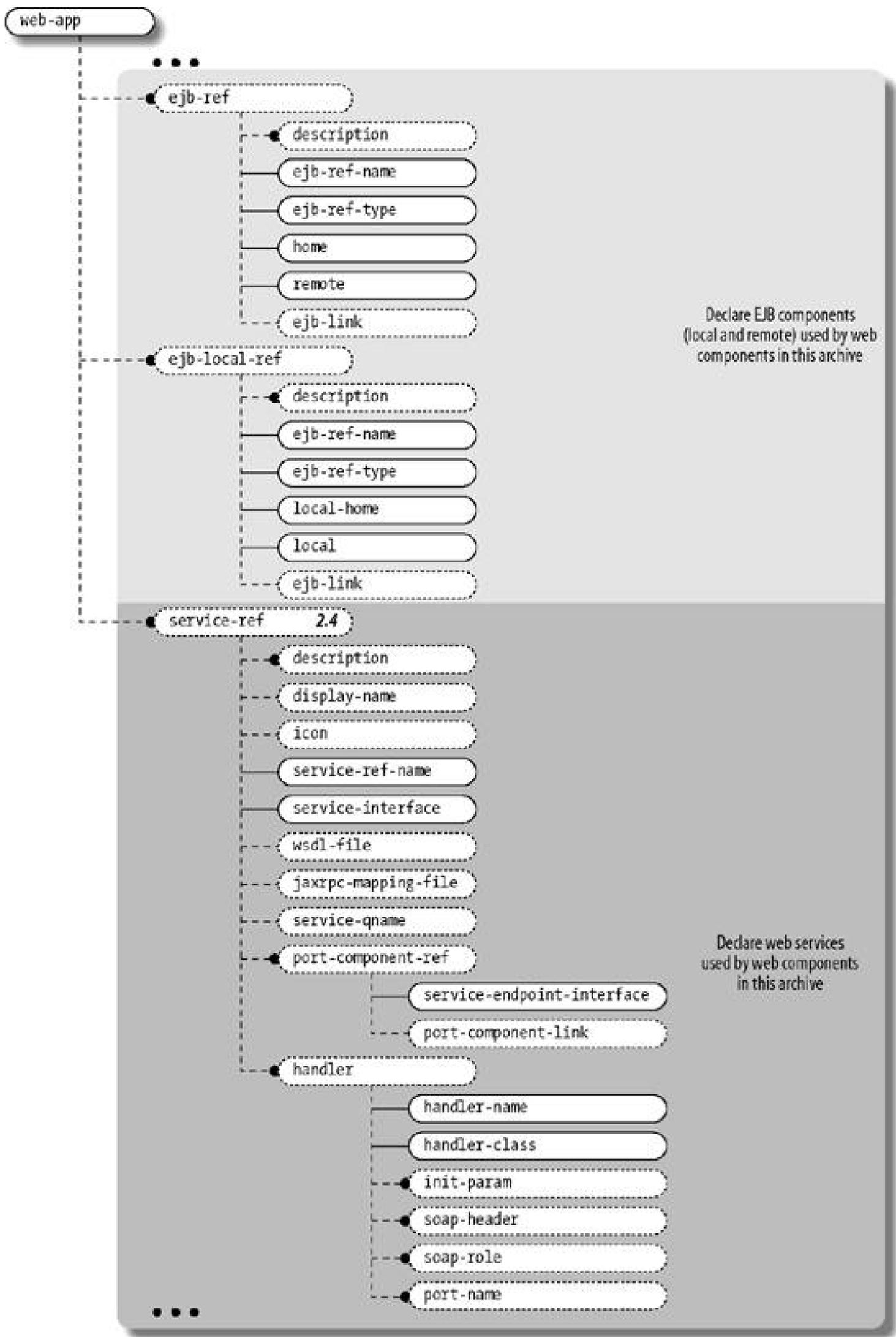
```

A.1.5. Component References

If your web components depend on other components (such as EJBs or web services), you declare these component references in the deployment descriptor to allow the container a chance to verify the availability of these dependencies. [Figure A-6](#) shows the deployment elements used for component references.

[Example A-6](#) shows sample component references, annotated with descriptive comments.

Figure A-6. Component reference elements



Example A-6. Annotated component references

```

<web-app . . .>
. . .
  <!-- An ejb-ref element is used to specify a remote
        reference to an EJB used by web components in this archive.  The
        ejb-local-ref element is used to declare a reference using local
        EJB interfaces.  The presence of an ejb-ref entry in a client's
        deployment descriptor is simply a placeholder that indicates that
        the client refers to this EJB using the indicated name, and
        therefore the expected EJB must be available.  At deploy time,
        if the EJB named here can't be resolved in the environment, then
        the deployment tools can catch this and raise a deployment-time
        error, avoiding a runtime error. -->
  <ejb-ref>
    <!-- Description of the EJB's use. -->
    <description>The cart EJB</description>
    <!-- The JNDI name used by the application client to look up
          the home interface for this EJB.  The name is relative to the
          java:comp/env context.  The J2EE specification
          recommends (but does not require) preceding JNDI references for
          EJBs with "ejb/".  This name has to resolve, eventually, to a real
          EJB home of the type described in this ejb-ref.  This means that
          the name used here has to be mapped (using the J2EE server's
          native tools) to a JNDI name that is the same as the deployed JNDI
          name of an EJB component, or an ejb-link element has to be given
          here that links this name directly to an EJB component in the same
          application archive (ear). -->
    <ejb-ref-name>ejb/ShoppingCartEJB</ejb-ref-name>
    <!-- The type of EJB referenced.  Allowed values are
          "Entity" or "Session" (message-driven beans have no home
          interfaces). -->
    <ejb-ref-type>Session</ejb-ref-type>
    <!-- The fully qualified class name of the remote home interface. -->
    <home>my.ejb.ShoppingCartHome</home>
    <!-- The fully qualified class name of the remote client interface -->
    <remote>my.ejb.ShoppingCart</remote>
    <!-- This element can be used to link this EJB reference to
          another EJB component in the same enclosing application jar file.
          The value of the link is the name of the EJB, as specified in the
          ejb-name element in its deployment descriptor.  You can also
          prepend the linked name with a relative path to the ejb-jar file
          of the target EJB, followed by a '#', e.g.,
          "../ejbs/dataEJBs.jar#Person".  This can be useful when there are
          multiple EJBs with the same ejb-name in the same application.
          If no ejb-link element is provided, then the EJB reference will
          simply be resolved directly to the JNDI name implied by the
          ejb-ref-name, e.g., "java:comp/env/ejb/name".  An EJB home of the
          expected type should be deployed to that name in order for
          the reference to be resolved correctly. -->

```

```

    <ejb-link>ShoppingCart-remote</ejb-link>
</ejb-ref>
<!-- Similar to the ejb-ref element, except this reference uses local
client and home interfaces, for those cases where the EJB is
running within the same application server instance -->
<ejb-local-ref>
  <!-- JNDI reference name. -->
  <ejb-ref-name>ejb/ShoppingCartEJB-local</ejb-ref-name>
  <!-- Type of EJB being referenced. -->
  <ejb-ref-type>Session</ejb-ref-type>
  <!-- Local home interface class name. -->
  <local-home>my.ejb.ShoppingCartLocalHome</local-home>
  <!-- Local client interface. -->
  <local>my.ejb.ShoppingCartLocal</local>
  <!-- Link this reference to an EJB in the same application archive. -->
  <ejb-link>ShoppingCart-local</ejb-link>
</ejb-local-ref>
<!-- Declare a dependence on a JAX-RPC interface to a SOAP web service.
At a minimum you need to specify a reference name and the JAX-RPC
interface to the service, but you can also specify a number of other
details. -->
<service-ref>
  <description>Neat web service</description>
  <display-name>Reference to web service X</display-name>
  <icon>
    <small-icon>/icons/my-small-icon.jpg</small-icon>
    <large-icon>/icons/my-large-icon.jpg</large-icon>
  </icon>
  <!-- JNDI name used by the web components to reference the
JAX-RPC interface to the service. -->
  <service-ref-name>service/MyWebService</service-ref-name>
  <!-- The interface/class that the service interface is expected
to satisfy. This can be a custom interface, perhaps one
mapped from the service WSDL, or it can simply be the JAX-RPC
Service interface, which all service interfaces must implement.
The container is responsible for verifying that the service
interface actually implements the interface given here. -->
  <service-interface>javax.xml.rpc.Service</service-interface>
  <!-- The path (within the web archive) to the WSDL for the service. -->
  <wsdl-file>WEB-INF/wsdl/myservice.wsdl</wsdl-file>
  <!-- The path (within the web archive) to the JAX-RPC mapping file,
specifying how to map between XML entities and Java entities
for the service. -->
  <jaxrpc-mapping-file>WEB-INF/myservice-mapping.xml</jaxrpc-mapping-file>
  <!-- The qualified name for the service within the WSDL file -->
  <service-qname>com.mypartner:cool-service</service-qname>
  <!-- Declare that the web components require a specific port from
the service to be mapped to a JAX-RPC interface. -->
  <port-component-ref>
    <!-- The interface that the mapped port must implement. -->
    <service-endpoint-interface>my.service.endpoint
  </service-endpoint-interface>

```

```

        <!-- You can optionally link this port reference to a JAX-RPC
            port component within the same application archive. The
            link value should match a port-component-name in a
            deployed webservices.xml deployment descriptor in one
            of the web or EJB archives in this application. -->
        <port-component-link>MyServicePort</port-component-link>
    </port-component-ref>
    <!-- The handler element is used to specify handlers for
        this service. -->
    <handler>
        <!-- A name for the handler. -->
        <handler-name>MySOAPHandler</handler-name>
        <!-- The full class name of the handler. The class must
            implement the javax.xml.rpc.handler.Handler interface -->
        <handler-class>my.soap.handler</handler-class>
        <!-- Specify initialization parameters, passed into the
            handler's init( ) method as part of the HandlerInfo -->
        <init-param>
            <!-- Name of the parameter. -->
            <param-name>param1</param-name>
            <!-- Value of the parameter. -->
            <param-value>value1</param-value>
        </init-param>
        <!-- The SOAP actor role that the handler should play as it
            processes the SOAP request. -->
        <soap-role>urn:my-service</soap-role>
        <!-- The qualified name of the SOAP header to be processed
            by the handler. -->
        <soap-header>com.partner:header1</soap-header>
        <!-- Name of the WSDL port for the handler.-->
        <port-name>service-port-x</port-name>
    </handler>
</service-ref>
. . .
</web-app>

```

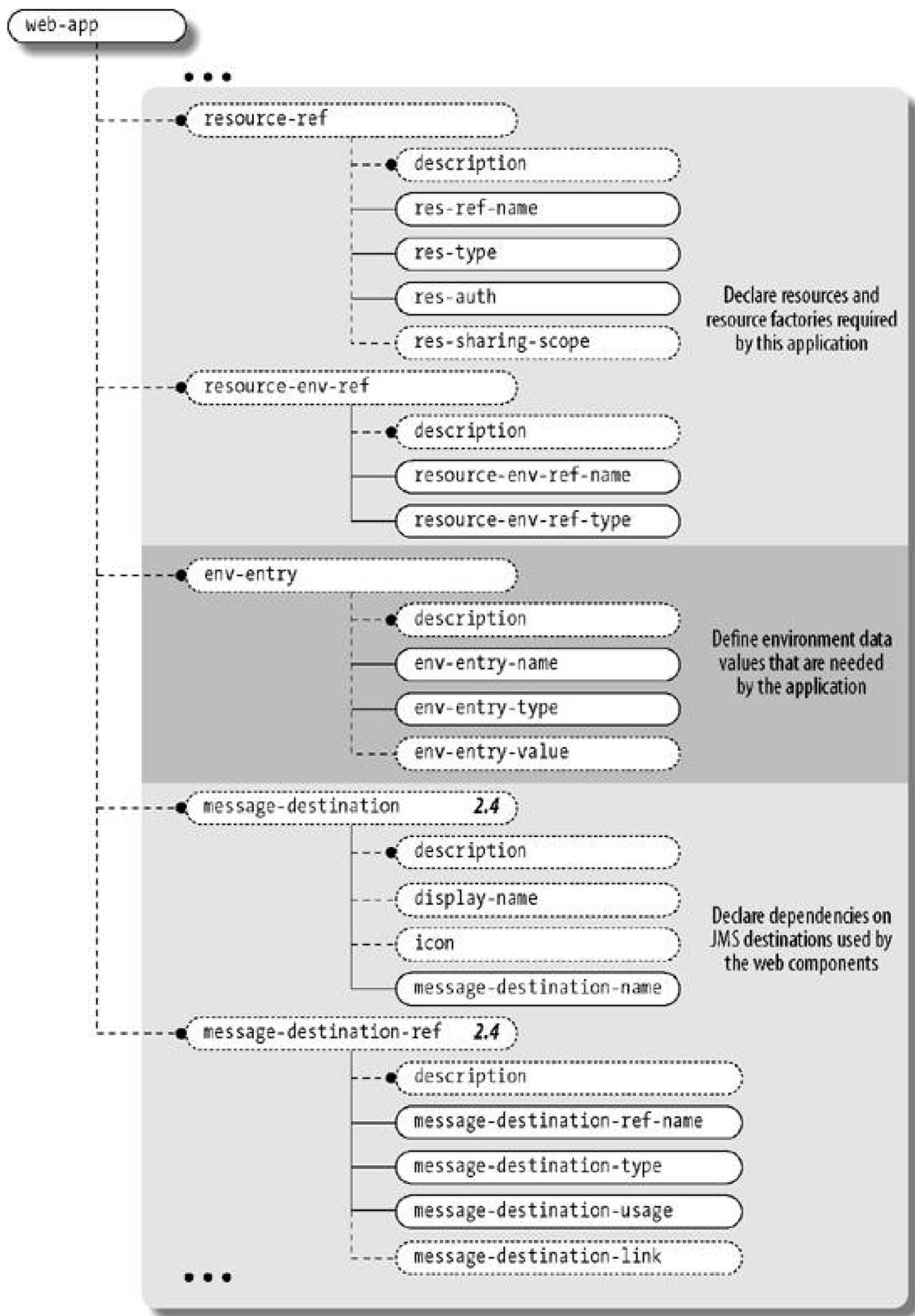
A.1.6. Resources and Environment Entries

When web components require various external resources and environment entries, they are declared in the deployment descriptor using the elements shown in [Figure A-7](#). The resources that can be declared include resource factories that generate connections to services (e.g., a JDBC `DataSource` that generates `Connections` to databases or JMS `ConnectionFactory` objects that generate `Connections` to JMS messaging services), resources themselves (e.g., URLs or JMS destinations), or basic environment data (e.g., string or numeric values, or basic data objects). All of the entities declared in this section of the descriptor are accessed at runtime through JNDI lookups and therefore need to be present in the server's JNDI service when the application is deployed.

[Example A-7](#) shows sample resource declarations and environment entries, annotated with

descriptive comments.

Figure A-7. Elements for resource and environment entries



Example A-7. Annotated resource and environment entries

```

<web-app . . .>
. . .
<!-- A reference to a resource factory used by the
web components. A resource factory is used to create connections
to a managed service. Examples of factories configurable using a
resource-ref are JDBC DataSources, JMS ConnectionFactories, and
JavaMail Sessions. You can also configure your own custom factories.
The presence of a resource-ref entry in a deployment descriptor is
simply a placeholder that indicates that the client refers to this
resource using the indicated name, and therefore the expected
resource must be deployed in the application server at the specified
JNDI name. At deploy time, if the resource named here can't be
resolved in the environment, then the container can catch this and
raise a deployment-time error, avoiding a runtime error. -->
<resource-ref>
  <!-- Description of the resource factory. -->
  <description>JMS connection factory for message service</description>
  <!-- JNDI name, relative to java:comp/env, used to look
up the resource factory. -->
  <res-ref-name>jms/notify-connection</res-ref-name>
  <!-- Class of the resource factory. -->
  <res-type>javax.jms.QueueConnectionFactory</res-type>
  <!-- Authorization model for the factory, either Container or
Application. -->
  <res-auth>Container</res-auth>
  <!-- This entry specifies whether connections obtained from the
resource factory are shareable (Shareable)
or not (Unshareable). -->
  <res-sharing-scope>Shareable</res-sharing-scope>
</resource-ref>
<!-- A reference to a general managed object used by the web components.
A resource-env-ref was used prior to J2EE 1.4 to declare JMS
destinations. In J2EE 1.4, JMS destinations are declared in the
message-destination and message-destination-ref elements below. -->
<resource-env-ref>
  <!-- Description of the env. resource. -->
  <description>Message queue for notifications</description>
  <!-- JNDI name, relative to java:comp/env, used to look up the
resource in the environment. -->
  <resource-env-ref-name>jms/notify-queue</resource-env-ref-name>
  <!-- Java type of the resource. -->
  <resource-env-ref-type>javax.jms.Queue</resource-env-ref-type>
</resource-env-ref>
<!-- Used to create environment entries that the
application client can access at runtime through JNDI in the
java:comp/env context. -->
<env-entry>
  <!-- Description of the environment entry. -->
  <description>Message notification alert threshold</description>
  <!-- JNDI name, relative to java:comp/env, used to look

```

```

        up the env entry. -->
<env-entry-name>email-alert-threshold</env-entry-name>
<!-- The Java type of the entry value.  The only types
      allowed for application client entries are Boolean, Byte,
      Character, String, Short, Integer, Long, Float, Double. -->
<env-entry-type>java.lang.Integer</env-entry-type>
<!-- The value of the entry.  The string used here will be used
      to initialize an instance of the type specified in
      the env-entry-type element, by passing it into its
      constructor. -->
<env-entry-value>12</env-entry-value>
</env-entry>
<!-- Declare names for message destinations that will be used to resolve
      the message destination references below.  The message destinations
      declared here must be resolved during deployment to actual, physical
      JMS destinations that are appropriate for the usage declared in the
      corresponding message-destination-ref elements. -->
<message-destination>
  <description>Another queue used for notifications</description>
  <display-name>Another notify queue</display-name>
  <icon>
    <small-icon>icons/my-small-jms-icon.jpg</small-icon>
    <large-icon>icons/my-large-jms-icon.jpg</large-icon>
  </icon>
  <!-- Declare a logical name for the message destination. -->
  <message-destination-name>my-notify-queue</message-destination-name>
</message-destination>
<!-- Declare a reference to a message destination, as it is used within
      the components in this archive.  This maps a JNDI name to one of the
      message destinations declared above, and specifies the type of
      destination expected and how the components use the destination. -->
<message-destination-ref>
  <!-- Specify the JNDI name (relative to java:comp/env) used by the
        components to reference this destination. -->
  <message-destination-ref-name
    >jms/notify-queue2</message-destination-ref-name>
  <!-- Specify the type of destination expected (Queue or Topic). -->
  <message-destination-type>javax.jms.Queue</message-destination-type>
  <!-- Specify how the components use the destination, in terms of
        message direction (Consumes, Produces, or ConsumesProduces).-->
  <message-destination-usage>Produces</message-destination-usage>
  <!-- A link to a message destination, using its logical name. -->
  <message-destination-link>my-notify-queue</message-destination-link>
</message-destination-ref>
  .
  .
  .
</web-app>

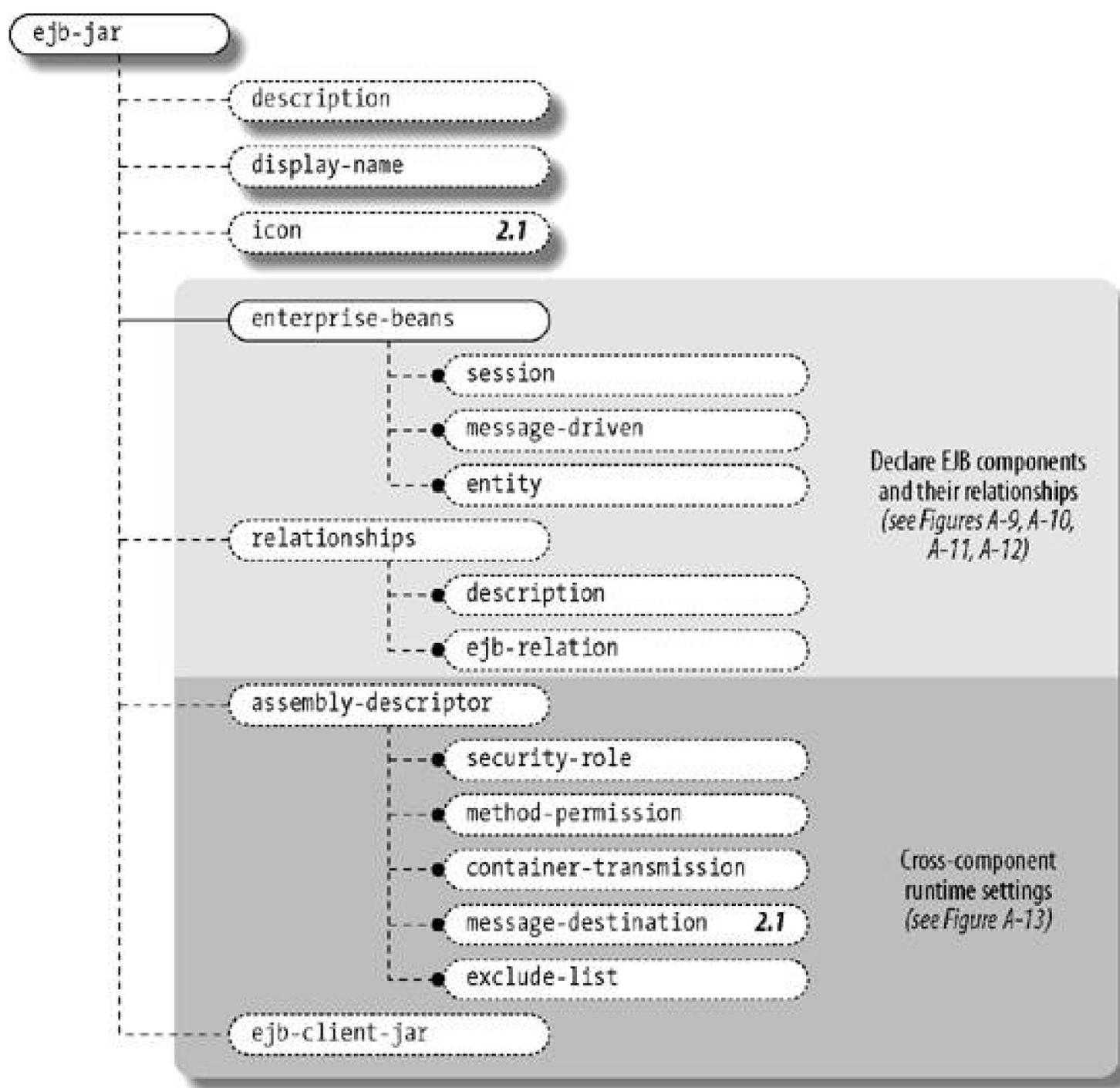
```

A.2. Enterprise JavaBeans (ejb-jar.xml)

Enterprise JavaBean components are deployed using a *ejb-jar.xml* deployment descriptor. The high-level descriptor summary for EJB deployment descriptors is shown in [Figure A-8](#). The following sections provide details on each section of the descriptor. In all the diagrams in this section, deployment elements that were introduced in the EJB 2.1 specification are labelled with a "2.1" indicator. All other elements were present in the 2.0 (and possibly earlier) version of the EJB specification.

The format for an empty *ejb-jar.xml* file is shown in [Example A-8](#).

Figure A-8. EJB deployment descriptor overview



Example A-8. EJB deployment descriptor header

```
<ejb-jar 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/ejb-jar_2_1.xsd"
  version="2.1">
  . . .
</ejb-jar>
```

The example references the XML schema for the EJB 2.1 deployment descriptor from the default location on the java.sun.com server. To reference the EJB 2.0 DTD instead (e.g., if you are using a J2EE 1.3 application server like JBoss 3.x or an EJB 2.0 container plug-in), you should start your deployment descriptor with the following `DOCTYPE` element and no schema reference:

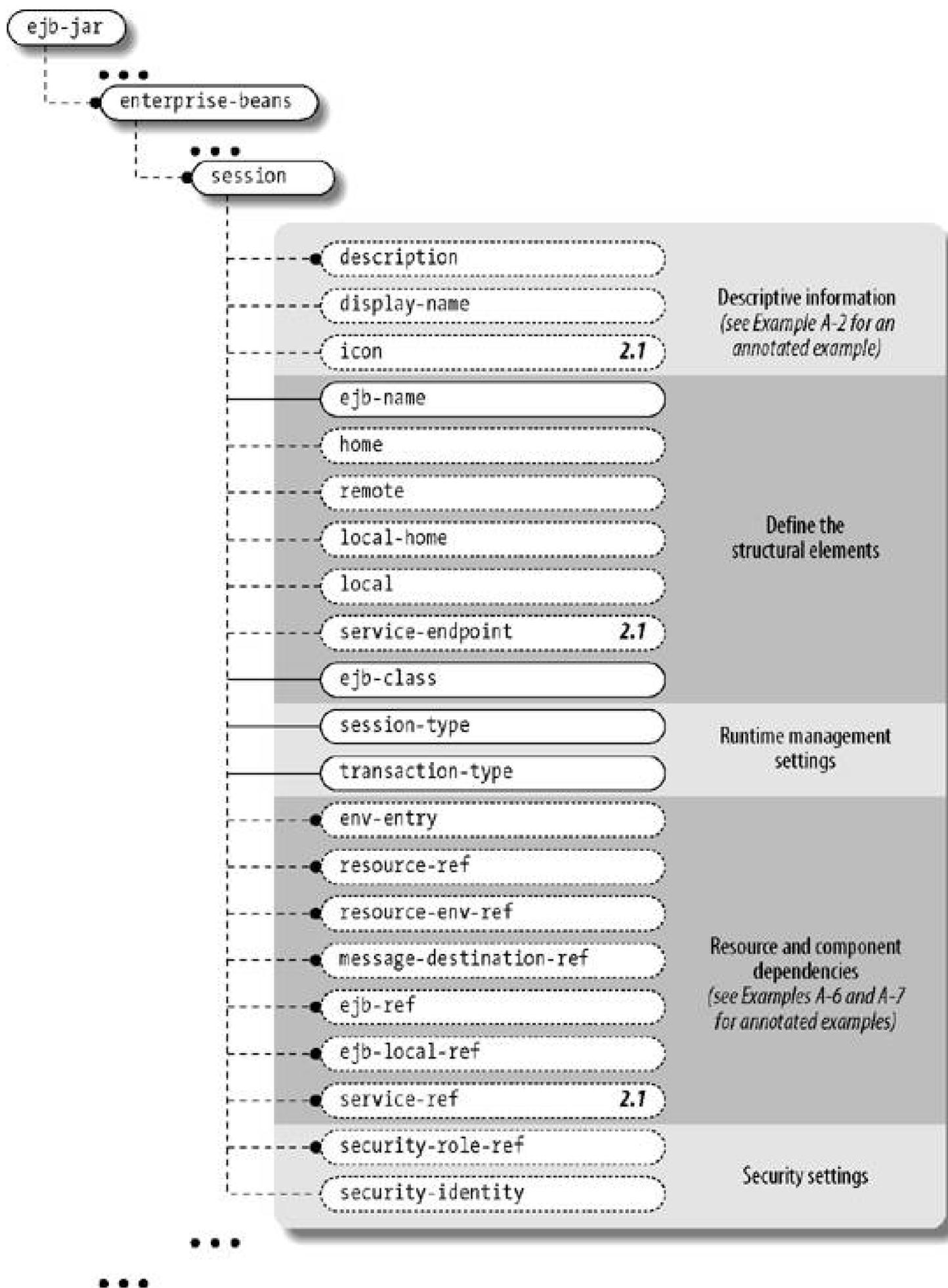
```
<!DOCTYPE ejb-jar PUBLIC
    "-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans 2.0//EN"
    "http://java.sun.com/dtd/ejb-jar_2_0.dtd">
<ejb-jar>
    . . .
</ejb-jar>
```

A.2.1. Session EJB Declarations

[Figure A-9](#) shows the elements used to declare session EJBs in your deployment descriptor.

[Example A-9](#) shows sample session EJB declarations, annotated with descriptive comments.

Figure A-9. Session EJB declarations



Example A-9. Annotated session EJB declarations

`<ejb-jar . . . >`

```

. . .
<enterprise-beans>
  <!-- Declare session EJBs included in this archive. -->
  <session>
    <!-- A description of the session EJB being declared. -->
    <description>desc</description>
    <!-- A display name to be used in various tools. -->
    <display-name>name</display-name>
    <!-- Icon to display in tools. -->
    <icon>
      <small-icon>my/icon/file</small-icon>
      <large-icon>my/icon/file</large-icon>
    </icon>
    <!-- A unique name to be used to refer to this EJB component,
         either within this DD or from other deployment
         descriptors in the same application assembly. Note that
         this name is not a JNDI name, it is simply a label used
         to refer to this EJB from other descriptor elements. -->
    <ejb-name>MySessionEJB</ejb-name>
    <!-- The fully qualified class name of the remote home
         interface for this EJB. A session EJB must have either
         a remote or local home interface, but doesn't require
         both. If an EJB has a remote home interface, it must
         have a remote client interface as well. -->
    <home>my.home.interface</home>
    <!-- The remote client interface for this EJB. A session EJB
         must have either a remote or local client interface,
         but doesn't require both. If an EJB has a remote home
         interface, it must have a remote client interface as
         well. -->
    <remote>my.remote.interface</remote>
    <!-- This specifies the fully qualified class name of the
         local home interface for this EJB. A session EJB must
         have either a remote and/or local home interface, but
         doesn't require both. If an EJB has a local home
         interface, it must have a local client interface as
         well. -->
    <local-home>my.local.home.interface</local-home>
    <!-- This declares the fully qualified class name for the
         local client interface for this EJB. A session EJB
         must have either a remote or local client interface,
         but doesn't require both. If an EJB has a local home
         interface, it must have a local client interface as
         well. -->
    <local>my.local.interface</local>
    <!-- If you want to expose a SOAP service interface to your
         session bean, you must indicate here the JAX-RPC service
         interface your service will implement. Only stateless
         session beans can be published as web services. You'll also
         need to include a webservices.xml deployment descriptor
         with your EJB archive, to fully describe the web service
         (see section "Web Services - (webservices.xml)"). -->

```

```

<service-endpoint>my.jaxrpc.service.interface</service-endpoint>
<!-- The fully qualified name of the implementation class for
the EJB. -->
<ejb-class>my.impl.class</ejb-class>
<!-- The type of the session bean. Allowed values are
"Stateful" or "Stateless". The EJB container uses the
session-type to determine what type of lifecycle
management to apply to the bean. It also allows the
server's deployment tools to validate the EJB (e.g.,
check its interfaces for required callbacks). -->
<session-type>Stateless</session-type>
<!-- The type of transaction management used by this EJB. The
allowed values are "Bean" and "Container". The EJB
container uses this value to determine what transaction
management rules to apply to the EJB at runtime. -->
<transaction-type>Container</transaction-type>
<!-- Descriptions of the env-entry, ejb-ref, ejb-local-ref,
resource-ref, and resource-env-ref elements can be
found in Examples A-6 and A-7. -->
<env-entry>
  <description>This env entry ...</description>
  <env-entry-name>env-var</env-entry-name>
  <env-entry-type>java.lang.String</env-entry-type>
  <env-entry-value>var-value</env-entry-value>
</env-entry>
<ejb-ref>
  <description>A remote EJB I use...</description>
  <ejb-ref-name>ejb/remoteEJB</ejb-ref-name>
  <ejb-ref-type>Entity</ejb-ref-type>
  <home>other.ejb.home.interface</home>
  <remote>other.ejb.client.interface</remote>
  <ejb-link>other-ejb-name</ejb-link>
</ejb-ref>
<ejb-local-ref>
  <description>A local EJB I use...</description>
  <ejb-ref-name>ejb/localEJB</ejb-ref-name>
  <ejb-ref-type>Entity</ejb-ref-type>
  <local-home>other.ejb.localhome.interface</local-home>
  <local>other.ejb.localclient.interface</local>
  <ejb-link>other-ejb-name2</ejb-link>
</ejb-local-ref>
<resource-ref>
  <description>A resource factory I use...</description>
  <res-ref-name>jdbc/MyDB</res-ref-name>
  <res-type>javax.sql.DataSource</res-type>
  <res-auth>Container</res-auth>
  <res-sharing-scope>Shareable</res-sharing-scope>
</resource-ref>
<resource-env-ref>
  <description>A resource I use...</description>
  <resource-env-ref-name>jms/MyQueue</resource-env-ref-name>
  <resource-env-ref-type>javax.jms.Queue

```

```

        </resource-env-ref-type>
    </resource-env-ref>
    <!-- A security-role-ref element declares a security
         role name that is used in this EJB component. Its use
         here is similar to its use in web components (see Example
         A-3). In EJBs, the role named here is used in a call
         to isCallerInRole( ) on the EJBContext. There must be a
         corresponding security-role element in the
         assembly-descriptor section of the descriptor, matching
         either the role-link value (if present) or the role-name
         (if no role-link is present). -->
    <security-role-ref>
        <!-- A text description of the role. -->
        <description>My admin role...</description>
        <!-- The role name as used in the EJB code. -->
        <role-name>admin</role-name>
        <!-- The role name referenced in the code can be linked
             to another security-role element (see the
             assembly-descriptor in Example A-13). If a
             role-link is not present, then the name used in the
             role-name must be declared as a security-role in the
             assembly-descriptor. -->
        <role-link>main-admin-role</role-link>
        <!-- -->
    </security-role-ref>
    <!-- The security-identity element is used to specify what
         identity is used to execute the methods of the EJB.
         If the use-caller-identity child element is used, then
         the identity of the caller is used to execute the EJB
         methods, and this identity is passed on to any EJB or
         other resources the EJB methods may invoke. If the run-as
         element is used, then the specified role name is used to
         execute the methods. -->
    <security-identity>
        <run-as>
            <role-name>admin</role-name>
        </run-as>
    </security-identity>
</session>
    . . .
</enterprise-beans>
    . . .
</ejb-jar>

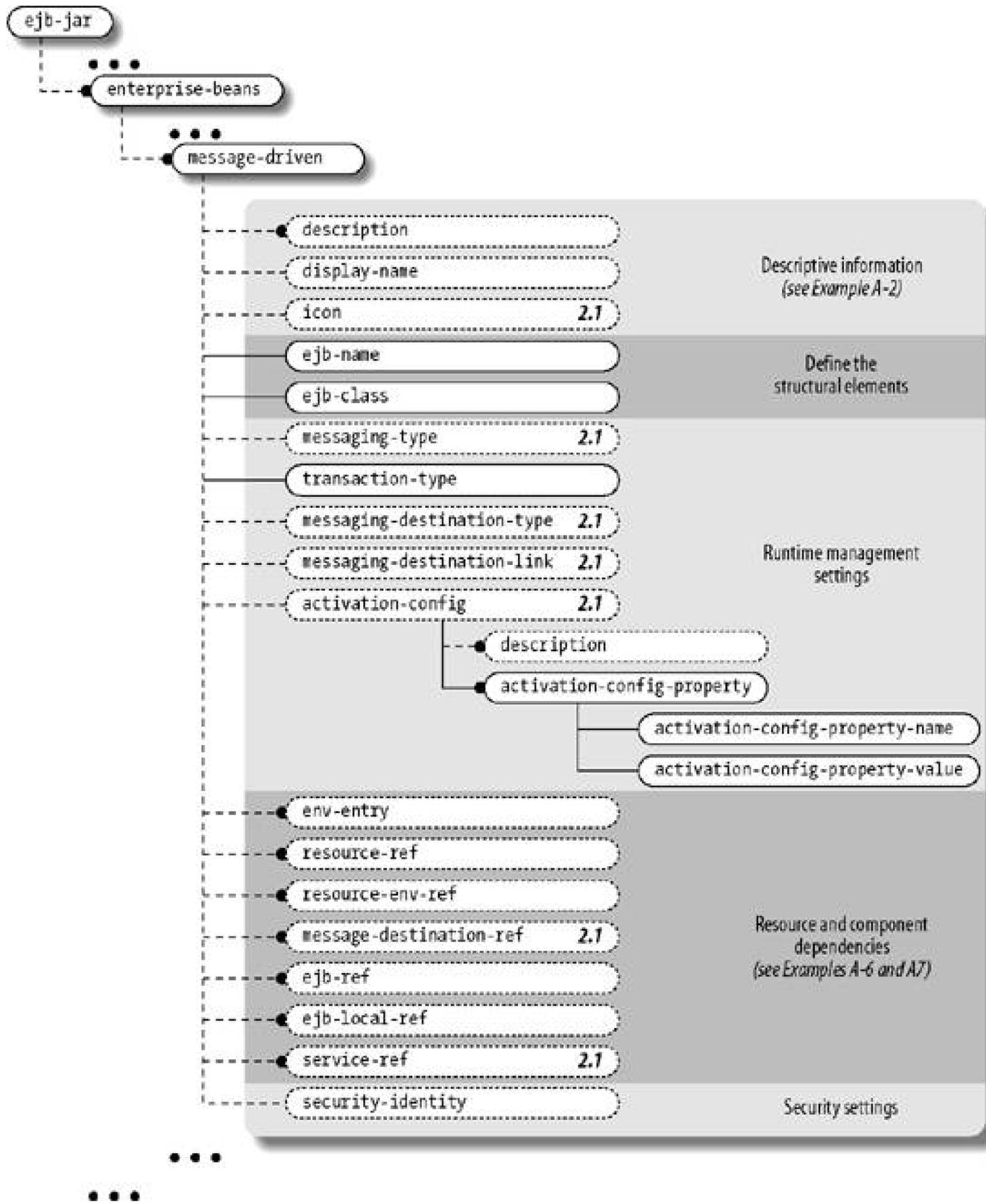
```

A.2.2. Message-Driven EJB Declarations

[Figure A-10](#) shows the elements used to declare message-driven beans.

[Example A-10](#) shows sample message-driven EJB declarations, annotated with descriptive comments

Figure A-10. Message-driven EJB declarations



Example A-10. Annotated message-driven EJB declarations

```
<ejb-jar . . . >
```

```

. . .
<enterprise-beans>
. . .
<!-- A message-driven element declares a message-driven EJB
      contained in this archive. -->
<message-driven>
  <description>desc</description>
  <display-name>name</display-name>
  <icon>
    <small-icon>my/icon/file</small-icon>
    <large-icon>my/icon/file</large-icon>
  </icon>
  <!-- For descriptions of the following elements, see the
        session element in Example A-9. -->
  <ejb-name>MyMessageEJB</ejb-name>
  <ejb-class>my.impl.class</ejb-class>
  <messaging-type>javax.jms.MessageListener</messaging-type>
  <transaction-type>Container</transaction-type>
  <!-- The activation-config specifies parameters to be used
        when initializing the message destination associated with
        this message-driven bean. The standard properties that can
        be set for JMS-based EJBs are acknowledgeMode,
        messageSelector, destinationType, and
        subscriptionDurability. Specific application servers may
        support other property values in their messaging
        implementations (especially non-JMS ones), but you can't
        expect these to be honored by other app servers. -->
  <activation-config>
    <!-- The messageSelector property specifies a message
          selector to be used to filter messages received by
          the bean. The value must be a valid JMS message
          selector expression -- see Appendix E for the
          syntax of message selectors. -->
    <activation-config-property>
      <activation-config-property-name
        >messageSelector</activation-config-property-name>
      <activation-config-property-value>
        X = 'a' AND Y = 'b'
      </activation-config-property-value>
    </activation-config-property>
    <!-- The acknowledgeMode property declares how the bean
          acknowledges receipt of messages, if the EJB uses
          bean-managed transaction management (specified in
          the transaction-type element). The allowed values
          are "Auto-acknowledge" or "Dups-ok-acknowledge",
          corresponding to the JMS acknowledge types. If this
          element is not present, then "Auto-acknowledge"
          is assumed. -->
    <activation-config-property>
      <activation-config-property-name>
        acknowledgeMode</activation-config-property-name>

```

```

        <activation-config-property-value>
            Auto-acknowledge</activation-config-property-value>
    </activation-config-property>
    <!-- The destinationType property is a hint to the EJB
        container as to what kind of JMS destination this
        bean should be associated with. The value should be
        full class name of the JMS destination type. -->
    <activation-config-property>
        <activation-config-property-name>
            destinationType</activation-config-property-name>
        <activation-config-property-value>
            javax.jms.Queue</activation-config-property-value>
    </activation-config-property>
    <!-- If a topic destination is specified, this element
        indicates whether a durable subscription should be
        used for this EJB or not. Allowed values are
        "Durable" or "NonDurable". If no value is provided,
        then the durability of the subscription is at the
        discretion of the EJB container. -->
    <activation-config-property>
        <activation-config-property-name>subscriptionDurability
        </activation-config-property-name>
        <activation-config-property-value>
            Durable</activation-config-property-value>
    </activation-config-property>
</activation-config>
<!-- For full descriptions of the env-entry, ejb-ref,
    ejb-local-ref, resource-ref, and resource-env-ref elements,
    see Examples A-6 and A-7. -->
<env-entry>
    <description>entry desc</description>
    <env-entry-name>name</env-entry-name>
    <env-entry-type>java.lang.String</env-entry-type>
    <env-entry-value>value</env-entry-value>
</env-entry>
<ejb-ref>
    <description>desc</description>
    <ejb-ref-name>ejb/name</ejb-ref-name>
    <ejb-ref-type>Entity</ejb-ref-type>
    <home>homeClassName</home>
    <remote>remoteClass</remote>
    <ejb-link>ejbName</ejb-link>
</ejb-ref>
<ejb-local-ref>
    <description>desc</description>
    <ejb-ref-name>ejb/name</ejb-ref-name>
    <ejb-ref-type>Entity</ejb-ref-type>
    <local-home>homeClassName</local-home>
    <local>remoteClass</local>
    <ejb-link>ejbName</ejb-link>
</ejb-local-ref>
<resource-ref>

```

```

        <description>desc</description>
        <res-ref-name>jdbc/name</res-ref-name>
        <res-type>javax.sql.DataSource</res-type>
        <res-auth>Container</res-auth>
        <res-sharing-scope>Shareable</res-sharing-scope>
    </resource-ref>
    <resource-env-ref>
        <description>desc</description>
        <resource-env-ref-name>context/name</resource-env-ref-name>
        <resource-env-ref-type>
            javax.jms.Queue</resource-env-ref-type>
    </resource-env-ref>
    <!-- For a full description of the security-identity element,
         see Example A-9. -->
    <security-identity>
        <run-as>
            <role-name>admin</role-name>
        </run-as>
    </security-identity>
</message-driven>
    . . .
</enterprise-beans>
    . . .
</ejb-jar>

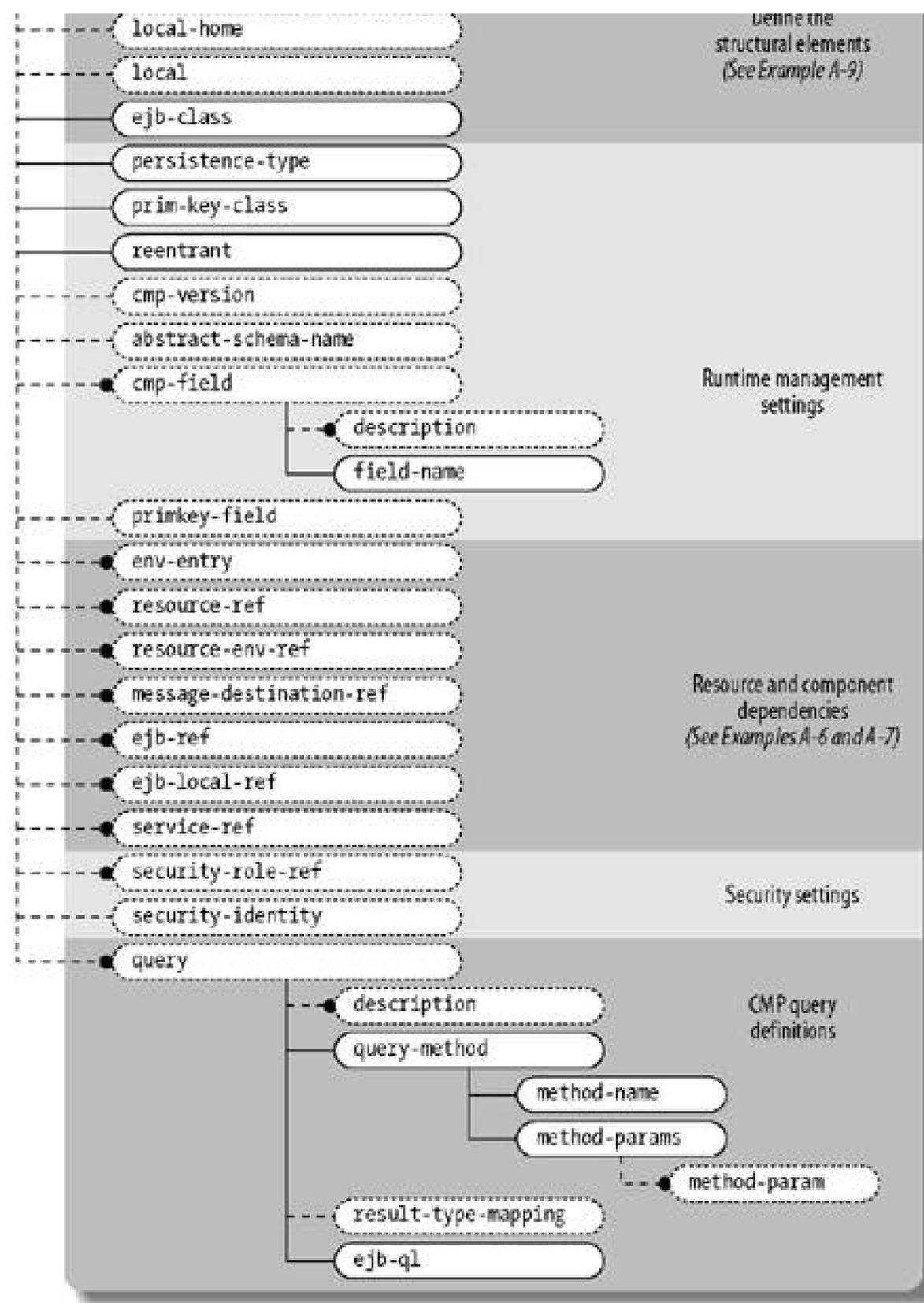
```

A.2.3. Entity EJB Declarations

[Figure A-11](#) shows the descriptor elements used to declare entity EJBs.

[Example A-11](#) shows sample entity EJB declarations, annotated with descriptive comments.

Figure A-11. Entity EJB declarations



Example A-11. Annotated entity EJB declarations

```

<ejb-jar . . . >
. . .
<enterprise-beans>
. . .
<!-- Each entity element declares an entity EJB contained in the
archive. In addition to the basic EJB details (home, client,
implementation classes, etc.), the elements here describe how

```

persistence should be managed for the entity bean, and for CMP, details on the abstract schema elements to be used for this EJB. -->

```

<entity>
  <description>My entity EJB</description>
  <display-name>my-entity-ejb</display-name>
  <!-- Icon for tools. -->
  <icon>
    <small-icon>my/icon/file</small-icon>
    <large-icon>my/icon/file</large-icon>
  </icon>
  <!-- For descriptions of the following elements, see the
        session element in Example A-9. -->
  <ejb-name>MyEntityEJB</ejb-name>
  <home>my.home.interface</home>
  <remote>my.remote.interface</remote>
  <local-home>my.local.home.interface</local-home>
  <local>my.local.interface</local>
  <ejb-class>my.impl.class</ejb-class>
  <!-- Indicate how the persistence for the entity EJB is managed.
        Allowed options are "Bean" or "Container". In you use
        bean-managed persistence, then you must provide the
        necessary persistence logic code in your EJB class. If
        you use container-managed persistence, there are
        additional deployment elements that must be defined
        below. -->
  <persistence-type>Container</persistence-type>
  <!-- The fully qualified class name of the type used for this
        EJB's primary key. -->
  <prim-key-class>classname</prim-key-class>
  <!-- Indicates whether the EJB's methods are reentrant or
        not. -->
  <reentrant>>false</reentrant>
  <!-- Indicates the version of container-managed persistence
        to be used for this EJB. Allowed values are "1.x",
        indicating CMP as defined in the EJB 1.1 specification,
        or "2.x", indicating CMP as defined in EJB 2.0/2.1. If this
        element isn't present, an EJB 2.x container will assume
        the value is 2.x. This element should be used only with
        EJBs that specify container-managed persistence in the
        persistence-type above. -->
  <cmp-version>2.x</cmp-version>
  <!-- If CMP is used, this element specifies the abstract
        "table" name for the EJB, to be used in EJB-QL queries. -->
  <abstract-schema-name>MyEntity</abstract-schema-name>
  <!-- If CMP is used, these elements declare the container-
        managed fields present on the EJB. -->
  <cmp-field>
    <!-- A description of the field. -->
    <description>A persisted field...</description>
    <!-- The name of the CMP field. In EJB 2.x CMP, the field
        name is an abstract "column" name, used in EJB-QL

```

```

        queries. It also implies the presence of
        JavaBean-style get/set accessors for this field on
        the EJB implementation class. -->
        <field-name>myPersistentField</field-name>
</cmp-field>
<!-- If CMP is used and if the EJB has a single-field primary
key, this element is used to specify which cmp-field is
the EJB's primary key. If the EJB has multiple fields
in its primary key, then a primary key class must be
used, and the primary key class (declared in the
prim-key-class element above) must have public data
members that map to cmp-fields on this EJB. -->
<primkey-field>primkey-fieldname</primkey-field>
<!-- For descriptions of the env-entry, ejb-ref, ejb-local-ref,
resource-ref, and resource-local-ref elements, see
Examples A-6 and A-7.-->
<env-entry>
    <description>This env entry ...</description>
    <env-entry-name>env-var</env-entry-name>
    <env-entry-type>java.lang.String</env-entry-type>
    <env-entry-value>var-value</env-entry-value>
</env-entry>
<ejb-ref>
    <description>A remote EJB I use...</description>
    <ejb-ref-name>ejb/remoteEJB</ejb-ref-name>
    <ejb-ref-type>Entity</ejb-ref-type>
    <home>other.ejb.home.interface</home>
    <remote>other.ejb.client.interface</remote>
    <ejb-link>other-ejb-name</ejb-link>
</ejb-ref>
<ejb-local-ref>
    <description>A local EJB I use...</description>
    <ejb-ref-name>ejb/localEJB</ejb-ref-name>
    <ejb-ref-type>Entity</ejb-ref-type>
    <local-home>other.ejb.localhome.interface</local-home>
    <local>other.ejb.localclient.interface</local>
    <ejb-link>other-ejb-name2</ejb-link>
</ejb-local-ref>
<resource-ref>
    <description>A resource factory I use...</description>
    <res-ref-name>jdbc/MyDB</res-ref-name>
    <res-type>javax.sql.DataSource</res-type>
    <res-auth>Container</res-auth>
    <res-sharing-scope>Shareable</res-sharing-scope>
</resource-ref>
<resource-env-ref>
    <description>A resource I use...</description>
    <resource-env-ref-name>jms/MyQueue</resource-env-ref-name>
    <resource-env-ref-type>javax.jms.Queue
    </resource-env-ref-type>
</resource-env-ref>
<!-- For descriptions of the security-role-ref and

```

```

        security-identity elements, see Example A-3. -->
<security-role-ref>
  <description>My admin role</description>
  <role-name>admin</role-name>
  <role-link>main-admin-role</role-link>
</security-role-ref>
<security-identity>
  <use-caller-identity/>
</security-identity>
<!-- If CMP is used, query elements are used to define the
      logic behind finder and select methods, using EJB-QL. -->
<query>
  <!-- A description of the query. -->
  <description>This is a finder query...</description>
  <!-- A declaration of the finder or select method that
        this query is bound to. -->
  <query-method>
    <!-- The name of the select or finder method. -->
    <method-name>findByMyPersistentField</method-name>
    <!-- A list of the method arguments, used to
          distinguish overloaded finder or select
          methods. The method-param elements must be
          listed here in order of the corresponding method
          signature. An empty method-params element can
          be used if the method has no arguments or if
          there is only a single version of the named
          method. -->
    <method-params>
      <!-- Each method-param element specifies the
            fully qualified Java type of a parameter
            in the method. -->
      <method-param>my.finder.arg.type</method-param>
    </method-params>
  </query-method>
  <!-- Used to specify whether EJBs returned as the result
        of this query method should be returned using their
        remote or local client interfaces. Allowed values
        are "Remote" or "Local". -->
  <result-type-mapping>Local</result-type-mapping>
  <!-- The ejb-ql element contains the EJB-QL query string
        that defines this finder or select method. See
        Appendix C for a full description of the EJB-QL
        syntax. -->
  <ejb-ql>SELECT e FROM my-ejb-table e WHERE
        e.my-persistent-field IS NOT NULL</ejb-ql>
</query>
</entity>
<entity>
  <ejb-name>MySlaveEJB</ejb-name>
  <ejb-class>my.slave.ejb.class</ejb-class>
  <persistence-type>Container</persistence-type>
  <prim-key-class>my.slave.primkey.class</prim-key-class>

```

```

        <reentrant>false</reentrant>
    </entity>
    . . .
</enterprise-beans>
. . .
</ejb-jar>

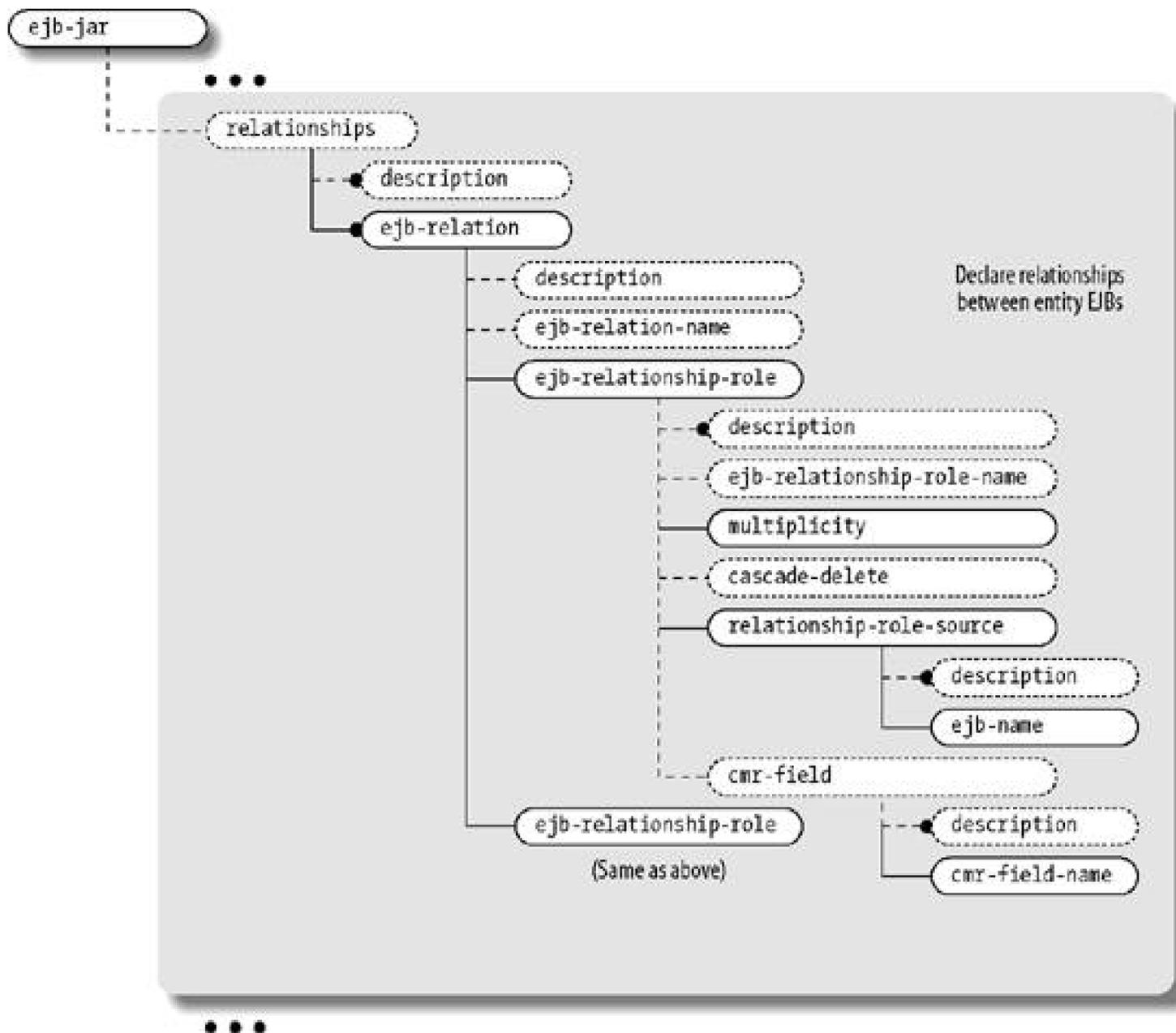
```

A.2.4. Entity EJB Relationship Declarations

Figure A-12 shows the elements used to declare relationships between entity EJBs.

Example A-12 shows sample entity relationship declarations, annotated with descriptive comments.

Figure A-12. Entity EJB relationship declarations



Example A-12. Annotated entity EJB relationship declarations

```

<ejb-jar . . .>
. . .
<!-- The relationships element defines persistence relationships
between CMP entity EJBs and specifies how to manage the
persistent relationships. -->
<relationships>
  <!-- An overall description of the relationships. -->
  <description>desc</description>
  <!-- Each ejb-relation element declares a persistence relationship
between two CMP entity EJBs declared in this archive. -->
  <ejb-relation>
    <!-- Description of the relation. -->
    <description>desc</description>
    <!-- A unique name for the relation. -->
    <ejb-relation-name>name</ejb-relation-name>
    <!-- The two roles involved in the relationship are defined
with two relationship-role elements. Each element
defines one end of the relation. -->
    <ejb-relationship-role>
      <!-- Description of the role. -->
      <description>The master role in the relation</description>
      <!-- The name of the role in the relationship. -->
      <ejb-relationship-role-name>master
</ejb-relationship-role-name>
      <!-- The multiplicity of this end of the relationship.
Allowed values are "One" or "Many". -->
      <multiplicity>One</multiplicity>
      <!-- If this element is present, it indicates that a
deletion of the EJB on the other end of the
relationship should invoke a cascading delete
of the EJB(s) on this end of the relationship. -->
      <cascade-delete/>
      <!-- The name of the EJB (from its ejb-name element) to be
used for this end of the relationship. -->
      <relationship-role-source>
        <!-- A description of the role source. -->
        <description>desc</description>
        <!-- The name of the EJB at this end of the
relationship. The name must correspond to the
ejb-name element of a CMP EJB declared in this
ejb-jar. -->
        <ejb-name>MyEntityEJB</ejb-name>
      </relationship-role-source>
      <!-- A field name to be used to access this relationship
from the source EJB. If this element is present,
JavaBean-style get/set accessors will be assumed to
exist on the EJB implementation class to access the
EJBs at the other end of this relationship. If this
element is not present, then the relationship

```

```

        cannot be accessed from this EJB. -->
    <cmr-field>
        <description>This field...</description>
        <cmr-field-name>slaves</cmr-field-name>
    </cmr-field>
</ejb-relationship-role>
<!-- The opposite role to that defined above. -->
<ejb-relationship-role>
    <description>desc</description>
    <ejb-relationship-role-name>slave
</ejb-relationship-role-name>
    <multiplicity>Many</multiplicity>
    <cascade-delete/>
    <relationship-role-source>
        <description>desc</description>
        <ejb-name>MySlaveEJB</ejb-name>
    </relationship-role-source>
    <cmr-field>
        <description>This field...</description>
        <cmr-field-name>master</cmr-field-name>
    </cmr-field>
</ejb-relationship-role>
</ejb-relation>
</relationships>
. . .
</ejb-jar>

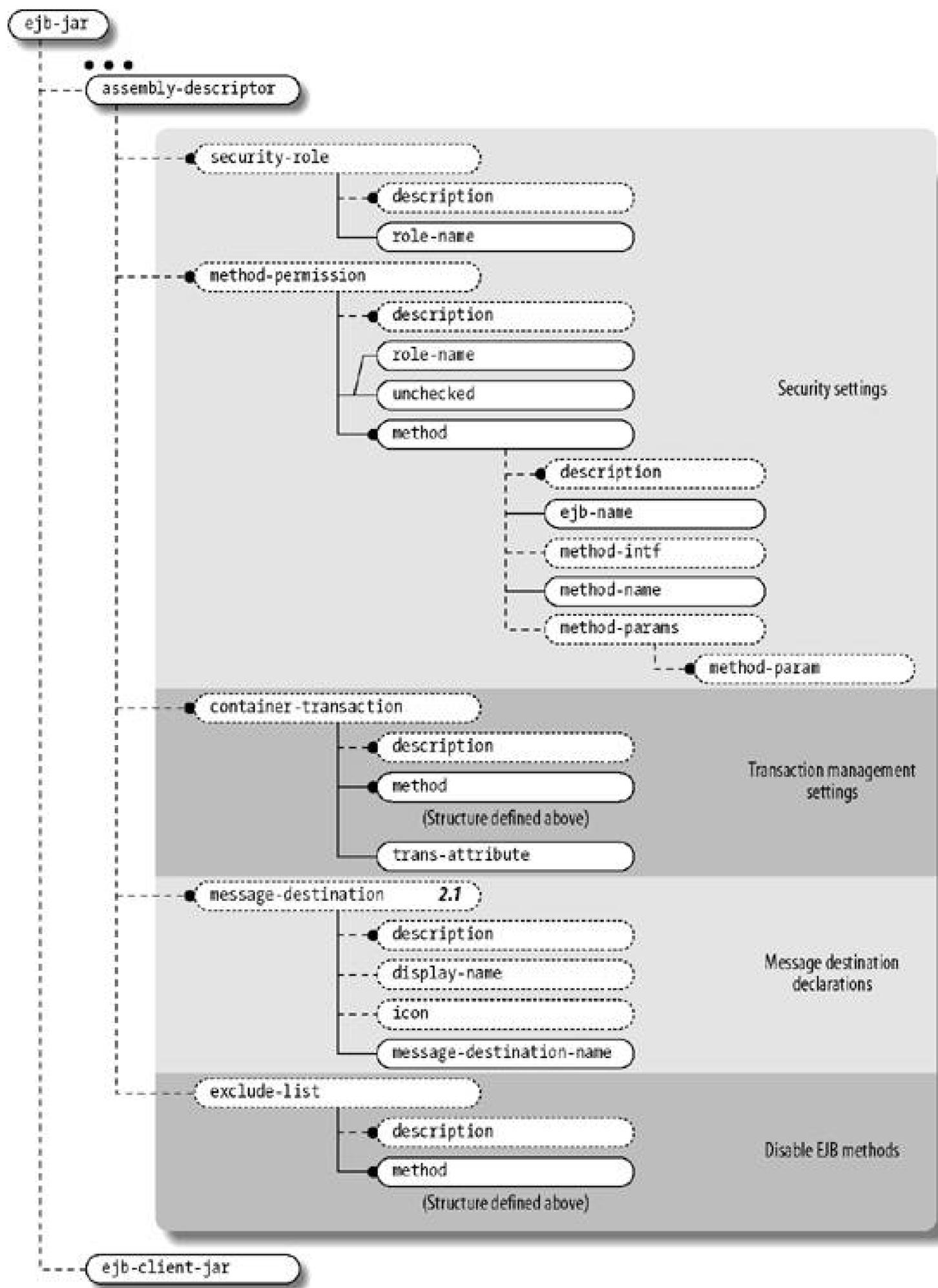
```

A.2.5. Cross-Component Runtime Settings

[Figure A-13](#) shows the elements used to define cross-component runtime settings for EJB components.

[Example A-13](#) shows sample cross-component runtime settings, annotated with descriptive comments.

Figure A-13. Cross-component runtime settings



Example A-13. Annotated cross-component runtime settings

```

<ejb-jar . . .>
. . .
<!-- This element contains various cross-component details about
how the EJBs contained in the ejb-jar file should be managed at
runtime. These details can be changed at deployment time by the
application assembler. -->
<assembly-descriptor>
<!-- Each security-role element declares a security role used in
EJBs in this ejb-jar module. Each named role must be mapped
to a physical group/identity within the runtime security
environment of the EJB/J2EE server. -->
<security-role>
<!-- Description of the role. -->
<description>desc</description>
<!-- The name of the role. This name must be mapped to a
physical group/identity in the security environment
in the EJB/J2EE server. -->
<role-name>main-admin-role</role-name>
</security-role>
<!-- Each method-permission element declares the permissions to
be used to control access to EJB methods. -->
<method-permission>
<!-- Description of the method permission. -->
<description>desc</description>
<!-- Each permission must include either a list of role-name
elements, indicating the roles being given permissions to
execute the methods in this method-permission, or it must
include an empty "unchecked" element, indicating that
the methods specified in the list are to be executed
without checking the client's identity. If a list of
role-name elements is given, each role-name must match a
security-role defined above. -->
<unchecked/>
<!-- A list of one or more method elements, specifying the
set of methods whose permissions are being set. Each
method element specifies one or more methods on a
specific EJB in this archive. Wildcards are used to
indicate multiple methods. -->
<method>
<!-- Description of the method. -->
<description>This method set...</description>
<!-- The name of the EJB whose methods are being specified.
The ejb-name element must match the ejb-name element
of an EJB declared in this descriptor. -->
<ejb-name>MySessionEJB</ejb-name>
<!-- Limits the method definitions to a specific interface
on the EJB. Allowed values of the method-intf
element are "Home" for the remote home interface,
"Remote" for the remote client interface,
"LocalHome" for the local home interface, and "Local"
for the local client interface. If this element is
not present, the methods are chosen from all of

```

```

        these interfaces on the EJB. -->
<method-intf>Local</method-intf>
<!-- The name of the method(s). The allowed values are
either a full method name, or an asterisk, "*",
indicating that all of the EJB's methods (limited
by the method-intf, if present) are being selected. -->
<method-name>setName</method-name>
<!-- A specification of the parameters in the method
signature. This element can be used to chose a
specific method when a method name has been
overloaded in an EJB. -->
<method-params>
    <!-- Each method-param element specifies the
fully qualified Java type of a parameter in
the method. -->
    <method-param>java.lang.String</method-param>
</method-params>
</method>
</method-permission>
<!-- Used to specify how transactions should be managed for a set
of methods on an EJB. -->
<container-transaction>
    <!-- Description of the element. -->
    <description>The transactions here...</description>
    <!-- A set of method(s) for which transaction management is
being defined. The method element is described above
as part of the method-permission element. -->
    <method>
        <description>This method...</description>
        <ejb-name>MyEntityEJB</ejb-name>
        <method-intf>Remote</method-intf>
        <method-name>getName</method-name>
        <method-params/>
    </method>
    <!-- The transaction support to be offered by the method(s)
specified above. Allowed values are "NotSupported",
"Supports", "Required", "RequiresNew", "Mandatory",
and "Never". -->
    <trans-attribute>NotSupported</trans-attribute>
</container-transaction>
<!-- Declare message destinations (queues or topics)
used by components in this module. Any destinations
declared here will need to be configured in the target
application server. -->
<message-destination>
    <message-destination-name
        >MyMessageDestination</message-destination-name>
</message-destination>
<!-- An exclude-list can be used to specify methods on the EJBs
in this archive that should not be called by any clients. -->
<exclude-list>
    <!-- Description of exclude list. -->

```

```
<description>These methods are excluded...</description>
<!-- List of methods that should be considered uncallable.
      The method element is described fully in the
      method-permission element, above. -->
<method>
  <description>Tis method is dangerous...</description>
  <ejb-name>MyEntityEJB</ejb-name>
  <method-name>doSomethingBad</method-name>
  <method-params>
    <method-param>java.lang.Object</method-param>
  </method-params>
</method>
</exclude-list>
</assembly-descriptor>
<!-- If this element is present, the J2EE server's deployment tools
      should generate a jar file with the given filename that contains
      all of the classes (including container-generated classes) required
      by a client in order to use the contained EJB components.  These
      classes include remote and local stub implementations for the EJB's
      remote home and client interfaces, among other things.  This jar
      file can then be included in other J2EE components or application
      clients if they reference the EJBs in this archive. -->
  <ejb-client-jar>myClientJar.jar</ejb-client-jar>
</ejb-jar>
```

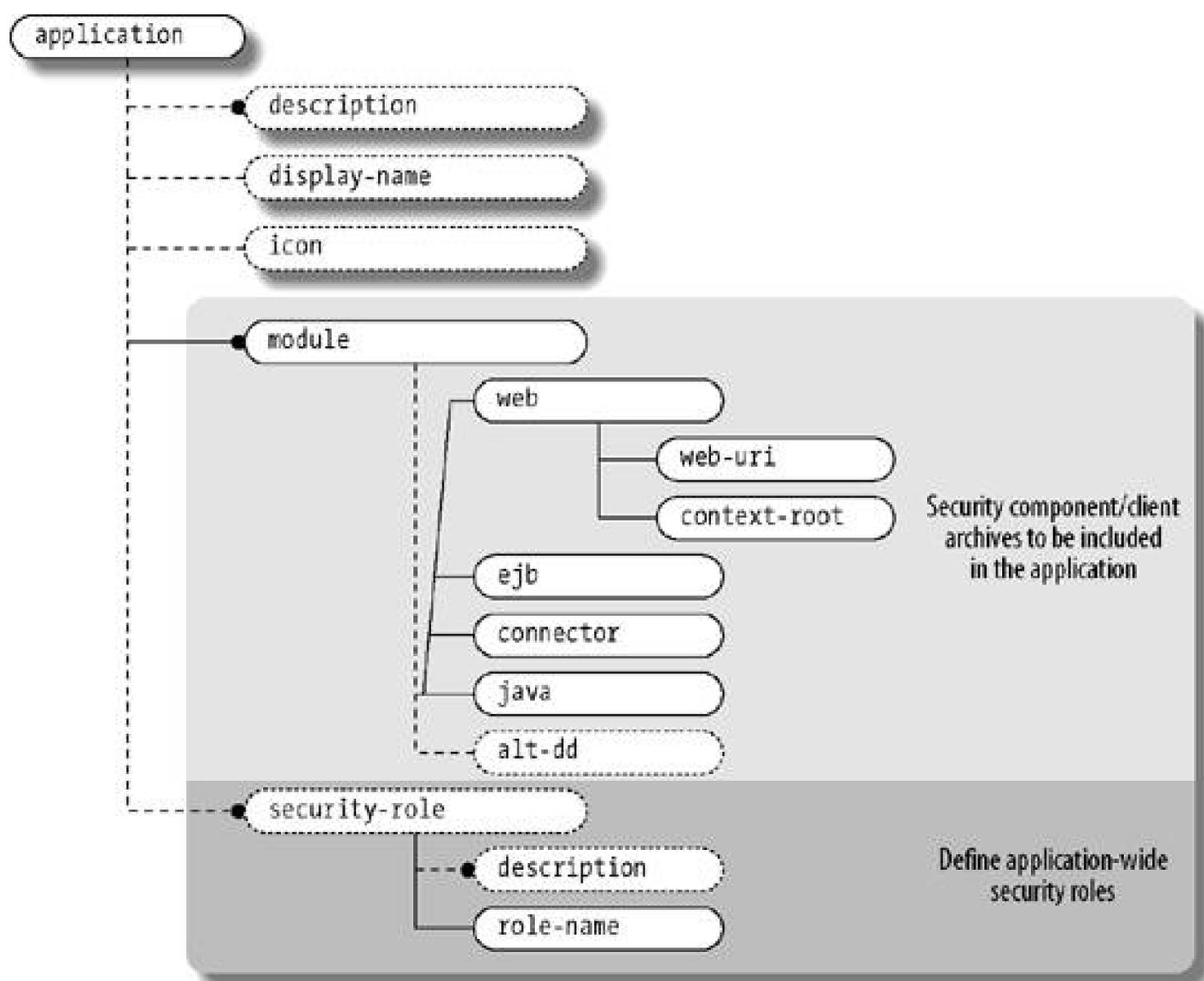
A.3. Application Archives (application .xml)

Enterprise application archives (ears) contain collections of component archives (web, EJB, and so on) that make up an application. An *application.xml* deployment descriptor is used to declare and configure the component archives, as well as declare applicationwide security roles. [Figure A-14](#) shows the structure of the application deployment descriptor.

[Example A-14](#) shows a sample application deployment descriptor, annotated with descriptive comments. Note that this example uses the application descriptor schema from the J2EE 1.4 specification, referencing the schema from the default location on the java.sun.com server. To use the J2EE 1.3 application descriptor DTD, you would need to replace the `application` element tag and its schema reference with the following empty tag and `DOCTYPE` entry:

```
<!DOCTYPE application PUBLIC
    "-//Sun Microsystems, Inc.//DTD J2EE Application 1.3//EN"
    "http://java.sun.com/dtd/application_1_3.dtd">
<application>
    . . .
</application>
```

Figure A-14. Application archive deployment descriptor overview



Example A-14. Annotated application deployment descriptor

```
<?xml version="1.0" encoding="UTF-8"?>
<application 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/application_1_4.xsd"
  version="1.4">
  <!-- Description of the application, for tools. -->
  <description>This application is...</description>
  <!-- Display name of the application. -->
  <display-name>My Application</display-name>
  <!-- Icons to be used to represent the app in tools. -->
  <icon>
    <small-icon>my/icon/file</small-icon>
    <large-icon>my/large/icon/file</large-icon>
  </icon>
  <!-- Each module element specifies an archive containing
    components or clients that are part of the application. -->
  <module>
    <!-- The connector element references the archive file for
```

a set of JCA Connectors. The file reference is relative to the root of the enterprise application archive (ear) file. -->

```
<connector>myConnector.rar</connector>
```

```
<!-- The alt-dd element can be used in any of the module
variations, to specify a deployment descriptor that should
be used in lieu of the descriptor in the component archive.
The descriptor must be of the appropriate format for the
type of archive, and the file reference is made relative
to the root of the ear file. -->
```

```
<alt-dd>my-alt-rar-dd.xml</alt-dd>
```

```
</module>
```

```
<module>
```

```
<!-- Include the specified EJB archive in the application. -->
```

```
<ejb>myEJBs.jar</ejb>
```

```
</module>
```

```
<module>
```

```
<!-- Include the specified J2EE client jar in the application. -->
```

```
<java>myClient.jar</java>
```

```
</module>
```

```
<module>
```

```
<!-- Include the specified web archive in the application. -->
```

```
<web>
```

```
<!-- The location of the web archive (war) file -->
```

```
<web-uri>myWebComps.war</web-uri>
```

```
<!-- The web context root to use for the web components in
this archive. -->
```

```
<context-root>myapp</context-root>
```

```
</web>
```

```
</module>
```

```
<!-- The security-role element can be used here much as it is used
in individual archive deployment descriptors
```

, but the roles

```
declared here can be used in any components in the archives
specified in the module entries above. See section X for
additional information on the security-role element. -->
```

```
<security-role>
```

```
<description>desc</description>
```

```
<!-- Name of the role, which must be mapped to a physical role
in the application server's runtime environment. -->
```

```
<role-name>app-level-admin</role-name>
```

```
</security-role>
```

```
</application>
```

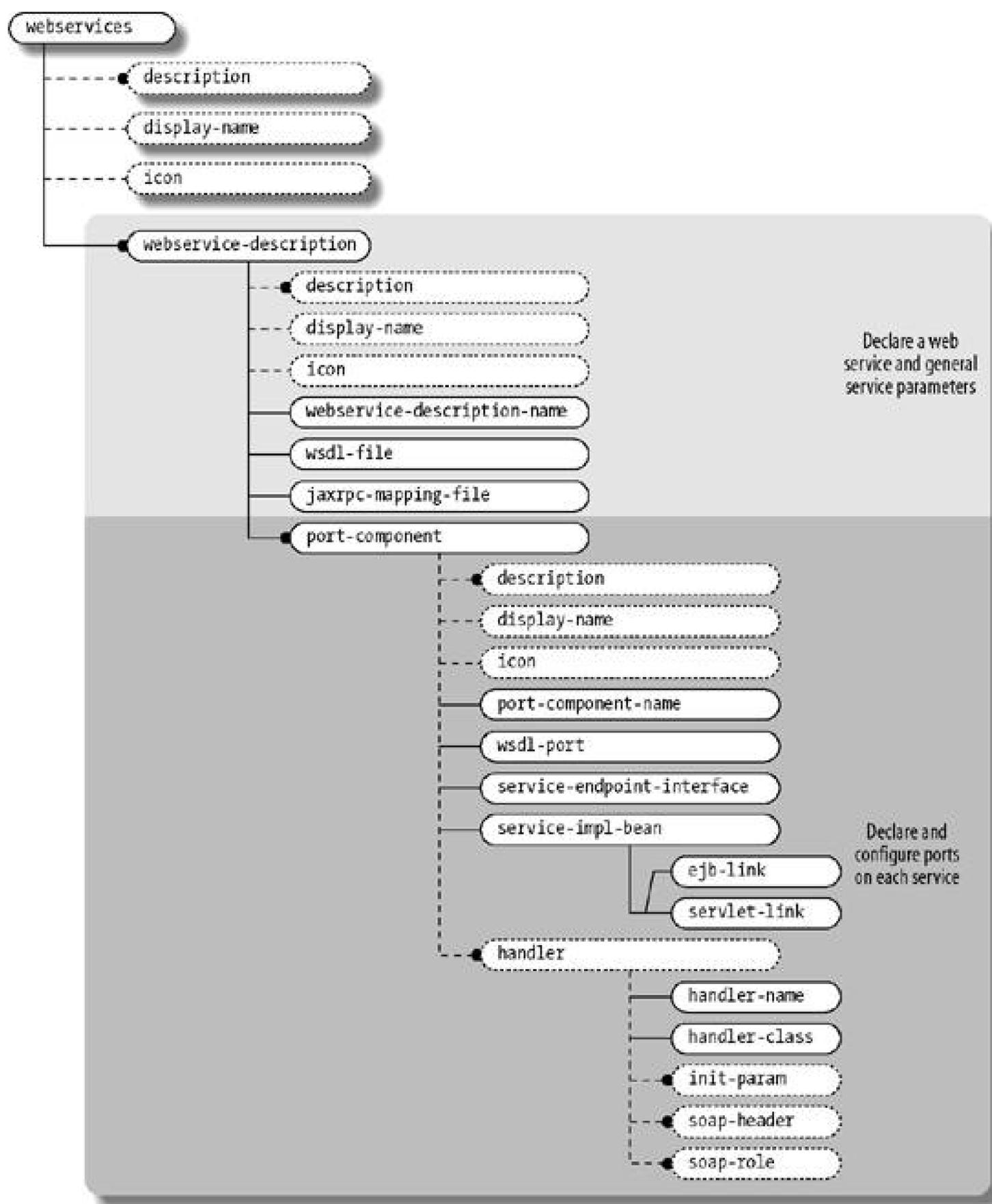
A.4. Web Services (webservices.xml)

SOAP web services are implemented in J2EE using JAX-RPC, and service implementations can take the form of a basic Java bean deployed as a web component or a stateless session EJB component. In either case, a *webservices.xml* deployment descriptor is used within the component archive (the war file or *ejb-jar.xml*/file) to declare and configure the web services implemented by the components in the archive.

[Figure A-15](#) shows the overall XML schema structure used in *webservices.xml* files.

[Example A-15](#) shows sample web service declarations, annotated with descriptive comments.

Figure A-15. Web service declarations



Example A-15. Annotated web service declarations

```
<?xml version="1.0" encoding="UTF-8"?>
<webservises 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/j2ee_web_services_1_1.xsd"
  version="1.1">
```

```

<description>Web services contained in my application</description>
<display-name>Web services</display-name>
<icon>
  <small-icon>icons/my-small-ws-icon.jpg</small-icon>
  <large-icon>icons/my-large-ws-icon.jpg</large-icon>
</icon>
<!-- Each webservice-description element defines a full web service
implemented somewhere in the surrounding component archive (web
or EJB). Each service specifies a WSDL file and a JAX-RPC
mapping file contained in the archive and one or more ports
that can be invoked by clients. Each port is described here,
in terms of the WSDL port element that it implements, the Java
entities that implement it, and optional runtime settings. -->
<webservice-description>
  <description>This service...</description>
  <display-name>A service</display-name>
  <icon>
    <small-icon>icons/small-ws1-icon.jpg</small-icon>
    <large-icon>icons/large-ws1-icon.jpg</large-icon>
  </icon>
  <!-- A unique name for this service definition. This name must be
unique within the surrounding archive. -->
  <webservice-description-name
    >MyWebService</webservice-description-name>
  <!-- Indicate the location of the WSDL file that describes this
service. The location is specified as a relative path from
the root of the surrounding archive. In web archives, the WSDL
file is typically placed in the WEB-INF directory. In EJB
archives, it is placed in the META-INF directory. -->
  <wsdl-file>WEB-INF/mybservice.wsdl</wsdl-file>
  <!-- Indicate the location of the JAX-RPC mapping file for this
service. The location is specified as a relative path from the
root of the surrounding archive. This XML file describes how
to map between SOAP XML and Java entities. The mapping file
is typically placed in the WEB-INF directory in web archives
and in the META-INF directory in EJB archives. See Section X
for a full reference on JAX-RPC mapping files. -->
  <jaxrpc-mapping-file
    >WEB-INF/mybservice-mapping.xml</jaxrpc-mapping-file>
  <!-- Each port on the service is declared with a port-component
element. -->
  <port-component>
    <description>A port provided by this service...</description>
    <display-name>Service port X</display-name>
    <icon>
      <small-icon>icons/my-small-port-icon.jpg</small-icon>
      <large-icon>icons/my-large-port-icon.jpg</large-icon>
    </icon>
    <!-- A unique name, within the surrounding archive, for this
port component. This name can be used in web and EJB
descriptors within service-ref entries, as the value
of a port-component-link. -->

```

```

<port-component-name>MyServicePort</port-component-name>
<!-- The qualified name (QName) of the port within the
      service's WSDL description. -->
<wsdl-port>MyServicePort</wsdl-port>
<!-- The Java interface that is used in the Java binding of
      the port. -->
<service-endpoint-interface>com.my.service.interface
</service-endpoint-interface>
<!-- Specify the component within this archive or application
      that implements this port. The service-impl-bean element
      can either point to the servlet-name of a web component,
      by using the servlet-link element, or it can point
      to the ejb-name of an EJB component, by using an ejb-link
      element. -->
<service-impl-bean>
  <servlet-link>MyServiceWebComponent</servlet-link>
</service-impl-bean>
<!-- The handler element is used to specify custom SOAP
      message handlers that should be put in place for this port
      in the service. Handlers can be triggered for any message
      targeted at the port, or they can handle specific
      messages based on their header entries. -->
<handler>
  <!-- A name for the handler. -->
  <handler-name>MySOAPHandler</handler-name>
  <!-- The full class name of the handler. The class must
      implement the javax.xml.rpc.handler.Handler interface. -->
  <handler-class>com.my.soap.handler</handler-class>
  <!-- Specify initialization parameters, passed into the
      handler's init( ) method as part of the HandlerInfo. -->
  <init-param>
    <!-- Name of the parameter. -->
    <param-name>param1</param-name>
    <!-- Value of the parameter. -->
    <param-value>value1</param-value>
  </init-param>
  <!-- The qualified name (QName) of a SOAP header to be
      processed by the handler. -->
  <soap-header>com.partner-header1</soap-header>
  <!-- A SOAP actor role that the handler should play as it
      processes the SOAP request. -->
  <soap-role>urn:my-service</soap-role>
</handler>
</port-component>
</webservice-description>
</webservices>

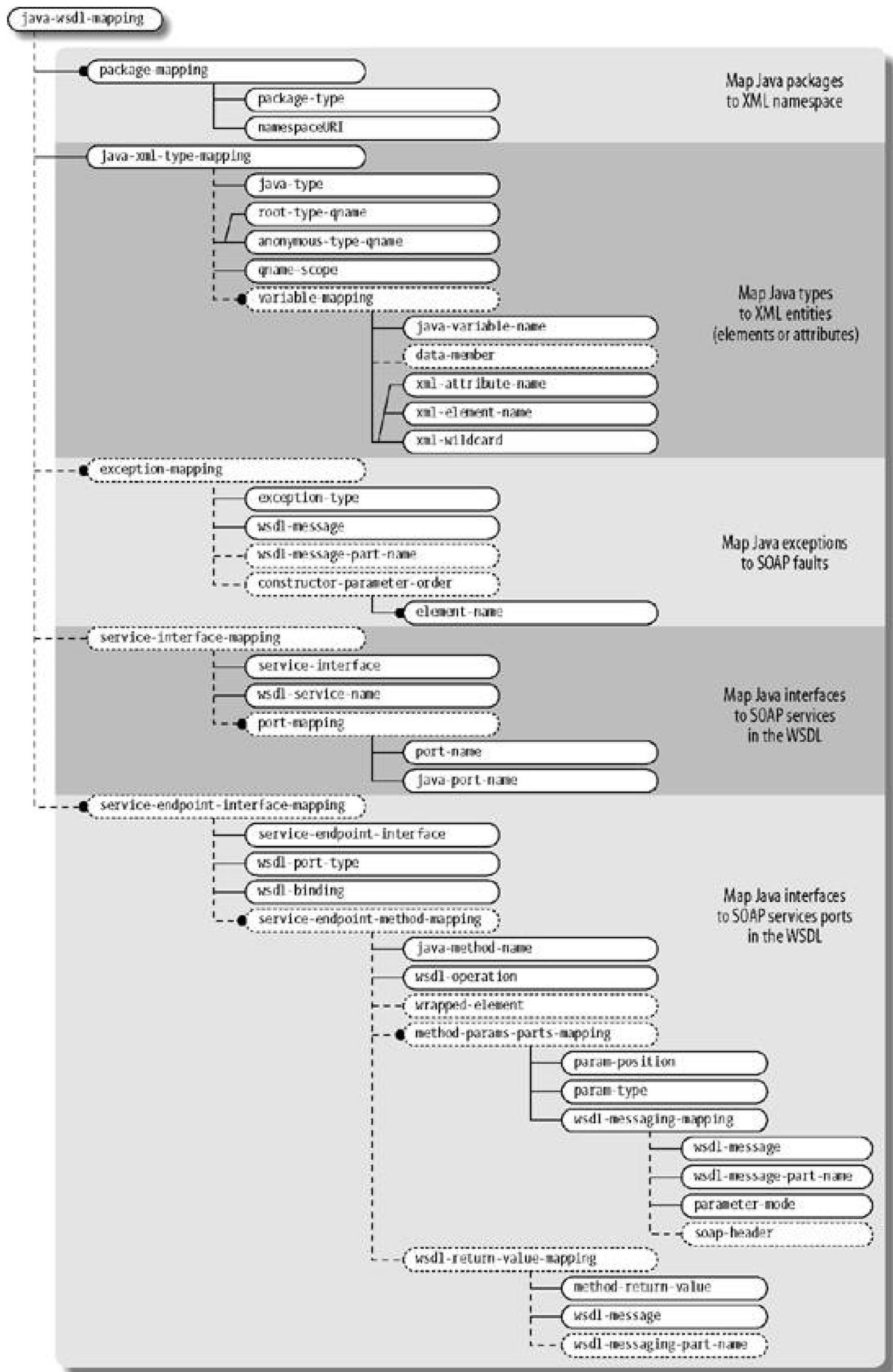
```

A.5. Web Service Java/WSDL Mappings

When deploying JAX-RPC web services in a J2EE application, each web service must have a mapping file that describes how the Java entities in the service implementation and the XML entities in the WSDL are mapped to each other. The mapping file is included in the component archive and specified in the *webservices.xml*/deployment descriptor. [Figure A-16](#) shows the structure of JAX-RPC mapping files.

[Example A-16](#) shows sample Java/WSDL mappings, annotated with descriptive comments.

Figure A-16. Web services Java/WSDL mappings



Example A-16. Annotated Java/WSDL mappings

```

<?xml version="1.0" encoding="UTF-8"?>
<java-wsdl-mapping
  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/j2ee_jaxrpc_mapping_1_1.xsd"
  version="1.1">
  <!-- Each package-mapping element specifies how a Java package
    is mapped to an XML namespace in the WSDL for the service. -->
  <package-mapping>
    <!-- The Java package for the mapping. -->
    <package-type>com.my.service.datatypes</package-type>
    <!-- The XML namespace to be mapped to this package. -->
    <namespaceURI>my-service-ns</namespaceURI>
  </package-mapping>
  <!-- Each java-xml-type-mapping element specifies how to map an
    XML element to a corresponding Java type. -->
  <java-xml-type-mapping>
    <!-- The full classname of the Java type. -->
    <java-type>com.my.service.datatypes.type1</java-type>
    <!-- The WSDL entity is identified using its qualified name (QName).
      If the WSDL entity has an anonymous QName, then an
      anonymous-type-qname element is used here. Otherwise, a
      root-type-qname element should be used. -->
    <root-type-qname>my-service-ns:type1</root-type-qname>
    <!-- Each qualified name must be scoped to the WSDL element type,
      by specifying simpleType, complexType, or element here. -->
    <qname-scope>complexType</qname-scope>
    <!-- If the WSDL entity has child elements that need to be mapped
      to corresponding properties on the Java type, then one or
      more variable-mapping elements can be used here. -->
    <variable-mapping>
      <!-- The name of the Java property or public data member for
        the mapping. -->
      <java-variable-name>id</java-variable-name>
      <!-- This empty element is used to indicate that the Java
        variable listed above is a public data member, not a
        JavaBeans property. -->
      <data-member/>
      <!-- The corresponding WSDL entity is specified as either
        an XML element, using an xml-element-name; an XML attribute,
        using an xml-attribute-name element; or an XML xsd:any
        element, using the empty xml-wildcard element -->
      <xml-attribute-name>id</xml-attribute-name>
    </variable-mapping>
  </java-xml-type-mapping>

```

```

<!-- Each exception-mapping entry maps a WSDL fault entity to a
      Java exception. -->
<exception-mapping>
  <!-- The fully qualified Java class name of the exception -->
  <exception-type>com.my.service.BadParameterException</exception-type>
  <!-- This element specifies the qualified name (QName) of the
        WSDL message for the fault. -->
  <wsdl-message>my-service-ns:BadParameterFaultMessage</wsdl-message>
  <!-- If the fault message has more than one part, you need to use
        this element to indicate which part is being mapped here. -->
  <wsdl-message-part-name>fault-part</wsdl-message-part-name>
  <!-- If the mapped WSDL fault element is a complex type, you can
        specify the order in which its elements are mapped to the
        constructor arguments on the exception class. -->
  <constructor-parameter-order>
    <!-- The name of an an element of the complex WSDL type. -->
    <element-name>fault-message</element-name>
  </constructor-parameter-order>
</exception-mapping>
<!-- Each service-interface-mapping element maps a WSDL service entity
      to a corresponding set of Java interfaces. At a minimum, the
      service itself must be mapped to a corresponding Java interface,
      but you can also specify how each port in the service is mapped
      to a getX method on generated service interfaces. -->
<service-interface-mapping>
  <!-- The fully qualified class name of the Java interface
        for the WSDL service. This can be preexisting, or it
        can be generated using tools in your web service engine. -->
  <service-interface>com.my.service.interface</service-interface>
  <!-- The qualified name of the WSDL service entity. -->
  <wsdl-service-name>my-service-ns:service1</wsdl-service-name>
  <!-- Each port-mapping entry generates a getX method on any service
        interfaces that are automatically generated using this
        mapping file. If "JavaName" is given as the java-port-name
        below, then a getJavaName( ) method will be generated on the
        service interface, and calling this method will return a
        Java interface for the port named in the port-name element -->
  <port-mapping>
    <!-- The WSDL port name to be bound to the Java name below. -->
    <port-name>port1</port-name>
    <!-- The name to use for the getX method on the generated
          service interface. In this case, the method will be
          getPort1, which will return a Java binding for the
          WSDL port named port1. -->
    <java-port-name>Port1</java-port-name>
  </port-mapping>
</service-interface-mapping>
<!-- Each service-endpoint-interface-mapping element specifies how a
      WSDL port is to be mapped to corresponding Java entities. -->
<service-endpoint-interface-mapping>
  <!-- The fully qualified Java class for the port interface. -->
  <service-endpoint-interface>

```

```

    com.my.service.port1</service-endpoint-interface>
<!-- The qualified name of the WSDL port to be mapped. -->
<wsdl-port-type>my-service-ns:port1</wsdl-port-type>
<!-- The qualified name of the binding of the WSDL port to be mapped.
A given WSDL port can potentially be associated with multiple
bindings (e.g., HTTP and SMTP bindings of the same service), so
this element is needed to uniquely identify the port/binding
being mapped. -->
<wsdl-binding>urn:my-service-ns/port1-binding1</wsdl-binding>
<!-- Each service-endpoint-method-mapping element defines how a
method on the Java interface is mapped to specific operations
on the WSDL port being mapped. If a default mapping is being
used, you don't need to include any method mappings here, but
these method mappings can be useful for customizing the
mapping. -->
<service-endpoint-method-mapping>
  <!-- The name of the Java method being mapped. -->
  <java-method-name>doServiceOperation1</java-method-name>
  <!-- The name of the WSDL operation within the port being mapped -->
  <wsdl-operation>serviceOperation1</wsdl-operation>
  <!-- If the operation uses a wrapped element with the same name
as the operation, then this element should be present. -->
  <wrapped-element/>
  <!-- The Java method arguments can be mapped to input messages
on the WSDL operation using method-param-parts-mapping
elements. -->
  <method-param-parts-mapping>
    <!-- The position of the method argument in the Java method
argument list. The position is indexed from zero. -->
    <param-position>0</param-position>
    <!-- The Java type of the method argument. -->
    <param-type>java.lang.String</param-type>
    <!-- The corresponding WSDL message is specified with this
element. -->
    <wsdl-message-mapping>
      <!-- The qualified name of the WSDL message. -->
      <wsdl-message>
        my-service-ns:operation1Input1</wsdl-message>
      <!-- The part of the message being mapped. If the method
is using a wrapped element (i.e., the wrapped-element
entry is specified in the method mapping), then
this part name references the name of a child element
in the wrapped element. If the method is not using
a wrapped element, then the part name here references
WSDL message part name. -->
      <wsdl-message-part-name>
        elementPart1</wsdl-message-part-name>
      <!-- This indicates whether the mapping is inbound only
(with respect to the SOAP service), outbound only,
or both directions. The allowed values are IN, OUT,
or INOUT. -->
      <parameter-mode>INOUT</parameter-mode>

```

```
        <!-- If this element is present, then this mapping
             is from a Java parameters to a SOAP header, rather
             than a WSDL message. -->
        <soap-header/>
    </wsdl-message-mapping>
</method-param-parts-mapping>
<!-- The mapping of the output messages (if there are any) on
     the WSDL operation to the Java return type of the method
     can be specified using this element. -->
<wsdl-return-value-mapping>
    <!-- The fully qualified Java type of the method return
         value. -->
    <method-return-value></method-return-value>
    <!-- The qualified name of the WSDL message being mapped. -->
    <wsdl-message>
        my-service-ns:operation1Output1</wsdl-message>
    <!-- The part of the message being mapped. If the method
         is using a wrapped element (i.e., the wrapped-element
         entry is specified in the method mapping), then
         this part name references the name of a child element
         in the wrapped element. If the method is not using a
         wrapped element, then the part name here references
         the WSDL message part name. -->
    <wsdl-message-part-name>
        elementPart1</wsdl-message-part-name>
    </wsdl-return-value-mapping>
</service-endpoint-method-mapping>
</service-endpoint-interface-mapping>
</java-wsdl-mapping>
```

[← PREV](#)

Appendix B. JavaServer Faces Tag Libraries

The JavaServer Faces API includes two JSP tag libraries that are used to create the user interface for your applications. The JSF tag library is divided into two parts. The HTML tag library provides tags to produce HTML output, and the core tag library provides utilities that are not specific to the HTML rendering model.

[← PREV](#)

B.1. JSF Core Tags

The JSF core tags provide access to all of the JSF features that do not directly relate to HTML rendering. Many of the HTML tags rely on core tags to specify optional features, such as validation functionality.

To use the JSF core tags in a JSP file, you must include the following statement in the file header:

```
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
```

Use of `f` as the tag namespace prefix is both conventional and highly recommended but not required.

B.1.1. <f:actionListener>

Adds an action listener implementation to the surrounding `UIComponent` tag, such as an `<h:commandLink>`. The tag must specify the fully qualified class name of a class implementing the `javax.faces.event.ActionListener` interface. If you wish to handle action events via a method on a backing bean, use the appropriate attribute on the UI component tag instead.

Table B-1. Attribute for <f:actionListener>

Attribution	Description
<code>type</code>	The fully qualified class name of a class that implements the <code>ActionListener</code> interface

B.1.2. <f:attribute>

Adds an attribute to the surrounding tag. Using this tag is functionally equivalent to setting the attribute value in the parent tag. Values set on the parent component supersede any attributes set using this tag.

Table B-2. Attributes for <f:attribute>

Attribute	Description
<code>name</code>	The name of the attribute to set
<code>value</code>	The value of the attribute to set

B.1.3. <f:convertDateTime>

Converts the value contained by the surrounding component (such as an `<h:outputText>` tag) into a formatted date and time value.

Table B-3. Attributes for <f:convertDateTime>

Attribute	Description
<code>dateStyle</code>	Date format, as defined by <code>java.text.DateFormat</code> . Valid values are <code>default</code> , <code>short</code> , <code>medium</code> , <code>long</code> , and <code>full</code> .
<code>locale</code>	Locale to use for date formatting and parsing.
<code>pattern</code>	Custom formatting pattern for this converter. Supersedes <code>dateStyle</code> , <code>type</code> , and <code>timeStyle</code> .
<code>timeStyle</code>	Time format, as defined by <code>java.text.DateFormat</code> . Valid values are <code>default</code> , <code>short</code> , <code>medium</code> , <code>long</code> , and <code>full</code> .
<code>timeZone</code>	<code>String</code> or <code>TimeZone</code> object used to interpret time zone information when parsing time values.
<code>type</code>	String value, either <code>date</code> , <code>time</code> , or <code>both</code> . Default is <code>date</code> .

B.1.4. <f:convertNumber>

Converts the value contained by the surrounding component (such as an `<h:outputText>` tag) into a formatted numerical value.

Table B-4. Attributes for <f:convertNumber>

Attribute	Description
<code>currencyCode</code>	String specifying the ISO-4217 currency code used for currency formatting.
<code>currencySymbol</code>	Currency symbol.
<code>groupingUsed</code>	Boolean value specifying whether grouping separators (such as commas in U.S. formatted numbers) will be used.
<code>integerOnly</code>	Boolean value specifying whether only the integer component of the input/output will be included.
<code>locale</code>	<code>String</code> or <code>Locale</code> object specifying the number styles to be used in this conversion.
<code>maxFractionDigits</code>	Integer value specifying the maximum number of digits to include in the fractional component of the output.
<code>maxIntegerDigits</code>	Integer value specifying the maximum number of digits to include in the integer component of the output.
<code>minFractionDigits</code>	Integer value specifying the minimum number of digits to include in the fractional component of the output.
<code>minIntegerDigits</code>	Integer value specifying the minimum number of digits to include in the integer component of the output.
<code>type</code>	Specifies parsing as a <code>number</code> , <code>currency</code> , or <code>percentage</code> . The default is <code>number</code> .

B.1.5. <f:converter>

Adds a converter to a UI component, based on the converter ID specified in the *faces-config.xml* file. While class names are often used as converter IDs, this tag does not tag a class name.

Table B-5. Attributes for <f:converter>

Attribute	Description
<code>converterId</code>	A converter ID specified in the JSF configuration file for the application

B.1.6. <f:facet>

Defines a facet for a component. Facets are used by various components, such as `<h:dataTable>`, to override particular display attributes. Each facet tag contains a set of additional tags that will be used instead of or in addition to the regular tag body in particular circumstances. For example, the

`<h:dataTable>` tag supports facets named `header` and `footer` that are used to render a header and a footer before and after iterating over a collection.

Table B-6. Attributes for `<f:facet>`

Attribute	Description
<code>name</code>	The name of this facet. Valid facet names depend on the component containing the facet.

B.1.7. `<f:loadBundle>`

Loads a resource bundle from the classpath and binds it to a variable. Once bound, the resource bundle can be accessed via the JSF expression language using the normal map syntax.

Table B-7. Attributes for `<f:loadBundle>`

Attribute	Description
<code>basename</code>	The fully qualified name of the resource bundle
<code>var</code>	The variable to bind this resource bundle to

B.1.8. `<f:param>`

Associates a parameter with a component. The meaning of the parameter depends on the component it is associated with. For example, parameters passed to the `<h:outputFormat>` tag are used to provide substitution values within a string.

The `binding` and `id` attributes are rarely used.

Table B-8. Attributes for `<f:param>`

Attribute	Description
<code>name</code>	The name of this parameter. This attribute is optional.
<code>value</code>	The value of the parameter. This may be a JSF EL expression.

Attribute	Description
<code>binding</code>	Property expression indicating the bean to which this component is bound. Not normally used with this tag.
<code>id</code>	Overrides the ID of this component.

B.1.9. <f:selectItem>

Describes an item for use within a multiselection component, such as an `<h:selectManyCheckbox>`.

Table B-9. Attributes for <f:selectItem>

Attribute	Description
<code>itemDescription</code>	Description of the item, intended primarily for development tool use
<code>itemDisabled</code>	Boolean expression indicating whether this item is disabled
<code>itemLabel</code>	Textual label for the item
<code>itemValue</code>	Value that will be passed back to the server as a request parameter for this item
<code>value</code>	JSF EL expression evaluating to a <code>SelectItem</code> object containing the information for this item

B.1.10. <f:selectItems>

Adds a set of `javax.faces.model.SelectItem` objects, as indicated by the value binding or value, to the parent component.

Table B-10. Attributes for <f:selectItems>

Attribute	Description
<code>binding</code>	Property expression indicating the bean to which this component is bound. Not normally used with this tag.

Attribute	Description
<code>id</code>	Overrides the component identifier for this component.
<code>value</code>	JSF EL expression pointing to a <code>List</code> or array of <code>SelectItem</code> objects.

B.1.11. <f:subview>

A container tag for JSF components within a separate JSP page included within a JSF view (via `<jsp:include>` or similar mechanisms).

Table B-11. Attributes for <f:subview>

Attribute	Description
<code>binding</code>	Property expression indicating the bean to which this component is bound. Not normally used with this tag.
<code>id</code>	Overrides the ID of this component.
<code>rendered</code>	True or false, indicating whether or not this subview should be rendered. This may be a JSF EL expression that evaluates to a Boolean.

B.1.12. <f:validator>

Adds a validator to the parent UI input component, based on the validator ID specified in the *faces-config.xml* file. While class names are often used as validator IDs, this tag does not tag a class name.

Validators must be specified within the tag for the input component that they are responsible for validating.

Table B-12. Attribute for <f:validator>

Attribute	Description
<code>validatorId</code>	A validator ID specified in the JSF configuration file for the application

B.1.13. <f:validateDoubleRange>

Adds a double range validator to the parent UI input component. With this tag in place, the parent component will accept values only within the boundaries specified by the tag. Both attributes are optional, although omitting both is silly.

Table B-13. Attributes for <f:validateDoubleRange>

Attribute	Description
<code>maximum</code>	The maximum value the field will accept (optional)
<code>minimum</code>	The minimum value the field will accept (optional)

B.1.14. <f:validateLength>

Adds a length validator to the parent UI input component. With this tag in place, the parent component will accept only input that meets the minimum and maximum length constraints specified. You may specify a minimum length, a maximum length, or both.

Table B-14. Attributes for <f:validateLength>

Attribute	Description
<code>maximum</code>	The maximum allowable length for the associated field (optional)
<code>minimum</code>	The minimum allowable length for the associated field (optional)

B.1.15. <f:validateLongRange>

Adds a long range validator to the parent UI input component. With this tag in place, the parent component will accept only values within the boundaries specified by the tag. This tag may be used to validate integer types as well as longs.

Table B-15. Attribute for <f:validateLongRange>

Attribute	Description
<code>maximum</code>	The maximum value the field will accept (optional)
<code>minimum</code>	The minimum value the field will accept (optional)

B.1.16. <f:valueChangeListener>

Adds a value change listener implementation to the surrounding `UIComponent` tag, such as `<h:inputText>`. The tag must specify the fully qualified class name of a class implementing the `javax.faces.event.ValueChangeListener` interface. If you wish to handle action events via a method on a backing bean, use the appropriate attribute on the UI component tag instead.

Table B-16. Attribute for <f:valueChangeListener>

Attribute	Description
<code>type</code>	The fully qualified class name of a class that implements the <code>ValueChangeListener</code> interface

B.1.17. <f:view>

This tag provides the root element for a JSF view. This tag must be present on every JSF JSP, and user interface component tags must be included within this tag.

Table B-17. Attributes for <f:view>

Attribute	Description
<code>locale</code>	Overrides the locale for this view

B.1.18. <f:verbatim>

Embeds markup in a JSF page.

Table B-18. Attribute for <f:verbatim>

Attribute	Description
<code>escape</code>	Boolean value indicating whether to escape <, > and & characters, allowing output of readable HTML. Defaults to <code>false</code> .



B.2. JSF HTML Tags

The JSF HTML tags provide access to components that relate specifically to the HTML rendering model, including all of the standard HTML input components.

Several tag attributes are shared by all of the HTML tags or all of the tags in a particular category, such as the data input tags.

Table B-19. Shared tag attributes

Attribute	Description	Supported by
<code>binding</code>	A JSF EL expression identifying a backing bean property to bind this component to.	All tags (but not relevant for some)
<code>converter</code>	The name of a data converter instance associated with this component.	Input and output tags
<code>id</code>	A custom identifier for this component, allowing access to the underlying HTML element via JavaScript.	All tags
<code>rendered</code>	Boolean expression indicating whether this component and its children should be rendered.	All tags
<code>required</code>	A Boolean expression indicating whether data must be entered into the component in order to perform an action on the surrounding form.	Input tags
<code>styleClass</code>	Specifies the name of the CSS style to be applied to the generated element. Multiple styles can be specified via a comma-separated list.	All but <code><h:column></code> and <code><h:inputHidden></code>
<code>validator</code>	The name of a validator associated with this component.	Input tags
<code>value</code>	A JSF EL expression indicating the current value of this component. Used for values that are not manipulated by the user. For user-entered data, the <code>binding</code> attribute should be used instead.	Input, output, and control tags
<code>valueChangeListener</code>	JSF EL expression identifying an event-handling method for use when the value of an input component changes. All value change listeners in a view will be processed when the view is submitted.	Input tags

B.2.1. Pass-Through Attributes

Since the JSF HTML tags are responsible for rendering HTML for consumption by the browser, they include pass-through attributes for standard HTML attributes and for DHTML event attributes, which are used to integrate JavaScript with your application. To minimize repetition, we'll summarize the pass-through attributes here rather than include them with each individual tag.

All standard HTML 4.0 elements are supported by the corresponding JSF UI tags. The most common attributes are listed in [Table B-20](#). Specific interpretation of some of these values depends on the element itself.

Table B-20. Common HTML 4.0 pass-through elements

Attribute	Description
<code>accesskey</code>	Identifies a hotkey that will give focus to an element (usually used in accessible browsers in combination with an external command).
<code>accept</code>	The content type or types generated by the form.
<code>alt</code>	Alternative text to display for this component in nongraphical browsers.
<code>border</code>	Width of the element's border, in pixels.
<code>charset</code>	Character set in use.
<code>cols</code>	Number of columns in a text area.
<code>coords</code>	Coordinates for image map elements.
<code>dir</code>	Text direction.
<code>disabled</code>	Indicates whether or not a form input element is disabled.
<code>maxlength</code>	The maximum length of the input allowed in a field.
<code>readonly</code>	Indicates that an input component is read-only.
<code>rows</code>	Number of rows in a TEXTAREA.
<code>shape</code>	The shape of an image map region. Acceptable values are <code>default</code> , <code>circle</code> , <code>poly</code> , and <code>rect</code> .
<code>size</code>	The size of an input field, for display purposes.

Attribute	Description
<code>style</code>	Inline CSS stylesheet information for the current element.
<code>tabindex</code>	The tab index for this element. Used by the browser to determine the order in which fields receive focus when the user presses the Tab key.
<code>target</code>	The name of the target frame for a link or a form submission.
<code>title</code>	A description of the element's content, generally used to create more easily accessible applications.
<code>width</code>	The width of this element.

The DHTML pass-through attributes are listed in [Table B-21](#). Like the HTML 4.0 attributes, DHTML attributes are only supported for HTML components that originally supported the attribute.

Table B-21. DHTML pass-through attributes

Attribute	Description
<code>onblur</code>	Script invoked when an element loses focus
<code>onchange</code>	Script invoked when an element's value changes
<code>onclick</code>	Script invoked when an element is clicked on
<code>ondblclick</code>	Script invoked when an element is double-clicked on
<code>onfocus</code>	Script invoked when an element gains focus
<code>onkeydown</code>	Script invoked when a key is pressed while the element has focus
<code>onkeypress</code>	Script invoked when a key is pressed and released while the element has focus
<code>onkeyup</code>	Script invoked when a key is released
<code>onmousedown</code>	Script invoked when the mouse is clicked while the pointer is positioned over the element
<code>onmousemove</code>	Script invoked when the mouse moves over the element

Attribute	Description
<code>onmouseout</code>	Script invoked when the mouse moves out of the element
<code>onmouseover</code>	Script invoked when the mouse moves over the element
<code>onreset</code>	Script invoked when the form is reset
<code>onselect</code>	Script involved when text within the element is selected
<code>onsubmit</code>	Script invoked when the form is submitted

B.2.2. <h:column>

Represents a column of data within a multicolumn user interface component. See <h:dataTable> for more information.

B.2.3. <h:commandButton>

Renders an HTML form submission button element and wires the button to an outcome.

Table B-22. Attributes for <h:commandButton>

Attribute	Description
<code>action</code>	JSF EL expression specifying the navigation outcome from clicking this button. The outcome may be hardcoded or produced by a backing bean method that returns a string.
<code>actionListener</code>	Method binding for an action listener for this button. Must map to a valid listener method.
<code>image</code>	Absolute or relative URL pointing to an image file. If specified, the <code>type</code> attribute will be ignored and the button will be generated as type <code>image</code> .
<code>immediate</code>	Indicates that the results of pressing this button should be applied to the application immediately (during the first phase of JSF request processing, before validation) rather than during the Invoke Application phase.
<code>type</code>	The type of button. Valid values are <code>submit</code> and <code>reset</code> .
<code>value</code>	The text displayed on the button.

B.2.4. <h:commandLink>

Renders an HTML link, which, when clicked, will submit the current view for processing by the JSF engine.

Table B-23. Attributes for <h:commandLink>

Attribute	Description
<code>action</code>	JSF EL expression specifying the navigation outcome from clicking the link. The outcome may be hardcoded or produced by a backing bean method that returns a string.
<code>actionListener</code>	Method binding for an action listener for this button. Must map to a valid listener method.
<code>charset</code>	The character encoding of the resource at the other end of the link.
<code>immediate</code>	Indicates that the results of clicking the link should be applied to the application immediately (during the first phase of JSF request processing, before validation) rather than during the Invoke Application phase.
<code>type</code>	The content type associate with the link (HTML pass-through).
<code>value</code>	The link text.

B.2.5. <h:dataTable>

Renders an HTML table, iterating through a set of JavaBeans and processing the body content for each bean. This tag also supports two facets, `header` and `footer`, for creating table headers and footers.

Table B-24. Attributes for <h:dataTable >

Attribute	Description
<code>first</code>	Index of the first row of data to display. Defaults to <code>0</code> , which is the first row.
<code>rows</code>	Number of rows to display. If omitted or set to <code>0</code> , all rows will be displayed.

Attribute	Description
<code>value</code>	Valid values for this attribute are an array, a collection, a <code>javax.faces.model.DataModel</code> , a <code>java.sql.ResultSet</code> , a <code>javax.servlet.jsp.jstl.sql.ResultSet</code> , or a <code>javax.sql.RowSet</code> .
<code>var</code>	The name of a request-scoped variable to which each object in the underlying data model will be assigned.
<code>columnClasses</code>	A comma-separated list of CSS classes to be applied, in order, to the columns in the table. Multiple classes may be applied to a single column by separating them by spaces within the comma-separated list. If there are more columns than CSS classes, the extra columns will not have stylesheet information.
<code>footerClass</code>	CSS class to apply to footer elements.
<code>headerClass</code>	CSS class to apply to header elements.
<code>rowClasses</code>	Comma-separated list of CSS classes to be applied in sequence to the rows within the table. If there are more rows than CSS classes, the sequence will repeat until the table is complete.
<code>width</code>	Width of the table.

B.2.6. <h:form>

Root element for an HTML form. Accepts attributes for all standard HTML form tag elements.

Table B-25. Attributes for <h:form>

Attribute	Description
<code>enctype</code>	The content type used to submit the form. The default is <code>application/x-www-form-urlencoded</code> .

B.2.7. <h:graphicImage>

Renders an HTML `` tag.

Table B-26. Additional attributes for <h:graphicImage>

Attribute	Description
<code>ismap</code>	Indicates that this image is associated with a server-side image map
<code>usemap</code>	The name of a client-side image map definition associated with this image
<code>value</code>	The path to the image

B.2.8. <h:inputHidden>

Renders a hidden form field. Used primarily when using JSF to generate forms that will be submitted to non-JSF destinations.

Table B-27. Attributes for <h:inputHidden>

Attribute	Description
<code>value</code>	The value for the field, if no binding is used

B.2.9. <h:inputSecret>

Renders an HTML password field. Supports pass-through attributes for the <input> HTML tag.

B.2.10. <h:inputText>

Renders an HTML text input field. Supports pass-through attributes for the <input> HTML tag.

B.2.11. <h:inputTextarea>

Renders an HTML text area input field. The content of the text area is specified by the value or binding attributes, not, as in the HTML <textarea> tag, by the tag contents.

Table B-28. Attributes for <h:inputTextarea>

Attribute	Description
<code>cols</code>	The number of columns in this text area
<code>rows</code>	The number of rows in this text area
<code>title</code>	Title of this text area, used for accessible browsing
<code>value</code>	JSF EL expression evaluating to the value of the text area, if no binding is used

B.2.12. <h:message>

Displays messages, such as validation errors, associated with a particular component. Specifically, the tag will display a single message associated with a particular component.

The various style and class attributes of this tag allow you to specify a CSS class for particular message types or to directly define the style to be used.

Table B-29. Attributes for <h:message>

Attribute	Description
<code>for</code>	Specifies the ID of the component to display a message for. Required.
<code>showDetail</code>	Indicates whether the message detail will be displayed. Default is <code>true</code> .
<code>showSummary</code>	Indicates whether the message summary will be displayed. Default is <code>false</code> .
<code>errorClass</code>	CSS style class for messages with severity level ERROR.
<code>errorStyle</code>	CSS style for messages with severity level ERROR.
<code>fatalClass</code>	CSS style class for messages with severity level FATAL.
<code>fatalStyle</code>	CSS style for messages with severity level FATAL.
<code>infoClass</code>	CSS style class for messages with severity level INFO.
<code>infoStyle</code>	CSS style for messages with severity level INFO.

Attribute	Description
<code>warnClass</code>	CSS style class for messages with severity level WARN.
<code>warnStyle</code>	CSS style for messages with severity level WARN.

B.2.13. <h:messages>

Displays an HTML table or list containing all of the messages for the current view. The standard `style` and `styleClass` attributes should be used to provide formatting.

The defaults for the `showDetail` and `showSummary` attributes are the reverse of the defaults for the `<h:message>` tag.

Table B-30. Attributes for <h:messages>

Attribute	Description
<code>globalOnly</code>	Indicates that only global messages (not associated with a particular component) are to be displayed. Default is <code>false</code> .
<code>layout</code>	Format to use when displaying the list of messages. Options are <code>"list"</code> (the default) and <code>"table"</code> .
<code>showDetail</code>	Indicates whether the message detail will be displayed. Default is <code>false</code> .
<code>showSummary</code>	Indicates whether the message summary will be displayed. Default is <code>True</code> .
<code>errorClass</code>	CSS style class for messages with severity level ERROR.
<code>errorStyle</code>	CSS style for messages with severity level ERROR.
<code>fatalClass</code>	CSS style class for messages with severity level FATAL.
<code>fatalStyle</code>	CSS style for messages with severity level FATAL.
<code>infoClass</code>	CSS style class for messages with severity level INFO.
<code>infoStyle</code>	CSS style for messages with severity level INFO.

Attribute	Description
<code>warnClass</code>	CSS style class for messages with severity level WARN.
<code>warnStyle</code>	CSS style for messages with severity level WARN.

B.2.14. <h:outputFormat>

Outputs formatted text output, following the rules for the `java.text.MessageFormat` class. To pass parameters to the message formatter, embed one or more `<f:param>` tags in the body of the `<h:outputFormat>` tag.

Table B-31. Attributes for <h:outputFormat>

Attribute	Description
<code>value</code>	The string to format

B.2.15. <h:outputLabel>

Generates an HTML `<label>` tag for a component.

Table B-32. Attributes for <h:outputLabel>

Attribute	Description
<code>for</code>	The ID of the component that this label references

B.2.16. <h:outputLink>

Generates an HTML link tag (`<a>`) around a specified piece of content. To display a link, the `<h:outputLink>` tag must surround another JSF tag, such as `<h:outputText>`, `<h:graphicImage>`, or `<f:verbatim>`, that provides the link text or image.

Table B-33. Additional attributes for <h:outputLink>

Attribute	Description
<code>title</code>	An optional title for this link
<code>value</code>	The destination of this link

B.2.17. <h:outputText>

Outputs a bound value or a specified value.

B.2.18. <h:panelGrid>

Generates an HTML table. Each JSF user interface component within the body of the tag will be used as a cell value. To combine multiple JSF elements into a single cell, enclose them in an `<h:panelGroup>` component.

This tag supports a `header` and `footer` facet (see `<f:facet>`) for rendering table headers and footers.

Table B-34. Attributes for <h:panelGrid>

Attribute	Description
<code>bgcolor</code>	Background color for this table, as a color constant or RGB value.
<code>border</code>	Width of the border around this table, in pixels. The default is 0.
<code>cellpadding</code>	Specifies the amount of space between the border of a cell and its contents.
<code>cellspacing</code>	Specifies the spacing between cells within the table and between cells and the edge of the table.
<code>columnClasses</code>	A comma-separated list of CSS classes to be applied, in order, to the columns in the table. Multiple classes may be applied to a single column by separating them by spaces within the comma-separated list. If there are more columns than CSS classes, the extra columns will not have stylesheet information.
<code>columns</code>	Specifies the number of columns in this table. The number of rows is determined automatically.
<code>footerClass</code>	CSS class to apply to any footer generated for this table.

Attribute	Description
<code>frame</code>	Indicates which sides of the table will be visible. Values are <code>none</code> (default), <code>above</code> , <code>below</code> , <code>hsides</code> (top and bottom), <code>vsides</code> (right and left), <code>lhs</code> (left side), <code>rhs</code> (right side), <code>box</code> (all four sides, with a line), and <code>border</code> (all four sides, width specified by the <code>border</code> attribute).
<code>headerClass</code>	CSS class to apply to any header generated for this table.
<code>rowClasses</code>	Comma-separated list of CSS classes to be applied in sequence to the rows within the table. If there are more rows than CSS classes, the sequence will repeat until the table is complete.
<code>rules</code>	Indicates which rules will be rendered between cells in the table. Values are <code>none</code> (the default), <code>groups</code> (between groups of rows only), <code>rows</code> (between individual rows), <code>cols</code> (between columns), and <code>all</code> (between all rows and columns).
<code>summary</code>	Summary of the contents of this table.

B.2.19. <h:panelGroup>

Groups several JSF components (contained in the tag body) into a single unit. Panel groups are used to control component rendering en masse and to group components for use within the `<h:panelGrid>` tag.

B.2.20. <h:selectBooleanCheckbox>

Renders an HTML checkbox, bound to a Boolean property on a backing bean or from a Boolean value attribute.

B.2.21. <h:selectManyCheckbox>

Renders a set of checkboxes. The binding for the component must be an array or a collection capable of holding the selected values. The content of the set of checkboxes must be provided via an `<f:selectItems>` tag or a set of one or more `<f:selectItem>` tags embedded within the body of the `<h:selectManyCheckbox>` tag.

Table B-35. Attributes for <h:selectManyCheckbox>

Attribute	Description
<code>layout</code>	Specifies the order in which the checkboxes should be laid out. Options are <code>lineDirection</code> (the default) and <code>pageDirection</code> (a vertical arrangement).

B.2.22. <h:selectManyListbox>

Renders a listbox using the HTML `<select>` tag. The binding for the component must be an array or a collection capable of holding the selected values. The contents of the listbox must be provided via an `<f:selectItems>` tag or a set of one or more `<f:selectItem>` tags embedded within the body of the `<h:selectManyListbox>` tag.

Table B-36. Attributes for <h:selectManyListbox>

Attribute	Description
<code>size</code>	Indicates the number of items to display on screen at once

B.2.23. <h:selectManyMenu>

Generates a multiselection menu. For HTML rendering, this is equivalent to a `<select>` tag with the `multiple` attribute set to `true` and the `size` attribute set to `1`. We're not saying that this is an entirely useless tag for HTML user interfaces, but let's just say we haven't found a use for it either.

Like `<h:selectManyListbox>`, the menu contents must be provided via an `<f:selectItems>` tag or a set of `<f:selectItem>` tags.

B.2.24. <h:selectOneListbox>

Renders a listbox using the HTML `<select>` tag. The binding for the component must be an array or a collection capable of holding the selected values. The contents of the listbox must be provided via an `<f:selectItems>` tag or a set of one or more `<f:selectItem>` tags embedded within the body of the `<h:selectManyListbox>` tag.

The listbox rendered by the tag allows only a single item to be selected at a time.

Table B-37. Attribute for <h:selectOneListbox>

Attribute	Description
<code>size</code>	Indicates the number of items to display on screen at once

B.2.25. <h:selectOneMenu>

For HTML, this tag is about as useful as the `<h:selectManyMenu>` tag, which isn't particularly useful. Renders a menu item, allowing a single selection. Slightly more useful for WML or other non-HTML output formats that support a native menu component.

B.2.26. <h:selectOneRadio>

Generates a set of radio buttons using a set of selection items.

Table B-38. Additional attributes for `<h:selectOneRadio>`

Attribute	Description
<code>layout</code>	Specifies the order in which the radio buttons should be laid out. Options are <code>lineDirection</code> (the default) and <code>pageDirection</code> (a vertical arrangement).

Appendix C. Enterprise JavaBeans Query Language Syntax

EJB Query Language (EJB QL) was introduced in the EJB 2.0 specification and enhanced in EJB 2.1 as a scheme for specifying the logic for finder and select methods on EJBs using container-managed persistence (CMP). As described in detail in [Chapter 6](#), the CMP model in EJB 2.1 includes an abstract schema defined around your entity beans and their relationships, where beans become abstract tables, the CMP fields on the beans become abstract columns of sorts, and the relationships between beans act as abstract foreign key constraints. EJB QL allows you to specify queries based on this abstract schema, to be used by the container in the implementation of finder and select methods for your entity beans.

EJB QL is similar in syntax (naturally) to SQL, but is much simpler, since EJB QL needs to deal only with selecting beans and CMP fields from the abstract schema. EJB QL can't be used to update or delete entity beans (the bean client interfaces are used for that), and it can't be used to adjust the structure of the abstract schema. "Schema changes" are accomplished using the *ejb-jar.xml* deployment descriptor: new entity beans are defined, or unneeded ones are removed, and the deployment descriptor is adjusted to include/exclude these beans and define relationships between them. Full details on EJB deployment descriptors can be found in [Appendix A](#), and some examples of their use are shown in [Chapter 6](#).

C.1. Basic Structure of EJB QL Queries

EJB QL queries contain the following four clauses:

SELECT clause (required)

Indicates what beans or CMP fields are being returned

FROM clause (required)

Indicates what subset of the abstract schema is being queried

WHERE clause (optional)

Indicates optional conditional logic to use in selecting the results

ORDER BY clause (optional) (added in EJB 2.1)

Indicates how the results should be sorted when returned

So an EJB QL query takes the following high-level form:

```
SELECT <select clause> FROM <from clause>
  [WHERE <where clause>] [ORDER BY <sort clause>]
```

Queries are declared in an EJB deployment descriptor using the following format:

```
...
<entity>
  ...
  <query>
    ...
    <ejb-ql>
      <[!CDATA[SELECT p FROM Person p WHERE p.name IS NOT NULL
        ORDER BY p.lastName, p.firstName]]>
    </ejb-ql>
    ...
  </query>
```

```
...  
</entity>  
...
```

The complete syntax for EJB deployment descriptors can be found in [Appendix A](#).

In a query, the **FROM** clause specifies the domain of the abstract schema that is being queried. This is done by declaring variables that are bound to different elements of the abstract schema. The **SELECT** clause is used to specify what elements of the schema you want returned as the result set, as a subset of the domain defined by the **FROM** clause. The optional **WHERE** clause is used to specify the conditions used to match the elements to return, and the optional **ORDER BY** clause is used to order the result set by one or more data fields contained in the results.

Since the **FROM** clause defines the variables involved in the query, we'll start there, then describe the **SELECT** clause. After these required clauses, we discuss the optional **WHERE** clause and its conditional expressions and the optional **ORDER BY** clause for sorting the returned results.

 PREY



C.2. FROM Clause

The **FROM** clause in an EJB QL query declares the subset of elements from the abstract schema that is the target of the query. Any elements referenced in the conditional expressions in the **WHERE** clause as well as the element that is specified as the return type in the **SELECT** clause must be included in the **FROM** clause declarations.

The **FROM** clause is made up of a list of variable declarations. Each variable is declared with a type and a unique variable name. Variable names can be any valid Java identifier. The types of the variables are taken from the abstract schema elements defined by the entity EJBs. These variables are declared as either *range variables* or *collection member variables*.

C.2.1. Range Variables

Range variables are query variables that take the type of one of the entity EJBs in your abstract schema. They are called range variables because their actual value can range over any of the actual entities present in the persistent store at the time that the query is executed (limited by the conditions in the **WHERE** clause, if present). For example, we used the following EJB QLquery for one of the finder methods on our **Profile** EJB in [Chapter 6](#):

```
SELECT OBJECT(p) FROM ProfileBean p WHERE p.entriesBytes IS NULL
```

In this query, the **FROM** clause declares a single range variable, **p**, and it's declared to range over the EJB type **ProfileBean**. **ProfileBean** is the abstract schema type that we declared for our **Profile** EJB. This is analogous to declaring a query variable in SQL that ranges over the rows in a particular table. In the case of EJB QL, rather than being satisfied by any of the rows in a table, the variable can take the value of any instance of the EJB present in the persistent store.

You can also optionally use the **AS** operator when declaring range variables:

```
SELECT OBJECT(p) FROM ProfileBean AS p WHERE p.entriesBytes IS NULL
```

The **AS** operator is simply an optional syntactic element and doesn't have any affect on the query itself.

You can declare multiple range variables of the same type in the **FROM** clause, of course:

```
SELECT OBJECT(p1) FROM ProfileBean p1, ProfileBean p2 WHERE ...
```

C.2.2. Collection Member Variables

A collection member variable is declared to range over the values obtained from an entity bean's container-managed relationship fields (CMR fields). These variables are declared using a path expression (prefixed by the `IN` operator) that specifies the particular CMR field on the bean. For example, suppose we extend our `Profile` EJB example to include a `Person` entity bean that contains a one-to-many CMR field that refers to all the `Profiles` owned by the `Person`. We made a similar extension in [Chapter 6](#), but in that case, we defined the relationship as one-to-one. Here, to demonstrate the collection member variable, we assume it's one-to-many. We can declare a collection member variable that ranges over all `Profiles` owned by a `Person` with an EJB QL query, such as:

```
SELECT DISTINCT OBJECT(prof) FROM Person p, IN(p.profiles) prof WHERE ...
```

In the `FROM` clause, we declare a range variable `p` that ranges over all `Person` beans, then declare a collection member variable `prof` that ranges over the members of the `profiles` `cmr-field` on the matching `Person` beans. This type of the collection member variable is the abstract schema type of the collection members. In this case, the `profiles` CMR field contains `ProfileBean` EJBs, so that is the type of the `prof` variable.



C.3. SELECT Clause

As with standard SQL queries, the `SELECT` clause in EJB QL can specify what you want returned from the persistent store. The `SELECT` clause takes the form:

```
SELECT [ DISTINCT ] [ <schema variable> | <aggregation expression> ]
```

The clause contains a single variable reference to indicate the type of result returned by the query, or an aggregation function can be used to calculate some formula on the results. The optional `DISTINCT` operator specifies that the return results should be unique. The variable reference can either be to a specific variable declared in the `FROM` clause, or it can be a path expression that refers to a single-valued CMR field on an abstract schema type. In our previous example, we used a specific variable reference:

```
SELECT OBJECT(p) FROM ProfileBean AS p WHERE p.entriesBytes IS NULL
```

This `SELECT` clause specifies `OBJECT(p)` as the return value of the query. If a single query variable is being used in the `SELECT` clause, then it has to be qualified with the `OBJECT` operator. The `p` variable is declared in the `FROM` clause as a range variable with type `ProfileBean`, so the query results are one or more `ProfileBeans` that match the conditions in the `WHERE` clause. In our other query example:

```
SELECT DISTINCT OBJECT(prof) FROM Person p, IN(p.profiles) prof WHERE ...
```

the `SELECT` clause specifies `OBJECT(prof)` as the return value of the query. The `prof` variable is declared in the `FROM` clause as a collection member variable that refers to the `profiles` CMR field in the `Person` bean. This CMR field contains EJBs with the abstract schema type of `ProfileBean`, so the query results again are `ProfileBean` objects.

You can also use path expressions to specify the return type of the query. Suppose, for example, that our `Person` abstract schema type has a one-to-one relationship named `employer` with an `Organization` bean that represents the organization that employs the person. We can use a path expression in our `SELECT` clause to have the query return the `Organizations` of the selected `Person` beans:

```
SELECT DISTINCT p.employer FROM Person p WHERE ...
```

Note that the path expression has to be single-valued. If a CMR field is many-valued, then you need to declare a collection member variable in the `FROM` clause and refer to that variable in the `SELECT` clause. We did this in the previous example when we used the `prof` variable to refer to the `ProfileBeans` associated with the `Person` beans selected by the query.

If the EJB QL query is for a finder method, then the type specified in the `SELECT` clause has to be the abstract schema type of an EJB, since finder methods can only return EJB references. If the query is for a select method, then the `SELECT` clause can also specify CMP fields on abstract schema types, using a path expression. If, for example, our `Person` bean has a `name` CMP field, we could use the following query for a select method on the `Person` bean:

```
SELECT p.name FROM Person p WHERE ...
```

In this case, the return type of the select method needs to match the type of the `name` CMP field.

C.3.1. Aggregation expressions

EJB 2.1 introduced several aggregation functions that can be used within the `SELECT` clause of queries. The argument to these aggregation functions is a path expression or a variable reference. The aggregation functions supported in EJB 2.1 are:

AVG

Calculate the average of the values of the argument. The argument data must be numeric.

COUNT

Return the number of results found in the argument expression.

MAX

Return the maximum value found in the argument expression. The argument data can be any orderable types (numeric, string, character, date).

MIN

Return the minimum value found in the argument expression. The argument data can be any orderable type (numeric, string, character, date).

SUM

Return the sum of the values in the aggregate expression. The argument data must be numeric.

You can also optionally use the `DISTINCT` keyword before the argument expression in aggregation functions. Any duplicate values are removed from the set before the function is applied to them. `NULL` values are always removed from the data set before applying the aggregation function, regardless of whether you use `DISTINCT` or not.

The following example returns the number of people whose first names are John:

```
SELECT COUNT(p) FROM Person p WHERE p.firstName = 'John'
```

This example returns the number of different first names contained in the `Person` data:

```
SELECT COUNT(DISTINCT p.firstName) FROM Person p
```

This example returns the average age of all people:

```
SELECT AVG(p.age) FROM Person p
```

This example returns (alphabetically) the last names of all the people in the database:

```
SELECT MAX(p.lastName) FROM Person p
```

C.3.2. DISTINCT Queries

The `DISTINCT` option is used in the `SELECT` clause to specify that any duplicate values in the result set should be eliminated. It makes sense only when used in EJB QL queries for methods that return a `Collection` of some kind. If a finder or select method returns a single value, then the result set needs to be unique by definition. If a select method is declared as `ajava.util.Set`, then the results set has to be distinct, so the EJB container has to assume the `DISTINCT` keyword in these cases even if it's not specified in the EJB QL for the method. For example, if we have a select method, such as the following, defined in an EJB implementation class:

```
public abstract java.util.Set.ejbSelectPreferredClients(...)
    throws FinderException;
```

and we put this entry in our *ejb-jar.xml*/deployment descriptor to specify the EJB QL for this method:

```
<ejb-ql>  
  <?[CDATA[SELECT OBJECT(c) FROM Customers c WHERE c.preferred IS NOT NULL]]>  
</ejb-ql>
```

the EJB container still executes the query as if the `DISTINCT` clause had been included in the EJB QL.



C.4. WHERE Clause

The **WHERE** clause of an EJB QL query is used to specify conditional logic that is used to select a subset of the beans references in the **FROM** clause. Any beans that you want to refer to in your conditional query logic must be declared in the **FROM** clause of the query.

The syntax of the **WHERE** clause mirrors the syntax of SQL **WHERE** clauses. The **WHERE** clause is made up of literals and variables composed into various conditional expressions. The variables used in a **WHERE** clause can be query variables declared in the **FROM** clause, path expressions derived from the query variables, or input parameters provided in the arguments to the corresponding finder or select method. The conditional expressions in the **WHERE** clause can be composed using arithmetic operators, comparison operators, or logical operators.

C.4.1. Literals

Literals are strings, numeric values, or Boolean values. String literals are enclosed in single quotes:

```
SELECT OBJECT(p) FROM ProfileBean p WHERE p.name = 'Fred'
```

If you want to include a single quote in a literal, use two single quotes:

```
SELECT OBJECT(p) FROM ProfileBean p WHERE p.name = 'Fred''s profile'
```

Numeric literals can be either exact integer values or approximate floating-point values. Integer values are specified using a sequence of digits with no decimal point:

```
SELECT OBJECT(prof) FROM ProfileBean prof WHERE prof.id < 1000
```

Floating-point values are specified using either scientific notation or a numeric value with a decimal point:

```
SELECT OBJECT(p) FROM Person p WHERE p.salary >
    +5e4 AND p.overhead < 1.32
```

Floating-point values can be in the range supported by the Java `double` type.

Boolean literals are specified using `true` or `false`. These values are case-insensitive.

C.4.2. Variables

Variables used in expressions in a `WHERE` clause can be query variables declared in the `FROM` clause, input parameter variables that refer to arguments to the finder or select method, or path expressions based on query variables or input parameter variables.

C.4.2.1. Simple query variables

Query variables can be used directly in conditional expressions when it is necessary to directly compare abstract schema types. This makes sense only in comparison expressions in which the equality of two entity beans is being compared. For example:

```
SELECT OBJECT(p) FROM Person p1, Person p2, Organization o1, Organization
    o2 WHERE o1 = p1.employer AND o2 = p2.employer AND o1 = o2 AND ...
```

In this case we're comparing the two `Organization` beans for equality. Equality between entity beans is based on the equality of their primary keys, (e.g., if the two primary keys are equal, then the two entities are considered equal).

C.4.2.2. Input parameter variables

Input variables refer to the arguments of the query's finder or select method. They are specified using a `$` followed by the integer index (starting with 1) of the method argument. For example, if we are defining a query for a finder method declared with the following signature:

```
abstract public Collection findEmployees(String companyName);
```

We can refer to the string argument in the query using an input parameter variable:

```
SELECT OBJECT(e) FROM Person e, Organization o
    WHERE o.name = $1 AND e.employer = o
```

If the input variable is an entity bean object with an abstract schema type, you can also use the input parameter variable in path expressions. For example, if we have a finder method on our `Person` bean:

```
abstract public Collection findCoWorkers(Person p);
```

a corresponding query might be:

```
SELECT OBJECT(w) FROM Person w, Organization o
    WHERE $1.employer = o AND w.employer = o
```

Input parameter variables can be used only in comparison expressions that use the basic comparison operators and the `MEMBER OF` comparison operator, as described in "[Comparison operators](#)" later in this appendix.

C.4.2.3. Path expression variables

Path expressions are specified using either query variables or input parameter variables. A path expression is a variable followed by one or more dot-delimited CMP field or CMR field references. In one of our earlier examples, we used a path expression to refer to the `name` CMP field on the `Organization` bean:

```
SELECT OBJECT(e) FROM Person e, Organization o
    WHERE o.name = $1 AND e.employer = o
```

Here, the path expression `o.name` uses the query variable `Organization o` declared in the `FROM` clause and then refers to the CMP field `name` on the variable.

References to `cmr-fields` on variables can be referenced further in multilevel path expressions. For example, a simplified version of the previous query is:

```
SELECT OBJECT(e) FROM Person e WHERE p.employer.name = $1
```

Here, we're referencing the CMR field `employer` on the `Person` bean, then further referencing the CMP field `name` on the resulting `Organization`. Referencing CMR fields in path expressions is possible only if the relationship is navigable from the bean at the root of the reference. So in this case, we're assuming that the `employer` CMR field is navigable from the `Person` bean, which allows us to reference it as an `Organization` and then further reference the `name` CMP field on the `Organization`.

Path expressions can evaluate to either single values or collection values. If, for example, a CMR field

is one-to-many or many-to-many, a reference to it in a path expression is collection-valued. Path expressions that are collection-valued can be used only in comparison expressions using the `IS EMPTY` operator or the `MEMBER OF` operator (described in "[Comparison operators](#)" later in this appendix).

C.4.3. Functions

EJB QL also includes several built-in functions that can be applied to literals, variables, or values derived from other expressions. They are listed in [Table C-1](#).

Table C-1. EJB QL built-in functions

Function	Description
CONCAT(<string1>, <string2>)	Concatenates the two strings and returns a string value.
SUBSTRING(<string>, <start>, <length>)	Extracts the specified substring from the first argument, and returns a string value or <code>null</code> if the specified substring can't be extracted (e.g., the start position is greater than the length of the string). The <code><start></code> argument is indexed from 1.
LOCATE(<string1>, <string2> [, <start>])	Finds the index of the location of <code><string1></code> in <code><string2></code> , optionally starting at the specified location in <code><string2></code> . The return value is an integer index, indexed from 1. If <code><string1></code> is not found in <code><string2></code> , a value of 0 is returned.
LENGTH(<string val>)	Returns the number of characters in the string value as an integer value.
ABS(<numeric val>)	Returns the absolute value of the given numeric value. The return type is the same as the type of the numeric expression used as the argument.
SQRT(<floating-point val>)	Returns the square root of the given floating-point value as a double.

C.4.4. Conditional Expressions

Conditional expressions are composed of a sequence of one or more other expressions, joined together by logical `AND` or `OR` operators. A single expression is made up of literals and/or variables joined by operators to result in a Boolean value. Conditional expressions follow the usual evaluation/precedence rules: they are evaluated left to right, taking into account operator precedence rules (as discussed next) and grouping of expressions using parentheses.

The operators available in EJB QL for creating conditional expressions fall into three categories: logical operators, arithmetic operators, and comparison operators.

C.4.4.1. Logical operators

The logical operators in EJB QL are **NOT**, **AND**, and **OR**, in precedence order, with the usual semantics: **NOT** is the unary negation operator, **AND** is the binary intersection operator, and **OR** is the binary union operator. These can be applied to any expressions that evaluate to a Boolean value.

C.4.4.2. Arithmetic operators

Arithmetic operators are applied to numeric variables and literals. They are listed in [Table C-2](#).

Table C-2. Arithmetic operators

Operator	Description
+ <numeric expression>	Unary positive operator. Specifies a positive literal value or a variable with an unchanged sign.
- <numeric expression>	Unary negation operator. Specifies a negative literal or a variable whose sign is negated.
<numeric exp> * <numeric exp>	Binary multiplication.
<numeric exp> / <numeric exp>	Binary division.
<numeric exp> + <numeric exp>	Binary addition.
<numeric exp> - <numeric exp>	Binary subtraction.

C.4.4.3. Comparison operators

The comparison operators are used to perform various value and range comparisons on variables and are listed in [Table C-3](#). Some comparison operators can be applied only to variables of a certain type; these restrictions are mentioned in the description where applicable.

Table C-3. Comparison operators

Operators	Description	Examples
<p><exp> = <exp></p> <p><exp> <> <exp></p>	<p>Equality and inequality operators, respectively. The two operands to these operators must be of compatible types: two <code>Strings</code>, two entity beans with the same abstract schema type, or two compatible numeric values.</p>	<p>...FROM Person p</p> <p>WHERE p.age = 65</p> <p>...FROM Employee p</p> <p>WHERE p.dept <> 'Finance'</p> <p>...FROM Employee p</p> <p>WHERE p.title = 'Director'</p> <p>...FROM Person p1,</p> <p>Person p2</p> <p>WHERE p1 <> p2 AND...</p> <p>...FROM Organization a,</p> <p>Person p,</p> <p>IN(p.affiliations) b</p> <p>WHERE a = b...</p>
<p><exp> > <exp></p> <p><exp> >= <exp></p> <p><exp> < <exp></p> <p><exp> <= <exp></p>	<p>Greater-than, greater-than-or-equal, less-than, less-than-or-equal operators, respectively. These operators can be applied only to numeric and <code>String</code> values, not to entity beans.</p>	<p>...FROM Item b</p> <p>WHERE b.size > 4</p> <p>...FROM Item a,</p> <p>Item b</p> <p>WHERE b.name > a.name</p> <p>...FROM Item b</p>

Operators	Description	Examples
		WHERE b.size <= 4
		...FROM Item a, Item b
		WHERE b.name >= a.name
<arith exp1 > [NOT] BETWEEN <arith-exp2 > AND <arith-exp3 >	Checks whether a given numeric value is between two other numeric values. The NOT operator checks that the first numeric value is not within the range. If any arithmetic expression used in a BETWEEN operator is null , the entire expression evaluates to NULL (boolean "unknown").	...FROM Person p WHERE p.age BETWEEN 18 AND 65 ...FROM Person p, Rate r WHERE p.salary NOT BETWEEN r.base AND 1.05 * r.base
<string-exp > [NOT] IN (str-literal [,str-literal]*)	Checks whether a given string value is contained (or not contained) in a list of string literals. The string values are matched exactly. If the string value to compare to the set is null , the expression evaluates to NULL (boolean "unknown").	...WHERE p.name IN ('John', 'Mary') ...WHERE p.dept NOT IN ('Finance', 'Human Resource', 'Corporate')
<string-exp > [NOT] LIKE <pattern > [ESCAPE <esc-char >]	Compares a string value to a pattern. The pattern consists of a sequence of characters, where _ is a wildcard that stands for any single character, and % stands for any sequence of characters. If you want to use the literal _ and % characters in the pattern, escape them with a character and specify the escape character with the optional ESCAPE clause. If the string to compare is null , the value of the LIKE expression is NULL (boolean "unknown").	...WHERE o.title LIKE '%Professor%' ...WHERE p.state_prov NOT LIKE '%Province'

Operators	Description	Examples
		<p>...WHERE p.version LIKE</p> <p>'__1._' ESCAPE '\'</p>
<exp> IS [NOT] NULL	Checks if the value of the expression is <code>null</code> . The expression must be single-valued and not evaluate to a collection.	<p>...FROM Person o</p> <p>WHERE o.employer</p> <p>IS NOT NULL</p> <p>...FROM Publication p</p> <p>WHERE p.title</p> <p>IS NULL</p>
<collection-exp> IS [NOT] EMPTY	Checks whether a given collection-valued expression is empty. The collection expression must evaluate to a collection of entity beans.	<p>...FROM Person p</p> <p>WHERE p.profiles</p> <p>IS NOT EMPTY</p> <p>...FROM Order o</p> <p>WHERE o.items</p> <p>IS EMPTY</p>
<exp> [NOT] MEMBER OF <collection-exp>	Checks whether a given single value is contained within a given collection. The value being checked for membership must be a singular value (not a collection), and the collection expression must evaluate to a collection of entity beans.	<p>...FROM Person p,</p> <p>Organization o</p> <p>WHERE p</p> <p>MEMBER OF</p> <p>o.employees</p> <p>...FROM Item i,</p> <p>Warehouse w</p> <p>WHERE</p> <p>i NOT MEMBER</p> <p>OF</p> <p>w.backorders</p>



C.5. ORDER BY Clause

EJB 2.1 introduced an `ORDER BY` clause, to mimic the functionality of the corresponding SQL clause. With an `ORDER BY` clause, you can instruct the container to sort the results of the query before returning them to the calling application. This can save programming time and potentially be more efficient than doing the sort in the application code. Note the key word, "potentially." If the EJB container is using an underlying relational engine to presort the results, then it's likely that this will be much faster than sorting the data yourself in Java code. But there are no guarantees about the expected performance of the container's sort algorithms, so using `ORDER BY` is not only a convenience, but also a leap of faith.

Note that, if you use an `ORDER BY` clause in your EJB QL, then the `SELECT` clause must specify either a single range variable (e.g., `OBJECT(p)`) or a simple path expression (e.g., `p.firstName`).

The `ORDER BY` clause has the following syntax:

```
ORDER BY <order item> [, <order item>]*
```

Each order item must be an orderable field of the entity specified in the `SELECT` clause. For example, if we wanted to select all people with the first name Judy, and order them by their last names, then we'd use a query such as:

```
SELECT OBJECT(p) FROM Person p WHERE p.firstName = 'Judy'  
      ORDER BY p.lastName
```

If we wanted to select all people, and sort them by last name and then by first names, we would use a query something like this:

```
SELECT OBJECT(p) FROM Person p  
      ORDER BY p.lastName, p.firstName
```

Appendix D. SQL Reference

There are dozens of different database management systems on the market today, from nearly as many vendors. Developing applications that are more or less database-independent requires a standardized interface to the underlying data. Since the early 1980s, this interface has been SQL, a sophisticated database manipulation language.^[1]

[1] The acronym expands out to either Structured Query Language (based on the original IBM acronym from the 1970s) or Standard Query Language (which has been more popular in recent years). Perhaps because of this confusion, most people just say "SQL," pronounced either "see-quel" or "ess-cue-ell."

Unlike Java, SQL is a declarative language. It allows users to specify particular actions on the database and retrieve the results of those actions. It specifies a set of standardized data types and standard error messages, but it lacks procedural constructs. There are no conditionals or loops standard in SQL.

There are several versions of the SQL standard. SQL-86 and SQL-89 have been superseded by SQL-92, which is supported by most database vendors, although there are a number of platform-specific variations. Many databases also include additional data types, operators, and functions beyond those specified in the SQL-92 standard. In addition, there are three levels of SQL-92 conformance: entry-level, intermediate, and full. Many products support only the entry-level SQL-92 standard, leaving out some advanced features. JDBC drivers are supposed to provide entry-level functionality and, for the most part, they do.

This appendix presents a brief introduction to the structure of a relational database system and a quick reference to the most commonly used SQL commands. The complete set of SQL commands is simply too large to cover here; even a concise SQL reference can run to several hundred pages. We have endeavored to provide the information that most client-side programmers need. For a complete introduction to most aspects of SQL, we highly recommend *SQL Clearly Explained* by Jan Harrington (AP Professional).

D.1. Relational Databases

Data storage and retrieval are two of the biggest tasks facing most enterprise applications. There are lots of ways to store data on a disk, but for large-scale applications, a relational database management system (RDBMS) is far and away the most popular choice.

Data in an RDBMS is organized into tables, which contain rows and columns. You can think of an individual table as a spreadsheet with a little more organization: column data types are fixed, and there may be rules governing the formatting of each column. This alone is enough for a database system (plain DBMS). A relational database system has one other key attribute: individual tables can be *related* based on some common data. [Figure D-1](#) shows three tables in a relational structure. The `CUSTOMERS` table is related to the `ORDERS` table based on the `CUSTOMER_ID` field, and the `ORDERS` table is related to the `ITEMS` table based on the `ORDER_ID` field. SQL provides a standardized means of accessing the data in tables and working with the relationships between tables.

Figure D-1. Three related tables

The highest-level organizational concept in an RDBMS system is a *cluster*.^[2] A cluster contains one or more *catalogs*, which are usually the highest-level object a programmer ever has to deal with. A catalog contains a group of schemas. A schema contains a group of tables and other objects and is generally assigned to a particular application or user account. Generally, the database administrator is responsible for dealing with clusters and catalogs. Most users work within a particular schema.

^[2] The naming comes from the older SQL standards, where a cluster represented physical storage space. This is no longer the case, but the name persists.

To reference a table within a particular schema, separate the schema name and the table name with

a dot:

schema_name.table_name

To reference a particular column within a particular schema:

schema_name.table_name.column_name

To set a particular schema as the default, use the `SET SCHEMA` SQL statement:

```
SET SCHEMA schema_name
```

When you log into a database, you are generally assigned a default schema. When accessing tables within the default schema, you may omit the schema name.

When creating new objects, the names you assign to them must be unique within the schema. SQL-92 allows names up to 128 characters, including letters, numbers, and the underscore (`_`) character.

D.2. Data Types

Each column in a table has a particular data type associated with it. SQL-92 defines a fairly broad set of data types. Different RDBMS packages provide extensions to these basic types, for multimedia data, Java objects, and so on. Oracle's `NUMBER` data type, for instance, allows database designers to specify the exact precision of integer and floating-point data types. The basic SQL-92 types are listed in [Table D-1](#). The names in parentheses are SQL-92-specified alternates for the main type names.

Table D-1. SQL-92 data types

SQL data type	Description
<code>INTEGER (INT)</code>	A signed integer value. The number of bits represented is implementation-dependent.
<code>SMALLINT</code>	A smaller signed integer value, used when storage or memory is at a premium. Generally 8 bits but implementation-dependent.
<code>NUMERIC</code>	A signed fixed-precision decimal. When creating a <code>NUMERIC</code> , you must specify the total length of the number (including the decimal point) and the number of decimal places. <code>NUMERIC(8, 4)</code> allows three digits, a decimal point, and four more digits.
<code>DECIMAL</code>	Defined the same way as a <code>NUMERIC</code> but may store additional precision (more decimal places).
<code>REAL</code>	A single-precision floating-point value. Range and accuracy are implementation-dependent.
<code>DOUBLE PRECISION (DOUBLE)</code>	A double-precision floating-point value. Range and accuracy are implementation-dependent, but are equal to or better than a <code>REAL</code> .
<code>BIT</code>	A fixed number (one or more) of bits. A length specifier (<code>BIT(n)</code>) is optional. The default size is 1 bit.
<code>BIT VARYING</code>	Storage for up to n bits (<code>BIT VARYING (n)</code>). Many databases have alternate implementations of this data type, such as Oracle's <code>VARBINARY</code> .
<code>DATE</code>	A date value (day, month, and year).
<code>TIME</code>	A time value. Precision is implementation-dependent.
<code>TIMESTAMP</code>	A date and time.
<code>CHARACTER (CHAR)</code>	A fixed-length character string. Specified as <code>CHAR(n)</code> . Unused characters are padded with blanks. The default size is 1.

SQL data type	Description
<code>CHARACTER VARYING (VARCHAR)</code>	A variable length string, up to size <i>n</i> . Specified as <code>VARCHAR(<i>n</i>)</code> .
<code>INTERVAL</code>	A date or time interval. Not directly supported by JDBC.

Note that the data types given here aren't the same as those in [Table 8-1](#) in [Chapter 8](#). When mapping between SQL types and Java types, a JDBC driver matches the physical data type (either SQL-92 or database-specific) and the closest Java type. For example, calling the `getType()` method of `ResultSet` on a `BIT VARYING` field generally returns a `Types.LONGVARBINARY` value. This translation allows most JDBC programs to switch between different databases without losing functionality. Also, many databases implement additional types: the `FLOAT` type is so common that many people think it is required by the specification.

[← PREV](#)

D.3. Schema Manipulation Commands

SQL includes two broad classes of commands. The first class is schema manipulation commands, which allow the creation, modification, and deletion of high-level database objects such as tables. This section describes these commands. We've provided a syntax summary for each command. In case you aren't familiar with the style, items in square brackets are optional or not always required, while items in curly braces are either always required or required within the context of some optional item. A vertical bar (|) indicates a choice, while an ellipsis indicates that an entry may be repeated any number of times. Items in uppercase *constant width* are part of the SQL statement, while items in lowercase *constant width italic* represent names and values that you supply when using the statement.

D.3.1. CREATE TABLE

As its name says, the `CREATE TABLE` command creates a table. Here's the syntax:

```
CREATE [ [ GLOBAL | LOCAL ] TEMPORARY ] TABLE table_name
  ( { column_name { data_type | domain_name } [ column_size ]
    [ column_constraint... ] ... }
    [ DEFAULT default_value ], ...
    [ table_constraint ], ...
    [ ON COMMIT { DELETE | PRESERVE } ROWS ] )
```

Here's a simple example:

```
CREATE TABLE BOOKS
(
  TITLE VARCHAR (25) PRIMARY KEY,
  AUTHOR VARCHAR(25) NOT NULL DEFAULT 'Unknown',
  EDITION INTEGER,
  PRICE NUMBER(6,2)
)
```

The `PRIMARY KEY` and `NOT NULL` identifiers are column constraints. The `NOT NULL` constraint prevents any entry in a column from being set to `null`. Here, it's combined it with a default value. `PRIMARY KEY` identifies the column that's used as the primary key (or main unique identifier) for the table. If a table has a primary key column (it doesn't have to), there can be only one such column; no row has a `null` value in the primary key column, and no two rows have the same primary key.

A table constraint affects every row in the table. **UNIQUE** is a common example:

```
CREATE TABLE BOOKS
(
  TITLE VARCHAR (25),
  AUTHOR VARCHAR(25),
  EDITION INTEGER,
  PRICE NUMBER(6,2),
  UNIQUE
)
```

Used as a table constraint, **UNIQUE** indicates that each row in the table must have a unique combination of values. You can also specify particular columns that must form a unique combination:

```
UNIQUE(TITLE, AUTHOR, EDITION)
```

This mandates only unique title/author/edition combinations. Note that **UNIQUE** can also be used as a column constraint.

We can use **PRIMARY KEY** as a table constraint to specify more than one column as the primary key:

```
CREATE TABLE BOOKS
(
  TITLE VARCHAR (25) NOT NULL,
  AUTHOR VARCHAR(25) NOT NULL,
  EDITION INTEGER NOT NULL,
  PRICE NUMBER(6,2),
  PRIMARY KEY (TITLE, AUTHOR, EDITION)
)
```

Since entry-level SQL-92 requires that primary keys remain not **null**, we use **NOT NULL** column constraints on the primary key columns in this case.

D.3.2. ALTER TABLE

The **ALTER TABLE** command allows you to modify the structure of an existing table. Here's the syntax:

```
ALTER TABLE table_name
  { ADD [COLUMN] column_name definition }
```

```

{ ALTER [COLUMN] column_name definition
  { SET DEFAULT default_value } | { DROP DEFAULT } }
{ DROP [COLUMN] COLUMN_NAME [ RESTRICT | CASCADE ] }
{ ADD table_constraint_definition }
{ DROP constraint_name [ RESTRICT | CASCADE ] }

```

Note that the modifications you can make are somewhat limited. While you can add and remove columns (subject to the requirements of any constraints that may have been placed on the table), you can't reorder columns. To perform major changes, you generally need to create a new table and move the existing data from the old table to the new table.

Here's a statement that adds two columns to a table:

```

ALTER TABLE BOOKS
  ADD PUBLISHED_DATE DATE,
  ADD PUBLISHER CHAR (30) NOT NULL

```

Note that the ability to specify multiple operations in an `ALTER TABLE` command is not part of the SQL specification, although most databases support this functionality.

Here's how to change the type of a column:[\[3\]](#)

^[3] If you look back at the syntax for `ALTER TABLE`, you'll see that the official syntax for this kind of operation is `ALTER`, although most databases use `MODIFY` instead.

```

ALTER TABLE BOOKS
  MODIFY PUBLISHER VARCHAR (25)

```

When this statement runs, the database attempts to convert all existing data into the new format. If this is impossible, the modification fails. In the previous example, if any record has a publisher entry of more than 30 characters, the statement might fail (exact behavior depends on the implementation). If you are converting from a character field to, say, an integer field, the whole `ALTER TABLE` command might fail entirely. At the minimum, such a change requires that all entries contain a valid string representation of an integer.

To allow `null` values in the `PUBLISHER` column, use `MODIFY`:

```

ALTER TABLE BOOKS
  MODIFY PUBLISHER NULL

```

To remove the `PUBLISHER` column entirely, use `DROP`:

```
ALTER TABLE BOOKS  
DROP PUBLISHER
```

The `ALTER TABLE` command is not required for entry-level SQL-92 conformance. Due to its extreme usefulness, however, it is supported by most DBMS packages, although it often varies from the standard. More esoteric features, such as the `RENAME` command, aren't supported by most packages. In general, it is not safe to count on anything beyond the basic `ADD`, `DROP`, and `MODIFY (ALTER)` commands.

D.3.3. DROP

The `DROP` command allows you to permanently delete an object within the database. For example, to drop the `BOOKS` table, execute this statement:

```
DROP TABLE BOOKS
```

`DROP` also can delete other database objects, such as indexes, views, and domains:

```
DROP INDEX index_name  
DROP VIEW view_name  
DROP DOMAIN domain_name
```

Once something has been dropped, it is usually gone for good—certainly once the current transaction has been committed, but often before.

D.4. Data Manipulation Commands

Empty tables aren't very useful, and, even once they've been populated with data, we need some way of getting that data back out. The SQL data manipulation commands allow you to read data from a table and to create, update, and remove existing data.

D.4.1. SELECT

The `SELECT` statement is the most important statement in SQL and also the most complex. It allows you to retrieve data from a table or a set of tables. Here's the syntax:

```
SELECT [ DISTINCT ]
      { summary_function, ... }
      | { data_manipulation_expression, ... }
      | { column_name, ... }
FROM
      { { table_name [ AS correlation_name ] }
      | { subquery [ AS correlation_name ] }
      | joined_tables }
[ WHERE predicate ]
[ GROUP BY column_name, ... [ HAVING group_selection_predicate ] ]
[ { UNION | INTERSECT | EXCEPT } [ ALL ]
  [ CORRESPONDING [ BY (column_name, ...) ] ]
  select_statement | { TABLE table_name }
                    | table_value_constructor ]
[ ORDER BY {{output_column [ ASC | DESC ]}, ...}
           | {{positive_integer [ ASC | DESC ]}, ...}]
```

The simplest possible `SELECT`, which displays all columns of all rows of a single table, looks like this:

```
SELECT * FROM BOOKS
```

If this statement is executed in a command-line SQL interpreter, the output might look like this:

```
TITLE          | AUTHOR          | EDITION | PRICE
-----+-----+-----+-----
```

Me	Garrison Keillor		1	24.99
Bleak House	Charles Dickens		57	8.99
A Tale Of Two Cities	Charles Dickens		312	4.99

To sort the output by title, we can add an `ORDER BY` clause to the statement:

```
SELECT * FROM BOOKS ORDER BY TITLE
```

Now the output is:

TITLE	AUTHOR	EDITION	PRICE
A Tale Of Two Cities	Charles Dickens	312	4.99
Bleak House	Charles Dickens	57	8.99
Me	Garrison Keillor	1	24.99

To select just the `TITLE` and `AUTHOR` columns:

```
SELECT TITLE, AUTHOR FROM BOOKS
```

To select a subset of records, use the `WHERE` clause:

```
SELECT * FROM BOOKS WHERE PRICE < 10.0
```

This returns the Charles Dickens books, but not the Garrison Keillor book. You can have multiple criteria:

```
SELECT * FROM BOOKS WHERE PRICE < 10.0 OR EDITION = 1
```

This returns all three books. If we had specified a price less than 10 and an edition equal to 1, we wouldn't have received any records back. The various predicates you can use are listed in [Table D-2](#). Note that not all of the more esoteric ones (such as `LIKE`) are supported by all databases.

Table D-2. SQL predicates

Operator	Meaning
=	Equals
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to
!= or < >	Not equal to (some implementations may support only one)
LIKE	Wildcard match
IS NULL	Checks for a null value
IN	Checks to see if a value is contained within a set
BETWEEN	Checks to see if a date value is between two other date values

The final four predicates in the table can be used with the `NOT` modifier (e.g., `NOT LIKE`, `IS NOT NULL`, etc.).

D.4.1.1. String comparisons

The `=` operator can generally be used for exact string comparisons. The `LIKE` operator allows wildcard searches using two wildcard characters: `%` to match any number of characters, and `_` to match, at most, one character. Here's a query that selects all records that have a `LAST_NAME` that contains the letters "for":

```
SELECT LAST_NAME FROM CUSTOMERS WHERE LAST_NAME LIKE '%for%'
```

This matches last names like Buford, Crawford, and Trefor, but may not match Fordham, since most databases implement `LIKE` in a case-sensitive manner (Microsoft Access is a notable exception to this rule). Case-insensitive searches generally require a single-case version of the column or the use of a case-adjusting function (we'll discuss functions later in this appendix).

D.4.1.2. Subqueries and joins

The `IN` predicate allows you to check whether a value appears in another set of values. The simplest way to use this feature is in a SQL statement like this:

```
SELECT * FROM BOOKS WHERE PRICE IN (5.95, 4.95, 7.95)
```

This is simply another form for:

```
SELECT * FROM BOOKS WHERE PRICE = 5.95 OR PRICE = 4.95 OR PRICE = 7.95
```

But we can do something more interesting with `IN`. Let's assume that we have a table, called `PRICES`, that holds all the prices we want to search on. In this case, we can generate the set of values using another query, as follows:

```
SELECT * FROM BOOKS WHERE PRICE IN
  SELECT PRICE FROM PRICES
```

Note that we didn't put parentheses around the second `SELECT` statement; use parentheses only when you are specifying the entire list manually.

Subqueries like this are useful, but they still restrict the output columns to those of a single table.

Rather than using subqueries, two tables are normally connected via a join. A *join* allows a query to include references to multiple tables and to restrict the output based on the relations between those tables. The basic join is an *equi-join* (or inner join): data in two tables is linked based on a shared value. An order-tracking database might include a `CUSTOMERS` table and an `ORDERS` table. The `CUSTOMERS` table has a customer identifier (`CUSTOMER_ID`) as its primary key. The `ORDERS` table also has a `CUSTOMER_ID` column, although not as a primary key, since there may be more than one order per customer (see [Figure D-1](#) earlier in this appendix for a graphical representation of these tables).

Here's the SQL to combine the two tables based on the `CUSTOMER_ID` column:

```
SELECT * FROM CUSTOMERS, ORDERS WHERE ORDERS.CUSTOMER_ID =
  CUSTOMERS.CUSTOMER_ID
```

Since it's an equi-join, it works just as well the other way around:

```
SELECT * FROM CUSTOMERS, ORDERS WHERE CUSTOMERS.CUSTOMER_ID =
    ORDERS.CUSTOMER_ID
```

In SQL-92, a `JOIN` operator performs the same operation with a slightly different syntax:^[4]

[4] Note that some databases allow you to use `JOIN` without the `ON` clause. In this case, the join operates on columns that have the same name.

```
SELECT CUSTOMERS.CUSTOMER_ID, CUSTOMERS.NAME, ORDERS.ORDER_ID, ORDERS.TOTAL
    FROM CUSTOMERS INNER JOIN ORDERS ON CUSTOMERS.CUSTOMER_ID =
        ORDERS.CUSTOMER_ID
```

This example indicates the specific columns to return. Using the data shown in [Figure D-1](#), the output of this query might look like this:

CUSTOMER_ID	NAME	ORDER_ID	TOTAL
1	Bob Copier	4	72.19
1	Bob Copier	6	21.07

Note that some databases require you to use `INNER JOIN` instead of `JOIN`, while others allow just `JOIN` because inner joins are the default join.

To join on more than one table, use multiple `JOIN` statements. Here we add an `ITEMS` table that includes the `ORDER_ID` from the `ORDERS` table:

```
SELECT * FROM ITEMS JOIN ORDERS JOIN CUSTOMERS
```

This query joins the `CUSTOMERS` and `ORDERS` tables based on `CUSTOMER_ID`, and the `ORDERS` and `ITEMS` tables based on `ORDER_ID`. The join is performed from left to right, so this three-table join goes from the largest table to the smallest table.

As useful as the `JOIN` keyword is, it is not required for entry-level SQL-92 implementations, so here's a three-column join using the syntax we started with:

```
SELECT CUSTOMERS.CUSTOMER_ID, CUSTOMERS.NAME, ORDERS.ORDER_ID, ORDER.TOTAL,
    ITEMS.ITEM_NO, ITEMS.COST FROM CUSTOMERS, ORDERS, ITEMS
    WHERE ORDERS.CUSTOMER_ID = CUSTOMERS.CUSTOMER_ID
    AND ITEMS.ORDER_ID = ORDERS.ORDER_ID
```

Again, using the data shown in [Figure D-1](#), the output from this query might look as follows:

CUSTOMER_ID	NAME	ORDER_ID	TOTAL	ITEM_NO	COST
1	Bob Copier	4	72.19	1280	16.72
1	Bob Copier	4	72.19	4129	41.10
1	Bob Copier	4	72.19	3017	14.37

So far, all we've talked about is equi-joins, (or inner joins). There are also outer joins, which don't require a matching key in both tables. An *outer join* includes all the records from one table and any records from another table that match the primary key of the first table. If there are no corresponding records in the second table, those columns are simply left blank in the result. Outer joins are divided into left outer and right outer joins. In a left join, the primary key table is on the left, and in a right join, it is on the right. Here's the syntax for a `LEFT JOIN` (or `LEFT OUTER JOIN`) on `CUSTOMERS` and `ORDERS`:

```
SELECT CUSTOMERS.CUSTOMER_ID, CUSTOMERS.NAME, ORDERS.ORDER_ID, ORDERS.TOTAL
FROM CUSTOMERS LEFT JOIN ORDERS ON CUSTOMERS.CUSTOMER_ID =
ORDERS.CUSTOMER_ID
```

This includes all the customer records and order records for all the customers that have them. Here's some possible output using the data shown in [Figure D-1](#).

CUSTOMER_ID	NAME	ORDER_ID	TOTAL
1	Bob Copier	4	72.19
1	Bob Copier	6	21.07
2	John Stapler		

If we were to do a `RIGHT JOIN` on `CUSTOMERS` and `ORDERS`, the result would be the same as an inner join, since there are no records in `ORDERS` that don't have a match in `CUSTOMERS`.

D.4.1.3. Groups

The `GROUP BY` clause allows you to collapse multiple records into groups with a common field. For instance, to select all the records in the `BOOKS` table grouped by `AUTHOR`:

```
SELECT AUTHOR FROM BOOKS GROUP BY AUTHOR
```

This returns one row for each distinct author in the table. This query is not really that useful though, since we can do the same thing with the `DISTINCT` keyword (`SELECT DISTINCT`). However, we can use an aggregate function on each of the groups to do something more useful:

```
SELECT AUTHOR, COUNT(*) FROM BOOKS GROUP BY AUTHOR
```

This query returns two columns: the author and the number of books by that author in the table. Here's the output, based on the `BOOKS` table we used earlier:

AUTHOR	COUNT(*)
Charles Dickens	2
Garrison Keillor	1

We'll talk more about aggregate functions later in this appendix.

D.4.2. INSERT

The `INSERT` statement loads data into a table. Here's the syntax:

```
INSERT INTO table_name
  [ (column_name, ...) ]
  subquery | { VALUES (val1, val2,...) } | DEFAULT VALUES
```

To load static data, simply specify the table and the actual data:

```
INSERT INTO CUSTOMERS VALUES (3, 'Tom Fax', '617 555-1214')
```

This statement inserts the values `3`, `Tom Fax`, and `617 555-1214` into the first three fields of a new row in the `CUSTOMERS` table. If there are more than three fields in the `CUSTOMERS` table, this statement fails. If you don't want to insert into every column, you can specify the columns you want to insert into:

```
INSERT INTO CUSTOMERS (CUSTOMER_ID, NAME) VALUES (3, 'Tom Fax')
```

Note, however, that this kind of statement can fail if we don't provide a value for a field that is specified as `NOT NULL`.

To add more than one row at a time and to add data from other tables, we can specify a subquery rather than a set of specific values. To fill the `JUNKMAIL` table with values from the `CUSTOMERS` and `ADDRESSES` tables, run this query:

```
INSERT INTO JUNKMAIL (NAME, ADDR, CITY, STATE, ZIP)
  SELECT NAME, ADDR, CITY, STATE, ZIP FROM CUSTOMERS JOIN ADDRESSES
```

The database first performs a join on `CUSTOMERS` and `ADDRESSES`. It matches on the `CUSTOMER_ID` field and outputs the `NAME` field from `CUSTOMERS` and the other fields from `ADDRESSES`. The rows from the join are then inserted into the `JUNKMAIL` table, which can now be used to fill our mailboxes with catalogs.

D.4.3. UPDATE

The `UPDATE` statement modifies data in one or more existing rows. It consists of one or more `SET` statements and an optional `WHERE` clause. If the `WHERE` clause is not present, the operation is performed on every row in the table. Here's the syntax:

```
UPDATE table_name
  SET { column_name = { value | NULL | DEFAULT }, ... }
  [ { WHERE predicate }
    | { WHERE CURRENT OF cursor_name } ]
```

Here's an example that updates a few fields:

```
UPDATE ADDRESSES
  SET ADDR = '1282 Country Club Drive', STATE='CA'
  WHERE CUSTOMER_ID = 432
```

This statement sets the `ADDR` and `STATE` fields of the `ADDRESSES` table to particular values on all records where `CUSTOMER_ID` equals 432. Sometimes we do want to run an `UPDATE` on all records. Here's an example that makes sure all the `STATE` fields are in uppercase:

```
UPDATE ADDRESSES SET STATE = UPPER(STATE)
```

Note how we can use a field from the table itself in the `SET` statement.

The `WHERE CURRENT OF` clause allows you to update the row at the current cursor location in a multiple-table row. This is not something that JDBC programmers need to concern themselves with, although it can be of interest to an underlying JDBC 2.0 or JDBC 3.0 driver.

D.4.4. DELETE

`DELETE` is very simple: it removes rows from a table. Here's the syntax:

```
DELETE FROM table_name
  [ { WHERE predicate }
  | { WHERE CURRENT OF cursor_name } ]
```

To delete all the rows in `ORDERS`:

```
DELETE FROM ORDERS
```

To delete a specific record:

```
DELETE FROM ORDERS WHERE ORDER_ID = 32
```

Once a row has been deleted, there is no way to recover it.

D.5. Functions

SQL is not a procedural language, but it does provide some data-transformation capabilities. In addition to the string concatenation operator (`||`), the SQL-92 specification defines two sets of functions: aggregate and value.

D.5.1. Aggregate Functions

In the section on the `SELECT` statement, we saw an *aggregate function* used to count the number of records within a group. Mainly aggregate functions act on all the records of query, counting rows, averaging fields, and so forth. For example, here's how to count the number of rows returned by a `SELECT` statement:

```
SELECT COUNT(*) FROM CUSTOMERS
```

Instead of returning each row of the `CUSTOMERS` table, this query returns a single-column, single-row result that contains the number of records in `CUSTOMERS`.

The other aggregate functions are `AVG`, `SUM`, `MAX`, and `MIN`. Unlike `COUNT`, which works on either a single column or all columns, the other functions work on only a single column. `AVG` and `SUM` can be applied against numerical data types only (integers, reals, etc.) while `MAX` and `MIN` work with any data type. Here are some examples:

```
SELECT MIN(AGE) FROM GUESTS
SELECT MAX(NAME) FROM GUESTS
SELECT AVG(AGE), SUM(AGE) FROM GUESTS
```

D.5.2. Value Functions

Value functions work on particular column values and return modified data. Some of them also generate values from system information.

D.5.2.1. Date/time functions

There are three date and time functions that retrieve the current date, current time, and current timestamp, respectively, from the database:

```
CURRENT_DATE
CURRENT_TIME[ (precision) ]
CURRENT_TIMESTAMP[ (precision) ]
```

`CURRENT_TIME` and `CURRENT_TIMESTAMP` accept an optional precision level, which specifies the decimal fractions of a second to be included in the time portion of the value. The current time zone is used with all these functions.

Here's how you might use these functions in a query:

```
SELECT * FROM ORDERS WHERE ORDER_DATE = CURRENT_DATE
INSERT INTO VISITORS (VISIT_TS) VALUES (CURRENT_TIMESTAMP)
```

Some databases have platform-specific commands that duplicate this functionality (these commands often predate SQL -92). Oracle's `SYSDATE` is one example. Note that these functions aren't required for entry-level SQL-92.

D.5.2.2. String manipulation functions

The concatenation operator, `||`, has been around since before the SQL-92 standard. It allows you to concatenate multiple column values and string literals. Say we have a table that contains `FIRST_NAME` and `LAST_NAME` fields, and we want to display them in a "last, first" form. Here's a SQL statement that returns a single column that does just that:

```
SELECT LAST_NAME || ', ' || FIRST_NAME FROM CUSTOMERS
```

In addition, the SQL-92 standard defines a number of other functions that can be used in SQL statements. `UPPER` and `LOWER` convert a column into uppercase or lowercase, respectively:

```
SELECT UPPER(LAST_NAME) FROM CUSTOMERS
SELECT LOWER(FIRST_NAME) FROM CUSTOMERS
```

These functions can also be used in `WHERE` predicates, for example, to produce a case-insensitive search:

```
SELECT * FROM CUSTOMERS WHERE UPPER(FIRST_NAME) LIKE 'WILL%'
```

Most databases support `UPPER` and `LOWER`, but they are required only for full SQL-92 conformance, not entry-level conformance.

The `TRIM` function removes characters from one or both ends of a string:

```
TRIM ([ [ LEADING | TRAILING | BOTH ] [ character ] FROM ] string )
```

Calling `trim` on a string trims leading `CURRENT_TIME` and `CURRENT_TIMESTAMP` and trailing whitespace. Here's how to trim just leading blanks:

```
SELECT TRIM(LEADING ' ' FROM FIRST_NAME) FROM CUSTOMERS
```

And here's how to trim all "-" characters from both sides of a string:

```
SELECT TRIM(BOTH '-' FROM FIRST_NAME) FROM CUSTOMERS
```

Like `UPPER` and `LOWER`, `trim` is required only for full SQL-92 conformance, although it is supported by most database implementations.

The `SUBSTRING` command extracts a given number of characters from a larger string. It is defined as:

```
SUBSTRING (source_string FROM start_pos FOR number_of_characters)
```

For example, to get each customer's initials, we might use the following query:

```
SELECT SUBSTRING (FIRST_NAME FROM 1 FOR 1),  
       SUBSTRING(LAST_NAME FROM 1 FOR 1)
```

`SUBSTRING` is required only for intermediate-level SQL-92 conformance.

D.6. Return Codes

The SQL-92 standard defines a set of `SQLSTATE` return codes. `SQLSTATE` is defined as a five-character string, with the leftmost two characters defining the error class, and the remaining three characters defining the error subclass. Some database vendors may extend these return codes; classes beginning with the numbers 5 through 9 and letters I through Z are reserved for such implementation-specific extensions. The `SQLSTATE` code for a particular JDBC action can be retrieved via the `getSQLState()` method of `SQLException`. [Table D-3](#) lists the `SQLSTATE` return codes defined in SQL-92.

Table D-3. SQL-92 SQLSTATE return codes

Class	Class definition	Subclass	Subclass definition
00	Successful completion	000	None
01	Warning	000	None
		001	Cursor operation conflict
		002	Disconnect error
		003	Null value eliminated in set function
		004	String data, right truncation
		005	Insufficient item descriptor areas
		006	Privilege not revoked
		007	Privilege not granted
		008	Implicit zero-bit padding
		009	Search expression too long for information schema
00A	Query expression too long for information schema		

Class	Class definition	Subclass	Subclass definition
02	No data	000	None
07	Dynamic SQL error	000	None
		001	Using clause doesn't match dynamic parameters
		002	Using clause doesn't match target specifications
		003	Cursor specification can't be executed
		004	Using clause required for dynamic parameters
		005	Prepared statement not a cursor specification
		006	Restricted data type attribute violation
		007	Using clause required for result fields
		008	Invalid descriptor count
08	Connection exception	000	None
		001	SQL client unable to establish SQL connection
		002	Connection name in use
		003	Connection doesn't exist
		004	SQL server rejected establishment of SQL connection
		006	Connection failure
		007	Transaction resolution unknown
		00A	Feature not supported
21	Cardinality violation	000	None
		001	Multiple server transactions

Class Class definition

22 Data exception

Subclass Subclass definition

000	None
001	String data, right truncation
002	Null value, no indicator
003	Numeric value out of range
005	Error in assignment
007	Invalid date-time format
008	Date-time field overflow
009	Invalid time zone displacement value
011	Substring error
012	Division by zero
015	Internal field overflow
018	Invalid character value for cast
019	Invalid escape character
021	Character not in repertoire
022	Indicator overflow
023	Invalid parameter value
024	Unterminated C string
025	Invalid escape sequence
026	String data, length mismatch
027	Trim error
000	None

23 Integrity constraint violation

Class	Class definition	Subclass	Subclass definition
24	Invalid cursor state	000	None
25	Invalid transaction state	000	None
26	Invalid SQL statement name	000	None
27	Triggered data change violation	000	None
28	Invalid authorization specification	000	None
2A	Syntax error or access rule violation in direct SQL statement	000	None
2B	Dependent privilege descriptors still exist	000	None
2C	Invalid character set name	000	None
2D	Invalid transaction termination	000	None
2E	Invalid connection name	000	None
33	Invalid SQL descriptor name	000	None
34	Invalid cursor name	000	None
35	Invalid condition number	000	None
37	Syntax error or access rule violation in dynamic SQL statement	000	None
3C	Ambiguous cursor name	000	None
3F	Invalid schema name	000	None
40	Transaction rollback	000	None
		001	Serialization failure
		002	Integrity constraint violation
		003	Statement completion unknown
42	Syntax error or access rule violation	000	None

Class Class definition

44 With check option violation

Subclass Subclass definition

000 None



Appendix E. JMS Message Selector Syntax

JMS message selectors are used by JMS clients to filter the messages that a JMS server delivers to a given `MessageConsumer`. A message selector is provided (optionally) when a `MessageConsumer` is created, using either the `QueueSession.createReceiver()` or the `TopicSession.createSubscriber()` method.

A message selector is a string that specifies a predicate to be applied to each message that the provider wants to deliver to a `MessageConsumer`. If the predicate evaluates to true, then the message is delivered; if false, then the message is not delivered. In point-to-point messaging, when messages are filtered out by a message selector, the message will remain on the queue until the client eventually reads it or the message times out and the server removes it from the queue. In publish-subscribe messaging, messages that are filtered are never delivered to the subscriber.

In this chapter, we provide the full syntax of JMS message selectors. For more details on using JMS in general, see the tutorial in [Chapter 11](#).



E.1. Structure of a Selector

A message selector is made up of one or more Boolean expressions, joined together by logical operators and grouped using parentheses. For example:

```
(<expression1> OR <expression2> AND <expression3>) OR
```

```
  (<expression4> AND NOT <expression5>) . . .
```

A message selector is evaluated left to right in precedence order. So in this example, `expression2` will be evaluated followed by `expression3` (since `AND` has higher precedence than `OR`), and if they evaluate to false, then `expression1` will be evaluated and so on.

Expressions are made up of literal values, identifiers (referring to either message headers or properties), conditional operators, and arithmetic operators.

E.2. Identifiers

An identifier refers to either a standard JMS header field name or a custom message property name. Any JMS header field name can be used as an identifier, except for `JMSDestination`, `JMSExpiration`, `JMSRedelivered`, and `JMSReplyTo`. `JMSDestination` and `JMSReplyTo` are `Destination` values, and message selector operators support only numeric, Boolean, or string values. The time at which message selectors are applied to messages is not specified in the JMS specification, so using the value of `JMSExpiration` wouldn't provide a consistent, well-defined result. Using `JMSRedelivered` in a selector could result in unexpected results. If, for example, a selector checks for `JMSRedelivered` being true, the first delivery attempt by a provider will fail the selector since the redelivered flag should be false, but the provider could then immediately redeliver the message and pass the selector making the redelivered part of the predicate ineffective.

Identifier names are case-sensitive and follow the same general rules as Java identifiers. They must start with a valid Java identifier start character as determined by the `java.lang.Character.isJavaIdentifierStart()` method e.g., a letter, currency symbol, or connecting punctuation character like "_" and must contain valid Java identifier characters as determined by the `Character.isJavaIdentifierPart()` method. You cannot use the following reserved words for identifiers: `NULL`, `TRUE`, `FALSE`, `NOT`, `AND`, `OR`, `BETWEEN`, `LIKE`, `IN`, `IS`, or `ESCAPE`.

The type of an identifier is the type of the header field or property being referenced as its value was set in the message. It's important to remember that the evaluation of an identifier in a message selector does not apply type conversion functions according to the context that it's used in. If you attempt to refer to a numeric property value in an expression with a string comparison operator, for example, then the expression will always evaluate to false. If the named header field or property is not present in a message, then the identifier evaluates to a `NULL` value.



E.3. Literals

String literals are indicated with single quotes. For example:

```
JMSType = 'updateAck'
```

If you need to use a single quote in a string literal, you use two single quotes. For example:

```
JMSCorrelationID = 'Joe''s message'
```

Numeric literals are either integer values or floating-point values. Integer values follow the rules for Java integer literals. They are all numerals, with no decimal point, and can have a value in the same range as a Java long value. For example:

```
42, 149, -273
```

Floating-point literals follow the syntax of Java floating-point literals. They are all numerals with a decimal point:

```
3.14, 98.6, -273.0
```

or are in scientific notation:

```
31.4e-1, 6.022e23, 2.998e8
```

Literals can also be the Boolean values `true` or `FALSE`.

E.4. Operators

Operators are used to compose identifiers and literals into larger expressions. Operators can be logical operators, arithmetic operators, or comparison operators.

E.4.1. Logical Operators

The logical operators are **NOT**, **AND**, and **OR**. These are in precedence order. These have the usual Boolean logic semantics. If a logical operator is applied to header fields or properties whose value is **NULL**, then the following rules apply:

- **AND**ing a **NULL** value with a **FALSE** value evaluates to **FALSE**; **AND**ing a **NULL** with a **true** or **NULL** value evaluates to a **NULL** (or unknown) value.
- **OR**ing a **NULL** value with a **true** value evaluates to **true**; **OR**ing a **NULL** with a **FALSE** or a **NULL** value evaluates to a **NULL** (or unknown) value.
- Applying **NOT** to a **NULL** value evaluates to a **NULL** (or unknown) value.

E.4.2. Arithmetic Operators

The arithmetic operators, in precedence order, are **+** and **-** (unary); ***** and **/**; and **+** and **-** (binary). These have the usual arithmetic semantics. Any arithmetic operator that is applied to one or more **NULL** values evaluates to a **NULL** value.

E.4.3. Comparison Operators

The comparison operators can be loosely grouped into equality comparisons and range comparisons. The basic equality comparison operators, in precedence order, are **=**, **>**, **>=**, **<**, **<=**, and **<>**. These binary operators have to be applied to two values of the same type, else the expression will always evaluate to **FALSE**. If either value is **NULL**, then the result of the comparison is **NULL**. There are also the equality operators **IS NULL** and **IS NOT NULL** to compare a value to **NULL**. An equality operator can also be used to check for the presence of a header or property. For example:

```
timezone IS NOT NULL AND country = 'United Kingdom'
```

There are also set and range comparison operators. The **BETWEEN** operator can be used to check the

range of numeric values. For example:

```
userid BETWEEN 00000000 AND 09999999  
currRate NOT BETWEEN 0.0 AND 0.9999
```

The **IN** operator can be used to perform membership operations on string values:

```
JMSType IN ('msgAck', 'queryAck', 'updateAck')  
JMSType NOT IN ('msgBroadcast', 'synchMessage')
```

There is also a string comparison operator, **LIKE**, that allows for wildcard matching on string values. A pattern is used for the right side of the **LIKE** operator. The pattern consists of a valid string literal where the underscore character, **_**, matches against any single character, while the **%** character matches any sequence of zero or more characters. For example:

```
JMSType like '%Ack'  
label not like 'Step _'
```

The **_** and **%** characters can be used in these string comparison operators if they are escaped by ****. For example:

```
slogan LIKE '99 44/100\% pure'
```

E.5. Expressions

Expressions are simply literals and identifiers assembled together using the various operators described before. A message selector must eventually evaluate to a Boolean value, so its combination of expressions must be structured to result in a Boolean value. Expressions can be grouped in a message selector using parentheses in order to control the order of evaluation.

Arithmetic expressions are composed of arithmetic operators used with numeric literals and identifier values. Arithmetic expressions can be combined to form compound arithmetic expressions. For example:

```
(userid + 10000) / (callerid - 10000)
```

Conditional expressions are made up of comparison and logical operators used with numeric, string, or Boolean literals or identifiers and evaluate to `TRUE`, `FALSE`, or `NULL` (i.e., unknown). Conditional expressions can also be combined to form compound conditional expressions. For example:

```
(JMSType like '%Ack') AND ((userid + 10000) / (callerid - 10000) < 1.0)
```

Notice that, although the last example includes an arithmetic expression "fragment," `[(userid + 10000) / (callerid - 10000)]`, it becomes part of a conditional expression when used with a comparison operator with the numeric literal `1.0`.

Every complete message selector must be a conditional expression. A message selector that evaluates to `true` matches the message; one that evaluates to `FALSE` or `NULL` does not match the message.

[← PREV](#)

Appendix F. FRMI Tools

In this appendix, the tools documented are the default RMI-related tools shipped with Sun's implementation of the JDK. Other vendors may ship their own versions of these tools with different behavior and options than those described here, or they may provide their own toolset altogether. Consult your vendor's documentation for details.

[← PREV](#)



rmic: The Java RMI Compiler

JDK 1.1 and later

Synopsis

```
rmic [ options ] fully-qualified-classnames
```

Description

The *rmic* compiler generates the stub and skeleton classes for remote objects that you've written. The *rmic* compiler is capable of generating stubs and skeletons for both the Java Remote Method Protocol (JRMP) or the CORBA Internet Interoperable Object Protocol (IIOP). Once you've compiled your remote objects using a standard Java compiler, such as *javac*, you need to run *rmic*, specifying the class names of your RMI implementation classes using their full packages.

For example, suppose you define an interface named `utils.remote.TimeServer` that extends `java.rmi.Remote` and write an implementation of this interface called `utils.remote.TimeServerImpl`. After compiling both with a Java compiler, you can run *rmic*, specifying `utils.remote.TimeServerImpl` as the class name argument.

The native RMI remote method protocol, JRMP, was updated in Java 2 Version 1.2. The *rmic* compiler in JDK 1.2 and later supports the generation of stubs that are compatible with JRMP Version 1.1, 1.2, or both. A key difference between the two is that 1.1 required the generation of both stub and skeleton classes for a remote interface, while 1.2 requires only the stub class (the server-side skeleton operations are implemented dynamically by the RMI runtime). The `-vcompat`, `-v1.1`, and `-v1.2` options, described in the next section, control what flavor of stubs and skeletons are generated when you run *rmic*.

Running *rmic* with 1.1 compatibility generates a skeleton class for each remote object type, named `xxx_Skel`, where the `xxx` is the name of the remote interface. The skeleton is responsible for receiving client requests on a server object and dispatching these requests to the actual remote object. A stub class, named `xxx_Stub`, is also generated; this is used for client references to the remote object. When a client gets a reference to a remote object, it receives an instance of the stub class, which forwards any method requests to the server object over the network. In our example, the stub and skeleton classes would be called `utils.remote.TimeServer_Stub` and `utils.remote.TimeServer_Skel`, respectively. When using just 1.2 compatibility (e.g., the `-v1.2` option), only the stub classes are generated.

Both the stub class and the skeleton class implement the same remote interface as your remote object implementation, so they can be type cast to the remote interface.

IIOP-compatible stubs and skeletons can also be generated with *rmic*, using the `-iiop` option. In CORBA parlance, the skeletons are called *ties*. The concepts are the same as JRMP-compatible stubs and skeletons, except that the generated classes speak the IIOP protocol, and the class names are

`_rmtinterface_stub` and `_rmiimpl_tie` (`rmtinterface` is the class name of the RMI remote interface, and `rmiimpl` is the class name of the RMI object implementation). It's also important to note that, when using IIOP-compatible stubs and skeletons, a client has to use the `javax.rmi.PortableRemoteObject.narrow()` method to safely cast remote object references to their expected remote interface type, rather than just using the Java cast operator.

Options

`-always`

`-alwaysgenerate`

Applicable only when used with the `-idl` or `-iiop` option. They tell `rmic` to regenerate the stub and tie classes, regardless of the timestamps on the classes.

`-bootclasspath pathlist`

Allows you to override the path used by `rmic` to find its bootstrap classes. This is useful if a custom RMI runtime implementation is used. The directories in the list are separated by colons in Unix environments (Linux, Mac OS X, etc.) and by semicolons in Windows environments.

`-classpath pathlist`

Provides the classpath `rmic` uses to find any required classes (remote object implementation, remote interface, and classes referenced by these). This option overrides the environment `CLASSPATH` and the default classpath. The directories in the list are separated by colons in Unix environments (including Mac OS X) and by semicolons in Windows environments.

`-d path`

The destination directory where the compiler should write the class files that are generated. If a `-d` option is given, the package of the generated classes places the stubs and skeletons in their proper subdirectories in the given destination. If the directories don't exist, `rmic` creates them for you. If no `-d` option is given, the stubs and skeletons are still generated to have the same package as the remote implementation, but the class files are placed in the current directory (e.g., the compiler assumes `-d`).

`-depend`

Forces the compiler to attempt to recompile interdependent classes whose class files are out of date with each other. Without this option, the compiler attempts to recompile only class files explicitly referenced in the command-line options. Note that this option was removed in Java 5.0, since it was seen as redundant with the same capabilities in the `javac` compiler.

-extdirs path

Overrides the location of runtime extensions. Normally, extensions are found in a standard location in the JRE, such as *lib/ext*. This option allows you to specify a custom directory to find these extension libraries.

-factory

Applicable only if used with the **-idl** option. It tells *rmic* to generate declarations for initializers for value types in the generated IDL, using the IDL **factory** keyword. This option is relevant only if your RMI-IIOP implementation class uses CORBA Objects By Value. This option was removed in Java 5.0.

-g

Includes debugging information in the generated stub and skeleton classes; for use with Java debuggers.

-idl

Uses the *rmic* compiler to generate CORBA IDL interfaces for your RMI remote objects, generating according to the CORBA Java Language to IDL Mapping specification. Note that the implementation class specified needs to be implemented as an RMI-IIOP remote object, as described in [Chapter 13](#). Remote objects that support only JRMP can't use the IIOP protocol and can't have IDL interfaces generated for them. The **-factory**, **-idlModule**, **-noValueMethods**, and **-idlFile** options are allowed only when used with the **-idl** option. The **-d** option can control where the IDL files are generated.

-idlFile javapackage[.class] idlfile

Valid only when used with the **-idl** option. It overrides the default mapping of IDL entities in the Java class to an IDL file. For example, to have the IDL from any entities in the `jent.corba` package mapped into a `jententities.idl` file:

```
> rmic -idl -idlFile jent.corba jententities.idl
jent.corba.MyIIOPObjImpl
```

-idlModule javapackage[.class] idlmodule

Valid only when used with the **-idl** option. It overrides the default package mapping of IDL entity references, allowing you to specify another IDL module to map them to in the generated IDL. For example, if you want the generated IDL to map entities referenced in the Java `jent.corba` package to a custom `corba::custom::mod` module:

```
> rmic -idl -idlModule jent.corba corba::custom::mod
jent.corba.MyIIOPObjImpl
```

`-iiop`

Instructs the compiler to generate IIOP-compatible stubs and ties instead of JRMP stubs and skeletons. The implementation class specified needs to be implemented as an RMI-IIOP remote object, as described in [Chapter 13](#). The `-always`, `-alwaysgenerate`, `-nolocalstubs`, `-noValueMethods`, and `-poa` options are allowed only when used with the `-iiop` option. The `-d` option can control where the stub and tie classes are generated.

`-J[javaoption]`

Passes the option immediately following the `-J` to the Java interpreter. There should be no spaces between the `-J` and the option to be passed to the interpreter.

`-keep`

`-keepgenerated`

Keeps the Java source files for the stub and skeleton classes generated by the compiler. The Java files will be written to the same directory as the class files, with or without a `-d` option specified. Without the `-keep` or `-keepgenerated` option present, the Java source files will be removed after the stub and skeleton classes are generated.

`-nolocalstubs`

Used with the `-iiop` option, to instruct the compiler to avoid generating IIOP stubs that are optimized for local clients. Without this option, the generated stub classes attempt to detect whether the client is running in the same process as the servant and optimizes the IIOP operations. With this option, these checks aren't generated, and the stub assumes that the servant is always remote.

`-noValueMethods`

Used in conjunction with the `-idl` option, to instruct the compiler to avoid generating `valuetype` methods and initializers in the generated IDL.

`-nowarn`

Instructs the `rmic` compiler to eliminate warning messages from its output. Only errors encountered during compilation are reported.

`-poa`

Used with the `-iiop` option, `-poa` tells `rmic` to generate POA-compatible servants. The generated server class inherits from the `org.omg.PortableServer.Servant` interface rather than

the default `org.omg.CORBA_2_3.portable.ObjectImpl` interface.

`-vcompat`

Creates stub and skeleton classes that are compatible with both JDK 1.1 and 1.2 (and later) versions of RMI. This option is enabled by default in JDK 1.4 and earlier and doesn't need to be specified in those environments. In Java 5.0, the default was switched to `-v1.2`.

`-verbose`

Prints verbose messages as compilation takes place, including which class is being compiled and class files that are loaded during compilation.

`-v1.1`

Creates stub and skeleton classes that are compatible with the JDK 1.1 version of RMI. These classes may not run in a 1.2 or higher Java runtime environment, unless its RMI runtime supports 1.1 compatibility.

`-v1.2`

Creates stub and skeleton classes that are compatible with the JDK 1.2 and higher RMI runtime. These classes will not run in a 1.1 Java runtime environment. This option is the default as of Java 5.0 and doesn't need to be specified in those environments. Prior to Java 5.0, the default was `-vcompat`.

Environment

`CLASSPATH`

An ordered list of directories, zip files, and/or jar files the `rmic` compiler uses to look for classes. This list is separated by colons in Unix environments (Linux, Mac OS X, etc.) and by semicolons in Windows environments. The list is searched, in order, for a given class when it is encountered during compilation. The compiler automatically appends the system classpath to the `CLASSPATH`, if it is specified. If the `CLASSPATH` isn't set in the environment, the compiler uses the current directory and the system `CLASSPATH` as its classpath. The `CLASSPATH` environment variable is overridden by the `-classpath` option to `rmic`.



rmid: The RMI Activation Daemon

JDK 1.2 and later

Synopsis

```
rmid [ options ]
```

Description

The *rmid* command starts an RMI activation daemon on the local host. The activation daemon services all requests to register activatable objects and is responsible for activating objects due to client requests to invoke methods on them.

If no port option is given, the activation daemon runs on a default port of 1098. Internally, the activation daemon creates a `java.rmi.activation.Activator` and its own RMI naming registry (listening to port 1098). The daemon binds a `java.rmi.activation.ActivationSystem` object to the name `java.rmi.activation.ActivationSystem` in its internal registry.

The *rmid* daemon provided by Sun requires you to specify a security policy file on the command line, using the `-J` option:

```
> rmid -J-Djava.security.policy=myrmid.policy ...
```

This policy file determines what operations are allowed when launching JVMs for activation groups. For example, a policy file for *rmid* might include permissions such as the following:

```
grant {
  permission com.sun.rmi.rmid.ExecPermission
    "/opt/rmid/bin/*";
  permission com.sun.rmi.rmid.ExecOptionPermission
    "-Djava.security.policy=/usr/rmi-app/act-group.policy";
};
```

This allows *rmid* to execute any commands in the `/opt/rmid/bin` directory and to start any activation groups using the command-line option to invoke the security policy file `act-group.policy`. When registering activation groups, you can specify what commands to use to start the group's JVM, what command-line arguments to include, and what Java properties to override using the `-D` option for the JVM. Permission to run the specified commands has to be given to *rmid* in its policy file using `com.sun.rmi.rmid.ExecPermission` entries. The ability to add command-line options to the activation startup command is granted using `com.sun.rmi.rmid.ExecOptionPermission` entries.

You can also control the application of security policies using the `sun.rmi.activation.execPolicy` JVM option on the `rmid` command:

```
> rmid -J-Dsun.rmi.activation.execPolicy=<default|policyClass|none> ...
```

If this option isn't used or is used with the value `default`, the `java.security.policy` file determines the ability of the activator to execute commands and add command-line options to the JVM startup commands. If a class name is provided as the value of this option, then instead of using the security policy file, call `checkExecCommand()` on this class before activating groups:

```
public void checkExecCommand(ActivationGroupDesc desc, String[] command)
    throws SecurityException {
    if (command or options are not allowed) {
        throw new SecurityException(...);
    }
}
```

If the value of the `execPolicy` property is `none`, `rmid` won't do any checking of security policies before running activation commands.

Options

`-C < cmdlineOption >`

Uses the given option as a command-line option to the Java VM for each activation group started by the daemon. This can pass default properties to the VMs, for example, or set their memory limits to some default value. The Java interpreter option should immediately follow the `-C` option, with no spaces.

`-J [javaoption]`

Anything immediately following the `-J` is passed as an option to the underlying Java interpreter. There should be no spaces between `-J` and the option to be passed to the interpreter.

`-log path`

Uses the given directory for any logging or temporary files needed by the activation daemon. If this option isn't specified, the daemon writes its logfiles to a new directory named `log` in the current directory.

`-port portnum`

Uses the specified port for the internal registry started by the activation daemon. If this option

isn't given, the daemon's internal naming registry runs by default on port 1098 of the local host. If you want the activation daemon's registry to listen to the default RMI registry port, start the daemon with this command (Unix version):

```
$ rmid -port 1099 &
```

`-stop`

Stops any activation daemon currently running on the port specified with the `-port` option or the default port of 1098 if none is specified.

Environment

CLASSPATH

An ordered list of directories, zip files, and/or jar files the *rmi*d daemon uses to look for classes. This list is separated by colons in Unix environments and semicolons in Windows environments. The list is searched, in order, for a given class when it is encountered during execution. The daemon automatically appends the system `CLASSPATH` to the classpath, if it is specified. If the `CLASSPATH` isn't set in the environment, the daemon uses the current directory and the system `CLASSPATH` as its classpath.

Synopsis

```
rmiregistry [ port ]
```

Description

The *rmiregistry* command starts a remote object naming registry on the current host. The RMI registry binds remote objects to names, so that remote clients can request object references by name. It uses the object references to invoke methods.

Internally, the *rmiregistry* command uses the `java.rmi.registry.LocateRegistry` class to instantiate a registry object. If no port is provided, the default port for the registry is 1099. Typically, the registry is run in the background on a server and remains running for the lifetime of the objects it contains. If the registry crashes and the registry is running in a separate Java VM from the actual remote objects, the remote objects are still available over RMI, and any remote references to these objects that existed before the crash are still valid. However, all the name bindings the objects had in the registry are lost and need to be re-created after a new registry is started.

Options

```
-J[javaoption]
```

Anything immediately following the `-J` is passed as an option to the underlying Java interpreter. There should be no spaces between the `-J` and the option to be passed to the interpreter.

Environment

CLASSPATH

An ordered list of directories, zip files, and/or jar files the *rmiregistry* command uses to look for classes. This list is separated by colons in Unix environments and semicolons in Windows environments. It is searched, in order, for a given class when it is encountered during execution. The registry automatically appends the system `CLASSPATH` to the classpath, if it is specified. If the `CLASSPATH` isn't set in the environment, the daemon uses the current directory and the system `CLASSPATH` as its classpath.



[← PREVIOUS](#)

serialver: The RMI Serial Version Utility

JDK 1.1 and later

Synopsis

```
serialver [ options ] fully-qualified-classnames
```

Description

The *serialver* utility generates a serial version ID you can use to mark a given class definition to track its versions as it evolves. The utility returns a static `int` member declaration you can paste into your Java class definition. For example, this command:

```
% serialver AccountImpl
```

generates output something like:

```
AccountImpl: static final long serialVersionUID = 37849129093280989384L;
```

If versioning of your remote object classes becomes a problem for clients, this utility can tag a class with a version ID that can be checked to see if the proper version is being exported by your server for a given client or if its local version is out of date. Serial version IDs are used by Java object serialization to uniquely identify class definitions and should be placed on any `Serializable` classes that you implement.

Options

```
-classpath pathlist
```

Provides the classpath *serialver* uses to find any required classes or resources. This option overrides the environment `CLASSPATH` and the default classpath. The directories in the list are separated by colons in Unix environments (including Mac OS X) and by semicolons in Windows environments.

```
-J[ javaoption]
```

Anything immediately following the `-J` is passed as an option to the underlying Java interpreter. There should be no spaces between the `-J` and the option to be passed to the interpreter.

`-show`

Uses the graphical version of the tool. The GUI interface allows you to type a fully qualified class name and click a button to see the serial version ID generated.

Environment

`CLASSPATH`

An ordered list of directories, zip files, and/or jar files the `serialver` utility uses to look for classes. This list is separated by colons in Unix environments and semicolons in Windows environments. The list is searched, in order, for a given class when it is encountered during execution. The utility automatically appends the system `CLASSPATH` to the classpath, if it's specified. If the `CLASSPATH` isn't set in the environment, the utility uses the current directory and the system `CLASSPATH` as its classpath.

Appendix G. IDL Reference

This chapter serves as a quick reference for CORBA's language-independent Interface Definition Language (IDL). It also gives a summary of the Java mapping of IDL (i.e., how IDL interface definitions are mapped by an IDL-to-Java compiler into equivalent Java interfaces.) For a complete, definitive reference on IDL and the Java mapping of IDL, consult the IDL specification issued by the OMG.

Since this reference combines an IDL reference with an overview of the Java mapping of IDL, we'll be using a combination of IDL and Java examples throughout. In order to make it clear which language is being used in each example, we've stolen a convention from the CORBA standards documents: a comment preceding the example code indicates which language is in use.

IDL, as the name implies, is a language for defining object interfaces. IDL is language-neutral, so interfaces defined in IDL can be converted to objects implemented in any language with an IDL mapping. One way to think of the role that IDL plays is to imagine that you already have a set of interacting objects defined, and IDL is a way to export a subset of those interfaces so that they can be accessed by remote entities. Any member variables or operations that you define in the IDL interfaces will be visible and accessible by remote entities, providing they can obtain a reference to instances of these interfaces (refer to [Chapter 14](#) for more details on the use of CORBA objects). If you proceed this way, the definitions of the variables and operations on the IDL interfaces would have to match those on the actual implementations you are exporting, according to the rules of the IDL mapping to the language they are written in.

Of course, this is not the typical way that you use IDL in practice^[1] You usually want to do an abstract design of a distributed application and its set of objects first, define the IDL interfaces for the objects that need to be used remotely, then compile these IDL interfaces into language-specific interfaces. Then implementations of the interfaces can be written in whatever programming language you are using. You are free to add additional operations and member variables to your language-specific implementations, but these features won't be accessible remotely unless you add them to the corresponding IDL interface.

[1] One notable exception is the task of wrapping legacy code with a CORBA frontend, but even here you would normally use middleware objects to interface directly to the legacy code.

The five high-level entities that you can define in an IDL specification are:

- Modules, which act as namespaces
- Interfaces to objects (with their operations and data attributes)
- Data types
- Constants
- Exceptions

These high-level entities are listed here in roughly hierarchical order. Modules contain other modules, interfaces, data types, constants, and exceptions. Interfaces contain data types, constants, and exceptions that are specific to that interface, along with the operations and attributes for the interface. We'll look at the IDL syntax used to define each of these, and in each case we'll see how the IDL is mapped into equivalent Java code. Before we do that, let's go over some of the basics of IDL: the keywords, identifiers, comments, and various types of literals.



G.1. IDL Keywords

[Table G-1](#) lists the reserved keywords in the IDL language.

Table G-1. IDL reserved keywords

abstract	float	sequence
any	in	short
attribute	inout	string
boolean	interface	struct
case	local	supports
char	long	switch
const	module	true
context	native	truncatable
custom	Object	typedef
default	octet	union
double	oneway	unsigned
enum	out	ValueBase
exception	private	valuetype
factory	public	void
false	raises	wchar
fixed	readonly	wstring

These keywords are case-sensitive. They can't be used as identifiers in any IDL construct.

G.2. Identifiers

Identifiers can name various IDL constructs, such as modules, interfaces, and constants. In IDL, an identifier has to follow these rules:

- It can be any length of alphanumeric characters from the ISO Latin-1^[2] character set (e.g., a-z, A-Z, 0-9, plus various characters with accents, graves, tildes, etc.), and the underscore character (`_`).

^[2] This refers to the ISO standard 8859, "Information Processing 8-bit single-byte coded graphic character sets Part 1: Latin alphabet No. 1," ISO 8859-1:1987. You can find a listing of the character set in the HTML 4.0 standard at <http://www.w3.org/TR/html4>.

- All characters in an identifier are significant.
- The first character must be an alphabetic character.
- Identifiers are case-insensitive, in the sense that two identifiers that differ only by case are considered a name collision and will cause an IDL compiler error. This rule stems from the fact that IDL needs to be mappable into many implementation languages, some of which are case-insensitive.
- Identifiers must be spelled and capitalized consistently throughout an IDL file.
- All IDL identifiers share the same namespace, so interfaces, modules, user-defined types, and so on within the same scope all must have unique identifiers. An interface named `List` and a module named `List` within the same scope will cause a name collision and an IDL compiler error. See "[Naming Scopes](#)" later in this appendix for more details.

G.2.1. Mapping Identifiers to Java

An IDL-to-Java compiler attempts to map all IDL identifiers unchanged into equivalent Java identifiers.

An exception is the case in which a mapped identifier conflicts with an identifier created automatically by the IDL compiler. IDL interfaces, for example, when they are mapped to Java, have two additional Java interfaces created for them, named using the name of the original interface with `Helper` and `Holder` appended (see [Chapter 14](#) for details on the purpose of these generated interfaces). So an interface named `List` will be mapped into a Java interface named `List`, but will also cause the creation of a Java interface named `ListHelper` and another named `ListHolder`. If there is another identifier in the IDL file that you've named `ListHelper` or `ListHolder`, its mapped Java identifier will have an underscore prepended to it (e.g., `_ListHelper`, `_ListHolder`), to avoid a conflict with the generated interface names. In general, identifiers automatically generated by the IDL compiler have precedence over other identifiers declared explicitly in the IDL file.

The other exception to the general rule of directly mapping IDL identifiers to Java identifiers is with a mapping that conflicts with Java keywords. In these cases, the mapped Java identifier has an underscore prepended to it. If, for example, you declared a constant named `package` (not a reserved keyword in IDL), it is mapped into a Java variable named `_package`.



[← PREV](#)

G.3. Comments

Comments in IDL follow the format of C, C++, and Java comments. A block comment starts with the character sequence `/*` and ends with the character sequence `*/`. A line comment begins with the character sequence `//`, and ends at the end of the line on which it begins.

G.3.1. Mapping Comments to Java

There are no rules for mapping IDL comments to Java. Many IDL-to-Java compilers will simply drop comments from IDL files during the conversion to Java, since in many cases, the comments refer to the IDL code and may not be totally relevant in the generated Java code.

[← PREV](#)

G.4. Basic Data Types

IDL supports the basic data types shown in [Table G-2](#). The table also shows the Java type that each is mapped to according to the standard IDL Java mapping. Note that there isn't a standard mapping defined for the `long double` IDL type, because Java doesn't have a basic data type that supports the precision required. Another important thing to note is that Java doesn't support unsigned types, such as `unsigned short`. So you'll see in the table that IDL `short` and `unsigned short` are both mapped to the Java `short` data type. You should be aware of this when writing implementations of IDL-generated Java interfaces, since it is up to you to either ensure that their values remain positive or deal with the fact that their values may in fact be set to negative values.

Table G-2. IDL basic data types, sizes, and Java mappings

IDL type specifier	Required size	Java data type
<code>short</code>	16 bits	<code>short</code>
<code>long</code>	32 bits	<code>int</code>
<code>long long</code>	64 bits	<code>long</code>
<code>unsigned short</code>	16 bits	<code>short</code>
<code>unsigned long</code>	32 bits	<code>int</code>
<code>unsigned long long</code>	64 bits	<code>long</code>
<code>char</code>	8 bits	<code>char</code>
<code>wchar</code>	Implementation-dependent	<code>char</code>
<code>string</code>	Unlimited	<code>java.lang.String</code>
<code>string<size></code>	<i>size chars</i>	<code>java.lang.String</code>
<code>wstring</code>	Unlimited	<code>java.lang.String</code>
<code>wstring<size></code>	<i>size wchar</i> s	<code>java.lang.String</code>
<code>boolean</code>	Implementation-dependent	<code>boolean</code>
<code>octet</code>	8 bits	<code>byte</code>
<code>Any</code>	Implementation-dependent	<code>org.omg.CORBA.Any</code>
<code>float</code>	IEEE single precision	<code>float</code>
<code>double</code>	IEEE double precision	<code>double</code>
<code>long double</code>	IEEE double extended	Not supported
<code>fixed</code>	31 decimal digits	<code>java.math.BigDecimal</code>

G.4.1. Strings and Characters

There are two character types included in IDL: `char` and `wchar`. A `char` represents an 8-bit character from a single-byte character set, such as ASCII. A `wchar` represents a wide character from any character set, including multibyte character sets like Kanji. The size of `wchar` is implementation-specific.

We've included the IDL `string` and `wstring` data types in this table as well, although technically they should be considered constructed data types (arrays of a basic data type, characters). Since they're so frequently used, it's useful to have them together with all of the IDL basic data types.

A `string` is the equivalent of an array of `chars`, and a `wstring` is an array of `wchars`. In both cases, there are two ways to specify a string type: with or without a size specification or in angle brackets following the type name. If you provide a size specification in your IDL declaration (e.g., `string<10> name`), the language mapping is responsible for enforcing the bounds limits of the string. If you don't provide a size specification, the `string` is allowed to grow to any size, limited only by the implementation language.

If support for a multibyte character set is important for your application, then it's best to declare all of your character and string data as `wchars` and `wstrings`. This way you'll be sure to get multibyte support in languages that support it.

G.4.1.1. Mapping strings and characters to Java

In the IDL Java mapping, both `char` and `wchar` are mapped to the Java `char` type, and both `string` and `wstring` types are mapped to the `java.lang.String` class. In Java, the `char` type represents a 2-byte Unicode character and can therefore support multibyte character sets by default.

When marshaling and unmarshaling data items during remote method calls, the ORB is responsible for performing bounds checks on the data members being set. If a value exceeds the bounds limits declared for the string member in the IDL specification of the interface, then an `org.omg.CORBA.BAD_PARAM` exception is thrown. If a character in a `string` falls outside the range supported by IDL, then an `org.omg.CORBA.DATA_CONVERSION` exception is thrown.

G.5. Constants and Literals

Literals are explicit values inserted into IDL code. Sometimes literals are used to specify a default value for interface attributes or to declare the value for a constant. Literals can be Boolean (`true` or `false`), numeric (integer, floating point, or fixed point), or character-based (a single character or a string).

Literals are most often used in IDL to initialize the value of constants. Constants are named variables that are restricted from being modified after being initialized. In IDL, a constant is declared using the syntax:

```
// IDL
const <type spec> <identifier> = <value>;
```

where `<type spec>` is any valid basic data type or declared interface type, `<identifier>` is any valid IDL identifier, and `<value>` is any IDL expression that evaluates to a literal value. The initialization expression can be a simple literal or it can be a complex expression combining multiple literals using logical or mathematical operators. You could declare a few useful numeric constants as follows, for example:

```
// IDL
const float half = 1 / 2;
const float quarter = 1 / 4;
```

Most of the operators present in C/C++, such as addition (+), subtraction (-), multiplication (*), and the logical and bitwise operators (`|`, `&`, `^`, `||`, `&&`, etc.) are supported by IDL.

G.5.1. Mapping Constants to Java

If an IDL constant is declared within an interface definition, then the constant is mapped to a `public static final` member on the corresponding Java interface.

If the IDL constant is declared outside of an interface definition, then a Java interface is created to hold the constant value as a `public static final` value. The generated interface has the same name as the IDL identifier given to the constant, and the static class member has the name `value`. So this IDL constant declaration:

```
// IDL
const float PI = 3.1416;
interface GeometricOperators {
    const long maxDims = 3;
    ...
}
```

causes the generation of a Java `GeometricOperators` interface:

```
public interface GeometricOperators extends foobarOperations,
    org.omg.CORBA.Object, org.omg.CORBA.portable.IDLEntity
{
    public static final int maxVals = (int)((int)-3);
    ...
}
```

as well as a separate interface to hold the IDL constant `PI` that is declared outside of any interface:

```
// Java
public final class PI {
    public static final float value = (float) (3.1416D);
}
```

In your Java code, you can reference the value of `PI` using `PI.value`, and the value of the interface-specific constant using `GeometricOperators.maxDims`.

G.5.2. Boolean Literals

There are two Boolean literals (naturally) in IDL. They are specified using the keywords `TRUE` and `false`. Their IDL type is `boolean`. In Java, they are mapped into the `boolean` values `TRUE` and `false`.

G.5.3. Numeric Literals

G.5.3.1. Integer literals

An integer value in IDL can be declared in decimal, octal, or hexadecimal notation. Any sequence of digits that doesn't start with a zero is considered a decimal integer value. If the sequence is all digits but starts with a zero, it's assumed to be an octal value. If the literal starts with `0X` or `0x`, it's taken to be a hexadecimal value.

G.5.3.2. Floating-point literals

A floating-point literal is a decimal integer, optionally followed by a decimal point and a fractional component and/or by the letter `e` or `E` followed by an exponent expressed as a decimal integer. Either the fractional component (with the decimal point) or the exponent (with the `e` or `E`) must be present for the literal to be interpreted as a floating-point value and not an integer. Similarly, either the initial integer component or the decimal point must be present. So, for example, these are valid floating-point literals:

```
2.34
0.31416e1
3E19
.0003413
```

G.5.3.3. Fixed-point literals

A fixed-point literal consists of a decimal integer, optionally followed by a decimal point and fractional component (expressed as a decimal value), followed by the letter `d` or `D`. Either the integer component or the fractional component must be present. The decimal point is optional. The trailing `d` or `D` must be present in order for the literal to be interpreted as a fixed-point value. The following are all valid fixed-point literals:

```
1.50d
.025d
1.333D
12d
```

G.5.3.4. Mapping numeric literals to Java

Numeric literals are mapped by taking into account the context in which they are used. Typically a literal initializes a constant, and the declared type of the constant has to be checked to determine whether the literal is valid for the type and how it should be mapped to a Java literal. For example, these two similar IDL constant declarations:

```
// IDL
const short largeVal = 2000;
const float largeFloatVal = 2000;
```

are mapped by Sun's `idlj` compiler to these Java declarations:

```
// Java
public static final short largeVal = (short) (2000);
public static final float largeFloatVal = (float) (2000);
```

Sun's *idlj* compiler does some type and range checking on the IDL literal before converting it to its Java form and inserting it into the cast operation shown previously. For example, if we change the first constant declaration to:

```
// IDL
const short largeVal = 2e5;
```

the *idlj* compiler emits an error saying that the value is too large for a short variable.

G.5.4. Character Literals

A character literal is a character specification enclosed in single quotes (e.g., 'a'). Character literals can be specified only with elements of the ISO 8859-1 character set. Some characters need to be specified with a sequence of more than one character. These include characters that are nonprintable or the single- and double-quote characters that are used to delimit string and character literals. These characters are specified with escape sequences, which start with a backslash character. [Table G-3](#) lists the escape sequences supported by IDL and the nonprintable characters they represent.

Table G-3. IDL escape sequences

Escape sequence	Meaning
\a	Alert
\\	Backslash
\b	Backspace
\r	Carriage return
\"	Double quote
\f	Formfeed
\x## (e.g., \x4e)	Hexadecimal number
\n	Newline
\### (e.g., \012)	Octal number
\?	Question mark

Escape sequence	Meaning
\'	Single quote
\t	Tab
\v	Vertical tab

Character literals, including the escape sequences listed in [Table G-3](#), are converted unchanged into Java literals.

G.5.5. String Literals

A string literal is a sequence of characters delimited by double-quote (") characters. If two string literals are adjacent to each other in an IDL file, then they are concatenated. So in this example:

```
// IDL
const string acctHolder = "Jim " "Farley";
```

the generated Java code is:

```
// Java
public static final String acctHolder = "Jim Farley";
```

If you want to use the double-quote character in a string literal, you have to use its escape sequence (see [Table G-3](#)).

G.6. Naming Scopes

Each IDL file that you create defines a namespace or naming scope for identifiers that you declare within that file. This namespace is further subdivided into nested scopes whenever you declare a new module, interface, structure, union, or exception in your IDL file. You can think of the naming scope within an IDL file as a sort of naming directory. By default, you start at the root of the directory, and each time you open a declaration of one of these items, you start a new subdirectory of the naming directory, named after the identifier you use for the item.

You can specify scopes using the `::` delimiter, which is analogous to the `.` delimiters in Java class names or the `/` or `\` delimiters in file directories. The root scope for the IDL file is represented as `::` by itself, and nested scopes are specified by adding their names, such as `::utils::math::MatrixOps`. The names in a scope name can refer to any identifiers that might exist in each scope. In this example case, `utils` and `math` might refer to modules (the `math` module is declared within the `utils` module), and `MatrixOps` might refer to an interface declared within the `math` module. The intermediate elements of a scoped name must refer to one of the IDL elements that define their own scopes (listed previously), but the final element of a scoped name can refer to any item with its own identifier, including constants, data members on interfaces, and so forth.

Within any particular scope in the naming scope of an IDL file (including the root scope), all identifier within that scope must be unique. Separate nested scopes off of one parent scope can have identical identifiers declared within them and can share identifiers with their parent scope as well, but two identifiers at the same level within a scope can't be the same. So you can legally declare the following in an IDL file:

```
// IDL
module utils {
    interface math {
        const float PI = 3.1416;
    };
    interface baking {
        const string PI = "apple";
    };
};
```

The two definitions of `PI` (`::utils::math::PI` and `::utils::baking::PI`) don't conflict, since each has a distinct absolute scoped name within the IDL file. You can't, however, declare a constant named `math` within the `utils` module, since its fully scoped name is also `::utils::math`, and conflicts with the name of the `math` interface.

Scoped names that begin with `::` are absolute names and are relative to the root file scope of the IDL file. Names that don't start with `::` are relative to the local scope in which they appear. So in our previous example, we could add two new constants to our `math` interface that use scoped names to

reference our versions of `PI`:

```
// IDL
module utils {
    interface math {
        const float PI = 3.1416;
        const float PIsquared = PI * PI;
        const string PIOfTheDay = ::utils::baking::PI;
    };
    interface baking {
        const string PI = "apple";
    };
};
```

The reference to `PI` in the definition of the `PIsquared` constant is relative to the `::utils::math` scope, so it refers to the `float` constant. The reference to `PI` in the `PIOfTheDay` definition is absolute and references the string definition of `PI` in the `baking` interface.

[← PREV](#)

G.7. User-Defined Data Types

In addition to the basic data types already described, IDL supports user-defined data types, which are aggregations of these basic types. These complex data types include arrays, sequences, enumerations, and constructed data types that you define yourself using structs and unions. We'll go over each of these in detail in this section.

Complex data types are used in IDL by first giving them a type name, then using the type name wherever you would use a basic data type name or an interface type name (e.g., declaring attributes method arguments, etc.). A name can be assigned to a complex data type in different ways:

- With structures, unions, and enumerations, the name is included in the declaration of the data type.
- A `typedef` can be used to assign a name to a specific type (basic or complex).

Before we go on to see how complex data types are declared in IDL, let's take a look at how `typedefs` are used to assign type names to these complex data types.

G.7.1. Typedefs

A `typedef` is used to associate a name with another data type. The syntax of an IDL `typedef` is:

```
typedef <type_spec> <identifier>
```

The `<type_spec>` can be any basic IDL data type, a user-defined data structure (structure, union, or enumeration), an IDL interface type, or a sequence. The `<identifier>` can be a simple IDL identifier, or it can include dimension specifications for an array. So the following are all valid `typedef` statements:

```
// IDL

typedef short myShort;
typedef long longArray[2][2];
typedef PrintServer pserver;
```

After declaring these `typedefs` in your IDL file, you can use `myShort`, `longArray`, and `pserver` as type names when declaring method arguments, return values, or interface attributes.

G.7.1.1. Mapping typedefs to Java

If an IDL `typedef` refers to a basic IDL type, then the Java equivalent to that type is used wherever the `typedef` identifier is used. So our `myShort typedef` shown in the previous section is replaced by the Java type `short` wherever it's used.

Any `typedefs` that refer to user-defined types are replaced by the mapped Java class or interface for the target IDL type. If the type used in an IDL `typedef` is itself a `typedef`, then its target type is found and so on until a final user-defined type or basic IDL type is found. So in this example:

```
// IDL
struct LinkedList {
    any item;
    any next;
};
typedef LinkedList DefList;
typedef DefList MyList;
```

wherever either `DefList` or `MyList` appears in the IDL file, it is mapped to the Java class generated for the `LinkedList` type, since both refer (directly or indirectly) to that type.

A helper class is also generated for each `typedef`, providing the usual helper methods in terms of the underlying mapped type. So a `myShortHelper` class is generated from our example, but its methods are declared in terms of the underlying `short` type (e.g., its `read()` method returns a `short`). If the underlying type is a sequence or array, then a holder class is also generated for the `typedef`.

G.7.2. Arrays

Arrays can be declared only within the context of a `typedef`. Once you've assigned the array type to a type name using the `typedef`, you can use the new type name to declare array members on interfaces. Since IDL doesn't provide a way to initialize array values, you can't declare array constants in IDL, since constants have to be initialized in their declaration.

To declare an array, simply add dimensions in brackets to a variable identifier. For example, to define a two-dimensional array of short values:

```
// IDL
typedef short short2x2Array[2][2];
```

IDL requires that you specify explicitly each dimension of the array, in order to support mappings to languages that have a similar requirement.

G.7.2.1. Mapping arrays to Java

Arrays are mapped into Java as arrays (naturally). So if we used our `short2x2Array` type defined earlier in an IDL interface:

```
// IDL
interface MatrixOps {
    attribute short2x2Array identity2D;
    ...
}
```

the corresponding Java code would look like so:

```
// Java
public interface MatrixOps {
    short[][] identity2D( );
    void identity2D(short[][] arg);
    ...
}
```

We'll look more at how interface attributes are mapped to Java later, but you can infer from this that the `short` IDL array is mapped into a `short` array in Java. The attribute is mapped into a get and set method for that attribute, and since Java doesn't allow array type specifiers to include dimensions, our declaration that the `identity2D` attribute be a 2 x 2 array has been lost in the mapping. It's up to you to provide an implementation of this interface that enforces the intended dimensions of the array within the Java interface.

In addition to the mapping of the array type to equivalent type specifiers, each array `typedef` in IDL causes the generation of corresponding helper and holder classes in Java. The type name specified in the IDL `typedef` is used as the prefix for the `xxxHelper` and `xxxHolder` class names. So our `short2x2Array` type has a `short2x2ArrayHelper` class and a `short2x2ArrayHolder` class generated for it. The helper class provides the static methods that read and write the array type over CORBA I/O streams, when the array type is used as a method argument or return type. These methods enforce the array dimensions that you dictate in your IDL `typedef`; if the array is not of the correct type when being marshaled, then the `write()` method throws an `org.omg.CORBA.MARSHAL` exception. The holder class is used whenever you use your array type as an `inout` or `out` method argument. For more details on the purposes on helper and holder classes, see [Chapter 14](#).

G.7.3. Sequences

An IDL sequence is a one-dimensional array. To declare a sequence, you need to declare the type of the elements in the sequence and, optionally, the maximum size of the sequence:

```
// IDL
typedef sequence<long, 2> longVector;
typedef sequence<short> unboundedShortVector;
typedef sequence<sequence<float, 2> > coordVector;
```

Like arrays, sequences have to be declared within a `typedef`, and then the new type name can be used for typing attributes, method arguments, and return values. Notice in our last example that the elements in a sequence can themselves be a sequence. Also notice that, if you don't provide a bound for a sequence of sequences, you need to put a space between the two `>` brackets, so that they aren't parsed as a `>>` operator.

G.7.3.1. Mapping sequences to Java

Sequences are mapped to Java almost identically to how arrays are. A sequence of a given IDL type becomes a Java array of the equivalent Java type. Sequences of sequences become two-dimensional arrays, and so forth. Holder and helper classes are generated for each sequence `typedef` as well, using the type name specified in the `typedef`. The `write()` method on the helper class enforces any size bounds that you specify on the sequence, throwing an `org.omg.CORBA.MARSHAL` exception if they don't match.

G.7.4. Structs

A fixed data structure is declared using the `struct` construct in IDL. A `struct` is declared using the following syntax:

```
// IDL
struct <type name> {
    <data member>;
    <data member>;
    ...
};
```

The `type name` is any valid identifier in IDL. Each `data member` is specified using a type spec and an identifier used to reference the member (similar to attributes on an interface, described later). You can use basic data types, arrays, sequences, and any other `typedefs` as types for members of a `struct`. You can declare a recursive structure (a structure that includes members of its own type) by using a `sequence` declaration:

```
// IDL
struct LispStringList {
    string car;
    sequence<LispStringList> cdr;
```

```
};
```

G.7.4.1. Mapping structs to Java

An IDL `struct` is mapped into a `public final` Java class with the same name as the `struct`. Each member of the `struct` is mapped to a public instance member on the Java class. The Java class includes a default constructor that leaves the member variables uninitialized and a constructor that accepts a value for each member. Our previous example `struct` is mapped to the following Java class:

```
// Java
public final class LispStringList {
    // Instance variables
    public String car;
    public LispStringList[] cdr;
    // Constructors
    public LispStringList( ) { }
    public LispStringList(String _car, LispStringList[] _cdr) {
        car = _car;
        cdr = _cdr;
    }
}
```

Each `struct` also has a Java holder class generated for it, which is used to marshal the data type when it's used as an `inout` or `out` method argument or as a method return value.

G.7.5. Enumerations

An enumeration in IDL declares an ordered list of identifiers, whose values are assigned in ascending order according to their order in the enumeration. An enumerator is given a type name so that the elements of the enumeration can be referenced. The syntax for declaring an IDL enumeration is:

```
// IDL
enum <type name> { <element name>, <element name>, ... };
```

The elements in the enumeration are guaranteed to be assigned actual values so that the comparator operators in the implementation language will recognize the order of the elements as specified in the `enum` declaration. So the first element is less than the second, the second is less than the third, and so on. An example `enum` declaration follows:

```
// IDL
enum ErrorCode { BadValue, DimensionError, Overflow, Underflow };
```

G.7.5.1. Mapping enumerations to Java

Each enumerated type that you declare in IDL is mapped to a public final Java class of the same name as the enumeration. The class holds a single `private int` instance member. A single private constructor is generated for the class, which takes an `int` argument used to initialize the `value` member.

For each element of the enumeration, two components are added to the Java class: a `static final int` data member and a `static` instance of the generated Java class. The static data member generated for each element is given a value that enforces the order of the elements in the enumeration, and the static class instance generated for each element is initialized with this same value. The static class instance is given the same name as the element in the enumeration, and the static data member is given the element's name prepended with an underscore. These two representations for each element of the enumeration let you reference the element value using either a corresponding `int` value or the generated Java class type. If the enumerated type is used as a method argument or return value in an IDL interface, your Java implementation will have to use the object versions of the elements.

Our example enumeration generates a Java class like the following:

```
// Java
public class ErrorCode implements org.omg.CORBA.portable.IDLEntity
{
    private          int _value;
    private static int _size = 4;
    private static ErrorCode[] _array = new ErrorCode [_size];

    public static final int _BadValue = 0;
    public static final ErrorCode BadValue = new ErrorCode(_BadValue);
    public static final int _DimensionError = 1;
    public static final ErrorCode DimensionError =
        new ErrorCode(_DimensionError);
    public static final int _Overflow = 2;
    public static final ErrorCode Overflow = new ErrorCode(_Overflow);
    public static final int _Underflow = 3;
    public static final ErrorCode Underflow = new ErrorCode(_Underflow);

    public int value ( )
    {
        return _value;
    }

    public static ErrorCode from_int (int value)
    {
```

```

    if (value >= 0 && value < _size)
        return _array[value];
    else
        throw new org.omg.CORBA.BAD_PARAM ( );
}

protected ErrorCode (int value)
{
    _value = value;
    _array[_value] = this;
}
}

```

So we can refer to the elements in the enumeration in our Java code using any of the following forms:

```

// Java
int error1 = ErrorCode._BadValue;
ErrorCode error2 = ErrorCode.Overflow;
int error2Val = error2.value( );

```

Each enumerated type also has a holder class generated for it, which is used whenever the enumerated type is used in IDL as an *out* or *inout* method argument. Although not strictly required by the IDL-Java mapping as defined by the OMG, an enumerated type might also have a helper class generated for it.

G.7.6. Unions

IDL unions are similar in nature to discriminated unions in C and C++. A single tag field, or *discriminator*, is used to determine the data element held by the union. Depending on the value of the discriminator field, a particular instance of the union type may hold a different data member. The union is declared using a *switch* statement to declare the various possible formats, or branches, of the union structure:

```

// IDL
union <type name> switch (<discriminator type>) {
    case <tag value>:
        [<data element>;]
    case <tag value>:
        [<data element>;]
    ...
    [default:]
        <data element>;
}

```

```
};
```

The discriminator for the union is declared using only the type for the discriminator (no identifier is given to the discriminator, since there is only a single discriminator per union type). The type for the discriminator must be an integer, character, Boolean, or enumerated type (no `strings`, `structs`, unions, arrays, or sequences allowed).

Each branch in the switch defines a data element that represents the value of the union if its discriminator is a given value. Each data member identifier in a union switch has to be unique. Multiple cases can be mapped to the same data element by listing them sequentially within the `switch`. A single optional `default` case can be given for any values not given their own cases. So for example, the following union:

```
// IDL
typedef sequence<long, 2> Coord2d;
typedef sequence<long, 3> Coord3d;
union MultiCoord switch (short) {
    case 1:
        long pos;
    case 2:
        Coord2d val2d;
    case 3:
    default:
        Coord3d val3d;
};
```

declares a type named `MultiCoord`, which represents a one-, two-, or three-dimensional coordinate, depending on the value of its discriminator value. The default is for the coordinate to be 3-D, so the case for a discriminator value of 3 is the same as the default case. Since a union can have only a single data member per case, you have to use `typedef` types for the coordinate values. Depending on the discriminator value, the union contains either a simple integer position, a `Coord2D` type that is declared as a `sequence` of two integer values, or a `Coord3D` type that is a `sequence` of three integer values.

If the discriminator value is given a value not listed in a case, the union consists of the data member in the default case, if present. If there is no default case, the union has only its discriminator value and no data members.

G.7.6.1. Mapping unions to Java

Each IDL union is mapped to a `public final` Java class of the same name as the union identifier. The class contains a single, default constructor. The class has some kind of data member for maintaining the value of the union discriminator (the details of which aren't dictated by the IDL-to-Java mapping) and a `discriminator()` method for accessing it as a `short` value. The standard also doesn't specify how data members for the union are implemented in the Java class. Each branch that you specify in the IDL union is mapped to an accessor method and modifier method for that branch; these methods

are named after the identifier given to the data member in the branch. If you use one of the modifier methods to set that branch of the union type, then the discriminator is automatically set to the corresponding value. If you attempt to access the value from a branch and the union is not set to that branch, then a `CORBA::BAD_OPERATION` exception (`org.omg.CORBA.BAD_OPERATION` in Java) is thrown. The return value types and method arguments for the `discriminator()` method and the case accessor/modifier methods are determined based on the standard type conversion rules for mapping IDL to Java.

Our `MultiCoord` union example is mapped to the following Java class by Sun's `idltojava` compiler:

```
// Java
public final class MultiCoord implements org.omg.CORBA.portable.IDLEntity
{
    private int _pos;
    private int[] _val2d;
    private int[] _val3d;
    private short _discriminator;
    private boolean _uninitialized = true;

    public MultiCoord ( )
    {
    }

    public short discriminator ( )
    {
        if (_uninitialized)
            throw new org.omg.CORBA.BAD_OPERATION ( );
        return _discriminator;
    }

    public int pos ( )
    {
        if (_uninitialized)
            throw new org.omg.CORBA.BAD_OPERATION ( );
        verifypos (_discriminator);
        return _pos;
    }

    public void pos (int value)
    {
        _discriminator = 1;
        _pos = value;
        _uninitialized = false;
    }

    private void verifypos (short discriminator)
    {
        if (discriminator != 1)
            throw new org.omg.CORBA.BAD_OPERATION ( );
    }

    public int[] val2d ( )
```

```
{
    if (_uninitialized)
        throw new org.omg.CORBA.BAD_OPERATION ( );
    verifyval2d (_discriminator);
    return _val2d;
}

public void val2d (int[] value)
{
    _discriminator = 2;
    _val2d = value;
    _uninitialized = false;
}

private void verifyval2d (short discriminator)
{
    if (discriminator != 2)
        throw new org.omg.CORBA.BAD_OPERATION ( );
}

public int[] val3d ( )
{
    if (_uninitialized)
        throw new org.omg.CORBA.BAD_OPERATION ( );
    verifyval3d (_discriminator);
    return _val3d;
}

public void val3d (int[] value)
{
    _discriminator = 3;
    _val3d = value;
    _uninitialized = false;
}

public void val3d (short discriminator, int[] value)
{
    verifyval3d (discriminator);
    _discriminator = discriminator;
    _val3d = value;
    _uninitialized = false;
}

private void verifyval3d (short discriminator)
{
    if (discriminator == 1 || discriminator == 2)
        throw new org.omg.CORBA.BAD_OPERATION ( );
}
}
```

Notice that Sun's *idlj* compiler implements the data branches in the union using a set of data members, one for each branch of the union. Other IDL-to-Java compilers can choose to implement the union differently.

In this example, the default case and the third case share the same branch, so no accessor or modifier method is generated for the default case. If we have a default case that is separate from all other explicit cases in the union (i.e., has its own branch), then an accessor and modifier method is generated for its branch as well. If two explicit cases are mapped to the same branch in the `switch`, then the Java modifier method generated for that branch sets the discriminator value to the value of the first case included for that branch. In these cases, another modifier method, which takes a second argument that is the value for the discriminator, is also generated. As an example, if we want to use a `Coord2D` for both 1D and 2D coordinates, we can modify our IDL union to have both case 1 and case 2 use the same branch:

```
typedef sequence<long, 2> Coord2d;
typedef sequence<long, 3> Coord3d;
union MultiCoord switch (short) {
  case 1:
  case 2:
    Coord2d val2d;
  case 3:
    Coord3d val3d;
  default:
    Coord3d valDef;
};
```

In this situation, the generated Java has an additional method included for the `val2d` branch:

```
public void val2d (short discriminator, int[] value) { ... }
```

This allows you to set the union to that branch and also specify which discriminator is intended. This can be useful in some cases, such as our modified `MultiCoord` example, where the value of the discriminator determines the usage for the object.

If no explicit `default` case is given in the union and if the listed cases don't completely cover the possible values for the discriminator, then the generated Java class includes a single method named `default()` that takes no arguments and returns a `void`. This serves as the modifier for the default case, setting the union discriminator to some unused value.

The union also has a holder and helper class generated for it in the mapped Java.

G.8. Exceptions

You can define exceptions in IDL, which are used to signal errors or other unusual circumstances that may occur during a remote method call. Exceptions are declared with a unique name and an optional set of data attributes:

```
// IDL
exception identifier { <data member>; <data member>; ...};
```

Each data member on the exception type is simply a type specification followed by a unique identifier for the data member. The data is used to provide the caller with additional information about what went wrong during the remote method call.

Using our geometric examples from earlier, we might define an exception that is thrown when a `MultiCoord` with unexpected dimensions is passed into a method:

```
// IDL
exception BadDimension {
    short expected;
    short passed;
};
```

The server object raising one of these exceptions can set these data values, and the client making the request can read these values to interpret what went wrong.

Exceptions can be declared within any module or interface scope in your IDL file.

G.8.1. Standard Exceptions

In addition to user-defined exceptions, a set of standard exceptions is defined within the CORBA module. These standard exceptions can be raised by any method, even though they aren't listed explicitly in the method definition. These exceptions can be referenced in IDL using the `CORBA::` scope (e.g., `CORBA::BAD_PARAM`). The standard CORBA exceptions are listed in [Table G-4](#). Every standard CORBA exception includes two data members: an `unsigned long` minor error code, which can further specify the type of error that occurred, and a `completion_status` enum that can be one of `COMPLETED_YES`, `COMPLETED_NO`, and `COMPLETED_MAYBE`. These status values indicate that before the exception was raised, the method was either completed, was never initiated, or is in an unknown state, respectively.

Table G-4. Standard CORBA exceptions

Exception name	Meaning
BAD_CONTEXT	Failure while accessing the context object.
BAD_INV_ORDER	Some methods were called out of their expected order.
BAD_OPERATION	An invalid method was called.
BAD_PARAM	An invalid argument was passed into a method.
BAD_TYPECODE	A bad type code was used.
CODESET_INCOMPATIBLE	The underlying protocol code sets between the client and the server are incompatible.
COMM_FAILURE	A communication failure occurred.
DATA_CONVERSION	Error while converting data.
FREE_MEM	Failed to free some memory.
IMP_LIMIT	Some implementation limit was exceeded.
INITIALIZE	The ORB initialization failed.
INTERNAL	An internal ORB error occurred.
INTF_REPOS	Error attempting to access interface repository.
INV_FLAG	An invalid flag was given.
INV_IDENT	Invalid identifier syntax was encountered.
INV_OBJREF	An invalid object reference was encountered.
INV_POLICY	An operation couldn't be performed because of incompatible security policy settings.
INVALID_TRANSACTION	An invalid transaction was used.
MARSHAL	An error occurred while marshaling method arguments or results.
NO_IMPLEMENT	The implementation for the method is not available.
NO_MEMORY	Failed to allocate dynamic memory needed to execute the request.
NO_PERMISSION	Not allowed to execute the method.
NO_RESOURCES	There were insufficient resources for the request.
NO_RESPONSE	No response received for request.
OBJ_ADAPTER	The object adapter encountered an error.
OBJECT_NOT_EXIST	The referenced object doesn't exist on the server.
PERSIST_STORE	An error occurred while accessing persistent storage.
TRANSACTION_REQUIRED	An operation requiring a transaction was called without one.
TRANSACTION_ROLLEDBACK	A transactional operation didn't complete because its transaction was rolled back.

Exception name	Meaning
TRANSIENT	A transient error occurred, but the method can be tried again.
UNKNOWN	An error occurred that the ORB couldn't interpret.

G.8.2. Mapping Exceptions to Java

User-defined exceptions are mapped to `public final` Java classes that extend `org.omg.CORBA.UserException`, which is derived directly from `java.lang.Exception`. Otherwise, the exception is mapped to Java in the same way as `structs`, as described previously. Each data member is mapped to a `public` data member of the corresponding type, and a set of constructors is defined for the exception class.



G.9. Module Declarations

A module is a name-scoping construct in IDL. It's similar to packages in Java or LISP or to namespaces in C++. A module is declared with the `module` keyword, followed by an identifier for the module and then the body of the module, enclosed in braces:

```
// IDL
module identifier { ... };
```

Modules can contain IDL interface definitions, constants, or user-defined types, such as `typedefs`, `structs`, unions, and enumerations.

G.9.1. Mapping Modules to Java

Modules in IDL are mapped to packages in Java, and nested modules are mapped to subpackages, with the innermost module being mapped to the rightmost subpackage. If we have the following interfaces and modules defined in IDL:

```
// IDL
module util{
    interface MatrixOps { ... };

    module dbase {
        interface Query { ... };
    };
};
```

the generated Java code includes an interface named `MatrixOps`, starting with this `package` statement:

```
// Java
package util;
```

and another interface named `Query`, with this `package` statement:

```
// Java  
package util.dbase;
```



G.10. Interface Declarations

An IDL interface is just a collection of data attributes and methods that define the semantics of the interface. Declaring an interface is another way to create a new data type in IDL, but unlike `structs` and unions, an interface can have both data members and methods that can be called on objects of its type. An interface is also a name-scoping construct, similar to a module. You can declare an IDL interface and simply include a set of constants that you want associated with that interface name. In this case, you have to specify the interface scope in order to refer to the constants from within other scopes.

An interface consists of the following elements:

```
// IDL
interface identifier [: inheritance spec] {
    interface body
};
```

The interface identifier can be any valid IDL identifier. The body of the interface can contain any of the following constructs:

- A user-defined type (`struct`, `union`, `typedef`, `enum`)
- A constant declaration
- An interface-specific exception declaration
- Data attributes
- Methods or operations

We've already seen the syntax for the first three of these items in earlier sections of this IDL overview. They become part of an interface simply by being declared within the braces of the body of the interface. In the next few sections, we'll see how to define interface attributes and methods, and then we'll look at how inheritance of IDL interfaces works.

G.10.1. Attributes

Attributes are data members that belong to interfaces. To readers familiar with JavaBeans, declaring an attribute on an interface is roughly analogous to adding a property to a Java bean. An attribute in an IDL interface indicates that the interface provides some way to read and (in most cases) write the attribute value.

The syntax for declaring an attribute within an interface body is:

```
// IDL
[readonly] attribute <type spec> <identifier> [, <identifier>, ...];
```

The attribute is signified by the `attribute` keyword, following by a type specification for the attribute and an identifier name. You can declare multiple attributes of the same type by providing their identifiers in a comma-delimited list after the type specifier:

```
// IDL
attribute short coord_x, coord_y, coord_z;
```

The type specifier can be any valid type, including IDL basic types, other interfaces, and user-defined types previously defined or declared in a `typedef`. For example:

```
// IDL
enum ErrorCode { BadValue, DimensionError, Overflow, Underflow };

interface AttrTest {
    struct coord {
        short x;
        short y;
    };
    attribute ErrorCode lastError;
    readonly attribute coord COG;
    attribute string name;
};
```

The optional `readonly` keyword can precede the attribute declaration. This indicates that the attribute can only be read externally and not directly written. This typically means that the value of this attribute is set only as a side effect of some other method(s). In our example, the `COG` attribute may represent the center of gravity of some geometric object, and we'll want that to be recomputed only as the result of other methods that change the geometry of the object.

G.10.2. Methods

Methods (or *operations*, to use the IDL vernacular) provide a way for remote clients to interact with the objects defined by an interface. A method declaration in IDL is composed of an identifier for the method, the type of data returned by the method, and a list of data arguments or parameters that the method accepts. An IDL method can also (optionally) be declared to use specific call semantics, to possibly raise certain exceptions during its execution, and to accept certain context variables from

the client environment.

The syntax of a method declaration within an IDL interface is:

```
// IDL
[<call semantics>] <return type> <identifier> ([<param>, <param>, ...])
[<exception clause>] [<context clause>];
```

The only required elements in a method declaration are the method identifier and the return type, so an example of the simplest form of method declaration is:

```
// IDL
boolean doSomething( );
```

This method simply returns a `boolean` flag when it is complete. It doesn't accept any arguments, uses the default call semantics, doesn't raise a nonstandard exception, and accepts no context variables from the client environment.

The return type for an IDL method can be any valid type, including user-defined types such as structs and other interfaces. If a method doesn't return any data, then the return type should be declared as `void`.

The identifier for a method is a valid IDL identifier. In IDL, two methods in the same interface can't have the same identifier (i.e., there is no method overloading, as there is in C++ and Java).

G.10.2.1. Arguments

The arguments to a method on an interface are declared within the parentheses following the method identifier and are separated by commas. The syntax for an individual method argument is:

```
<arg direction> <arg type> <identifier>
```

The identifier is any valid IDL identifier, and the argument type is any valid IDL type, including user-defined types.

The `direction` specification indicates whether the argument is passed into the server, returned from the server, or both. The direction specification can have one of three values: `in`, `out`, or `inout`. An argument tagged as `in` is passed only from the client to the server object. An argument tagged as `out` is not taken from the client, but its value is set by the server and returned if the method returns successfully. An `inout` argument is passed in both directions; the object/data from the client is passed to the server, and the server may modify the data and have the updates returned back to the

client if the method returns successfully.

For example, suppose we want to change the `doSomething()` method that we just declared, have it pass a directive to the server object in the form of a string, telling it what to do, and have another string passed in as the thing to act on. In this case, we want the first argument (the string directive) to be input-only, and we want the modified object to be passed in and returned in its changed state. So we declare the method as follows:

```
boolean doSomething(in string whatToDo, inout string whatToDoItTo);
```

If a method raises an exception during its execution, then the values of any `out` or `inout` arguments to the method are undefined. They may or may not have been modified by the method before the exception was raised and execution was halted.

G.10.2.2. Exceptions

If a method on an interface can raise any exceptions during its execution, then you have to declare this in IDL by adding a clause to the method declaration that lists all of the exceptions that can be raised by the method. This is similar to the `throws` clause on Java methods. The syntax for the `raises` clause looks like:

```
// IDL
raises (<exception type>, <exception type>, ...)
```

Every exception that you list in this clause has to be defined earlier.

Every method that you declare on an IDL interface can potentially throw one of the standard ORB exceptions that we mentioned earlier (see [Table G-4](#)). You can't list these standard exceptions in the `raises` clause for your methods.

For example, suppose we define our own `BadDirective` exception for our `doSomething()` method, which is raised if the client passes in a string directive that the server object doesn't understand. We modify the method declaration to look like the following:

```
// IDL
boolean doSomething(in string whatToDo, inout string whatToDoItTo)
    raises (BadDirective);
```

Again, we have to declare the `BadDirective` exception, and any data it contains, earlier in the IDL file

G.10.2.3. Context values

IDL supports the concept of a client context, which can contain name/value pairs that describe the client's environment in some way. You might have an authenticated username stored in the client's context, for example. The name of a value is a string, and its value is an `Any` object. The interface to the context is provided by the IDL `Context` interface, and a mapping of this interface must be provided in any language-specific binding of the CORBA standard.

You can add a context clause to your method declarations, which indicates which client context variables should be propagated to the server when the method is invoked. The server object can then query these context variables during the execution of the method. The syntax for adding a context clause to your method declaration is:

```
// IDL
context (<var name>, <var name>, ...)
```

Each `<var name>` is a string literal that names the context variable to be propagated to the server when the method is called.

Suppose that when we invoke our `doSomething()` method we want to be able to log who is making the request. We'll look for a `username` variable in the client context and assume that it is the authenticated identity of the client. We can specify that this context variable should be included in the method call by adding a context clause to our method declaration:

```
// IDL
boolean doSomething(in string whatToDo, inout string whatToDoItTo)
    raises (BadDirective) context ("username");
```

A Java client might use this method like so:

```
// Java
// Get our context
Context ctx = ORB.get_default_context( );
// Add our username to the context
Any username = new Any( );
username.insert_string("JimF");
ctx.set_one_value("username", username);
// Call the remote method
obj.doSomething("anything", "entity");
```

Since we declared the `doSomething()` method to include the `username` context variable in its invocations, this variable appears in the server's context and can be queried during execution of the method.[\[3\]](#)

^[3] Sun's implementation of the Java IDL binding doesn't support context variables. The `Context` interface is available in the Java IDL API, but context clauses on IDL methods aren't represented in the generated Java code, and no context data is transferred to the server.

You might ask when this context feature can be used as opposed to adding a method argument to the method declaration. We could have just as easily added another string argument to our declaration for the `doSomething()` method:

```
boolean doSomething(in string whatToDo, inout string whatToDoItTo,
    in string username) raises BadDirective;
```

One argument for using context variables is to make things easier on the client when certain data for a method is optional. Rather than including an explicit argument and forcing the user to add a `nil` value of some kind to the method call (`null` in Java, for example), you can make the optional data a context variable and the user can choose to set it or not. In most cases, you'll find the context variables used rarely, if at all.

G.10.2.4. Call semantics

If you don't specify any call semantics at the start of your method declaration, then the default semantics are "at most once." This means that if a method call returns with no exceptions, then the method was called a single time on the server object. If an exception is raised, then the method is called at most once (the exception occurred either before the method was invoked or during execution of the method).

You can choose to use alternate call semantics for your method by including a call attribute at the start of your method declaration. In the current CORBA standard, only a single alternative is available, which is called "best-effort" semantics. In this case, whether the method call returns successfully or not, there's no guarantee that the method was actually invoked on the server object. The difference between the default semantics and best-effort semantics is roughly equivalent to the difference between TCP and UDP IP network connections and their handling of data packets.

Specify best-effort call semantics by adding the keyword `oneway` to the start of your method declaration:

```
// IDL
oneway void tryToDoSomething(in whatToDo);
```

If you specify that a method is `oneway`, then the return type of the method has to be `void`, and it can't have any `out` or `inout` arguments. The method is effectively called asynchronously, so the client can't synchronously receive return data from the server object.

G.10.3. Interface Inheritance

You can inherit attributes and methods from other IDL interfaces by deriving your interface from them. The syntax for declaring the inheritance of an interface in its header is:

```
interface identifier : <parent interface>, <parent interface>, ... {
```

The parent interfaces can be any predefined interfaces, in the same module as this interface or in different modules. If the parent interfaces are from other modules, you need to use the `::` scope specifier to identify them.

G.10.3.1. Method and attribute inheritance

A derived interface inherits all of the attributes and methods from its parent interfaces. Although IDL allows for multiple inheritance, it's illegal to have two inherited attributes or methods with the same identifier. You also can't declare an attribute or method within your interface with the same name as an inherited attribute or method (i.e., *overload* a method or attribute). So if you have two interfaces declared as follows:

```
// IDL
interface A {
    boolean f(int float x);
};
interface B {
    void f( );
};
```

you can't define a new interface that derives from both of these interfaces, since the definition of the method `f()` is ambiguous. Notice that, unlike C++ and Java, IDL uses only the name for the method as its unique identifier, and not the entire method signature. This rule is a result of IDL's multilanguage support, since some languages may be similarly limited.

G.10.3.2. Constant, type, and exception inheritance

A derived interface also inherits any constants, user-defined types, or exceptions defined in its parent interfaces. They can be referred to in the derived interface as if they had been defined within the interface. If we define the following base interface, for example:

```
// IDL
interface Server {
    exception ServiceInterrupted {};
```

```

        boolean doSomething(in string what) raises (ServiceInterrupted);
};

```

we can use the `ServiceInterrupted` exception that we defined within the `Server` interface in another interface by naming its scope:

```

// IDL
interface PrintServer {
    boolean printSomething(in string what)
        raises (Server::ServiceInterrupted);
};

```

Alternatively, we could derive the `PrintServer` from the `Server` interface, and then the exception can be used as if it existed in the `PrintServer` scope:

```

// IDL
interface PrintServer : Server {
    boolean printSomething(in string what) raises (ServiceInterrupted);
};

```

It is legal to define constants, types, and exceptions in a derived interface using the same names as those in its parent interfaces. If you do this, though, you need to refer to them unambiguously in your interface declaration, using fully scoped names if necessary. If you declare your own `ServiceInterrupted` exception in the `PrintServer` interface, for example, you need to provide a scope for the exception in the `raises` clause, in order for the IDL compiler to know which version you're referring to:

```

// IDL
interface PrintServer : Server {
    exception ServiceInterrupted { string printerName; };
    boolean printSomething(in string what)
        raises (PrintServer::ServiceInterrupted);
};

```

If you don't provide a scope, the IDL compiler throws back an error about `ServiceInterrupted` being ambiguous.

G.10.3.3. IDL early binding

It's important to realize that IDL does "early binding" of constants, user-defined types, and

exceptions as it compiles your IDL. This means that the definition of a constant, type, or exception is bound to a particular reference within an interface as it's encountered in your IDL file not after all definitions have been examined. If we have the following IDL definitions, for example:

```
// IDL
struct Coord {
    short x;
    short y;
};

interface GeometricObj {
    attribute Coord cog;
};

interface GeometricObj3D : GeometricObj {
    struct Coord {
        short x;
        short y;
        short z;
    };
    attribute Coord cog3D;
};
```

then the `cog` attribute in the `GeometricObj` interface is the global `Coord` type (with `x` and `y` members only), since at the time that the `cog` attribute is encountered in the IDL file, this is the binding definition for `Coord`. The `GeometricObj3D` interface inherits this attribute with this type, but the `cog3D` attribute declared in the `GeometricObj3D` interface is the `GeometricObj3D::Coord` type (with `x`, `y`, and `z` members), since at that point, the `Coord` struct within the `GeometricObj3D` scope is defined and is the default reference of the relative `Coord` type used in the `cog3D` declaration.

G.10.4. Mapping Interfaces to Java

As you might expect, each interface that you define in IDL is mapped to a public interface in Java. Helper and holder classes are also generated for each interface the names of these interfaces are generated using the identifier of the IDL interface, with `Helper` and `Holder` appended to it.

The Java interface extends the `org.omg.CORBA.Object` interface. Any inheritance specification that you provide in your IDL interface is mapped directly to interface inheritance in Java, using `extends` clauses. So our earlier `PrintServer` example, inheriting from `Server`, is mapped into a Java interface that begins with the following:

```
// Java
public interface GeometricObj3D
    extends org.omg.CORBA.Object, GeometricObj { ...
```

G.10.4.1. Helper and holder classes

The helper class generated for an interface includes a static `narrow()` method that allows you to safely downcast a CORBA `Object` reference to a reference of the interface type. If the `Object` isn't of the expected type, then an `org.omg.CORBA.BAD_PARAM` exception is thrown. The helper class also includes other static methods that let you read or write objects of the interface type over I/O stream: or insert/extract an object of this type from an `Any` value.

The holder class is used whenever the interface is used as the type for an `out` or `inout` method argument. The holder class is responsible for marshaling the contents of the object to the server object for the method call (for `inout` arguments), and then unmarshaling the (possibly updated) return value. The holder class has a constructor defined that lets you wrap the holder around an existing instance of the original interface, and it has a public value member that lets you access the object argument both before and after the method call. You can see more details on using holder classes in [Chapter 14](#).

G.10.4.2. Attributes

Each attribute that you declare on the IDL interface is mapped into two accessor methods, each with the same name as the attribute. So an attribute declared within an IDL interface as follows:

```
// IDL
attribute string name;
```

is mapped into these two methods on the corresponding Java interface:

```
// Java
String name( );
void name(String n);
```

If you include the `readonly` tag on your IDL attribute declaration, the Java interface has only the read accessor method and not the update accessor.

G.10.4.3. Methods

Methods declared on your IDL interface are mapped one-to-one to methods on the Java interface. The return values and any `in` arguments are mapped directly to their corresponding types in Java. Any `out` or `inout` arguments in the IDL method are mapped to their holder classes in the Java method. This includes basic IDL types, which have their own holder classes defined for them in the standard Java mapping. So this IDL method:

```
// IDL
boolean setPrintServer(in PrintServer server,
    out PrintServer previousServer,
    out long requestsHandled);
```

be mapped into the following Java method on the corresponding interface:

```
// Java
boolean setPrintServer(PrintServer server,
    PrintServerHolder previousServer,
    IntHolder requestsHandled);
```

Notice that the last argument is declared a `long` in IDL, which is mapped to `int` in Java. The `IntHolder` class is therefore used in the mapped Java method.

To use this method, we have to create holder objects for the output parameters, then check their values after the method call.

```
// Java
PrintServer newServer = ...;
PrintServerHolder prevHolder = new PrintServerHolder( );
IntHolder numReqHolder = new IntHolder( );
xxx.setPrintServer(newServer, prevHolder, numReqHolder);
int numReq = numReqHolder.value;
PrintServer prevServer = prevHolder.value;
```

We don't need to initialize the contents of the holders, since they are being used for `out` parameters. If they are used for `inout` parameters, we have to initialize their contents at construction time or set their `value` members directly.

If there is a `raises` clause on your IDL method declaration, then it is mapped to an equivalent `throws` clause on the Java method. The context clause and call semantics (`oneway`) on an IDL method declaration affect only the implementation of the generated Java method, not its signature.

G.11. Value Type Declarations

Value types are a new IDL construct introduced when Objects by Value was adopted in the CORBA specification, for Version 2.3. Prior to the adoption of Objects by Value, there were two argument-passing semantics available in IDL. Interface types were passed by reference, and basic data types were passed by value. But there wasn't any way to pass an object by value between remote agents, in the way that Java serialization provides when using Java RMI. The CORBA Objects by Value specification extends IDL to include a new entity, called a *value type*, that is similar in syntax to an interface or a `struct`.

Value types are declared using the `valuetype` IDL keyword:

```
valuetype Coord3DVal { ... };
```

Any entity declared as a `valuetype` is passed by value when used as the argument to an operation. In other words, the servant of the operation receives a copy of the entity, not a remote reference to the entity residing on the caller, as is the normal case for IDL interfaces.

Value types can be declared with a custom modifier, which indicates that it will use custom marshaling code provided by the developer in an implementation class:

```
custom valuetype Coord3DValCustom { ... };
```

This modifier alters the nature of the generated "native" code, as described in the next section.

Value types can also inherit other value types, which have semantics similar to interface inheritance: the value type inherits all of the attributes, operations, state members, and initializers of its parent(s). The syntax of value type inheritance is the same as interface inheritance:

```
valuetype childVal : parentVal { ... };
```

Value type definitions can contain everything an interface can, as well as state members and initializers. Any members declared as state members are considered part of the state of the value type, which needs to be marshaled and transmitted over the network when this `valuetype` is used in a remote operation call. Any attributes declared in a `valuetype` aren't considered part of the state that is passed by value instead, they're considered part of the local state of the entity. State members are declared using either a `public` or `private` modifier, followed by a type specification and the

declaration of the member identifier itself. Public state members are publicly available to the application code; private members are accessible only in the implementation code and during marshal operations. It's the responsibility of the language mapping to determine how this is implemented in a particular language.

An initializer is a declaration of a constructor for this `valuetype`. Initializer declarations are identical to operation declarations, except that they start with a special factory modifier. An initializer should have arguments that are sufficient for the entity's state to be initialized fully.

For example, suppose we want to declare an object-like version of our `Coord3d` typedef, in that we want our `Coord3d` to be able to have operations of its own. But we still want to be able to pass these coordinates by value in remote operations. We can declare a new `valuetype`:

```
valuetype Coord3DVal {
    // The state that should be passed by value
    private Coord3d coord;
    // A local short attribute, not passed in remote operations
    attribute short localShort;
    // Initializer for this type
    factory makeCoord(in long x, in long y, in long z);
    // Operations to access state, etc.
    long getX( );
    long getY( );
    long getZ( );
};
```

The `Coord3DVal` valuetype has a single state member, `coord`, that holds the coordinate value as a `Coord3D` (our earlier typedef). This state data is marshaled and passed by value when a `Coord3DVal` is used as an argument in a remote operation. The `coord` member is declared as `private`, so it is not directly accessible to the application code. The `Coord3DVal` also has a local attribute, `localShort`, that is not considered part of the state and is not transferred in remote operation calls. We declare a single initializer, `makeCoord()`, for our `valuetype`, which takes the three coordinate values for the `Coord3d` state member. Finally, our `valuetype` has some operations that provide access to the individual coordinate values of the `Coord3d` state member.

G.11.1. Mapping Valuetypes to Java

For the most part, valuetypes are mapped to Java along the same lines as interfaces, but with some key differences. A valuetype is mapped into an abstract Java class of the same name, rather than into a Java interface. Each attribute is mapped to instance variables in the same manner that interface attributes are mapped, using accessors that suit the modifiers on the attribute. State members, however, are mapped directly to instance variables. Public state members are mapped to public instance variables, and private state members become protected instance variables. Regular IDL operations are mapped to methods on the Java class in the same way as interface operations. Initializers, however, are mapped into a separate Java interface, named `xxxValueFactory`, where `xxx` is the name of the valuetype. Each initializer declared in the IDL valuetype is mapped to an equivalent method on the `xxxValueFactory` interface. To complete the mapping of the valuetype, you

need to provide concrete implementations of both the mapped abstract class and the `XXXValueFactory` interface.

Our `Coord3dVal` valuetype is mapped by *idlj* to the following Java abstract class:

```
public abstract class Coord3DVal implements org.omg.CORBA.portable.StreamableValue
{
    // The state that should be passed by value
    protected int coord[] = null;

    private static String[] _truncatable_ids = {
        test.Coord3DValHelper.id ( )
    };

    public String[] _truncatable_ids( ) {
        return _truncatable_ids;
    }

    // A local short attribute, not passed in remote operations
    public abstract short localShort ( );

    // A local short attribute, not passed in remote operations
    public abstract void localShort (short newLocalShort);

    // Operations to access state, etc.
    public abstract int getX ( );

    public abstract int getY ( );

    public abstract int getZ ( );

    public void _read (org.omg.CORBA.portable.InputStream istream)
    {
        this.coord = test.Coord3dHelper.read (istream);
    }

    public void _write (org.omg.CORBA.portable.OutputStream ostream)
    {
        test.Coord3dHelper.write (ostream, this.coord);
    }

    public org.omg.CORBA.TypeCode _type ( )
    {
        return test.Coord3DValHelper.type ( );
    }
}
```

The generated class inherits from `org.omg.CORBA.portable.StreamableValue`, which ensures that it is

streamable over an I/O stream. The generated `_write()` and `_read()` methods provide the code to marshal and unmarshal this value type when it is passed in a remote CORBA method call.

In addition, a `Coord3DValValueFactory` interface is generated to hold our single initializer method:

```
public interface Coord3DValValueFactory extends
    org.omg.CORBA.portable.ValueFactory
{
    // Initializer for this type
    Coord3DVal makeCoord (int x, int y, int z);
}
```

Once you provide the concrete implementations of these interfaces, an application creates one of these valuetypes by constructing a concrete subclass of the `Coord3DValValueFactory` interface and calling its `makeCoord()` method to obtain a `Coord3DVal`. This object can then be passed by value into any CORBA remote method.

[← PREV](#)

[← PREV](#)

Appendix H. HJava IDL Tools

This appendix provides details on the IDL-related tools provided with Sun's JDK implementation. Other JDK implementations or J2EE environments may supply their own versions of these tools, or other tools altogether. Consult your vendor's documentation for details on their tools.

[← PREV](#)



idlj: The Java IDL Compiler

Since JDK 1.3

Synopsis

```
idlj [ options ] IDLfile
```

Description

The *idlj* compiler generates Java source files from interfaces, modules, and type descriptions defined in IDL. The compiler implements the standard IDL-to-Java mapping defined by the OMG.

The *idlj* compiler accepts the standard IDL preprocessor directives, such as `#include`, which includes declarations from another IDL file, and `#define`, which defines symbols to be used for conditional compilation with `#ifdef` directives.

Run with no arguments at all, *idlj* generates all the client-side Java classes mapped from the specified IDL file. To generate both client and server-side Java bindings, you need to use the `-fall` option described later.

As of JDK 1.4, *idlj* generates POA-compliant server-side classes. The server skeletons inherit from `org.omg.PortableServer.Servant` and use the POA operations to implement the skeleton. In JDK 1.3, *idlj* generates server classes based on the nonstandard `ImplBase` inheritance scheme. The `ImplBase` scheme can also be used in JDK 1.4, by using the `-oldImplBase` option described later.

Options

`-d symbol`

Defines the named symbol during preprocessing of the IDL file. Invoking *idlj*:

```
% idlj -d foobar Account.idl
```

is equivalent to placing the following preprocessing directive at the top of the *Account.idl* file:

```
#define foobar
```

`-emitAll`

By default, `idlj` emits Java mappings only for entities found directly in the specified IDL file. This option causes `idlj` to emit Java mappings for all entities found in this file and in any files `#include`'d in the file.

`-f <side>`

Controls whether to generate client- and/or server-side mappings and which flavor of mapping to generate. The following values are supported:

`client`

Default value. Emits only the client-side mappings for the IDL entities.

`server`

Emits only the server-side mappings. In JDK 1.3, it generates `ImplBase`-style skeletons by default; as of JDK 1.4, it generates POA-compliant skeletons by default.

`serverTIE`

Emits server-side mappings based on the `TIE` delegation model. See [Chapter 14](#) for more details on the `TIE` model.

`all`

Generates both client and server mappings. This is equivalent to specifying `-fclient -fserver` on the command line.

`allTIE`

Generates both client and server mappings, but the server mappings follow the `TIE` delegation model. This is equivalent to specifying `-fclient -fserverTIE` on the command line.

`-i <path>`

The given path should be used to search for files specified in `#include` directives found in the IDL file. You can include multiple paths using multiple `-i` options on the command line, or you can use a delimited list of paths with a single `-i` option. The paths are delimited by colons on Unix variants (Linux, Mac OS, Solaris, etc.), and by semicolons on Windows.

`-J <javaoption>`

Passes the specified option directly to the underlying Java virtual machine. There is no space between the `-J` and the option. For example, use the following to run the JVM in verbose mode

```
> idlj -J-verbose ...
```

`-keep`

Don't overwrite a generated file if it already exists. By default, the compiler overwrites existing Java files. This is handy if you've customized generated Java files and want to only regenerate other files emitted from the IDL.

`-noWarn`

Don't print warning messages if encountered when compiling the IDL.

`-oldImplBase`

This version of `idlj` generates POA-compliant server classes by default. This option instructs the compiler to generate the `ImplBase`-style server skeletons used in older versions of Java IDL (1.3 and 1.2).

`-pkgPrefix <entity> <prefix>`

Overrides the normal package prefix given to the named entity. Normally entities such as an IDL module, interface, value type, and so on are placed into a package based on the modules it falls within. This option lets you override this procedure for a given entity using a specific package name.

`-pkgTranslate <moduleName> <newName>`

Wherever the string specified by `<moduleName>` is found in an IDL identifier, it's replaced with the `<newName>` in the generated Java files. For example, if you have an IDL file containing a module `Servers` with an interface `PrintServer`, this invocation of `idlj`:

```
> idlj -pkgTranslate Servers com.myorg.Servers servers.idl
```

causes the `PrintServer` interface to map to the package `com.myorg.Servers`. You can use multiple `-pkgTranslate` options on the command line.

`-skeletonName <prefix>%<postfix>`

It allows you to customize the name of generated server skeletons, where `%` represents the name of the IDL interface being mapped. If you have an IDL file containing an interface `PrintServer`, the following:

```
> idlj -skeletonName foo%bar servers.idl
```

generates a server skeleton class named `fooPrintServerbar`, in a file of the same name. By default, skeletons are named `<interfaceName>POA` if POA-compliant server classes are generated; they are named `_<interfaceName>ImplBase` if `ImplBase`-style server classes are generated. This custom naming scheme is applied only to the skeleton files, it doesn't affect any `TIE` delegates (use the `-tieName` option to customize these).

```
-td <path>
```

The given path should be used as the destination for the generated Java source files. If a Java class is mapped to a Java package, the compiler automatically creates subdirectories in the given destination to store the Java files. If this option isn't used, the compiler generates the Java files in the current directory.

```
-tieName <prefix>%<postfix>
```

This option allows you to customize the name of generated `TIE` delegates, where `%` represents the name of the IDL interface being mapped. If you have an IDL file containing an interface named `PrintServer`, the following:

```
> idlj -fallTIE -tieName foo%bar servers.idl
```

generates a `TIE` delegate class named `fooPrintServerbar`, in a file of the same name. By default, `TIE` delegate classes are named `<interfaceName>POATie` if POA-compliant server classes are generated; they are named `<interfaceName>_Tie` if `ImplBase`-style server classes are generated. This naming scheme is applied only to the `TIE` files; it doesn't affect any skeletons. Use the `-skeletonName` option to customize these.

```
-verbose
```

Runs the compiler in verbose mode, printing out progress messages as the IDL is parsed.

```
-version
```

Prints the version information for `idlj`. The compiler doesn't compile any files when this option is specified; it simply prints the version and exits.

orbd: Naming Service Daemon

Since JDK 1.4

Synopsis

```
orbd -ORBInitialPort <ns-port> [ options ]
```

Description

The *orbd* tool is a server daemon that runs a CORBA Naming Service. It's a replacement for the *tnameserv* tool, which is still provided as part of Sun's JDK (described later). *orbd* supports both transient and persistent CORBA servants running within its Naming Service. For transient naming services, *orbd* can be used in a similar fashion to *tnameserv*. The Naming Service is started and told to listen on a particular port using the `-ORBInitialPort` option, and CORBA servants are registered directly to this Naming Service on the given port. These servants are transient, in the sense that their references in the Naming Service last only for the lifetime of the *orbd* process.

To access the persistent services of *orbd*, you must use *servertool* (described later in the appendix). The *orbd* daemon has a persistent reference manager internally and a database of persistent references. The daemon can use these to reinitialize a servant if it dies or to reinitialize a servant reference if the *orbd* daemon dies and is restarted. On their own, CORBA servants can access the default transient services of *orbd*, but *servertool* can invoke *orbd*'s persistent reference services using an internal protocol between itself and *orbd*. To make a CORBA servant persistent, register it with *orbd* using the *servertool*/command interface.

To start the *orbd* daemon, run it from the command line with (at a minimum) an initial port specified:

```
> orbd -ORBInitialPort 1050
```

If you want to register persistent CORBA server objects with *orbd*, start *servertool* using the same port so that it connects to the running *orbd* process:

```
> servertool -ORBInitialPort 1050
```

This presents the *servertool* command-line interface, which allows you to register CORBA servers with *orbd*, as described later.

Options

`-defaultdb < path>`

Specifies the directory where *orbd* should maintain its persistent information in a file called *orbd.db*. By default, the current directory is used for this file.

`-J< javaoption>`

Passes the given option directly to the underlying Java virtual machine. There is no space between the `-J` and the option. For example, to run the JVM in verbose mode, use:

```
> orbd -J-verbose ...
```

`-ORBInitialPort < portnum>`

Required. The standard CORBA property controls the initial port used for services, such as the Naming Service. Normally it's optional for CORBA servants and tools, but the *orbd* daemon requires that this option be specified on the command line. The default port for the Naming Service is port 900, but on some Unix variants, ports less than 1024 are restricted to root access, so you need to specify a higher port number, such as 1050.

`-port < port>`

Customizes the port number used by *orbd* for requests to reactivate persistent servers. This option is useful when the default port of 1049 isn't available.

`-serverPollingTime <time>`

Sets the frequency (in milliseconds) with which the daemon checks on its persistent servers. If not specified, the default time is 1000 ms.

`-serverStartupDelay < time>`

Sets the delay *orbd* inserts between the time it restarts a persistent server and the time it forwards the next request targeted for that server. This gives the server a chance to initialize itself. If not specified, the default time is 1000 ms.



servertool

Since JDK 1.4

Synopsis

```
servertool -ORBInitialPort <ns-port> [ options ] [ commands ]
```

Description

The *servertool* is used with the *orbd* Naming Service to provide persistent CORBA server objects. A CORBA server registered with an *orbd* daemon using *servertool* is managed as a persistent named server. If the server object dies, the next request made of that server through *orbd* causes *orbd* to reactivate the server object using information in *orbd*'s persistent store. This information is registered with *orbd* using *servertool*'s command interface.

Options

-J< *javaoption* >

The given option is passed directly to the underlying Java virtual machine. There is no space between the *-J* and the option. For example, to run the JVM in verbose mode, use:

```
> servertool -J-verbose ...
```

-ORBInitialPort < *ns-port* >

Required. This is the standard CORBA property that controls the initial port used for services such as the Naming Service. Normally it's optional for CORBA servants and tools, but the *servertool* requires it be specified on the command line. The default port for the Naming Service is port 900, but on some Unix variants, ports less than 1024 are restricted to root access, so you need to specify a higher port number, such as 1050. The port number specified when starting *servertool* should match the *ORBInitialPort* used to start the *orbd* daemon.

Commands

The following commands are executed using the *servertool* command line or prompt. You can specify a single command from the list on the command line when you invoke *servertool*, *servertool* executes the command and exits. If you don't provide a command on startup, *servertool* presents a prompt, allowing you to execute the command.

```
getserverid -applicationName < appName>
```

Prints the server identifier for a server object, given its application name.

```
help
```

Prints a list of the available commands.

```
list
```

Lists details about all the persistent servers registered with the `orbd` daemon.

```
listactive
```

Lists details about all currently active server objects (i.e., all servers that are registered and running, not awaiting activation or reactivation).

```
listappnames
```

Lists the application names for all the persistent servers registered with the `orbd` daemon.

```
locate -serverid < serverID> [ -endpointType < type> ]
```

```
locate -applicationName < +> [ -endpointType < type> ]
```

Prints details of the ports used by the ORB(s) created for a particular server object. The `server` object can be specified using either its server identifier (obtained using the `getserverid`, `list`, or `listactive` commands) or using its application name (obtained using the `list`, `listappnames`, or `listactive` commands). If an endpoint type is specified, only ports of that type are listed. The only port type that seems to be supported in Sun's JDK is `IIOP_CLEAR_TEXT`.

If the specified server object isn't running when this command is invoked, it is activated.

```
locateperorb -serverid < serverID> [ -orbid < orbID> ]
```

```
locateperorb -applicationName < appName> [ -orbid < orbID> ]
```

The same as `locate`, except that it allows you to specify a specific ORB started by the server object and prints ports only for that ORB. If no `-orbid` option is used, an empty ORB name (i.e., " ") is used. The names given to ORBs that are started by a given server object are obtained using the `orblast` command.

```
orblast -serverid < serverID>
```

```
orblast -applicationName < appName>
```

Lists the identifiers for the ORBs created by a server object. If the specified server object isn't running, it is activated.

```
quit
```

Quits the *servertool*/process but doesn't stop the *orbd* daemon.

```
register -server < serverClass> -classpath < pathlist> [ -applicationName < name> ] [ -args < serverArgs> ] [ -vmargs < javaoptions> ]
```

Registers a new CORBA server object with the persistent naming services of *orbd*. The `-server` option specifies the fully qualified name of a server object that registers CORBA objects with the Naming Service. The specified class must have a `main()` method, which is executed by the daemon to activate and reactivate the CORBA server. The server object is activated automatically when this *register* command is invoked. If the server object also has an `install()` method with the following signature:

```
public static void install(org.omg.CORBA.ORB orb)
```

then this method is also invoked once when this command is invoked in *servertool*. This provides a way for the server object to have a one-time setup of persistent resources.

The `-classpath` option specifies where to search for the classes required by the server object.

The `-applicationName` option specifies a name to associate with the server object. This name is used in the information printed by the *list*, *listappnames*, and *listactive* commands and can also unregister a server object using the *unregister* command.

The `-args` option specifies arguments to give to the server object when its `main()` method is invoked.

The `-vmargs` option specifies arguments to pass to the JVM when the server object is activated.

```
shutdown -serverid < serverID>
```

```
shutdown -applicationName < appName>
```

Shuts down (deactivates) a server object registered with the ORB. If the server class has a `shutdown()` method, it's called when this method is invoked.

```
startup -serverid <serverID>
```

```
startup -applicationName <appName>
```

Manually reactivates a server object. If the specified server is already running, the command returns an error and does nothing.

```
unregister -serverid <serverID>
```

```
unregister -applicationName <appName>
```

Removes a server object from the Naming Service's persistence services. The server object can be referenced using either its server identifier (which can be obtained using the *getserverid*, *list*, or *listactive* commands) or using its application name (which can be obtained using the *list*, *listappnames*, or *listactive* commands).

If the server object has a static `uninstall()` method with the signature:

```
public static void uninstall(org.omg.CORBA.ORB orb)
```

then this method is invoked when `unregister` is called on the server object. This lets the server object clean up any persistent resources.



tnameserv: Transient Naming Service Daemon

Since JDK 1.2

Synopsis

```
tnameserv [ options ]
```

Description

The *tnameserv* daemon is a transient implementation of the CORBA Naming Service provided with Java IDL. *tnameserv* allows remote CORBA objects to bind to names with the naming directory of the service. Remote clients can connect to *tnameserv* through standard CORBA APIs and ask for references to these objects by name or browse through the objects bound to names in the directory. This daemon runs a Naming Service implementation whose object bindings persist only for the lifetime of the daemon process. The *orbd* tool, described earlier, provides both persistent and transient naming services.

Options

`-J< javaoption >`

Passes the given option directly to the underlying Java virtual machine. There is no space between the `-J` and the option. For example, use the following to run the JVM in verbose mode

```
> tnameserv -J-verbose ...
```

`-ORBInitialPort < portnum >`

Listens to the specified port for client requests on the Naming Service. The default port is 900. This option corresponds to the Java property `org.omg.CORBA.ORBInitialPort`.

About the Authors

Jim Farley is a technology strategist, architect, manager, and author. His current focus is his role as the technology solutions architect for the Harvard Business School, working to assemble enterprise systems for both the instructional and back-office operations at the school. Jim also consults for various organizations in strategic technology and development, lectures in the computer science department at Harvard's Division of Continuing Education, and publishes articles in a number of forums. Jim's history with computing, enterprise and otherwise, has spanned numerous domains, including manufacturing, artificial intelligence, multimedia, collaboration tools, and content management.

William Crawford has developed web-based enterprise applications since 1995, including one of the first web-based electronic medical record systems, and some of the first enterprise-level uses of Java. He's also consulted for Children's Hospital Boston, Harvard Medical Center, and several Fortune 500 companies. He was chief technology officer at Invantage, Inc., and principal software architect at Perceptive, Informatics, Inc. He is also the coauthor of O'Reilly's J2EE Design Patterns and Java Servlet Programming. Will has taken a temporary leave from enterprise Java development to pursue MBA and MS degrees at MIT.

Prakash Malani has extensive experience in architecting, designing, and developing software in many application domains, such as entertainment, finance, retail, medicine, communications, and interactive television. He practices and mentors leading technologies such as J2EE, the Unified Modeling Language (UML), and XML. Prakash also teaches (and preaches) best practices and design patterns at institutions such as California State Polytechnic University, Pomona. He has published various articles in industry-leading publications including JavaWorld and WebLogic Developers Journal.

John G. Norman has been developing web-based products and services for more than 10 years, and has been working with technology since he was introduced in the mid 1970s to an IBM 1130 running FORTRAN and APL. Since 1999 he's worked on social networking products, business analytics, and e-commerce at companies such as Cambridge Incubator, Veritas Medicine, Elytics, and H3.com, where he is currently the chief software architect. John has taught literature at Ohio State, new media at the New School, and engineering at Harvard University's Extension Division. He holds a PhD in English literature from Harvard.

Justin Gehtland, a professional programmer, instructor, speaker, and pundit since 1992, has developed real-world applications using VB, COM, .NET, Java, Perl, and a slew of obscure technologies since relegated to the trash heap of history. His focus on "connected" applications led him to COM+, ASP/ASP.NET, and JSP. Justin is the coauthor of Effective Visual Basic and Windows Forms Programming in Visual Basic .NET (both Addison-Wesley). He is currently the Agility columnist on TheServerSide.NET, works as a consultant through his company Relevance, LLC, and teaches for DevelopMentor.

Colophon

Our look is the result of reader comments, our own experimentation, and feedback from distribution channels. Distinctive covers complement our distinctive approach to technical topics, breathing personality and life into potentially dry subjects.

The animal appearing on the cover of *Java Enterprise in a Nutshell*, Third Edition, is a sand dollar (*Echinarachnius parma*). The sand dollar is a flattened, rigid, disk-shaped marine invertebrate related to sea urchins and sea stars. It is found in large numbers on the sandy bottoms in the coastal waters of many parts of the world. The sand dollar's shell, or test, is often perforated with petal-shaped slots arranged around a central point. The mouth is located in this central position on the underside of the shell. The shell is covered with spines of varying lengths. These spines aid the sand dollar in locomotion and enable it to burrow just below the surface of the sandy bottom. In this sand, the sand dollar finds the tiny organic material it feeds on, pushing the food toward its mouth with tiny tube feet. Additional tube feet on the upper side of the sand dollar are used for breathing.

The sand dollar's flower-like appearance and its abundance in many parts of the world have made it a favorite of shell collectors. Scientists have also taken an interest in this small invertebrate. The sand dollar is frequently used in the study of mitosis, the process of cell division. It is believed that a better understanding of mitosis may lead to a better understanding of cancer.

Abby Fox was the production editor for *Java Enterprise in a Nutshell*, Third Edition. Norma Emory was the copyeditor, and Genevieve Rajewski was the proofreader. Matt Hutchinson and Marlowe Shaeffer provided quality control. Lydia Onofrei, Loranah Dimant, and Jansen Fernald provided production assistance. Johnna VanHoose Dinse wrote the index.

Edie Freedman designed the cover of this book, using a 19th-century engraving from the Dover Pictorial Archive. Karen Montgomery produced the cover layout with Adobe InDesign CS using Adobe's ITC Garamond font.

David Futato designed the interior layout. This book was converted by Keith Fahlgren from Microsoft Word to FrameMaker 5.5.6 with a format conversion tool created by Erik Ray, Jason McIntosh, Neil Walls, and Mike Sierra that uses Perl and XML technologies. The text font is Linotype Birka; the heading font is Adobe Myriad Condensed; and the code font is LucasFont's TheSans Mono Condensed. The illustrations that appear in the book were produced by Robert Romano, Jessamyn Read, and Lesley Borash using Macromedia FreeHand MX and Adobe Photoshop CS. The tip and warning icons were drawn by Christopher Bing. This colophon was written by Clairemarie Fisher O'Leary.

[← PREV](#)

Index

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[J\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#)

[← PREV](#)

[← PREV](#)

Index

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[J\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#)

[& \(ampersand\), combining with search filter](#)

[* \(asterisk\) wildcard character in LDAP searches](#)

[| \(vertical bar\), | \(OR\) operator, combining with search filter](#)

[← PREV](#)

[← PREV](#)

Index

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[J\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#)[abstract persistence schema, entity beans](#)[acceptChanges\(\) \(CachedRowSet\)](#)[Account interface \(example\)](#)[defining and implementing](#)[AccountClient class \(example\)](#)[AccountImpl class \(example\)](#)[implementation of Account class](#)[AccountReg class \(example\)](#)[ACID properties, transactions](#)[acknowledge\(\) \(Message\)](#)[Action class, Struts](#)[RequestProcessor](#)[action tags, JSPs](#)[JavaBeans and](#)[ActionForm class, Struts](#)[ActionServlet, Struts and](#)[Activatable class 2nd 3rd 4th](#)[constructors](#)[activating](#)[activation groups](#)[defining activatable objects](#)[passing arguments to activatable objects before activation](#)[registering activatable objects](#)[registering objects](#)[RMI activation service, features of](#)[rmid activation daemon](#)[ActivationDesc class](#)[ActivationGroup class](#)[entire group, becoming inactive](#)[ActivationGroupDesc class](#)[ActivationGroupID class](#)[ActivationID class 2nd](#)[ActivationMonitor class](#)[Activator interface](#)[activatable objects registry](#)[active object references, transferring](#)[activeObject\(\) \(ActivationGroup\)](#)[addBatch\(\)](#)[CallableStatement interface](#)[PreparedStatement interface](#)[Statement interface](#)

[addNamingListener\(\), EventContext and EventDirContext](#)

[addRowSetListener\(\) \(RowSet\)](#)

[aggregate functions, SQL](#)

[AlreadyBoundException class](#)

[ALTER TABLE statement \(SQL\)](#)

[AND operator, combining search filter with](#)

annotations

[introduction](#)

[J2SE tutorial](#)

[tools](#)

[XDoclet](#)

[custom](#)

[deployment descriptors](#)

[processing](#)

[ANSI-92 SQL state codes](#)

[Ant](#)

[application assemblies](#)

[application deployment](#)

[buildfiles](#)

[portability](#)

[properties](#)

[targets](#)

[builds, unit tests](#)

[component deployment](#)

[introduction](#)

[JUnit and](#)

[batch testing](#)

[test result format](#)

[overview](#)

[property files](#)

[targets, modular](#)

tasks

[directory tasks](#)

[enterprise](#)

[file tasks](#)

[Java tasks](#)

[utilities](#)

[versions](#)

Apache

[Ant and](#)

[Cocoon content management system](#)

[Xalan \(XSLT transformer\)](#)

[Xerces parser](#)

[XML project](#)

APIs

[distributed computing and](#)

[Enterprise JavaBeans and 2nd](#)

Hibernate API

[configuration and](#)

[events](#)

[interceptors](#)

[Query class](#)

- [Session object](#)
- [SessionFactory](#)
- [IDL](#)
- [J2EE security](#)
- [JACC](#)
- [Java RMI](#)
- [JAXP](#)
- [JCE](#)
- [JNDI](#)
- [JSF](#)
- [JSP](#)
- [JTA 2nd](#)
- [messaging](#)
- [Servlet API 2nd](#)
- [web UIs](#)
- [appendChild\(\)](#)
- [applets](#)
 - [classes, loading 2nd](#)
 - [deployment, Java drivers and](#)
 - [JDBC-ODBC Bridge, requirements for using](#)
- [application archives](#)
 - [deployment descriptors](#)
- [application assembly 2nd](#)
 - [Ant](#)
 - [example](#)
 - [module declaration](#)
 - [security](#)
- [application clients](#)
- [application layer security](#)
 - [authentication](#)
 - [authorization](#)
 - [declarative vs. programmatic](#)
- [application.xml, deployment descriptors](#)
- [applications](#)
 - [database, debugging](#)
 - [object references \(application-specific\)](#)
 - [server, network hardware and](#)
 - [web applications](#)
 - [structure](#)
- [archives, application archives](#)
- [arguments](#)
 - [IDL methods](#)
 - [initialization constructors](#)
 - [remote methods](#)
- [arithmetic operators](#)
 - [JMS message selectors](#)
- [arrays](#)
 - [IDL](#)
 - [update counts](#)
- [assembling applications, Ant](#)
- [assembly, introduction](#)
- [assertion methods \(JUnit\)](#)

[attachments to email messages](#)

attributes

[Attributes interface](#)

[DOM 2nd](#)

[IDL interfaces](#)

[parsing with SAX](#)

authentication

[application layer security](#)

[Authenticator class](#)

[client connections to JMS provider](#)

[EJB-tier](#)

[JAAS and](#)

[JNDI and](#)

[usernames and passwords 2nd](#)

[web-tier](#)

[Authenticator class](#)

[final methods for information about an authentication request](#)

authorization

[application layer security](#)

[EJB-tier](#)

[auto-commit mode \(transactions\) 2nd](#)

[Axis deployment model](#)

[← PREV](#)

Index

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[J\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#)

[backing beans, JSF](#)

[BAD_PARAM class](#)

[base document element \(XML\)](#)

[batch testing, JUnit and Ant](#)

[batch updates \(JDBC\)](#)

[BatchUpdateException class](#)

[bean implementation \(EJBs\)](#)

[bind\(\)](#)

[Context interface](#)

[NamingContext interface](#)

[Registry interface](#)

[bind_new_context\(\) \(NamingContext\)](#)

[binding](#)

[languages to CORBA](#)

[names to objects in naming service](#)

[objects to contexts](#)

[remote objects to names](#)

[bindings](#)

[for naming events](#)

[listing for JNDI context](#)

[black box testing](#)

[BLOBs \(binary large objects\)](#)

[BMP \(bean-managed persistence\)](#)

[BMTD \(Bean-Managed Transaction Demarcation\)](#)

[body tags, JSP](#)

[Boolean literals, IDL](#)

[branch, CORBA naming directory](#)

[browse command](#)

[browsers, transforming XML into HTML for](#)

[browsing](#)

[message queues](#)

[naming systems](#)

[buildfiles \(Ant\)](#)

[portability](#)

[properties](#)

[targets](#)

[builds \(Ant\), unit tests](#)

[built-in objects, JSP](#)

[business logic in EJB component model](#)

[BytesMessage class](#)



[← PREV](#)

Index

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[J\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#)

C/C++ languages

[IDL interfaces, converting to Java](#)

[Java and IDL syntax, modeled upon
nonclass data structure definitions](#)

[wrapping code with Java interface](#)

[CachedRowSet object](#)

[caching in Hibernate](#)

[Cactus 2nd](#)

[architecture](#)

[asynchronous code testing](#)

[model](#)

[running tests](#)

[test proxies](#)

[testing enterprise components](#)

[testing JSPs](#)

[testing process](#)

[testing servlet filters](#)

[testing servlets](#)

[writing tests](#)

[call semantics, IDL methods](#)

[CallableStatement interface](#)

[stored procedures](#)

casting

[CORBA remote objects to specific remote interfaces](#)

[Node to Element object and adding attributes](#)

[returned Context or DirContext for event source](#)

[catch blocks \(SQLException\)](#)

[cd command \(NamingShell application\)](#)

[changed transactions](#)

[character literals, IDL](#)

characters

[char strings](#)

[CLOBs \(character large objects\) 2nd](#)

[IDL](#)

[characters\(\)](#)

[children of a context, listing](#)

[ClassCastException class](#)

classes

[activatable remote objects](#)

[activation groups](#)

[configuring clients and servers](#)

- [_container-specific](#)
- [DII](#)
- [inner](#)
- [Java, IDL interfaces vs.](#)
- [loading dynamically](#)
- [loading remote](#)
- [mapping, Hibernate configuration](#)
- [names for \(JDBC\)](#)
- [public](#)
- [remote interface, implementing](#)
- [RMI, for remote object implementations](#)
- [security](#)
- [stubs and skeletons, generating](#)
- [classloading, models for deployment](#)
- [RMI](#)
- [classpath](#)
- [Ant](#)
- [local default class loader](#)
- [RMI registry](#)
- [clearBatch\(\) \(Statement\)](#)
- [clearBody\(\) \(Message\)](#)
- [clearParameters\(\) \(PreparedStatement\)](#)
- [clearProperties\(\) \(Message\)](#)
- [client interfaces \(EJBs\)](#)
- [client stub interfaces \(EJBs\)](#)
- [client stubs 2nd 3rd 4th 5th](#)
 - [_classes, loading](#)
 - [_classes, regenerating \(RMI/IIOP\)](#)
 - [_compiling](#)
 - [_configuring for remote class loading](#)
 - [_idltojava compiler, generating 2nd](#)
 - [_Java to CORBA object](#)
 - [_language-specific, generating](#)
 - [_original objects, changing](#)
 - [_relationships with skeletons and remote objects](#)
 - [_remote object classes](#)
 - [_remote objects, not instantiated](#)
 - [_RemoteObject as superclass for](#)
 - [_RMI architecture](#)
- clients
 - [_as Java objects \(RMI\)](#)
 - [_authenticated JMS connections](#)
 - [_distinguishing from servers in distributed object system](#)
 - [_EJB client](#)
 - [_factory classes](#)
 - [_freeing resources from connections and sessions](#)
 - [_general setup](#)
 - [_identifiers for](#)
 - [_Java IDL \(CORBA\)](#)
 - [_JMS](#)
 - [_key interfaces](#)
 - [_looking up remote object and calling method on it](#)

- [messaging](#)
- [point-to-point messaging](#)
- [receiving messages](#)
- [remote method arguments and return values](#)
- [remote method requests, server objects](#)
- [RMI 2nd](#)
- [RMI registry and naming services](#)
- [sending messages](#)
- [sessions](#)
- [setting RMISecurityManager for class loading](#)
- [SMTP](#)
- [stringified object reference, using](#)
- [temporary destinations, creating](#)
- web services
 - [DII approach](#)
 - [dynamic proxy approach](#)
 - [static proxy approach](#)
 - [writing](#)
- [CLOBs \(character large objects\) 2nd](#)
 - [Clob class](#)
- [cloneNode\(\) \(Node\)](#)
- [cloning a message](#)
- close()
 - [Connection interface 2nd](#)
 - [Folder class](#)
 - [Session interface](#)
- [clusters, RDBMS](#)
- [CMP \(container-managed persistence\), entity beans \(EJBs\)](#)
- [CMR \(container-managed relationship\) fields](#)
 - [EJB relationships](#)
- [CMTD \(container-managed transaction demarcation\) 2nd](#)
 - [EJBs, transactions and 2nd](#)
- [code compile, Ant tasks](#)
- [codebase for non-default class loader classes](#)
- collection member variables
 - [EJB QL](#)
- [collection types, Hibernate configuration](#)
- columns
 - [data types](#)
 - [Java objects, storing in database tables](#)
 - [null values in](#)
 - [values, retrieving](#)
- commands
 - [rmiid](#)
 - [rmiregistry](#)
- SQL
 - [data manipulation](#)
 - [DELETE](#)
 - [INSERT](#)
 - [schema manipulation commands](#)
 - [SELECT](#)
 - [UPDATE](#)

comments

[IDL](#)

[XML, ignoring with DOM parser](#)

[commit phase, distributed transactions](#)

commit()

[Connection interface](#)

[XASession class](#)

[committing transactions](#)

[auto-commit mode](#)

communication protocols

comparison operators

[EJB QL statements](#)

[SQL statements](#)

[IIOP](#)

[IMAP and POP-3](#)

[JRMP](#)

[POP and SMTP](#)

[SOAP](#)

[transport protocol layer, RMI](#)

[comparison operators](#)

[JMS message selectors](#)

[compiling code, Ant tasks](#)

[component model, EJB](#)

component modules

[assembly](#)

[deployment descriptors](#)

[EJB](#)

[jar files](#)

[web](#)

[component references, deployment descriptors and](#)

[components](#)

[deployment and](#)

[compound names](#)

[CompoundName class](#)

concurrency

[support information](#)

[transactions, optimistic](#)

conditional expressions

[arithmetic operators](#)

[EJB QL functions](#)

[functions, comparison operators](#)

[logical operators](#)

configuration

Hibernate

[class mapping](#)

[collection types](#)

[global configuration](#)

[polymorphic mappings](#)

[relationship mappings](#)

[types](#)

[Hibernate API](#)

[JNDI services](#)

- [naming conventions](#)
- [JSF](#)
- [Struts](#)
- [configuration information in database metadata](#)
- [connect\(\) \(Service\)](#)
- [Connection interface 2nd](#)
 - [createStatement\(\)](#)
 - [isolation modes, listed](#)
 - [JMS provider connections](#)
 - [setSavepoint\(\)](#)
 - [transactions \(JDBC\), managing](#)
- [connection pooling](#)
- [ConnectionFactory interface 2nd](#)
- [ConnectionPoolDataSource interface](#)
- [connections to JMS provider, closing at session end](#)
- [connections, database](#)
 - [pooling](#)
- [constants, IDL](#)
 - [InitialContext](#)
 - [mapping to Java](#)
- [constructors](#)
 - [Activatable class](#)
 - [activatable objects](#)
 - [ActivationDescr class](#)
 - [for remotely loaded classes](#)
 - [in server implementation](#)
 - [NameComponent class](#)
 - [reactivation](#)
 - [remote object, initializing, registering, activating](#)
 - [RMI implementation classes, throwing RemoteException](#)
- [containers](#)
 - [EJB session beans](#)
 - [EJBs 2nd](#)
 - [servlet containers](#)
- [content management systems](#)
- [ContentHandler interface](#)
- [Context interface 2nd](#)
 - [keys for JNDI property names](#)
 - [rebind\(\) method](#)
- [context values, IDL methods](#)
- [contexts](#)
 - [binding objects to](#)
 - [browsing a naming system](#)
 - [creating and destroying](#)
 - [JNDI](#)
 - [listing bindings of](#)
 - [listing children for](#)
 - [looking up objects](#)
 - [naming system providers](#)
- [cookies](#)
- [CORBA](#)
 - [adding objects to NamingContext](#)

- [architecture](#)
- [creating objects](#)
- [DII API](#)
- [distributed object services](#)
- [DSI API](#)
- [IDL](#)
- [IIOP 2nd](#)
- [IORs](#)
- [language independence](#)
- [Naming Service, registering server object with object references, narrowing](#)
- [ORBs](#)
- [remote classes, finding](#)
- [remote object references](#)
- [RMI objects, accessing from](#)
- [RMI with JNI vs.](#)
- [root object class](#)
- [CREATE TABLE statement \(SQL\)](#)
- [createAttribute\(\) \(Document\)](#)
- [createAttributeNS\(\) \(Document\)](#)
- [createConsumer\(\) \(Session\)](#)
- [createDurableSubscriber\(\) \(TopicSession\) 2nd](#)
- [createElement\(\) \(Document\)](#)
- [createElementNS\(\) \(Document\)](#)
- [createProducer\(\) \(Session\)](#)
- [createPublisher\(\) \(TopicSession\)](#)
- [createQueueBrowser\(\) \(QueueSession\)](#)
- [createQueueSession\(\) \(QueueConnection\)](#)
- [createReceiver\(\) \(QueueSession\) 2nd](#)
- [createRegistry\(\) \(LocateRegistry\)](#)
- [createSender\(\) \(QueueSession\)](#)
- [createStatement\(\) \(Connection\) 2nd 3rd](#)
- [createSubcontext\(\)](#)
 - [Context interface](#)
 - [DirContext interface](#)
- [createSubscriber\(\) \(TopicSession\) 2nd](#)
- [createTopicSession\(\) \(TopicConnection\)](#)
- [Crimson parser](#)
- [cross-database APIs](#)
- [cursor position \(result sets\)](#)
- customizing
 - [JMS message properties](#)
 - [URI resolution for XSL files](#)

[← PREV](#)

Index

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[J\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#)

[data grids, JSF tables](#)

[data manipulation statements, SQL](#)

[data sources for multipart email messages](#)

[data types](#)

[binary and character \(large objects\)](#)

[columns, returning \(ResultSet\)](#)

[custom message properties](#)

[DatabaseMetaData methods, returning](#)

[declaring](#)

[IDL](#)

[characters](#)

[strings](#)

[user-defined](#)

[in remote method arguments](#)

[Java handling](#)

[Java types corresponding to](#)

[Java/SQL mapping](#)

[large](#)

[object, Java handling of](#)

[primitive](#)

[relational databases and](#)

[serialization](#)

[SOAP messages and](#)

[SOAP, mapping to Java](#)

[SQL](#)

[stored procedures output](#)

[streams of in messages](#)

[WSDL, mapping to Java](#)

[DatabaseMetaData interface](#)

[scrolling and concurrency support](#)

[databases](#)

[accessing \(result sets\)](#)

[changes, losing](#)

[commands, database-independent \(JDBC\)](#)

[configuration information, obtaining](#)

[connections](#)

[creating](#)

[differences among systems](#)

[editing data, inserting and deleting rows \(ResultSet\)](#)

[insertions, preventing \(transactions\)](#)

[interacting with JDBC](#)

- [Java-aware](#)
- [JDBC drivers](#)
- [null values, handling](#)
- [null values, inserting](#)
- [objects \(Java types\), inserting](#)
- [relational](#)
 - [SQL and](#)
 - [scalar functions](#)
 - [SQL strings, processing](#)
 - [stored procedures 2nd](#)
 - [tables, displaying contents of](#)
 - [transactions](#)
 - [update counts, getting](#)
 - [updateable ResultSet, differences](#)
 - [web applications](#)
- [DataSource interface](#)
- [date/time functions, SQL](#)
- [dates and times, JDBC classes for](#)
- [DCE-CIOP \(DCE Common Inter-ORB Protocol\)](#)
- [debugging database applications, using warnings for](#)
- declarations
 - [IDL interfaces](#)
 - [web components, deployment descriptors and](#)
 - [XML](#)
- [declarative transactions](#)
- [DefaultHandler class](#)
- defining
 - [activatable remote objects 2nd](#)
 - [key RMI classes for implementations](#)
 - [remote RMI objects](#)
 - [writing class to implement remote interface](#)
- [delegates \(CORBA remote object stubs\)](#)
- [DELETE statement \(SQL\)](#)
- [deleteRow\(\) \(ResultSet\)](#)
- deleting
 - [message from IMAP store](#)
 - [rows from ResultSet](#)
- deployment
 - [applications, Ant](#)
 - [component references](#)
 - [components \(Ant\)](#)
 - [EJBs](#)
 - [entity beans](#)
 - [security-related attributes](#)
 - [introduction](#)
 - [J2EE](#)
 - [container-specific classes](#)
 - [descriptors](#)
 - [server-specific configuration](#)
 - [JNDI configuration services](#)
 - [message-driven EJBs](#)
 - [physical](#)

- [resource references](#)
- [runtime classloading models](#)
- [web services](#)
 - [Axis model](#)
 - [J2EE](#)
 - [Java implementation code](#)
- [deployment descriptors 2nd](#)
 - [application archives](#)
 - [component modules](#)
 - [EJBs 2nd](#)
 - [web components](#)
 - [web services](#)
 - [WSDL mappings](#)
 - [XDoclet](#)
 - [XML](#)
- [Destination interface](#)
- [destinations for messages](#)
 - [addresses](#)
 - [temporary, clients creating](#)
- [destroySubcontext\(\) \(Context\)](#)
- [development tools](#)
 - [Ant](#)
 - [Cactus](#)
 - [Hibernate](#)
 - [Java Annotation](#)
 - [JUnit](#)
 - [Struts](#)
 - [XDoclet 2nd](#)
- [development, Struts and](#)
- [DII \(Dynamic Invocation Interface\) 2nd 3rd](#)
 - [web service clients 2nd](#)
- [DirContext interface](#)
 - [createSubcontext\(\)](#)
- [<%@ > directives](#)
- [directives in JSP](#)
- [directory services](#)
 - [accessing](#)
 - [creating directory entries](#)
 - [modifying directory entries](#)
 - [search command](#)
 - [search controls](#)
 - [search criteria](#)
 - [search results](#)
 - [searching a directory](#)
- [directory tasks \(Ant\)](#)
- [directory trees, servlet context](#)
- [dirty reads \(transactions\)](#)
- [DISTINCT option, SELECT clause](#)
- [distributed applications](#)
 - [remote interfaces, defining](#)
 - [with Java and non-Java components](#)
- [distributed component model](#)

[distributed computing](#)

[EJB component model](#)

[garbage collection](#)

[Java RMI, garbage collection](#)

[transactions](#)

[distributed objects](#)

[clients and servers, distinguishing](#)

[CORBA standard services](#)

[RMI, interacting with](#)

[distributed transactions](#)

[JMS destination](#)

[scenarios](#)

[two-phase commit](#)

[Document class](#)

[Document Type Definitions](#)

[DocumentBuilder interface 2nd](#)

[DocumentBuilderFactory class 2nd](#)

[DocumentCondenser class \(DOM example\)](#)

[DocumentFragment class](#)

[DOM \(Document Object Model\) 2nd](#)

[element attributes](#)

[extending trees](#)

[getting a parser](#)

[manipulating DOM trees](#)

[navigating DOM tree](#)

[DOMImplementation interface](#)

[DOMParser class](#)

[DOMResult class](#)

[DOMSource class](#)

[DriverManager class](#)

[drivers, JDBC](#)

[currently available, listing of](#)

[escape sequences, translating](#)

[JDBC-ODBC Bridge 2nd](#)

[loading \(example\)](#)

[null values](#)

[registering with DriverManager](#)

[results sets \(scrollable and updateable\)](#)

[support levels, accommodating](#)

[types, listing of](#)

[DROP statement \(SQL\)](#)

[DSI \(Dynamic Skeleton Interface\) API 2nd](#)

[DTDs \(Document Type Definitions\)](#)

[orders.xml document \(example\)](#)

[validating XML document against with SAX parser](#)

[DynaActionForms, Struts](#)

[dynamic class loading \(RMI\)](#)

[configuring clients and servers for](#)

[loading classes from applets](#)

[Dynamic Invocation Interface \(DII\) API 2nd 3rd](#)

[Dynamic Method Interface \(CORBA\)](#)

[dynamic proxy, web services clients 2nd](#)

[Dynamic Skeleton Interface \(DSI\) API 2nd
dynamic web pages](#)



[← PREV](#)

Index

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[J\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#)

[e-commerce](#)

[ear files](#)

[ear task](#)

[EJB \(Enterprise JavaBeans\)](#)

[bean implementation](#)

[beans](#)

[creating](#)

[finding](#)

[client stub interfaces](#)

[code artifacts](#)

[component model](#)

[overview](#)

[component modules](#)

[components 2nd](#)

[container callbacks](#)

[message-driven beans](#)

[containers](#)

[deployment](#)

[security-related attributes](#)

[deployment descriptors 2nd 3rd](#)

[distributed components](#)

[distributed computing and](#)

[enterprise computing scenarios](#)

[e-commerce for mail-order business, enabling](#)

[transactional integrity for e-commerce business](#)

[entity beans 2nd](#)

[abstract persistence schema](#)

[BMP](#)

[CMP](#)

[container callbacks](#)

[deployment](#)

[finder methods](#)

[implementation](#)

[lifecycle](#)

[primary keys](#)

[select methods](#)

[handles](#)

[home interfaces](#)

[identities](#)

[deployment](#)

[interfaces](#)

- [client interfaces](#)
- [home interfaces 2nd](#)
- [JTA \(Java Transaction API\)](#)
- [local clients](#)
- [message-driven beans 2nd](#)
 - [container callbacks](#)
 - [deployment](#)
- [object implementation](#)
- [packages](#)
- [remote clients](#)
- [session beans](#)
 - [container management](#)
 - [stateful](#)
 - [stateless](#)
 - [transaction support](#)
- [testing, Cactus](#)
- [transactions](#)
 - [BMTD](#)
 - [CMTD 2nd](#)
 - [session facade](#)
- [tutorial](#)
- [version 3.0](#)
- [version covered](#)
- [web services](#)
 - [deployment](#)
- [EJB API](#)
- [EJB client](#)
- [EJB object](#)
- [EJB QL](#)
 - [finder and select methods](#)
 - [FROM clause](#)
 - [collection member variables](#)
 - [range variables](#)
 - [functions](#)
 - [conditional expressions](#)
 - [ORDER BY clause](#)
 - [query structure](#)
 - [SELECT clause](#)
 - [DISTINCT option](#)
 - [WHERE clause](#)
 - [literals](#)
 - [variables](#)
- [ejb-jar files](#)
 - [Ant buildfiles](#)
 - [creating from bean classes](#)
- [ejb-jar.xml, deployment descriptors](#)
- [EJB-tier authentication](#)
- [EJB-tier authorization](#)
- [element nodes](#)
- [elements, XML](#)
 - [attributes \(DOM\)](#)
 - [parsing with SAX 2nd](#)

email

[basics of](#)

[JavaMail and](#)

[empty elements \(XML\)](#)

[encodings, web services](#)

[endDocument\(\)](#)

[enterprise computing](#)

[mail-order enterprise](#)

[scenarios](#)

[enterprise tasks, Ant](#)

[entity beans \(EJBs\) 2nd](#)

[abstract persistence schema](#)

[BMP](#)

[CMP](#)

[container callbacks](#)

[deployment](#)

[finder methods](#)

[implementation](#)

[lifecycle](#)

[primary keys](#)

[select methods](#)

[entity reference nodes \(XML\), expanding \(DOM parser\)](#)

[enumerations, IDL](#)

[environment entries, deployment descriptors and](#)

[equals\(\) \(RemoteObject\)](#)

error handling

[naming services 2nd](#)

servlet responses

[status codes](#)

[servlet responses and](#)

[error\(\) \(ErrorHandler\)](#)

[ErrorHandler interface](#)

errors

[binding remote objects to names](#)

[in remote sessions](#)

[JDBC, handling](#)

[marshaling exception](#)

[remote object URLs](#)

[escape sequences](#)

[components \(JDBC\)](#)

[dates \(example\)](#)

[dates \(ISO\)](#)

[stored procedures, accessing \(JDBC\)](#)

[event notification, JNDI](#)

[listen command](#)

[registering event listeners](#)

[sources of events](#)

[writing event listeners](#)

[EventContext interface](#)

[addNamingListener\(\)](#)

[EventDirContext interface](#)

[addNamingListener\(\)](#)

[filters specifying target of listener](#)

events

[asynchronous, JMS](#)

[Hibernate API](#)

[JSF user interface](#)

[synchronicity in SAX parsing cycle](#)

exceptions

[IDL 2nd](#)

[standard](#)

[IDL methods](#)

[JNDI](#)

[name, object not found at specified](#)

[object references, narrowing \(CORBA\)](#)

[objects, binding to NamingContext](#)

[servlets](#)

[execute\(\) \(Statement\)](#)

executeBatch()

[PreparedStatement interface](#)

[Statement interface](#)

[executeQuery\(\) \(Statement\)](#)

executeUpdate()

[PreparedStatement interface](#)

[Statement interface](#)

[expirable leases on remote objects](#)

[exportObject\(\) \(Activatable\)](#)

[using instead of initialization constructors](#)

[expression tag](#)

[<%- %> expression tag](#)

[expression tags](#)

expressions

[message selectors \(JMS\)](#)

[nesting in searches](#)

[expunge\(\) \(Folder\)](#)

[← PREV](#)

Index

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[J\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#)

factory classes

[custom, client and server socket](#)

[defining \(example\)](#)

[for remote object references](#)

[for XML parsers](#)

[factory objects](#)

[failed tests \(JUnit\)](#)

[fatalError\(\) \(ErrorHandler\)](#)

[file tasks \(Ant\)](#)

[FileDataSource class](#)

[filesystem factory class \(JNDI\)](#)

filtering

[messages](#)

[QueueBrowser, using](#)

filters

[listener targets for naming services](#)

[request dispatching](#)

[search filter strings](#)

[servlets](#)

[fixed-point literals, IDL](#)

[flat transaction model \(FTM\)](#)

floating-point literals

[IDL](#)

[flow control, JSTL](#)

[folders, message](#)

forms

[servlet requests](#)

[Struts tags](#)

[validation, JSF](#)

[forName\(\) \(Class\)](#)

[forward-only result sets](#)

[fragments of documents \(Document fragment nodes\)](#)

[framework, Struts as](#)

FROM clause

[EJB QL](#)

[collection member variables](#)

[range variables](#)

[HQL](#)

[FTM \(flat transaction model\)](#)

functions

[C/C++, invoking with Java methods](#)

- conditional expressions
 - [comparison operators](#)
- [EJB QL](#)
 - [conditional expressions](#)
- [scalar \(databases\)](#)
- SQL
 - [aggregate](#)
 - [value functions](#)



Index

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[J\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#)

garbage collection

[distributed, for RMI objects 2nd](#)

[RMI](#)

[generic property accessors](#)

[get\(\) \(Attributes\)](#)

[get/setProperty\(\) methods, Message interface](#)

[getAsciiStream\(\) \(ResultSet\)](#)

[getAttributes\(\) \(Node\) 2nd](#)

[getAutoCommit\(\) \(Connection\)](#)

[getBinaryStream\(\)](#)

[Blob class](#)

[ResultSet interface](#)

[getBytes\(\) \(Blob\)](#)

[getCharacterStream\(\) \(Clob\)](#)

[getClassName\(\) \(NameClassPair\)](#)

[getColumnCount\(\) \(ResultSetMetaData\)](#)

[getColumnLabel\(\) \(ResultSetMetaData\)](#)

[getColumns\(\) \(ResultSetMetaData\)](#)

[getColumnTypeName\(\) \(ResultSetMetaData\)](#)

[getConnection\(\) \(DriverManager\)](#)

[getDefaultFolder\(\) \(Store\)](#)

[getDefaultInstance\(\) \(Session\)](#)

[getDefaultTransactionIsolation\(\) \(DatabaseMetaData\)](#)

[getDOMImplementation\(\)](#)

[getErrorCode\(\) \(SQLException\)](#)

[getFeature\(\) \(TransformerFactory\)](#)

[getFirstChild\(\) \(Node\)](#)

[getFlags\(\) \(Message\)](#)

[getFloat\(\) \(CallableStatement\)](#)

[getFolder\(\) \(Session\) 2nd](#)

[getInstance\(\) \(Session\)](#)

[getMaxConnections\(\) \(DatabaseMetaData\)](#)

[getMessageNumber\(\) \(Message\)](#)

[getMetaData\(\) \(Connection\)](#)

[getMoreResults\(\) \(Statement\)](#)

[getName\(\) \(NameClassPair\)](#)

[getNewBinding\(\) \(NamingEvent\)](#)

[getNextException\(\) \(SQLException\)](#)

[getNextSibling\(\) \(Node\)](#)

[getNodeTypes\(\) \(Node\)](#)

[getNodeValue\(\) \(Node\)](#)

[getObject\(\)](#)
 [Binding class](#)
 [ResultSet interface](#)
[getOldBinding\(\) \(NamingEvent\)](#)
[getPasswordAuthentication\(\) \(Authenticator\)](#)
[getPrivileges\(\) \(ResultSetMetaData\)](#)
[getProviders\(\) \(Session\)](#)
[getRegistry\(\) \(LocateRegistry\)](#)
[getRequestingProtocol\(\) \(Authenticator\)](#)
[getRequestingSite\(\) \(Authenticator\)](#)
[getResultSet\(\) \(Statement\)](#)
[getSQLState\(\) \(SQLException\)](#)
[getStore\(\)](#)
 [Session class](#)
 [Store class](#)
[getString\(\) \(ResultSet\)](#)
[getSubString\(\) \(Clob\)](#)
[getTableTypes\(\) \(ResultSetMetaData\)](#)
[getTransport\(\) \(Session\)](#)
[getUnicodeStream\(\) \(ResultSet\)](#)
[getUpdateCount\(\) \(Statement\)](#)
[getURL\(\) \(DatabaseMetaData\)](#)
[getURLName\(\) \(Service\)](#)
[getWarnings\(\)](#)
 [CallableStatement interface](#)
 [Connection interface](#)
 [PreparedStatement interface](#)
 [ResultSet interface](#)
 [Statement interface](#)
[getXMLReader\(\) \(SAXParser\)](#)
[global configuration, Hibernate](#)
[groups, SELECT statement \(SQL\)](#)

[← PREV](#)

Index

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[J\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#)

handlers

[EJBs](#)

handlers, SAX

[ContentHandler interface](#)

[DefaultHandler class](#)

[ErrorHandler interface](#)

hardware (networks), server applications and

[hashCode\(\) \(RemoteObject\)](#)

[HEAD requests](#)

[header files \(JNI C/C++\), generating with javah](#)

headers

[IDL interfaces](#)

[JMS messages 2nd](#)

helper classes

[helper classes, IDL/Java conversion](#)

Hibernate

[caching](#)

configuration

[class mappings](#)

[collection types](#)

[global configuration](#)

[polymorphic mappings](#)

[relationship mappings](#)

[types](#)

[HQL \(Hibernate Query Language\)](#)

[introduction](#)

[Java and](#)

[JDBC and](#)

[performance](#)

[pluggability](#)

[RDBMS and](#)

[reflection](#)

[sample application](#)

[security](#)

[transactions](#)

[transparency](#)

Hibernate API

[configuration and](#)

[events](#)

[interceptors](#)

[Query class](#)

[Session object](#)

[SessionFactory](#)

[hierarchical property files \(Ant\)](#)

[holder classes, IDL](#)

[IDL/Java conversion](#)

[objects, passing as inout](#)

[home interfaces \(EJBs\) 2nd](#)

[JNDI and](#)

hosts

[ORBs, specifying](#)

[RMI registry daemon, running](#)

HTML

[converting XML file into with XSL stylesheet](#)

[document condenser \(DOM\)](#)

[HTML tags, JSF](#)

[HTTP requests](#)

[HTTP responses](#)

[HTTP servlets](#)

[HttpServletRequest interface](#)

[HttpServletResponse interface](#)

[HttpSessionBindingListener](#)

[hyperlinks, JSPs](#)

[← PREV](#)

Index

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[J\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#)

[id field \(NameComponent\)](#)

identifiers

[IDL](#)

[mapping to Java](#)

[JMS clients](#)

[IDL \(Interface Definition Language\) 2nd 3rd](#)

[comments](#)

[constants](#)

[mapping to Java](#)

[CORBA-enabled interface, defining](#)

[data types](#)

[characters](#)

[strings](#)

[user-defined](#)

[declaration language](#)

[default ORB implementation](#)

[describing in CORBA](#)

[example, complete](#)

[exceptions](#)

[standard](#)

[identifiers](#)

[mapping to Java](#)

[initial object references, storing as root objects in Naming Service](#)

[interface declarations](#)

[interface inheritance](#)

interfaces

[attributes](#)

[declarations](#)

[mapping to Java](#)

[methods](#)

[Java RMI interface, mapping to](#)

[Java stubs to CORBA objects](#)

[Java, differences from](#)

[Java-to-IDL-mapping](#)

[keywords](#)

[literals](#)

[Boolean](#)

[character literals](#)

[numeric](#)

[string literals](#)

methods

- [arguments](#)
- [call semantics](#)
- [context values](#)
- [exceptions](#)
- [modules](#)
 - [declarations](#)
- [naming scopes](#)
- [Naming Service 2nd](#)
- [properties \(nonstandard\)](#)
- [properties \(standard\)](#)
- [remote method using inout parameter](#)
- [RMI vs.](#)
- [sequences](#)
- [tools](#)
 - [naming service daemon](#)
 - [orbd](#)
 - [servertool](#)
 - [tnameserv](#)
 - [Transient Naming Service Daemon](#)
- [unions](#)
- [value types](#)
- [IDL compiler](#)
- [IDL tools](#)
 - [IDL compiler](#)
 - [idlj](#)
- [IDL-to-Java compiler](#)
 - [stub and skeleton classes, generating](#)
- [ignoreableWhitespace\(\)](#)
- [IIOP \(Internet Inter ORB Protocol\)](#)
- [IMAP protocol](#)
 - [URLs](#)
- [implementation](#)
 - [beans \(EJBs\)](#)
 - [EJB objects](#)
 - [web services](#)
- [importNode\(\) \(Document\)](#)
- [inactiveGroup\(\) \(ActivationGroup\)](#)
- [inactiveObject\(\) \(ActivationGroup\)](#)
- [incoming messages, handling](#)
- [inheritance, interface inheritance, IDL](#)
- [init\(\)](#)
 - [ORB class](#)
 - [PTPMessagingClient \(example\)](#)
 - [servlet initialization and](#)
- [initial context, JNDI](#)
 - [command for loading](#)
 - [NamingShell application, establishing for](#)
- [initial references, listing \(ORB\)](#)
- [InitialContext interface](#)
 - [obtaining from JNDI service of JMS provider](#)
- [initialization constructors](#)
- [initialization methods \(ORBs\)](#)

[initialization, servlet contexts](#)

[inner classes](#)

[input parameter variables](#)

[input/output, from XSLT transformations](#)

[INSERT statement \(SQL\)](#)

[insertBefore\(\)](#)

[insertRow\(\) \(ResultSet\)](#)

[integration testing, unit testing and](#)

[interceptors, Hibernate API](#)

[interface inheritance, IDL](#)

interfaces

[client \(EJBs\)](#)

[client stub interfaces \(EJBs\)](#)

[CORBA IDL](#)

[DOM, mapping to node types](#)

[home interfaces \(EJBs\) 2nd](#)

[JNDI and](#)

[HttpServletRequest](#)

[IDL](#)

[attributes](#)

[declarations](#)

[mapping to Java](#)

[methods](#)

[JMS](#)

[JSF events](#)

[remote object, unknown](#)

[ServletContext](#)

[ServletRequest](#)

[internationalization, JSTL and](#)

[Internet Inter-ORB Protocol \(IIOP\)](#)

[InternetAddress class](#)

[Interoperable Object References 2nd](#)

[IORs](#)

[isNull\(\) \(ResultSet\)](#)

[ISO date escape sequences \(JDBC\)](#)

[isolation levels, transactions 2nd](#)

[JDBC support, listed](#)

isolation, transactions

[levels](#)

[violations](#)

[← PREV](#)

Index

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[J\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#)

[J2EE \(Java 2 Enterprise Edition\)](#)

[deployment](#)

[container-specific classes](#)

[descriptors](#)

[server-specific configuration](#)

[security](#)

[limitations](#)

[overview](#)

[transport layer security](#)

[XML Encryption](#)

[web services deployment](#)

[J2SE \(Java 2 Standard Edition\)](#)

[annotations tutorial](#)

[JAAS \(Java Authentication and Authorization Service\)](#)

[authentication and](#)

[JACC \(Java Authorization Contract for Containers\)](#)

[JAF \(JavaBeans Activation Framework\)](#)

[JavaMail, using with](#)

[mail messages, assigning content and MIME type](#)

[JAR \(Java Archive\) files](#)

[activation.jar \(JAF\)](#)

[application, finding XML parser](#)

[mail.jar \(JavaMail\)](#)

[Java](#)

[and XML](#)

[DOM](#)

[IDL, differences \(list\)](#)

[JAXP](#)

[SAX](#)

[using XML documents](#)

[Java annotations](#)

[overview](#)

[tools](#)

[Java API for XML Registries \(JAXR\)](#)

[Java beans, scope](#)

[Java classes, IDL interfaces vs.](#)

[Java IDL](#)

[overview](#)

[Java objects](#)

[Java packages \(IDL modules vs.\)](#)

[Java Remote Method Protocol \(JRMP\)](#)

[Java Runtime Environment \(JRE\)](#)

[Java Servlet API](#)

Java VM

[activation daemon](#)

[activation groups](#)

[codebase property for nondefault class loaders](#)

[libraries \(native functions\), loading](#)

[running](#)

[Java-aware databases](#)

[java.rmi package](#)

[java.rmi.activation package](#)

[java.rmi.server package](#)

[java.rmi.server.codebase property](#)

[java.sql package](#)

JavaBeans

[JSP action tags](#)

[RowSet interface](#)

[javah tool, generating JNI C/C++ header files](#)

JavaMail 2nd

[authenticators, username and password](#)

[creating messages](#)

[displaying](#)

[email and](#)

[incoming messages, handling](#)

[installing and configuring](#)

[layers](#)

[message stores](#)

[multipart messages](#)

[provider registries](#)

[providers of messaging services](#)

[retrieving messages](#)

[searches and message management](#)

[sending messages 2nd](#)

[sessions](#)

[URL names](#)

[javax.servlet.GenericServlet class](#)

[javax.servlet.http.HttpServlet class](#)

[javax.servlet.Servlet interface](#)

[JAX-RPC \(Java API for XML Remote Procedure Calls\) 2nd 3rd 4th](#)

[introduction](#)

[SOAP and](#)

[Web service clients](#)

[writing web services](#)

[WSDL and](#)

[JAXP \(Java API for XML Processing\) 2nd 3rd](#)

[custom URI resolution](#)

[data sources](#)

[determining data source support](#)

[DOM](#)

[generating SAX parsers](#)

[getting a DOM parser](#)

[parser objects and XSLT processors](#)

[parser or processor, getting](#)

[transformation classes](#)

[XSLT](#)

[JAXR \(Java API for XML Registries\)](#)

[JCE \(Java Cryptography Extension\)](#)

[JDBC \(Java Database Connectivity\)](#)

[architecture](#)

[basic functionalities \(example\)](#)

[batch updates](#)

[BLOB and CLOB objects](#)

[commands, database-independent](#)

[currently available drivers, listing of](#)

[data types, mapping to primitive Java types](#)

[date and time classes](#)

[drivers 2nd](#)

[escape syntax, stored procedures](#)

[Hibernate and](#)

[JDBC-ODBC Bridge 2nd](#)

[metadata](#)

[platform independence](#)

[registering](#)

[ResultSet \(Version 2.0 vs. 1.0\)](#)

[savepoints](#)

[SQL statements](#)

[transactions](#)

[types of](#)

[URLs](#)

[JDBC-ODBC Bridge Drivers](#)

[URLs](#)

[JDBMS systems, storing Java objects](#)

[JDK, RMI registry daemon](#)

[JMS \(Java Message Service\) 2nd](#)

[accessing content of](#)

[browsing queues](#)

[client \(example\)](#)

[clients](#)

[durable subscriptions](#)

[elements of](#)

[filtering](#)

[header fields and properties](#)

[key interfaces](#)

[message selectors](#)

[expressions](#)

[identifiers](#)

[literals](#)

[operators](#)

[structure](#)

[messages](#)

[point-to-point and publish-subscribe styles](#)

[point-to-point messaging](#)

[publish-subscribe messaging](#)

[transactional messaging](#)

- [types of](#)
- [JMSReplyTo header](#)
- [JMX \(Java Management Extensions\)](#)
- [JNDI \(Java Naming and Directory Interface\) 2nd](#)
 - [API and SPI \(service provider interface\)](#)
 - [architecture](#)
 - [binding objects to](#)
 - [browsing a naming system](#)
 - [command for loading initial context](#)
 - [configuration services](#)
 - [naming conventions](#)
 - [contexts](#)
 - [CORBA Naming Service, registering remote object with](#)
 - [creating and destroying contexts](#)
 - [directories, searching](#)
 - [directory entries, creating](#)
 - [directory entries, modifying](#)
 - [directory services, accessing](#)
 - [EJBs, home interfaces](#)
 - [event notification](#)
 - [InitialContext class](#)
 - [introduction](#)
 - [JMS provider, services for 2nd](#)
 - [listing bindings of](#)
 - [listing children of](#)
 - [looking up object in naming system \(example\)](#)
 - [looking up objects](#)
 - [naming system service providers](#)
 - [NamingShell application](#)
 - [registering event listeners](#)
 - [remote objects, binding to CORBA Naming Service](#)
 - [running the shell](#)
 - [search command](#)
 - [search controls](#)
 - [search criteria](#)
 - [search results](#)
 - [sources of events](#)
 - [writing event listeners](#)
- [JNI \(Java Native Interface\)](#)
 - [using with RMI vs. using CORBA](#)
 - [wrapping C/C++ code with](#)
- [joins, SELECT statement \(SQL\)](#)
- [JRE \(Java Runtime Environment\)](#)
- [JRMP \(Java Remote Method Protocol\)](#)
- [JSF \(JavaServer Faces\) 2nd](#)
 - [actions](#)
 - [dynamic](#)
 - [applications](#)
 - [sample](#)
 - [structure](#)
 - [configuration file](#)
 - [forms, validation](#)

[introduction](#)

[lifecycle](#)

managed beans

[action methods](#)

[backing beans](#)

[model beans](#)

[managed beans and](#)

[navigation rules](#)

[dynamic actions and](#)

[sample application](#)

tables

[data grids](#)

[editing](#)

[panel grids](#)

[user interface, events](#)

[validation](#)

[custom validators](#)

[forms](#)

[messages](#)

[views](#)

[web UI components](#)

[<f:actionListener> JSF core tag](#)

[<f:convertNumber> JSF core tag](#)

[<f:convertDateTime> JSF core tag](#)

[<f:attribute> JSF core tag](#)

[<f:loadBundle> JSF core tag](#)

[<f:facet> JSF core tag](#)

[<f:converter> JSF core tag](#)

[<f:selectItems> JSF core tag](#)

[<f:selectItem> JSF core tag](#)

[<f:param> JSF core tag](#)

[<f:validator> JSF core tag](#)

[<f:validateLength> JSF core tag](#)

[<f:validateDoubleRange> JSF core tag](#)

[<f:subview> JSF core tag](#)

[<f:view> JSF core tag](#)

[<f:verbatim> JSF core tag](#)

[<f:valueChangeListener> JSF core tag](#)

[<f:validateLongRange> JSF core tag](#)

[JSF core tags](#)

[<f:actionListener>](#)

[<f:attribute>](#)

[<f:convertDateTime>](#)

[<f:converter>](#)

[<f:convertNumber>](#)

[<f:facet>](#)

[<f:loadBundle>](#)

[<f:param>](#)

[<f:selectItem>](#)

[<f:selectItems>](#)

[<f:subview>](#)

[<f:validateDoubleRange>](#)

[<f:validateLength>](#)
[<f:validateLongRange>](#)
[<f:validator>](#)
[<f:valueChangeListener>](#)
[<f:verbatim>](#)
[<f:view>](#)

JSF EL

[<h:dataTable>](#) JSF HTML tag
[<h:commandLink>](#) JSF HTML tag
[<h:commandButton>](#) JSF HTML tag
[<h:column>](#) JSF HTML tag
[<h:inputHidden>](#) JSF HTML tag
[<h:graphicImage>](#) JSF HTML tag
[<h:form>](#) JSF HTML tag
[<h:message>](#) JSF HTML tag
[<h:inputTextarea>](#) JSF HTML tag
[<h:inputText>](#) JSF HTML tag
[<h:inputSecret>](#) JSF HTML tag
[<h:outputLabel>](#) JSF HTML tag
[<h:outputFormat>](#) JSF HTML tag
[<h:messages>](#) JSF HTML tag
[<h:panelGrid>](#) JSF HTML tag
[<h:outputText>](#) JSF HTML tag
[<h:outputLink>](#) JSF HTML tag
[<h:selectManyMenu>](#) JSF HTML tag
[<h:selectManyListbox>](#) JSF HTML tag
[<h:selectManyCheckbox>](#) JSF HTML tag
[<h:selectBooleanCheckbox>](#) JSF HTML tag
[<h:panelGroup>](#) JSF HTML tag
[<h:selectOneRadio>](#) JSF HTML tag
[<h:selectOneMenu>](#) JSF HTML tag
[<h:selectOneListbox>](#) JSF HTML tag

JSF HTML tags

[<h:column>](#)
[<h:commandButton>](#)
[<h:commandLink>](#)
[<h:dataTable>](#)
[<h:form>](#)
[<h:graphicImage>](#)
[<h:inputHidden>](#)
[<h:inputSecret>](#)
[<h:inputText>](#)
[<h:inputTextarea>](#)
[<h:message>](#)
[<h:messages>](#)
[<h:outputFormat>](#)
[<h:outputLabel>](#)
[<h:outputLink>](#)
[<h:outputText>](#)
[<h:panelGrid>](#)
[<h:panelGroup>](#)
[<h:selectBooleanCheckbox>](#)

[<h:selectManyCheckbox>](#)

[<h:selectManyListbox>](#)

[<h:selectManyMenu>](#)

[<h:selectOneListbox>](#)

[<h:selectOneMenu>](#)

[<h:selectOneRadio>](#)

[pass-through attributes](#)

[JSF tag library](#)

[JSP API](#)

[JSPs \(JavaServer Pages\)](#)

[action tags](#)

[JavaBeans and](#)

[ActionForm class, Struts](#)

[custom tags, testing](#)

[directives](#)

[dynamic web pages](#)

[EL](#)

[expression tags](#)

[granularity of](#)

[hyperlinks](#)

[introduction](#)

[Java beans, scope](#)

[objects, built-in](#)

[overview](#)

[tags](#)

[communications](#)

[custom](#)

[EL and](#)

[libraries](#)

[whitespace and](#)

[testing, Cactus](#)

[JSTL \(JSP Standard Tag Library\)](#)

[internationalization and](#)

[URLs and](#)

[JSTL EL, Struts tags](#)

[<c:when> JSTL tag](#)

[<c:set> JSTL tag](#)

[<c:remove> JSTL tag](#)

[<c:out>JSTL tag](#)

[<c:if> JSTL tag](#)

[<c:choose> JSTL tag](#)

[<c:otherwise> JSTL tag](#)

[<c:forEach> JSTL tag](#)

[<c:url> JSTL tag](#)

[<fmt:message> JSTL tag](#)

[<fmt:bundle> JSTL tag](#)

[<fmt:setBundle> JSTL tag](#)

[JSTL tags](#)

[<c:choose>](#)

[<c:forEach>](#)

[<c:if>](#)

[<c:otherwise>](#)

[<c:out>](#)

[<c:remove>](#)

[<c:set>](#)

[<c:url>](#)

[<c:when>](#)

[for internationalization](#)

[<fmt:bundle>](#)

[<fmt:message>](#)

[<fmt:setBundle>](#)

[JTA \(Java Transaction API\) 2nd 3rd](#)

[transactions, distributed](#)

[JUnit](#)

[Ant and](#)

[batch testing](#)

[build control](#)

[test result format](#)

[assertion methods](#)

[Cactus](#)

[failed tests](#)

[introduction](#)

[overview](#)

[runtime model](#)

[test fixtures](#)

[TestCases](#)

[TestSuites](#)

[← PREV](#)

Index

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[J\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#)

keywords

[_escape sequences \(JDBC\)](#)

[IDL](#)

[kind field \(NameComponent\)](#)

[← PREV](#)

[← PREV](#)

Index

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[J\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#)

language independence

[CORBA](#)

[inter-ORB communications](#)

[large data types](#)

[lazy loading, Hibernate relationships](#)

[LDAP \(Lightweight Directory Access Protocol\)](#)

[creating InitialContext with factory class](#)

[directory servers, prohibiting new entries in](#)

[search filter strings](#)

[leaf, CORBA naming directory](#)

[leases, expirable \(on remote objects\)](#)

libraries

[native code, wrapping with Java](#)

[native functions, loading](#)

[tag libraries, JSP](#)

[lifecycle of servlet](#)

[LIKE statement](#)

[list\(\) \(Context\)](#)

[list\(\) \(Registry\)](#)

[list_initial_references\(\) \(ORB\)](#)

[listBindings\(\) \(Context\)](#)

literals

[IDL](#)

[Boolean](#)

[character literals](#)

[numeric](#)

[string literals](#)

[JMS message selectors](#)

[WHERE clause, EJB QL](#)

[loadLibrary\(\) \(System\)](#)

[local clients, EJBs](#)

[local names of XML elements](#)

[LocateRegistry interface](#)

[remote objects, looking up](#)

[locatorsUpdateCopy\(\) \(DatabaseMetaData\)](#)

logical operators

[conditional expressions](#)

[JMS message selectors](#)

[login information, associating with URLName objects](#)

[logins, database](#)

[looking up objects in naming systems](#)

[JNDI context](#)
[lookup\(.\) \(Naming\) 2nd](#)



[← PREV](#)

Index

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[J\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#)

[mail-order enterprise](#)

[main\(\)](#)

[AccountReg class \(example\)](#)

[PTPMessagingClient \(example\)](#)

[MalformedURLException class](#)

[managed beans](#)

[JSF](#)

[action methods](#)

[backing beans](#)

[model beans](#)

[scope](#)

[many-to-many relationships](#)

[many-to-one relationships](#)

[MapMessage class](#)

[mapping](#)

[constants to Java \(IDL\)](#)

[IDL identifiers to Java](#)

[IDL interfaces to Java](#)

[IDL to Java](#)

[object-oriented languages to CORBA specifications](#)

[requests, servlet contexts](#)

[MarshalException class 2nd](#)

[marshaling](#)

[encoding URL in marshaled bytestream](#)

[unmarshaling method arguments](#)

[MarshaledObject class 2nd](#)

[match\(\) \(SearchTerm\)](#)

[message destinations](#)

[Message interface 2nd 3rd](#)

[subclasses with different message bodies](#)

[message selectors \(JMS\)](#)

[expressions](#)

[identifiers](#)

[literals](#)

[operators](#)

[arithmetic](#)

[comparison](#)

[logical](#)

[structure](#)

[message validation, JSF](#)

[message-driven beans \(EJBs\) 2nd](#)

- [_container callbacks](#)
- [_deployment](#)
- [message-oriented middleware \(MOM\)](#)
- [MessageConsumer interface 2nd 3rd 4th](#)
- [MessageListener interface](#)
 - [_implementation \(example\)](#)
- [MessageProducer interface 2nd](#)
- [messages, JMS](#)
 - [_accessing content of](#)
 - [_filtering](#)
 - [_header fields and properties](#)
- [messaging](#)
 - [_JavaMail](#)
 - [_JMS](#)
- [messaging clients](#)
- [messaging provider](#)
- [metadata, JDBC](#)
- [method permissions, EJB deployment](#)
- methods
 - IDL
 - [_arguments](#)
 - [_call semantics](#)
 - [_context values](#)
 - [_exceptions](#)
 - [_IDL interfaces 2nd](#)
 - [_parameter modifiers](#)
 - [_return values or exceptions](#)
 - [_servlet contexts](#)
- Microsoft
 - [_ODBC 2nd](#)
 - [_Windows](#)
- MIME types
 - [_JavaMail support for](#)
 - [_multipart messages, creating](#)
- [MimeMessage class](#)
- [model beans, JSF](#)
- [modifiers \(IDL methods\)](#)
- [modifyAttributes\(\) \(DirContext\)](#)
- modules
 - [_application assembly](#)
 - [_IDL, declarations](#)
- [modules \(IDL\)](#)
 - [_nesting within files](#)
- [MOM \(message-oriented middleware\)](#)
- [moveToCurrentRow\(\) \(ResultSet\)](#)
- [moveToInsertRow\(\) \(ResultSet\)](#)
- [multicast objects \(remote object communication\)](#)
- [multipart messages](#)
 - [_displaying](#)
 - [_sending](#)
- [Multipurpose Internet Mail Extensions \(see MIME types\)](#)
- [MVC \(model-view-controller\)](#)

[Struts](#)



[← PREV](#)

Index

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[J\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#)

[NameAlreadyBoundException class](#)

[NameClassPair class](#)

[NameComponent class 2nd](#)

[NamedNodeMap class](#)

[NamedValue class](#)

names

[children of Context \(JNDI\)](#)

[fully-qualified, for remote objects](#)

[NamespaceChangeListener interface](#)

[NamespaceChangeLogger class \(example\)](#)

namespaces

[support for, requesting](#)

[XML 2nd](#)

[naming context, obtaining object reference from](#)

naming conventions

[JNDI configuration services](#)

[two-level binding](#)

[Naming interface 2nd 3rd](#)

[looking up remote objects \(example\)](#)

[lookup\(\)](#)

[naming scopes, IDL](#)

[Naming Service \(CORBA\) 2nd](#)

[references, narrowing](#)

[naming service daemon \(IDL\)](#)

[naming services](#)

[binding names to objects](#)

[converting to JNDI for communication with CORBA](#)

[JNDI, looking up object](#)

[non-RMI, using to register RMI remote objects](#)

[registering server object with](#)

[RMI 2nd](#)

naming systems

[browsing](#)

[service providers for JNDI](#)

[NamingContext interface](#)

[NamingEnumeration interface](#)

[NamingEvent interface](#)

[NamingException class](#)

[namingExceptionThrown\(\) \(NamingListener interface\)](#)

[NamingManager class](#)

[NamingShell application](#)

- [_cd command](#)
- [_create command](#)
- [_destroy command](#)
- [_help and exit commands](#)
- [_initctx command](#)
- [_list command](#)
- [_listen command](#)
- [_running the shell](#)
- [_search command](#)

[narrow\(\)](#)

- [_NamingContextHelper class](#)
- [_PortableRemoteObject interface 2nd](#)

[native code, RMI and](#)

[Native Interface API](#)

[native messaging systems](#)

- [_proprietary message formats](#)
- [_wrapping messages with JMS format](#)

[native-API partly-Java drivers](#)

[native-protocol all-Java drivers](#)

[navigation rules, JSF](#)

- [_dynamic actions](#)

[NDS \(Novell Directory Services\)](#)

[nesting search expressions](#)

[net-protocol all-Java drivers](#)

[network protocols, CORBA](#)

[networks, heterogeneous nature of](#)

[newInstance\(\)](#)

- [_ActivationGroup class](#)
- [_DocumentBuilderFactory class](#)
- [_parser or processor, retrieving from Java program](#)
- [_SAXParserFactory class](#)
- [_TransformerFactory class](#)

[NewsAddress class](#)

[next\(\) \(ResultSet\)](#)

[NNTP \(Usenet\) news, addresses](#)

[nodes, DOM](#)

- [_importing into documents](#)
- [_listing of important types](#)
- [_manipulating](#)

[non-object-oriented languages \(CORBA specifications and\) 2nd](#)

[nonclass data structure definitions \(C++ and IDL\)](#)

[NoPermissionException class](#)

[NotFound class 2nd](#)

[Novell Directory Services \(NDS\)](#)

[NTM \(nested transaction model\)](#)

[null values](#)

- [_database columns](#)
- [_inserting with PreparedStatement](#)
- [_rows, inserting](#)

[nullsAreSortedHigh\(\) \(DatabaseMetaData\)](#)

[numeric literals, IDL](#)

- [_fixed-point](#)

[floating-point](#)



[← PREV](#)

Index

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[J\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#)

Object interface

[CORBA](#)

[direct communication with remote object](#)

[remote method requests, sending](#)

[object registry \(RMI\)](#)

[object relational mapping \(ORM\)](#)

object serialization

[marshaling method arguments](#)

[remote method arguments](#)

[unmarshaling](#)

[object-oriented database management systems \(OODBMS\)](#)

object-oriented programming

[Java](#)

[languages, mapping to CORBA specifications](#)

[object_to_string\(\) \(ORB\)](#)

[objectAdded\(\) \(NamespaceChangeListener\)](#)

[ObjectChangeListener interface](#)

[implementation \(example\)](#)

[ObjectImpl class](#)

[ObjectMessage class](#)

[objectRemoved\(\) \(NamespaceChangeListener\)](#)

objects

[activation service](#)

[adding to NamingContext \(CORBA\)](#)

[binding to NamingService](#)

[CORBA](#)

[creating](#)

[data types, IDL vs. Java handling of](#)

[distributed garbage collection](#)

[EJBs](#)

[implementation](#)

[Java types, inserting into databases](#)

[JSP, built-in](#)

[naming/registry service](#)

[non-serializable, throwing MarshalException](#)

[remote](#)

[remote distribution, enabled](#)

[remote vs. nonremote](#)

[storing and manipulating, support for](#)

[subcontexts](#)

[support for](#)

- [transport methods, deciding](#)
- [ODBC \(Open Database Connectivity\)](#)
 - [JDBC URL subprotocol \(odbc\)](#)
- [oj \(outer join\) keyword](#)
- [OMG, Java IDL mapping](#)
- [one-to-many relationships](#)
- [one-to-one relationships](#)
- [onMessage\(\) \(MessageListener\) 2nd](#)
- [OODBMS \(object-oriented database management systems\)](#)
- [open source tools](#)
- [open source, Struts as](#)
- [open\(\), Folder class](#)
- [OperationNotSupportedException class](#)
- operators
 - [arithmetic](#)
 - [JMS message selectors](#)
 - [combining search filters with](#)
 - [comparison](#)
 - [JMS message selectors](#)
 - [logical](#)
 - [JMS message selectors](#)
 - [SQL predicates](#)
- [optimistic concurrency, transactions](#)
- [OR operator, combining search filter with](#)
- Oracle databases
 - [JDBC-Thin driver, URLs for](#)
 - [PL/SQL stored procedure \(example\)](#)
- [ORBClass property](#)
- [ORBInitialHost property](#)
- [ORBInitialPort property](#)
- ORBs (Object Request Brokers)
 - [communications between](#)
 - [default implementation \(IDL\)](#)
 - [implementation not standardized](#)
 - [initializing](#)
 - [listing](#)
 - [properties, specifying \(methods\)](#)
 - [reference, initializing](#)
 - [remote object references](#)
 - [services provided](#)
 - [stringified object references and](#)
- [ORBSingletonClass property](#)
- [ORDER BY clause, EJB QL](#)
- order processing form (XML)
 - [converting from SAX into HTML with XSL](#)
 - [transforming into SAXResult](#)
- [order processing with SAX parser \(example\)](#)
- [org.omg.CORBA package 2nd](#)
- [org.omg.CORBA.portable package](#)
- [org.omg.CosNaming package 2nd](#)
- [org.w3c.dom package](#)
- [ORM \(object relational mapping\)](#)

[outer join keyword](#)



[← PREV](#)

Index

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[J\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#)

[packaged](#)

[packaging EJBs](#)

[panel grids, JSF tables](#)

[parameters](#)

[batch updates \(CallableStatement\)](#)

[clearing \(PreparedStatement\)](#)

[specifying, prepareStatement\(\)](#)

[stored procedures](#)

[parse\(\), SAXParser class](#)

[parsers for XML](#)

[creating with JAXP](#)

[DocumentBuilder class](#)

[DOM](#)

[getting a parser](#)

[handlers](#)

[manipulating DOM trees](#)

[navigating DOM tree](#)

[retrieving from a Java program](#)

[SAX](#)

[using](#)

[Part interface](#)

[setting message content](#)

[pass-through attributes, JSF HTML tags](#)

[passwords](#)

[for database connections](#)

[JavaMail, authenticating 2nd](#)

[JNDI contexts, authenticating](#)

[path expression variables](#)

[paths, Ant](#)

[peer-to-peer systems](#)

[client vs. server](#)

[clients and servers in](#)

[performance](#)

[databases, precompiled SQL statements and](#)

[escape-sequence processing, affecting](#)

[factory class vs. RMI registry](#)

[Hibernate](#)

[transaction isolation levels vs.](#)

[persistence](#)

[remote object references](#)

[remote references 2nd](#)

- [physical deployment](#)
- [platform independence \(JDBC\)](#)
- [plug-ins, Struts](#)
- [pluggability, Hibernate](#)
- [point-to-point messaging 2nd](#)
 - [browsing queues](#)
 - [object references](#)
 - [queueing messages](#)
 - [receiving messages](#)
 - [XA interfaces for](#)
- [policy, security](#)
- [polymorphic mappings, Hibernate](#)
- [pooling database connections](#)
- [POP \(Post Office Protocol\)](#)
- [POP-3 protocol](#)
- [PortableRemoteObject interface 2nd](#)
- [ports](#)
 - [activation daemon](#)
 - [exported remote objects, specifying for](#)
 - [for exported objects, specifying](#)
 - [ORBs, specifying](#)
 - [registry daemons, listening on](#)
- [POST requests](#)
- [prepare phase, distributed transactions and](#)
- [prepareCall\(\)](#)
 - [CallableStatement interface](#)
 - [Connection interface](#)
- [PreparedStatement interface 2nd](#)
 - [batch updates](#)
 - [parameters, setting \(example\)](#)
 - [setBlob\(\) and setClob\(\)](#)
- [prepareStatement\(\) \(Connection\)](#)
- [primary keys, entity beans \(EJBs\)](#)
- [primitive data types](#)
 - [streams of, in JMS messages](#)
- [printQueue\(\)](#)
- [printWarnings\(\) \(SQLWarning\)](#)
- [processing orders with SAX parser \(example\)](#)
- [processors, XSL](#)
- [processors, XSLT](#)
 - [retrieving from a Java program](#)
- [programmatic transactions](#)
- [properties](#)
 - [buildfiles \(Ant\)](#)
 - [JavaMail system](#)
 - [jaxp.properties, setting default XML parser](#)
 - [JMS messages](#)
 - [JMS messages, clients creating](#)
 - [JNDI context](#)
 - [ORB initialization](#)
 - [transactions](#)
- [property files, Ant](#)

[providers of messaging services, accessing with JavaMail](#)

[proxies, Hibernate relationship mappings](#)

[public classes](#)

[publish-subscribe messaging](#)

[client \(example\)](#)

[creating publishers and subscribers](#)

[durable subscriptions](#)

[XA interfaces for](#)



[← PREV](#)

Index

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[J\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#)

[Query class, Hibernate API](#)

[query variables](#)

[QueueBrowser class](#)

[QueueConnection interface](#)

[QueueConnectionFactory interface](#)

[queueing messages](#)

[QueueReceiver interface 2nd](#)

[_associating MessageListener with](#)

[QueueSender interface](#)

[QueueSession interface](#)

[← PREV](#)

Index

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[J\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#)[range variables](#)[EJB QL](#)[RDBMS \(relational database management system\)](#)[clusters](#)[Hibernate and](#)[reactivation constructors 2nd](#)[read-only result sets](#)[rebind\(\)](#)[Context interface 2nd](#)[Registry class](#)[receive\(\) \(QueueReceiver\)](#)[receiving messages, JMS client](#)[rcv command](#)[rcv_synch command](#)[references](#)[remote object, persistence of](#)[to local and remote registries, obtaining](#)[RefFSContextFactory class](#)[reflection, Hibernate](#)[register\(\) \(Activatable\)](#)[registerGroup\(\) \(ActivationSystem\)](#)[registering](#)[AccountReg class \(example\)](#)[activatable remote objects](#)[activation groups](#)[JNDI event listeners](#)[listen command](#)[remote objects on local host](#)[server object with CORBA Naming Service](#)[without instantiating](#)[registerListener\(\)](#)[PTPMessagingClient class \(example\)](#)[registerOutParameter\(\) \(CallableStatement\)](#)[registries, JavaMail providers](#)[Registry interface 2nd](#)[remote objects, looking up](#)[registry, RMI 2nd](#)[activatable class, registering](#)[dynamic class loading and](#)[factory classes](#)[local and remote, obtaining references to](#)

- [registering remote objects without activation](#)
- [run by rmid activation daemon](#)
- [services provided](#)
- [vendor implementations of](#)
- relational databases
 - [not object-oriented](#)
 - [SQL and](#)
 - [relationship mappings, Hibernate configuration](#)
- relationships
 - [many-to-many](#)
 - [many-to-one](#)
 - [one-to-many](#)
 - [one-to-one](#)
- [remote clients, EJBs](#)
- [remote interface 2nd](#)
 - [defining for remote objects](#)
 - [implementation \(example\)](#)
 - [objects enabled for remote distribution, tagging](#)
 - [remote exceptions, declaring](#)
 - [specifying for remote objects](#)
 - [stub and skeleton classes](#)
- remote methods
 - [arguments](#)
 - [executing with DII and DSI](#)
 - [implementation, server \(IDL/Java conversion\)](#)
 - [inout parameter](#)
 - [out or inout arguments](#)
 - [return values](#)
- [remote objects](#)
 - [activatable, defining 2nd](#)
 - [activating](#)
 - [activation daemon](#)
 - [activation groups](#)
 - [activation service \(RMI\)](#)
 - [binding to CORBA Naming Service \(JNDI\)](#)
 - [binding to names](#)
 - [class names, finding](#)
 - [classes \(RMI registry\)](#)
 - [clients accessing](#)
 - [defining](#)
 - [defining activatable objects](#)
 - [delegates](#)
 - [factory classes](#)
 - [getting from other remote objects](#)
 - [IDL \(interfaces, describing\)](#)
 - [implementing 2nd](#)
 - [interaction, starting](#)
 - [interface, defining](#)
 - [interfaces not available](#)
 - [looking up with LocateRegistry and Registry interfaces](#)
 - [looking up with Naming interface](#)
 - [native code, exporting \(RMI and JNI\)](#)

- [nonremote objects vs.](#)
- [persistent remote references](#)
- [references as printable character strings](#)
- [registering](#)
- [registering activatable objects](#)
- [remote reference, generating](#)
- [stubs, skeletons and objects, relationships among](#)

remote references

- [clients, obtaining \(IDL/Java conversion\)](#)
- [copying objects vs.](#)
- [CORBA](#)
- [expiring](#)
- [factory classes for](#)
- [finding, methods for](#)
- [persistence of 2nd 3rd](#)
- [point-to-point, nonpersistent](#)
- [RMI](#)
- [stubs \(server objects\)](#)
- [to client stubs](#)

[RemoteException class 2nd](#)

[RemoteObject class](#)

[RemoteServer class 2nd](#)

[removeChild\(\)](#)

[repeatable reads \(transactions\) 2nd](#)

[replaceChild\(\)](#)

[replication \(remote objects\)](#)

request dispatching

- [filters](#)

[Request interface](#)

[request level](#)

[RequestProcessor, Struts](#)

- [Action class and](#)

requests

- [JSF](#)
- [servlets](#)
 - [dispatching](#)
 - [forms](#)
 - [HEAD](#)
 - [HTTP](#)
 - [POST](#)
- [web service](#)

reset()

- [BytesMessage class](#)
- [StreamMessage class](#)

[resolve_initial_references\(\) \(ORB\) 2nd 3rd 4th](#)

[resource adapters](#)

resource management

- [cleaning up JMS connections and sessions](#)
- [database connections, closing](#)
- [for servers, factory objects and](#)

resources

- [deployment and](#)

[deployment descriptors and](#)

[resources, transactions](#)

[responses, servlets](#)

[error handling](#)

[request dispatching](#)

[Result interface](#)

[result sets](#)

[deleting rows](#)

[multiple](#)

[SQL queries](#)

[structure, discovering](#)

[updateable](#)

[ResultSet interface 2nd](#)

[BLOBs and CLOBs](#)

[cursor location \(JDBC 2.0\)](#)

[JDBC 1.0 vs. JDBC 2.0](#)

[large, handling](#)

[navigating](#)

[result set types, listed](#)

[rows, inserting, deleting, editing data](#)

[scroll-sensitive, updateable](#)

[updating \(single table without joins\)](#)

[ResultSetMetaData interface](#)

[database table, contents and data types \(example program\)](#)

[retrieving JavaMail messages](#)

[handling incoming messages](#)

[message stores](#)

[searches and message management](#)

[return codes, SQL](#)

[return values \(remote methods\)](#)

[serializable, required](#)

[RMI \(Remote Method Invocation\)](#)

[accessing RMI objects from CORBA](#)

[activation daemon 2nd](#)

[activation groups](#)

[architecture](#)

[classes, loading](#)

[clients, accessing remote objects](#)

[compiler 2nd](#)

[configuring clients and servers for](#)

[converting to JNDI to communicate with CORBA](#)

[defining activatable object](#)

[distributed garbage collection](#)

[dynamic class loading](#)

[factory classes](#)

[from applets](#)

[interfaces \(elements\)](#)

[key RMI classes for implementations](#)

[language dependence](#)

[naming/registry](#)

[native method calls and](#)

[object registry](#)

- [_object services](#)
- [persistent remote references](#)
- [registering activatable objects](#)
- [registering remote objects](#)
- [registry](#)
- [registry and naming services](#)
- [remote method arguments and return values](#)
- [remote object activation 2nd](#)
- [serial version utility](#)
- [serialver utility](#)
- [stub/skeleton layer](#)
- [stubs and skeletons, creating](#)
- [using with JNI vs. using CORBA](#)
- [writing class to implement remote interface](#)

[rmi protocol \(object URLs\)](#)

[RMI tools](#)

[RMI/IIOP 2nd](#)

- [RMI communicating with CORBA remote objects](#)
- [TCP/IP protocol base](#)
- [tnameserv utility](#)

[rmic \(RMI compiler\)](#)

- [IDL mapping of Java RMI interface, generating stub and skeleton classes, generating 2nd 3rd updated, RMI/IIOP](#)

[RMIClassLoader class](#)

[rmid activation daemon](#)

- [RMI registry, running](#)

[rmid command](#)

[RMIProcess class](#)

[RMISecurityManager class 2nd](#)

- [setting for remote class loading](#)

[rollback phase, distributed transactions](#)

[rollback\(\)](#)

- [CMTD and Connection interface 2nd](#)
- [stateless session beans and XASession class](#)

[rolling back transactions](#)

[root \(CORBA naming directory\)](#)

[root object class \(CORBA\)](#)

[rows](#)

- [accessing](#)
- [editing data, inserting or deleting](#)
- [insertions, preventing](#)
- [navigating \(ResultSet\)](#)

[RowSet interface](#)

[RowSetListener interface](#)

[← PREV](#)

Index

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[J\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#)

[SAAJ \(SOAP with Attachments API for Java\)](#)

[introduction](#)

[web service deployment](#)

[web services and](#)

[clients, writing](#)

[savepoints \(JDBC 3.0\)](#)

[SAX \(Simple API for XML parsing\) 2nd](#)

[ContentHandler interface](#)

[DefaultHandler class](#)

[DOM, combining with to parse XML documents](#)

[ErrorHandler interface](#)

[handlers](#)

[parser, using](#)

[transforming XML document to SAXResult](#)

[SAXParser class](#)

[parsing document directly from](#)

[SAXParserFactory class 2nd](#)

[SAXParserFactoryImpl class](#)

[SAXResult class](#)

[SAXSource class](#)

[scalar database functions](#)

[scope](#)

[Java beans, in JSPs](#)

[JSF managed beans](#)

[naming scopes, IDL](#)

[request level](#)

[Struts ActionForm beans 2nd](#)

[scoping](#)

[searches](#)

[targets for JNDI event listeners](#)

[scrolling](#)

[result sets 2nd](#)

[support information \(DatabaseMetaData\)](#)

[SearchControls class](#)

[searching](#)

[directories](#)

[JavaMail messages](#)

[search command](#)

[search controls](#)

[search criteria](#)

[search results](#)

[SearchResult class](#)

[SearchTerm class](#)

security

[application assembly](#)

[application layer security](#)

[authentication](#)

[authorization](#)

[authentication](#)

[EJB-tier](#)

[web-tier](#)

[authorization](#)

[EJB-tier](#)

[web-tier](#)

configuration

[deployment descriptors and](#)

[CORBA policy services](#)

[EJB deployment](#)

[Hibernate](#)

[J2EE](#)

[limitations](#)

[overview](#)

[XML encryption](#)

[JAAS and](#)

[servlets](#)

[SSO \(Single Sign-On\)](#)

[transport layer security](#)

[confidentiality](#)

[integrity](#)

web components

[EJB-tier authentication](#)

[EJB-tier authorization](#)

[web-tier authentication](#)

[web-tier authorization](#)

[web-tier transport-level security](#)

[security manager, remote class loading, allowing](#)

security policies

[changing to allow remote class loading](#)

[remote class resolution](#)

[remote classes, loading](#)

SELECT clause

[EJB QL](#)

[DISTINCT option](#)

[HQL](#)

[SELECT statement \(SQL\)](#)

[groups](#)

[joins](#)

[string comparisons](#)

[subqueries](#)

[semicolons \(IDL vs. Java\)](#)

[send command](#)

send()

[QueueSender class 2nd](#)

- [Transport class](#)
- [sending JMS messages 2nd](#)
 - [multipart](#)
- [sendMessage\(\)](#)
- [separator characters in JNDI compound names](#)
- [sequences, IDL](#)
- [Serializable interface](#)
 - [remote method arguments](#)
- serialization
 - [holder class objects](#)
 - [MarshaledObject](#)
 - [primitive data types](#)
 - [remote method arguments](#)
 - [Streamable object \(CORBA\)](#)
- [serialver utility](#)
- [server skeletons 2nd](#)
 - [classes, loading](#)
 - [classes, regenerating \(RMI/IIOP\)](#)
 - [compiling](#)
 - [converting remote request into method call on server object](#)
 - [generating with rmic 2nd](#)
 - [method invocations, forwarding to server implementation object](#)
 - [relationships with remote objects and stubs](#)
 - [RemoteObject as superclass for](#)
 - [RMI architecture](#)
- [server tool \(IDL\)](#)
- [servers](#)
 - [as Java objects \(RMI\)](#)
 - [configuring for remote class loading](#)
 - [constructing with given name](#)
 - [distinguishing from clients in distributed object system](#)
 - [EJB](#)
 - [J2EE, support for JMS](#)
 - [Java implementation, lacking](#)
 - [LDAP, prohibiting directory entries](#)
 - [native libraries, loading](#)
 - [registering RMI-enabled objects](#)
 - [registering with CORBA Naming Service](#)
 - [remote activation of server objects](#)
 - [remote interface, implementing](#)
 - [remote requests, handling](#)
 - [resource management](#)
- RMI
 - [implementing with RemoteServer](#)
 - [loading skeleton and implementation classes](#)
 - [RMI/JNI or CORBA, interface to native code](#)
- [SMTP](#)
- [ServerStub class](#)
- [Service class](#)
 - [retrieving URLName objects](#)
- [service providers, JavaMail implementations](#)
- [Services API](#)

[Servlet API 2nd](#)

[exceptions](#)

[javax.servlet.GenericServlet class](#)

[javax.servlet.http.HttpServlet class](#)

[javax.servlet.Servlet interface](#)

[servlet contexts](#)

[directory tree](#)

[initialization](#)

[methods](#)

[request mapping](#)

[ServletContext interface](#)

[ServletRequest interface](#)

[servlets 2nd](#)

[containers](#)

[cookies](#)

[environment](#)

[exceptions](#)

[filters](#)

[granularity of](#)

[HTTP](#)

[in e-commerce application \(example\)](#)

[initialization, custom](#)

[instantiating parser factories in](#)

[lifecycle](#)

[requests](#)

[dispatching](#)

[forms](#)

[HEAD](#)

[HTTP](#)

[POST](#)

[responses](#)

[error handling](#)

[request dispatching](#)

[security](#)

[session tracking](#)

[testing servlets \(Cactus\)](#)

[testing, Cactus](#)

[writing](#)

[session beans \(EJBs\)](#)

[container management](#)

[stateful](#)

[stateless](#)

[transaction support](#)

[session facade, EJB transactions](#)

[Session interface, JMS provider/client sessions](#)

[Session object](#)

[Hibernate API](#)

[session tracking](#)

[SessionFactory, Hibernate API](#)

[sessions](#)

[acknowledging received messages](#)

[authenticators for usernames and passwords](#)

- [closing](#)
- [JavaMail](#)
- [JMS](#)
- [providers of messaging services](#)
- [transactional messaging](#)
- [setAutoCommit\(\) \(Connection\)](#)
- [setBlob\(\) \(PreparedStatement\)](#)
- [setBytes\(\) \(Blob\)](#)
- [setClientID\(\) \(TopicConnection\)](#)
- [setClob\(\) \(PreparedStatement\)](#)
- [setDataHandler\(\)](#)
- [setDocumentLocator\(\)](#)
- [setEscapeProcessing\(\) \(Statement\)](#)
- [setFeature\(\) \(SAXParserFactory\)](#)
- [setFlag\(\) \(Message\)](#)
- [setMessageListener\(\) \(QueueReceiver\) 2nd](#)
- [setNamespaceAware\(\)](#)
 - [DOMParser class](#)
 - [SAXParser class](#)
 - [SAXParserFactory class](#)
- [setNodeValue\(\) \(Node\)](#)
- [setNull\(\) \(PreparedStatement\)](#)
- [setObject\(\) \(PreparedStatement\)](#)
- [setReturningObjFlag\(\) \(SearchControls\)](#)
- [setSavepoint\(\) \(Connection\)](#)
- [setSearchScope\(\) \(SearchControls\) 2nd](#)
- [setString\(\)](#)
 - [Clob class](#)
 - [PreparedStatement interface](#)
- [setText\(\) \(Part\)](#)
- [setTransactionIsolation\(\) \(Connection\)](#)
- [setURIResolver\(\) \(Transformer\)](#)
- [setValidating\(\)](#)
 - [DOMParser class](#)
 - [SAXParser class](#)
 - [SAXParserFactory class](#)
- [SMTP \(Simple Mail Transfer Protocol\)](#)
 - [addresses for mail](#)
- [SOAP \(Simple Object Access Protocol\)](#)
 - [introduction](#)
 - [mapping to Java](#)
 - [messages](#)
 - [data types](#)
- [socket programming, RMI vs.](#)
- [sockets, customized factories for](#)
- [software testing](#)
- [source code, XDoclet and](#)
- [Source interface](#)
- [spam, searching for and deleting](#)
- [SPI \(service providers interface\)](#)
 - [JNDI 2nd](#)
 - [separator characters for compound names](#)

[SQL \(Structured Query Language\)](#)

[ALTER TABLE statement](#)

[ANSI-92 SQL state codes](#)

[CREATE TABLE statement](#)

[data types 2nd](#)

[DATE, TIME, and TIMESTAMP](#)

[DROP statement](#)

[executing as batch](#)

[executing multiple](#)

[executing multiple queries simultaneously](#)

functions

[aggregate](#)

[value functions](#)

[mapping to Java primitive types](#)

[precompiled, running repeatedly](#)

[query results](#)

[relational databases](#)

[return codes](#)

[SQL-92](#)

statements

[data manipulation](#)

[DELETE](#)

[INSERT](#)

[schema manipulation](#)

[SELECT](#)

[UPDATE](#)

[statements in JDBC](#)

[stored procedures](#)

[SQLException class 2nd](#)

[SQLWarning class](#)

[SSO \(Single Sign-On\), security](#)

[start\(\) \(QueueConnection\)](#)

[startDocument\(\) \(Content Handler\)](#)

[startElement\(\) \(ContentHandler\) 2nd](#)

[Statement interface 2nd](#)

[PreparedStatement vs. Statement objects](#)

[queries, executing multiple](#)

[static methods \(helper class\)](#)

[static proxy, web services clients 2nd](#)

[Store class](#)

[stored procedures \(databases\)](#)

[batch updates](#)

[Streamable interface](#)

[StreamMessage class](#)

[StreamResult class](#)

[StreamSource class](#)

[string literals, IDL](#)

string manipulation functions

[SQL](#)

[String objects \(arrays\), initial objects listing](#)

[string_to_object\(\) \(ORB\)](#)

[stringified object references 2nd](#)

strings

[comparisons, SELECT statement \(SQL\)](#)

[DatabaseMetaData methods](#)

[IDL](#)

[Java, converting to native](#)

[object references as printable characters](#)

[read-only attribute \(IDL\)](#)

[SQL, processing \(databases\)](#)

[structs, IDL](#)

Struts

[Action class](#)

[RequestProcessor](#)

ActionForms

[DynaActionForms](#)

[ActionServlet and](#)

[as framework](#)

[as open source](#)

[configuration file](#)

[introduction](#)

[MVC controller](#)

[plug-ins](#)

[RequestProcessor](#)

[Action class and](#)

[sample application](#)

[tags](#)

[form handling](#)

[JSTL Expression Language](#)

[validator](#)

[views](#)

[ActionForm class](#)

[scope](#)

[validation](#)

[stub and skeleton classes, regenerating for RMI/IIOP](#)

[stub/skeleton layer, RMI](#)

[stubs](#)

[as remote references to server objects](#)

[client and server](#)

[converting from stringified object references](#)

[styles, web services](#)

[stylesheets](#)

[subcontexts \(NamingService\), binding objects to](#)

[subqueries, SELECT statement \(SQL\)](#)

[subscriptions to topics](#)

Sun Microsystems

[Crimson XML parser](#)

[JNDI API classes and providers \(web site\)](#)

[RowSet implementations](#)

[support for transactions, databases](#)

[synchronous messaging](#)

[System class, loading native libraries](#)

[system properties for XML parsers](#)



[← PREV](#)

Index

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[J\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#)

tables

JSF

[data grids](#)

[editing](#)

[panel grids](#)

[modifying structure](#)

[<%= %> tag](#)

[<% %> tag](#)

[<%! %> tag](#)

tags

[attributes of](#)

JSP

[action tags](#)

[communication](#)

[custom](#)

[EL and](#)

[libraries](#)

[whitespace and](#)

[JSTL](#)

[Struts](#)

[form handling](#)

[JSTL Expression Language](#)

[text within, parsing](#)

[XML](#)

targets (Ant)

[buildfiles](#)

[modular](#)

tasks (Ant)

[directory tasks](#)

[enterprise](#)

[file tasks](#)

[Java tasks](#)

[classpaths](#)

[code compile](#)

[paths](#)

[running programs](#)

TCP/IP

[remote references, point-to-point](#)

[RMI/IIOP and](#)

[templates, XSL](#)

[temporary destinations for messages, creation by JMS clients](#)

[TestCases \(JUnit\)](#)

testing

[black box testing](#)

Cactus

[asynchronous code](#)

[enterprise components](#)

[JSPs](#)

[process](#)

[servlet filters](#)

[test proxies](#)

[testing EJBs](#)

[writing tests](#)

[coverage](#)

[EJBs, Cactus](#)

[result formatting](#)

[white box testing](#)

[TestSuites \(JUnit\)](#)

[text nodes](#)

[TextMessage interface 2nd](#)

[thread safety, parser factories and](#)

threads

[database updates, performing](#)

[PTPMessagingClient \(example\)](#)

[threadsafe servlets](#)

[ties \(CORBA skeletons\)](#)

[Time class](#)

timestamps

[JMS message headers](#)

[Timestamp class](#)

[tnameserv utility \(RMI/IIOP\)](#)

[TopicConnection interface](#)

[TopicConnectionFactory interface](#)

[topics, publish-subscribe messaging](#)

[durable subscriptions to](#)

[TopicSession interface](#)

[creating publishers](#)

[toString\(\) \(RemoteObject\)](#)

[tracking sessions](#)

[TRANSACTION_READ_UNCOMMITTED class](#)

[transactional messaging](#)

transactions

[best practices](#)

[changed transactions](#)

[concurrency, optimistic](#)

[connections supporting](#)

[declarative](#)

[distributed](#)

[JMS destination](#)

[scenarios](#)

[two-phase commit](#)

[EJBs](#)

[BMTD](#)

[CMTD 2nd](#)
[session facade](#)
[Hibernate](#)
[isolation](#)
[levels 2nd](#)
[violations](#)
[JDBC supporting, listed](#)
[models](#)
[changed transactions](#)
[NTM](#)
[operations](#)
[overview](#)
[programmatic](#)
[properties](#)
[resources](#)
[rolling back to savepoint](#)
[setting](#)
[XML documents for](#)
[transformation, XSLT, JAXP data sources](#)
[Transformer class](#)
[setURIResolver\(\)](#)
[TransformerFactory class 2nd](#)
[getFeature\(\)](#)
[Transient Naming Service Daemon \(IDL\)](#)
[transparency, Hibernate](#)
[transport layer security](#)
[confidentiality](#)
[integrity](#)
[web-tier transport-level security](#)
[transport protocols, RMI architecture](#)
[tree structure, DOM](#)
[extending trees](#)
[manipulating \(DocumentCondenser example\)](#)
[navigating](#)
[troubleshooting](#)
[two-level name binding](#)
[typedefs, IDL](#)
[Types class](#)
[types, Hibernate](#)

[← PREV](#)

Index

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[J\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#)

[unbind\(\), Registry interface](#)

[UnicastRemoteObject class 2nd](#)

[nonpersistent references](#)

[RemoteException and](#)

[unions, IDL](#)

[unit testing](#)

[Ant builds](#)

[dependencies](#)

[integration testing and](#)

[introduction](#)

[tools](#)

[units](#)

[Unix systems, RMI registry daemon](#)

[unmarshaling](#)

[classes, finding](#)

[UPDATE statement \(SQL\)](#)

[updateBlob\(\) \(ResultSet\)](#)

[updateClob\(\) \(ResultSet\)](#)

[updating](#)

[batch updates \(JDBC\)](#)

[existing rows, editing data](#)

[JDBC 2.0 support](#)

[SQL UPDATE statements](#)

[support information \(DataBaseMetaData\)](#)

[update counts 2nd](#)

[URIs, resolving for XSL files](#)

[URLDataSource class](#)

[URLs](#)

[bundling into MarshalledObject](#)

[class file name, encoding \(marshaling\)](#)

[class names, loading](#)

[context provider, for JNDI](#)

[database connections, using for](#)

[for class bytecodes](#)

[for JavaMail services](#)

[JDBC](#)

[JSTL and](#)

[object names, fully qualified](#)

[odbc subprotocol](#)

[RMI remote objects](#)

[Usenet news, addresses for](#)

[user-defined data types, IDL](#)

[arrays](#)

[enumerations](#)

[sequences](#)

[structs](#)

[typedefs](#)

[unions](#)

[usernames and passwords](#)

[for database connections](#)

[JavaMail 2nd](#)

[utilities, Ant](#)





Index

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[J\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#)

[validating XML documents against DTDs](#)

[validation \(JSF\)](#)

[custom validators](#)

[forms](#)

[messages](#)

[value functions, SQL](#)

[value types, IDL](#)

[variables](#)

[collection member variables](#)

[EJB QL](#)

[input parameter](#)

[JSTL](#)

[path expression](#)

[query variables](#)

[range variables](#)

[WHERE clause](#)

[EJB QL](#)

[vendor naming services, RMI registry and](#)

[views, Struts](#)

[ActionForm class](#)

[scope](#)

[validation](#)

[virtual machine](#)

[← PREV](#)

Index

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[J\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#)

[W3C \(World Wide Web Consortium\)](#)

[DOM](#)

[XML specification, web site](#)

[wait\(\)](#)

[warning\(\) \(ErrorHandler\)](#)

[warnings \(SQL\)](#)

[wasNull\(\) \(ResultSet\)](#)

[web applications](#)

[databases and](#)

[non-HTML content](#)

[structure](#)

[web components](#)

[deployment descriptors](#)

[component references](#)

[cross-component settings](#)

[declarations](#)

[resources](#)

[security configuration](#)

[web handler declarations](#)

[security](#)

[EJB-tier authentication](#)

[EJB-tier authorization](#)

[web-tier authentication](#)

[web-tier authorization](#)

[web-tier transport-level security](#)

[web handlers, deployment descriptors and](#)

[web services](#)

[clients](#)

[DII approach](#)

[dynamic proxy approach](#)

[static proxy approach](#)

[writing](#)

[deployment](#)

[Axis mode](#)

[J2EE](#)

[Java implementation code](#)

[SAAJ and](#)

[deployment descriptors](#)

[development](#)

[EJB web services](#)

[encodings](#)

- [implementation](#)
- [introduction](#)
- [overview](#)
- [SAAJ and 2nd](#)
- [styles](#)
- [writing, simple services](#)
- web UIs
 - [components](#)
- [WEB-INF directory](#)
- [web-tier authentication](#)
- [web-tier authorization](#)
- [web-tier transport-level security](#)
- [web.xml deployment descriptor](#)
 - [component references](#)
 - [cross-component settings](#)
 - [resources](#)
 - [security configuration](#)
 - [web component declarations](#)
 - [web handler declarations](#)
- [WebRowSet interface](#)
- [well-formed XML documents 2nd](#)
- WHERE clause
 - [EJB QL](#)
 - [literals](#)
 - [variables](#)
 - [HQL](#)
- [white box testing](#)
- whitespace
 - [handling by SAX XML parser](#)
 - [JSP tags](#)
- wildcard searches
 - [DatabaseMetaData methods](#)
 - [escape sequences, JDBC](#)
- Windows systems
 - [filename separator character](#)
 - [RMI registry daemon](#)
- [World Wide Web Consortium](#)
- [WSDL \(Web Services Description Language\) 2nd](#)
 - [mapping to Java](#)
 - [mappings, deployment descriptors](#)
 - [operations](#)

[← PREV](#)

Index

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[J\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#)

[XAConnectionFactory class](#)

[XAConnections class](#)

[Xalan XSL engine](#)

[XASessions class](#)

[XDoclet 2nd](#)

[\[annotations\]\(#\)](#)

[\[custom\]\(#\)](#)

[\[deployment descriptors\]\(#\)](#)

[\[processing\]\(#\)](#)

[\[source code generation\]\(#\)](#)

[\[tutorial\]\(#\)](#)

[Xerces parser \(Apache\)](#)

[XML](#)

[\[advantages of\]\(#\)](#)

[\[deployment descriptors\]\(#\)](#)

[\[documents, using\]\(#\)](#)

[\[DTDs \\(orders.xml example\\)\]\(#\)](#)

[\[getting a parser\]\(#\)](#)

[\[handlers\]\(#\)](#)

[\[JAXP 2nd\]\(#\)](#)

[\[JAXP data sources\]\(#\)](#)

[\[manipulating DOM trees\]\(#\)](#)

[\[navigating DOM tree\]\(#\)](#)

[\[parser or processor, getting\]\(#\)](#)

[\[parsing with DOM\]\(#\)](#)

[\[parsing with SAX\]\(#\)](#)

[\[specification, web site for\]\(#\)](#)

[\[using a SAX parser\]\(#\)](#)

[\[well-formed XML\]\(#\)](#)

[XML Encryption](#)

[XML Schema](#)

[XMLReader class](#)

[XSL stylesheets](#)

[XSLT \(XSL for Transformations\)](#)

[\[JAXP data sources\]\(#\)](#)

[\[JAXP processors for\]\(#\)](#)