

Mastering Enterprise JavaBeans™

Second Edition

Ed Roman

Scott Ambler

Tyler Jewell

Wiley Computer Publishing



John Wiley & Sons, Inc.

NEW YORK • CHICHESTER • WEINHEIM • BRISBANE • SINGAPORE • TORONTO

Publisher: Robert Ipsen
Editor: Robert M. Elliott
Developmental Editor: Emilie Herman
Managing Editor: John Atkins
Associate New Media Editor: Brian Snapp
Text Design & Composition: MacAllister Publishing Services, LLC

Designations used by companies to distinguish their products are often claimed as trademarks. In all instances where John Wiley & Sons, Inc., is aware of a claim, the product names appear in initial capital or ALL CAPITAL LETTERS. Readers, however, should contact the appropriate companies for more complete information regarding trademarks and registration.

This book is printed on acid-free paper. ∞

Copyright © 2002 by The Middleware Company. All rights reserved.

Published by John Wiley & Sons, Inc.

Published simultaneously in Canada.

No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, except as permitted under Sections 107 or 108 of the 1976 United States Copyright Act, without either the prior written permission of the Publisher, or authorization through payment of the appropriate per-copy fee to the Copyright Clearance Center, 222 Rosewood Drive, Danvers, MA 01923, (978) 750-8400, fax (978) 750-4744. Requests to the Publisher for permission should be addressed to the Permissions Department, John Wiley & Sons, Inc., 605 Third Avenue, New York, NY 10158-0012, (212) 850-6011, fax (212) 850-6008, E-Mail: permreq@wiley.com.

This publication is designed to provide accurate and authoritative information in regard to the subject matter covered. It is sold with the understanding that the publisher is not engaged in professional services. If professional advice or other expert assistance is required, the services of a competent professional person should be sought.

Library of Congress Cataloging-in-Publication Data:

ISBN: 0-471-41711-4

Printed in the United States of America.

10 9 8 7 6 5 4 3 2 1

To my wonderful wife, Younhi.

—*Ed Roman*

Acknowledgments	xix
Preface	xxi
Introduction	xxv
About the Author	xxxii
Part One Overview	1
Chapter 1 Overview	3
The Motivation for EJB	4
Divide and Conquer to the Extreme	5
Component Architectures	12
Introducing Enterprise JavaBeans	13
Why Java?	14
EJB as a Business Solution	14
The EJB Ecosystem	16
The Bean Provider	17
The Application Assembler	17
The EJB Deployer	18
The System Administrator	19
The Container and Server Provider	19
The Tool Vendors	20
Summary of Roles	20
The Java 2 Platform, Enterprise Edition (J2EE)	22
The J2EE Technologies	23
Summary	26
Chapter 2 EJB Fundamentals	29
Enterprise Beans	29
Types of Beans	30
Distributed Objects: The Foundation for EJB	32
Distributed Objects and Middleware	34
Explicit Middleware	34
Implicit Middleware	35
What Constitutes an Enterprise Bean?	37
The Enterprise Bean Class	37

	The EJB Object	38
	The Home Object	44
	The Local Interfaces	46
	Deployment Descriptors	50
	Vendor-Specific Files	51
	Ejb-Jar File	51
	Summary of Terms	52
	Summary	54
Chapter 3	Writing Your First Bean	55
	How to Develop an EJB Component	55
	The Remote Interface	57
	The Local Interface	58
	The Home Interface	58
	The Local Home Interface	59
	The Bean Class	62
	The Deployment Descriptor	66
	The Vendor-Specific Files	67
	The Ejb-jar File	67
	Deploying the Bean	68
	The Optional EJB Client JAR file	68
	Understanding How to Call Beans	69
	Looking up a Home Object	70
	Running the System	74
	The Server-Side Output	75
	The Client-Side Output	75
	Implementing Component Interfaces	75
	A Solution	76
	Summary	77
Part Two	The Triad of Beans	79
Chapter 4	Introduction to Session Beans	81
	Session Bean Lifetime	81
	Session Bean Subtypes	82
	Stateful Session Beans	82
	Stateless Session Beans	83
	Special Characteristics of Stateful Session Beans	84
	Achieving the Effect of Pooling with Stateful Beans	85
	The Rules Governing Conversational State	86
	Activation and Passivation Callbacks	87

Method Implementation Summary	89
A Simple Stateful Session Bean	89
Life Cycle Diagrams for Session Beans	100
Summary	103
Chapter 5 Introduction to Entity Beans	105
Persistence Concepts	105
Java Object Serialization	106
Object-Relational Mapping	106
Object Databases	109
What Is an Entity Bean?	109
About the Files that Make up an Entity Bean	112
Features of Entity Beans	112
Entity Beans Survive Failures	112
Entity Bean Instances Are a View into a Database	113
Several Entity Bean Instances May Represent the Same Underlying Data	114
Entity Bean Instances Can Be Pooled	116
There Are Two Ways to Persist Entity Beans	118
Creation and Removal of Entity Beans	119
Entity Beans Can Be Found	121
You Can Modify Entity Bean Data without Using EJB	123
Entity Contexts	124
getEJBLocalObject() / getEJBObject()	124
getPrimaryKey()	125
Summary	126
Chapter 6 Writing Bean-Managed Persistent Entity Beans	127
Entity Bean Coding Basics	127
Finding Existing Entity Beans: ejbFind()	129
Bean-Managed Persistence Example: A Bank Account	136
Account.java	137
AccountLocal.java	138
AccountHome.java	138
AccountLocalHome.java	138
AccountPK.java	139
AccountBean.java	143
AccountException.java	156
Client.java	156
The Deployment Descriptor	156
The Container-Specific Deployment Descriptor	161
Setting up the Database	161
Running the Client Program	161
Server-Side Output	162
Client-Side Output	163

Putting It All Together: Walking through a BMP Entity Bean's Life Cycle	163
Summary	166
Chapter 7 Writing Container-Managed Persistent Entity Beans	167
Features of CMP Entity Beans	167
CMP Entity Beans Are Subclassed	167
CMP Entity Beans Have No Declared Fields	168
CMP Get/Set Methods Are Defined in the Subclass	170
CMP Entity Beans Have an Abstract Persistence Schema	172
CMP Entity Beans Have a Query Language	173
CMP Entity Beans Can Have ejbSelect() Methods	175
Implementation Guidelines for Container-Managed Persistence	176
Container-Managed Persistence Example: A Product Line	180
Product.java	181
ProductLocal.java	182
ProductHome.java	182
ProductLocalHome.java	184
ProductPK.java	184
ProductBean.java	187
The Deployment Descriptor	191
The Container-Specific Deployment Descriptor	195
Client.java	196
Running the Client Program	196
The Life Cycle of a CMP Entity Bean	200
Summary	200
Chapter 8 Introduction to Message-Driven Beans	201
Motivation to Use Message-Driven Beans	201
The Java Message Service (JMS)	203
Messaging Domains	204
The JMS API	206
Integrating JMS with EJB	211
What Is a Message-Driven Bean?	212
Developing Message-Driven Beans	214
The Semantics	214
A Simple Example	215
Advanced Concepts	223
Message-Driven Bean Gotchas	225
Message Ordering	225
Missed ejbRemove() Calls	226
Poison Messages	228

How to Return Results Back to Message Producers	230
The Future: Asynchronous Method Invocations	235
Summary	236
Chapter 9 Adding Functionality to Your Beans	237
Calling Beans from Other Beans	237
Default JNDI Lookups	238
Understanding EJB References	239
Resource Factories	241
Environment Properties	244
Understanding EJB Security	245
Security Step 1: Authentication	246
Security Step 2: Authorization	257
Security Propagation	266
Understanding Handles	268
Home Handles	269
Summary	270
Part Three Advanced Enterprise JavaBeans Concepts	271
Chapter 10 Transactions	275
Motivation for Transactions	276
Atomic Operations	276
Network or Machine Failure	277
Multiple Users Sharing Data	278
Benefits of Transactions	279
The ACID Properties	280
Transactional Models	282
Flat Transactions	282
Nested Transactions	284
Other Transactional Models	286
Enlisting in Transactions with Enterprise JavaBeans	286
Underlying Transaction System Abstraction	287
Declarative, Programmatic, and Client-Initiated Transactions	287
Choosing a Transaction Style	290
Container-Managed Transactions	292
EJB Transaction Attribute Values	294
Programmatic Transactions in EJB	300
CORBA's Object Transaction Service (OTS)	300
The Java Transaction Service (JTS)	301
The Java Transaction API (JTA)	301
Declarative versus Programmatic Transactions Example	304

Transactions from Client Code	306
Transactional Isolation	307
The Need for Concurrency Control	308
Isolation and EJB	309
The Dirty Read Problem	311
The Unrepeatable Read Problem	312
The Phantom Problem	313
Transaction Isolation Summary	314
Isolation and EJB	315
Pessimistic and Optimistic Concurrency Control	316
Distributed Transactions	316
Durability and the Two-Phase Commit Protocol	317
The Transactional Communications Protocol and Transaction Contexts	318
Designing Transactional Conversations in EJB	320
Summary	323
Chapter 11 BMP and CMP Relationships	325
The CMP and BMP Difference	326
Cardinality	326
1:1 Relationships	328
1:N Relationships	332
M:N Relationships	336
Directionality	344
Implementing Directionality with BMP	344
Implementing Directionality with CMP	345
Directionality May Not Map to Database Schemas	347
Bidirectional or Unidirectional?	349
Lazy Loading	349
Aggregation vs. Composition and Cascading Deletes	350
Relationships and EJB-QL	352
Recursive Relationships	353
Circular Relationships	354
Referential Integrity	355
Relationships, Referential Integrity, and Client Code	357
Summary	360
Chapter 12 Persistence Best Practices	361
When to Use Entity Beans	362
Control	362
Parameter Passing Analogy	362
Procedural Versus Object-Oriented	363
Caching	363

Enforcement of Schema Independence	364
Ease of Use	364
Migration	364
Rapid Application Development	365
Choosing between CMP and BMP	365
Code Reduction and Rapid Application Development	365
Performance	366
Bugs	366
Control	367
Application Server and Database Independence	367
Relationships	368
Learning Curve and Cost	368
Choosing the Right Granularity for Entity Beans	368
Persistence Tips and Tricks	370
Beware the Object-Relational Impedance Mismatch	370
Hard-Coded Versus Soft-Coded SQL	370
When to Use Stored Procedures	371
Normalizing and Denormalizing	373
Use Your EJB Object Model to Drive Your Data Model	375
Follow a Good Data Design Process	375
Use Surrogate Keys	376
Understand the Impacts of Database Updates	377
Versioning EJB Components	377
Living with a Legacy Database Design	379
Handling Large Result Sets	387
Summary	390
Chapter 13 EJB Best Practices and Performance Optimizations	391
When to Use Stateful versus Stateless	391
When to Use Messaging versus RMI-IIOP	393
How to Guarantee a Response Time with Capacity Planning	397
How to Achieve Singletons with EJB	398
Wrap Entity Beans with Session Beans	398
Performance-Tuning Entity Beans	400
Choosing between Local Interfaces and Remote Interfaces	401
How to Debug EJB Issues	402
Partitioning Your Resources	404
Assembling Components	405
Developing Components to Be Reusable	406
When to Use XML in an EJB System	407
Legacy Integration with EJB	408
Summary	410

Chapter 14 Clustering	411
Overview of Large-Scale Systems	411
What Is a Large-Scale System?	412
Basic Terminology	413
Partitioning Your Clusters	415
Instrumenting Clustered EJBs	416
How EJBs Can Be Clustered	419
The Concept of Idempotence	420
Stateless Session Bean Clustering	421
Stateful Session Bean Clustering	423
Entity Bean Clustering	425
Message-Driven Bean Clustering	429
Other EJB Clustering Issues	430
First Contact	430
Initial Access Logic	430
Summary	431
Chapter 15 Starting Your EJB Project on the Right Foot	433
Get the Business Requirements Down	433
Decide Whether J2EE is Appropriate	434
Decide Whether EJB Is Appropriate	434
Staff Your Project	438
Design Your Complete Object Model	439
Implement a Single Vertical Slice	439
Choose an Application Server	442
Divide Your Team	443
Invest in Tools	445
Invest in a Standard Build Process	446
Next Steps	446
Summary	447
Chapter 16 Choosing an EJB Server	449
J2EE 1.3 Brand	450
Pluggable JRE	450
Conversion Tools	450
Complex Mappings	451
Third-Party JDBC Driver Support	451
Lazy-Loading	451
Deferred Database Writes	451

Pluggable Persistence Providers	451
In-Memory Data Cache	452
Integrated Tier Support	452
Scalability	452
High Availability	453
Security	453
IDE Integration	454
UML Editor Integration	454
Intelligent Load Balancing	455
Stateless Transparent Fail-over	455
Clustering	455
Java Management Extension (JMX)	456
Administrative Support	456
Hot Deployment	456
Instance Pooling	456
Automatic EJB Generation	457
Clean Shutdown	457
Real-Time Deployment	457
Distributed Transactions	458
Superior Messaging Architecture	458
Provided EJB Components	458
J2EE Connector Architecture (JCA)	459
Web Services	459
Workflow	459
Open Source	460
Specialized Services	460
Nontechnical Criteria	461
Summary	462
Chapter 17 EJB-J2EE Integration: Building a Complete Application	463
The Business Problem	463
A Preview of the Final Web Site	464
Scoping the Technical Requirements	468
Object Model for the Business Logic Tier	469
Object Model for the Presentation Tier	475
Example Code	482
Summary	488

Part Four	Appendixes	489
Appendix A	RMI-IIOP and JNDI Tutorial	491
	Java RMI-IIOP	492
	Remote Method Invocations	492
	The Remote Interface	493
	The Remote Object Implementation	496
	Stubs and Skeletons	497
	Object Serialization and Parameter Passing	499
	Passing By-Value	500
	Object Serialization	500
	What Should You Make Transient?	502
	Object Serialization and RMI-IIOP	503
	The Java Naming and Directory Interface (JNDI)	505
	Naming and Directory Services	506
	Problems with Naming and Directories	507
	Enter JNDI	508
	Benefits of JNDI	509
	JNDI Architecture	509
	JNDI Concepts	511
	Programming with JNDI	515
	Integrating RMI-IIOP and JNDI	517
	Binding an RMI-IIOP Server to JNDI	518
	Looking up an RMI-IIOP Server with JNDI	519
	Summary	520
Appendix B	CORBA Interoperability	523
	What Is CORBA?	523
	CORBA as the Basis for EJB	524
	Why Should I Care about CORBA?	524
	Drawbacks of CORBA	525
	Understanding How CORBA Works	525
	Object Request Brokers	525
	OMG's Interface Definition Language	526
	OMG IDL Maps to Concrete Languages	528
	CORBA Static Invocations	529
	CORBA's Many Services	531
	The Need for RMI-IIOP	531
	The Need for RMI-CORBA Interoperability	532
	Combining RMI with CORBA	533
	Steps to Take for RMI and CORBA to Work Together:	
	An Overview	538
	RMI-IIOP Client with a CORBA Object Implementation	538

CORBA Client with an RMI-IIOP Object Implementation	539
Bootstrapping with RMI-IIOP and CORBA	540
The Big Picture: CORBA and EJB Together	540
Sample Code	541
Summary	543
Appendix C Deployment Descriptor Reference	545
How to Read a DTD	545
The Header and Root Element	546
Defining Session Beans	547
<session>	547
Defining Entity Beans	549
<entity>	549
<cmp-field>	551
<query>	552
<query-method>	552
<method-params>	553
Defining Message-Driven Beans	553
<message-driven>	553
<message-driven-destination>	555
Defining Environment Properties	555
<env-entry>	556
Defining EJB References	556
<ejb-ref>	557
<ejb-local-ref>	558
Defining Security	558
<security-role-ref>	558
<security-identity>	559
<run-as>	559
Defining Resource Factories	560
<resource-ref>	560
<resource-env-ref>	561
Defining Relationships	561
<relationships>	562
<ejb-relation>	562
<ejb-relationship-role>	563
<relationship-role-source>	563
<cmr-field>	564
Defining the Assembly Descriptor	564
<assembly-descriptor>	565
<security-role>	566
<method-permission>	566
<container-transaction>	567

<exclude-list>	567
<method>	568
<method-params>	568
Appendix D The EJB Query Language (EJB-QL)	569
Overview	569
A Simple Example	570
The Power of Relationships	571
EJB-QL Syntax	572
The FROM Clause	572
The WHERE Clause	574
The SELECT Clause	578
Truth Tables	581
Final Note	583
Summary	583
Appendix E EJB Quick Reference Guide	585
Session Bean Diagrams	586
Stateless Session Bean Diagrams	587
Stateful Session Bean Diagrams	589
Entity Bean Diagrams	592
Message-Driven Bean Diagrams	597
EJB API Reference	598
EJBContext	599
EJBHome	600
EJBLocalHome	601
EJBLocalObject	601
EJBMetaData	602
EJBObject	602
EnterpriseBean	603
EntityBean	604
EntityContext	610
Handle	610
HomeHandle	611
MessageDrivenBean	611
MessageDrivenContext	611
SessionBean	612
SessionContext	614
SessionSynchronization	614
Exception Reference	616
Transaction Reference	617
Index	621

ACKNOWLEDGEMENTS

This book has been a project spanning several years. Many have commented that the first edition was one of the best technical books they ever read. What's made this book a reality are the many people that aided in its development.

We took a big risk in developing the second edition of this book and decided to build the book *on the Web*. We received feedback from around the world when writing this book, and thus we have an evolving list of contributors and reviewers. The list is too large to mention here but is available at www.TheServerSide.com.

As a special thanks, we'd like to acknowledge the great folks over at John Wiley & Sons. They have been absolutely outstanding throughout this book's evolution. In particular, we'd like to thank Bob Elliott, Emilie Herman, and Bob Ipsen for their incredible efforts.

As I write these words, I can't help but think back to an inflection point that occurred in my life almost three years ago. I remember sitting in my cubicle at Trilogy Software, an e-commerce company in Austin, Texas, lost in deep middleware thoughts. My challenge was to devise an interesting load-balancing strategy for our in-house application server, which we called the backbone.

The backbone was a superb software system. It was cleanly written, easy to use, and boasted some very high-end features—features such as distributed object support, object-relational mapping, and extensible domain object modeling. It had almost anything you needed for Internet development. It was a worthy investment for Trilogy.

I was part of a task force to add enterprise features to this backbone, such as transaction control, security, and load-balancing. Our goal was to improve the backbone into a product worthy of large-scale deployment.

So that day, after hours of racking my brain, I finally finished crafting what I believed to be a highly creative and optimal load-balancing strategy. Looking for feedback, I walked to my friend Court Demas' office. Court is one of those developers who can really pick apart almost any design and expose its flaws—a unique quality that only a few developers I know have.

Walking into Court's office, I was expecting a typical developer-level conversation, and that's what I received. We turned the design inside and out, marking up my freshly printed hard copy with scribbles and other unintelligible comments that only we could understand. Finally, satisfied that we had reached a conclusion, I thanked Court and walked toward the door, prepared to implement the changes we had agreed upon.

But I didn't make it that far. Court said something to me that would change my way of thinking. His comment baffled and confused me at first, but would eventually result in a complete paradigm shift and career move for me. What did Court say? Nothing profound, but simply, "You know Ed, this stuff is really what Enterprise JavaBeans is for."

At first, I had no idea what he was talking about. Enterprise JavaBeans? What's that? Something like regular JavaBeans? Eventually, Court managed to explain to me what EJB was. And once he explained it, I knew that Trilogy had to do a 180-degree turn or lose its competitive advantage.

You see, EJB is a specification for a server-side component marketplace. EJB enables you to purchase off-the-shelf components from one vendor, combine them with components from another vendor, and run those components in an application server written by yet a third vendor. This means companies can collaborate on the server side. EJB enables you to buy, rather than build, elements of server-side applications.

The EJB value proposition had strong ramifications for Trilogy. EJB represented a way for Trilogy to get out of the middleware business and concentrate on its e-commerce strategic efforts. This meant discarding the backbone completely in favor of a third-party vendor's architecture. Not only would this reduce Trilogy's maintenance costs, but it would also solidify its software suite, since their middleware would now be written by professionals who had been in the business for 20 years. This proposition would eventually lead to Trilogy forming an entirely new business unit.

I decided to start researching EJB and pushing for Trilogy to adopt it. I went to the Sun Microsystems Web page, downloaded the EJB 1.0 specification in PDF form, and printed it out. Back then, the specification was about a third of the size it is today.

Understanding the specification turned out to be much more challenging than downloading it. The specification was written for system-level vendors and was not meant to be a tutorial for end developers. The section on entity beans, for example, took me a good two months to really grasp, as the notion of persistent components was new to me.

This arduous struggle with understanding the EJB specification is what eventually led me to write this book for you. This book represents everything I wish I had when I first started using EJB in 1998. So what is this book about? Well, it may be more accurate to tell you what this book is *not*. This is not EJB propaganda. It is not a book on how to write EJB code on any single application server. This is not a nice book that paints a perfect picture of the EJB world. Nor is it an advertisement for any particular EJB product or a campaign to rid the world of Microsoft.

The goal of this book is to help you. I want you to be able to craft solid, secure, and scalable server-side deployments. As you read this book, you'll learn how to design, implement, and deploy EJB solutions. This book covers both the vision and the reality of EJB from an independent developer's perspective. I hope it will prepare you for the challenges you will face.

I wish the grass was greener and that I could write a book on how clean and portable EJB is; but the truth is that this technology is not perfect, and you should know exactly what the imperfections are. I will expose you to the gruesome and incompatible parts of EJB and also explain how the industry is solving these problems.

Indeed, the newer specifications (especially EJB 2.0) improve portability and reduce incompatibilities tremendously. I hope that by the time you're done reading this book, you are convinced that the vision of EJB is solid, and the future is very bright.

My hope is that I can save you time and energy, and aid you in designing well-crafted server-side deployments. But this is merely the beginning. The EJB marketplace is just getting started, and there's a whole lot more work ahead. I encourage you to take an active role in the middleware industry and to work with me taking EJB to the next level. Feel free to write your experiences, tips, and design strategies, and post them on TheServerSide.com to share with others. Our goal is to increase our knowledge of EJB as a community, and together, we can do it.

Ed Roman

This book is a tutorial on Enterprise JavaBeans (EJB). It's about EJB concepts, methodology, and development. This book also contains a number of advanced EJB topics, giving you a practical and real-world understanding of the subject. By reading this book, you will acquire a *deep* understanding of EJB.

Make no mistake about it—what you are about to read is *not* easy. EJB incorporates concepts from a wealth of areas, including distributed computing, databases, security, component-driven software, and more. Combining them is a magnificent stride forward for the Java community, but with that comes a myriad of concepts to learn and understand. This book will teach you the concepts and techniques for authoring reusable components in Java, and it will do so from the ground up. You need only to understand Java to understand this book.

While you're reading this book, you may want to download the EJB specification, available on <http://java.sun.com>.

Goals for This Edition

The first edition of this book came out in 1999. We had to make some tough calls when writing the second edition, and we are confident you'll like them. Here are our goals:

- **To update the book for EJB 2.0.** EJB 2.0 has many new useful features that we will detail throughout the book.
- **To be broad and also deep.** We do not regurgitate the complete EJB specification in this book, nor do we cover every last detail of EJB in this book. Rather, we cover the most important parts of EJB, leaving room to discuss advanced issues. For a complete reference while you are coding, search through the EJB specification using Adobe Acrobat. Readers who are looking for a well-written book that is interactive, fun to read, and covers the basics through advanced subjects have come to the right place.

- **To be concise.** Your time as a reader is extremely valuable, and you're likely waiting to read a stack of books besides this one. Given that most people don't have time to read 1,000-plus-page books, we actually wanted to *reduce* the size of this book as much as possible. So we've tightened things up and eliminated redundant examples. This way, you can get to actually program with EJB, rather than reading a book for months on end. The irony of this story is that it was harder for us to write a shorter book than a long book!
- **To be a book for developers.** This book is not intended for high-level businessmen. This is a technical book for a technical audience.
- **To write a book the right way.** This book's primary author, Ed Roman, has taken his skills in training and knowledge transfer and applied them to this book. Thus, we've infused this book with the following attributes:
 - **A conversational style.** When you read this book, sometimes you'll feel like you're almost having a discussion with us. We think this is far superior to spending eons trying to re-read a formal writing style over and over again.
 - **Use of diagrams and bulleted lists.** The adage a picture is worth a thousand words applies here. These tactics are great for breaking up blocks of text. They keep things varied and make the book a much faster read.
 - **A consistent voice.** Even though several coauthors wrote this book, you'll hear one voice. This was done to combine best-of-breed knowledge from several expert coauthors, while maintaining a uniform look and feel throughout the book.
- **To be an introductory book, but also to get quickly into advanced topics.** We figured that the average developer has had enough of books that merely skim the surface. We wanted to write a book that pushed beyond the basics. Our approach when writing this book was to always err on the side of being advanced. To achieve this, we did an immense amount of research. We participated in the mailing lists, performed many real-world projects, attended conferences and seminars, and networked with the top experts throughout the world.
- **To be vendor-neutral.** All vendor-specific deployment steps are externalized to the book's accompanying source code. This makes this book useful for any EJB server.
- **To add useful EJB information garnered from our instructor-led training classes.** Having taught EJB/J2EE for years, we have learned significantly

from our students. We have interlaced this book with many of our own students' questions and answers in relevant sections.

- **To take all the source code and make it available online.** By making the code available on the Web, you know it's the latest version. This will ensure the code you receive works right the first time.

Organization of the Book

The text is organized into the following five parts:

Part 1 is a whirlwind introduction to EJB programming. Part 1 serves as a great overview for people in a hurry. While Part 1 is essential information to EJB newcomers, veterans will also find nuggets of useful knowledge as well. The following chapters are covered:

Chapter 1 is a tour of enterprise computing. We'll talk about components, distributed frameworks, and containers. We'll also introduce EJB and J2EE.

Chapter 2 moves onto the fundamentals of building an EJB system, including the tricky concept of request interception. We'll also look at the files that makeup an enterprise bean.

Chapter 3 shows you how to put together a simple enterprise bean. We'll also learn how JNDI is used in EJB, and see how to call that bean from a client.

Part 2 devotes exclusive attention to programming with EJB. We'll see how to use the triad of beans: entity beans, session beans, and message-driven beans. We'll cover the basics of writing each type of bean, including an example as well as detailed life cycle diagrams.

Chapter 4 covers session beans. We'll look at the difference between stateful and stateless session beans, how to code a session bean, and what's going on behind-the-scenes with session beans.

Chapter 5 is a conceptual introduction to entity beans. We'll look at persistence concepts, what makes entity beans unique, and the files involved when building entity beans.

Chapter 6 covers bean-managed persistent (BMP) entity beans. We'll see how to program a BMP entity bean, and also look at what's happening behind the scenes with BMP.

Chapter 7 covers container-managed persistent (CMP) entity beans. We'll focus in on the exciting new advances that EJB 2.0 has introduced, we'll learn how to program a CMP entity bean, and also look at what's happening behind the scenes with CMP.

- Chapter 8** covers message-driven beans. We'll first review the Java Message Service (JMS), which is a pre-requisite for learning message-driven beans. We'll then dive in and understand how to program with message-driven beans.
- Chapter 9** discusses the EJB environment, along with services provided by the container. This includes security, environment properties, resource factories, references between beans, and handles.
- Part 3** is the most exciting part of the book, and covers advanced EJB concepts. The following chapters are included:
- Chapter 10** tackles transactions. Transactions are a crucial topic for anyone building an EJB deployment that involves state. We'll discuss transactions at a conceptual level, and how to apply them to EJB. We'll also learn about the Java Transaction API (JTA).
- Chapter 11** covers relationships between entity beans. This is a critical concept for anyone performing complex persistence. We'll understand the concepts of cardinality, directionality, referential integrity, and cascading deletes. We'll also see how to code relationships for *both* CMP and BMP entity beans.
- Chapter 12** covers persistence best practices. You'll learn exciting concepts such as how to choose between session beans and entity beans, how to choose between BMP and CMP, and survey a collection of persistence best practices that we've assembled from our knowledge and experience.
- Chapter 13** covers EJB design strategies to help your projects succeed. You'll learn about interesting topics such as how to choose between local interfaces and remote interfaces, how to choose between stateful and stateless systems, and how to choose between a 3-tier and 4-tier deployment.
- Chapter 14** discusses clustering in large-scale EJB systems. You'll learn about how clustering works behind-the-scenes, and learn a few strategies for how containers might achieve clustering. This is a critical topic for anyone building a system that involves several machines working together.
- Chapter 15** covers EJB project management. We'll talk about how to get your project started on the right foot. This includes how to choose whether EJB is right for you, how to build a first-pass of your system, and how to divide up your development team.
- Chapter 16** covers how to choose an EJB server. We'll describe our methodology for how an organization can compare and contrast different vendors' offerings. We'll also list our set of criteria for what we would want in an EJB server.
- Chapter 17** shows how to build a real-world J2EE system using EJB components. We'll see how the EJB components should be used *together* in an

enterprise, as well as how to connect them with clients such as Java Servlets and JavaServer Pages (JSPs). We'll also demonstrate how to design an EJB object model using UML.

The Appendices are a collection of ancillary EJB topics. Some developers may want to read the appendices, while some may not need to do so.

Appendix A teaches you Java Remote Method Invocation over the Internet Inter-ORB Protocol (RMI-IIOP) and the Java Naming and Directory Interface (JNDI). These technologies are pre-requisites for using EJB. If you're just starting down the EJB road, you must read this appendix first.

Appendix B discusses how to integrate EJB and CORBA systems together. We'll learn about how EJB and CORBA are interoperable through RMI-IIOP, and see sample code for calling an EJB component from a CORBA client.

Appendix C is a deployment descriptor reference guide. This will be useful for you later, when you're writing a deployment descriptor and need a guide.

Appendix D covers the new EJB query language (EJB-QL) in detail.

Appendix E is an API and diagram reference guide. This is useful when you need to look up the purpose of a method or class in EJB.



Throughout the book, this icon will signal a tip, note, or other helpful advice on EJB programming.



In a similar paradigm to our training courses, the content of this book is very interactive. We have taken our knowledge of adult learning, and scattered boxes like this throughout the book. Each box asks you a question to get you thinking. The answers to the questions are posted on the book's accompanying Web site. What do you think the benefits are of this paradigm?

Illustrations in the Text

Almost all of the illustrations in this book are written in the Unified Modeling Language (UML). UML is the de facto standard methodology for illustrating software engineering concepts in an unambiguous way. If you don't know UML, pick up a copy of *The Unified Modeling Language User Guide* (Addison-Wesley, ISBN 0201571684), which illustrates how to effectively use UML in your everyday software. UML is a highly important achievement in object-oriented methodology. It's a common mechanism for engineers to

communicate and design, and it forces you to abstract your object model prior to implementation. We cannot stress its use enough.

The Accompanying Web Site

This book would not be complete without a way to keep you in touch after it was published. A Web site is available for resources related to this book. There you'll find

- All of the source code you see in this book. The code comes complete with build scripts, ready to build and run. It should be portable to a variety of application servers that are EJB 2.0- and J2EE 1.3-compliant.
- Updates to the source code examples.
- Links to EJB resources.
- Error corrections from the text.

The Web site is at www.wiley.com/compbooks/roman.

Feedback

When you begin your EJB programming, we're sure you'll have many experiences to share with other readers as well. Feel free to email examples, case studies, horror stories, or tips that you've found helpful in your experiences, and we'll post them on the Web site.

Send bug reports to bookbugs@middleware-company.com.

Send general communications to Ed Roman at:
edro@middleware-company.com.

From Here

Now that we've gotten the logistics out of the way, let's begin our exploration of Enterprise JavaBeans with Part 1, an introduction to EJB concepts and programming.

ABOUT THE AUTHOR

Ed Roman is one of the world's leading authorities on high-end middleware technologies. He has been heavily involved with Sun Microsystems' enterprise Java solutions from their inception, and has designed, built, and deployed a variety of enterprise applications, including architecting and developing complete application server products. He devotes a significant amount of time towards influencing and refining Sun's enterprise specifications, contributes regularly to middleware interest mailing lists, and regularly speaks at middleware-related conferences.

Ed is CEO of The Middleware Company (www.middleware-company.com), a firm specializing in EJB, J2EE, and XML-based Web Services training and consulting. The mission of The Middleware Company is to educate and aid in the design, development, and deployment of middleware solutions. Are you or your company making a purchase decision, performing EJB design work, integrating a legacy system to EJB, performing e-commerce-related deployments, or working on any other middleware endeavors? If you need some assistance, The Middleware Company can be a valuable resource.

Ed also is CEO of TheServerSide.com, which is the de facto J2EE community Web site. Every day, thousands of developers get together on TheServerSide.com to share EJB design patterns, hear about the latest EJB news, ask and answer EJB development questions, and read articles. After you've read this book, visit TheServerSide.com to catch up on the latest EJB information. TheServerSide.com is a completely free service and is intended to help the community.

And last but not least, if you want to get involved in the middleware field, Ed is always looking for great people who want to work on exciting projects using the latest technologies. You can reach him at edro@middleware-company.com.

About the Coauthors

Tyler Jewell oversees BEA's technology evangelism efforts, which are chartered to use print and speaking media to deepen developers' respect for enterprise technologies and BEA products. Tyler is an experienced developer,

lecturer, and author. He has worked on more than 40 e-business development projects, has delivered over 200 speeches, and has published nearly 6,000 pages of content worldwide.

Tyler is a co-author of *Professional Java Server Programming—J2EE 1.3* (Wrox, 2001). He is a member of O'Reilly's editorial advisory panel and maintains a monthly J2EE column at www.onjava.com. He also is a technology adviser to TheServerSide.com.

In his spare time, Tyler is an avid volleyball and poker enthusiast and a connoisseur of fine red wines. He can be reached at tyler@bea.com.

Scott W. Ambler is president and a senior consultant of Ronin International, www.ronin-intl.com, a software services consulting firm that specializes in software process mentoring, Agile Modeling (AM), and object/component-based software architecture and development. He is also founder and thought leader of the Agile Modeling (AM) methodology, www.agilemodeling.com.

Scott is the author of the books *The Object Primer, 2nd Edition* (2001), *Building Object Applications That Work* (1997), *Process Patterns* (1998), and *More Process Patterns* (1999), and co-author of *The Elements of Java Style* (2000), all published by Cambridge University Press. He is author of the forthcoming *Agile Modeling* (Autumn 2001) from John Wiley & Sons. He is also co-editor with Larry Constantine of the Unified Process series from R&D books (2000-2001). Scott is a contributing editor with *Software Development* magazine (www.sdmagazine.com), a contributor to *IBM DeveloperWorks* (www.ibm.com/developer), and a columnist with *Computing Canada*.

Scott's personal Web site, www.ambysoft.com, has a wide variety of white papers, including the AmbySoft Inc. "Coding Standards for Java," which are available for free download. In his spare time, Scott studies T'ai Chi and the Goju Ryu and Kobudo styles of karate. Scott has spoken at a wide variety of international conferences including software development, UML world, object expo, Java expo, and application development.

Overview



In Part 1, we introduce the server-side development platform that is the *Java 2 Platform, Enterprise Edition* (J2EE), of which the *Enterprise JavaBeans* (EJB) component architecture is a vital piece. J2EE is a conglomeration of concepts, programming standards, and innovations—all written in the Java programming language. With J2EE, you can rapidly construct distributed, scalable, reliable, and portable secure server-side deployments.

Chapter 1 begins by exploring the need for a server-side component architecture such as EJB. You'll see the rich needs of server-side computing, such as scalability, high availability, resource management, and security. We'll look at each of the different parties that are involved in an EJB deployment. We'll also survey the J2EE server-side development platform.

Chapter 2 moves on to the Enterprise JavaBeans fundamentals. We'll look at the concept of *request interception*, which is crucial for understanding how EJB works. We'll also look at the different files that go into a bean and how they work together.

Chapter 3 gets down and dirty with EJB programming. Here, we'll write our first simple bean. We'll show how to code each of the files that compose the bean, and we'll also look at how to call that bean from clients.

Overview

Enterprise JavaBeans (EJB) is a server-side component architecture that simplifies the process of building enterprise-class distributed component applications in Java. By using EJB, you can write scalable, reliable, and secure applications without writing your own complex distributed component framework. EJB is about rapid application development for the server side; you can quickly and easily construct server-side components in Java by leveraging a prewritten distributed infrastructure provided by the industry. EJB is designed to support application portability and reusability across any vendor's enterprise middleware services.

If you are new to enterprise computing, these concepts will be clarified shortly. EJB is a complicated subject and thus deserves a thorough explanation. In this chapter, we'll introduce EJB by answering the following questions:

- What plumbing do you need to build a robust distributed object deployment?
- What is EJB, and what value does it add?
- Who are the players in the EJB ecosystem?

Let's kick things off with a brainstorming session.

The Motivation for EJB

Figure 1.1 shows a typical business application. This application could exist in any vertical industry and could solve any business problem. Here are some examples:

- A stock trading system
- A banking application
- A customer call center
- A procurement system
- An insurance risk analysis application

Notice that this application is a *distributed system*. We broke up what would normally be a large, monolithic application and divorced each layer of the application from the others, so that each layer is completely independent and distinct.

Take a look at this picture, and ask yourself the following question based purely on your personal experience and intuition: *If we take a monolithic application and break it up into a distributed system with multiple clients connecting to multiple servers and databases over a network, what do we need to worry about now (as shown in Figure 1.1)?*

Take a moment to think of as many issues as you can. Then turn the page and compare your list to ours. Don't cheat!

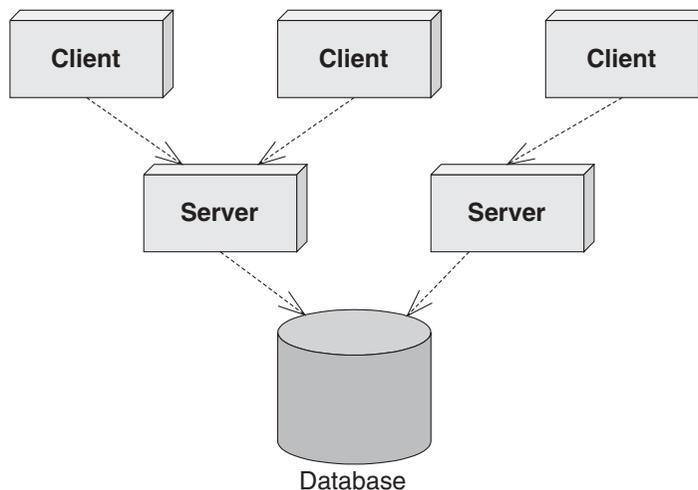


Figure 1.1 Standard multitier deployment.

In the past, most companies built their own middleware. For example, a financial services firm might build some of the middleware services above to help them put together a stock trading system.

These days, companies that build their own middleware risk setting themselves up for failure. High-end middleware is hideously complicated to build and maintain, requires expert-level knowledge, and is completely orthogonal to most companies' core business. Why not buy instead of build?

The *application server* was born to let you buy these middleware services, rather than build them yourself. Application servers provide you with common middleware services, such as resource pooling, networking, and more. Application servers allow you to focus on your application and not worry about the middleware you need for a robust server-side deployment. You write the code specific to your vertical industry and deploy that code into the runtime environment of an application server. You've just solved your business problem by *dividing and conquering*.

Divide and Conquer to the Extreme

We've just discussed how you can gain your middleware from an application server, empowering you to focus on your business problem. But there's even better news: You may be able to buy a partial solution to the business problem itself.

To achieve this, you need to build your application out of *components*. A component is code that implements a set of well-defined interfaces. It is a manageable, discrete chunk of logic. Components are not entire applications—they cannot run alone. Rather, they can be used as puzzle pieces to solve some larger problem.

The idea of software components is very powerful. A company can purchase a well-defined module that solves a problem and combine it with other components to solve larger problems. For example, consider a software component that computes the price of goods. We'll call this a *pricing component*. You hand the pricing component information about a set of products, and it figures out the total price of the order.

The pricing problem can get quite hairy. For example, let's assume we're ordering computer parts, such as memory and hard drives. The pricing component figures out the correct price based on a set of *pricing rules* that may include:

Base prices of a single memory upgrade or a single hard disk

Quantity discounts that a customer receives for ordering more than 10 memory modules

Things to Consider When Building Large Business Systems

By now you should have a decent list of things you'd have to worry about when building large business systems. Here's a short list of the big things we came up with. Don't worry if you don't understand all of them yet—you will.

- **Remote method invocations.** We need logic that connects a client and server via a network connection. This includes dispatching method requests, brokering of parameters, and more.
- **Load balancing.** Clients must be directed to the server with the lightest load. If a server is overloaded, a different server should be chosen.
- **Transparent fail-over.** If a server crashes, or if the network crashes, can clients be rerouted to other servers without interruption of service? If so, how fast does fail-over happen? Seconds? Minutes? What is acceptable for your business problem?
- **Back-end integration.** Code needs to be written to persist business data into databases as well as integrate with legacy systems that may already exist.
- **Transactions.** What if two clients access the same row of the database simultaneously? Or what if the database crashes? Transactions protect you from these issues.
- **Clustering.** What if the server contains state when it crashes? Is that state replicated across all servers, so that clients can use a different server?
- **Dynamic redeployment.** How do you perform software upgrades while the site is running? Do you need to take a machine down, or can you keep it running?
- **Clean shutdown.** If you need to shut down a server, can you do it in a smooth, clean manner so that you don't interrupt service to clients who are currently using the server?
- **Logging and auditing.** If something goes wrong, is there a log that we can consult to determine the cause of the problem? A log would help us debug the problem so it doesn't happen again.
- **Systems Management.** In the event of a catastrophic failure, who is monitoring our system? We would like monitoring software that paged a system administrator if a catastrophe occurred.
- **Threading.** Now that we have many clients connecting to a server, that server is going to need the capability of processing multiple client requests simultaneously. This means the server must be coded to be multi-threaded.
- **Message-oriented middleware.** Certain types of requests should be *message-based* where the clients and servers are very loosely coupled. We need infrastructure to accommodate messaging.
- **Object life cycle.** The objects that live within the server need to be created or destroyed when client traffic increases or decreases, respectively.

- **Resource pooling.** If a client is not currently using a server, that server's precious resources can be returned to a *pool* to be reused when other clients connect. This includes sockets (such as database connections) as well as objects that live within the server.
- **Security.** The servers and databases need to be shielded from saboteurs. Known users must be allowed to perform only operations that they have rights to perform.
- **Caching.** Let's assume there is some database data that all clients share and make use of, such as a common product catalog. Why should your servers retrieve that same catalog data from the database over and over again? You could keep that data around in the servers' memory and avoid costly network roundtrips and database hits.
- **And much, much, *much* more.**

Each of these issues is a separate service that needs to be addressed for serious server-side computing. These services are needed in any business problem and in any vertical industry. And each of these services requires a lot of thought and a lot of plumbing to resolve. Together, these services are called *middleware*.

Bundling discounts that the customer receives for ordering *both* memory and a hard disk

Preferred customer discounts that you can give to big-name customers

Locale discounts depending on where the customer lives

Overhead costs such as shipping and taxes

These pricing rules are in no way unique to ordering computer parts. Other industries, such as health care, appliances, airline tickets, and others need the same pricing functionality. Obviously, it would be a huge waste of resources if each company that needed complex pricing had to write its own sophisticated pricing engine. Thus, it makes sense that a vendor provides a generic pricing component that can be reused for different customers. For example:

1. The U.S. Postal Service can use the pricing component to compute shipping costs for mailing packages. This is shown in Figure 1.2.
2. An automobile manufacturer can use the pricing component to determine prices for cars. This manufacturer may set up a Web site that allows customers to get price quotes for cars over the Internet. Figure 1.3 illustrates this scenario.

- An online grocery store can use the pricing component as one discrete part of a complete *workflow* solution. When a customer purchases groceries over the Web, the pricing component first computes the price of the groceries. Next, a different vendor's component bills the customer with the generated price. Finally, a third component fulfills the order, setting things in motion for the groceries to be delivered to the end user. We depict this in Figure 1.4.

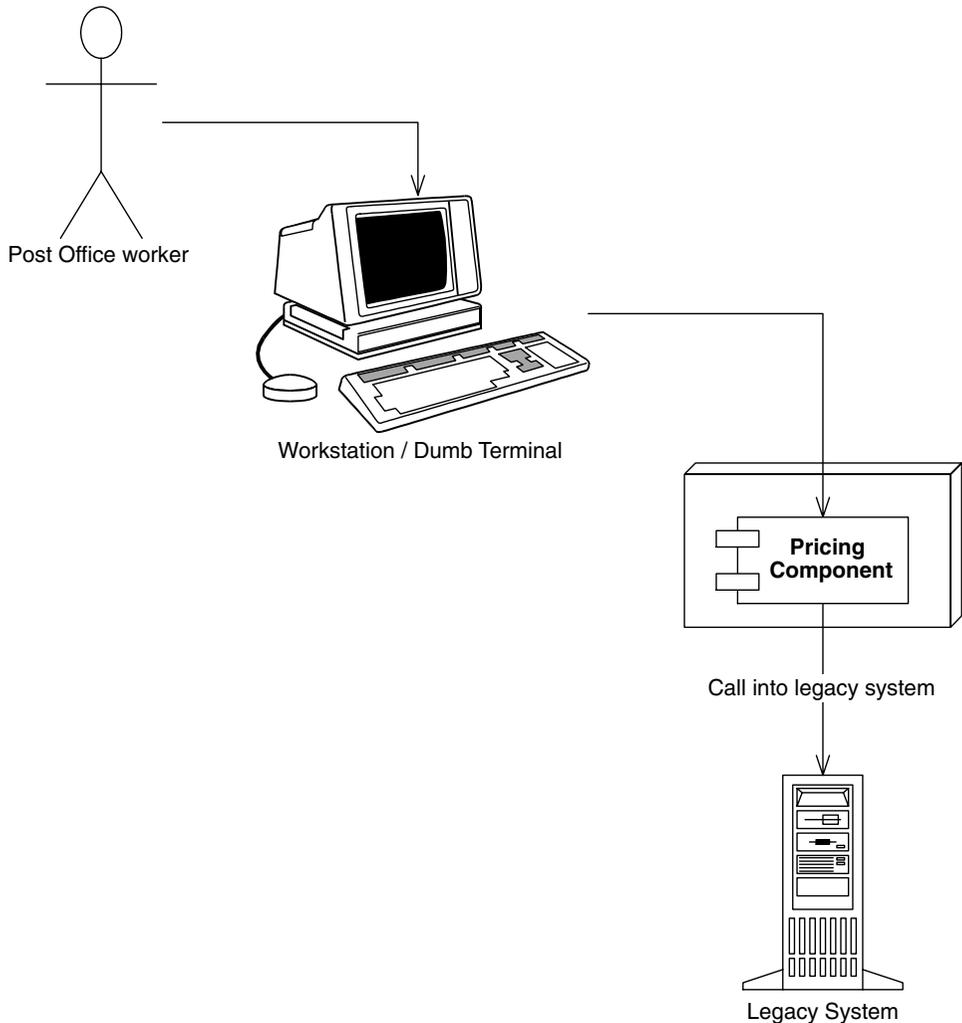


Figure 1.2 Reusing a pricing component for the U.S. Postal Service.

Reusable components are quite enticing because components promote rapid application development. An IT shop can quickly assemble an application from prewritten components rather than writing the entire application from scratch. This means:

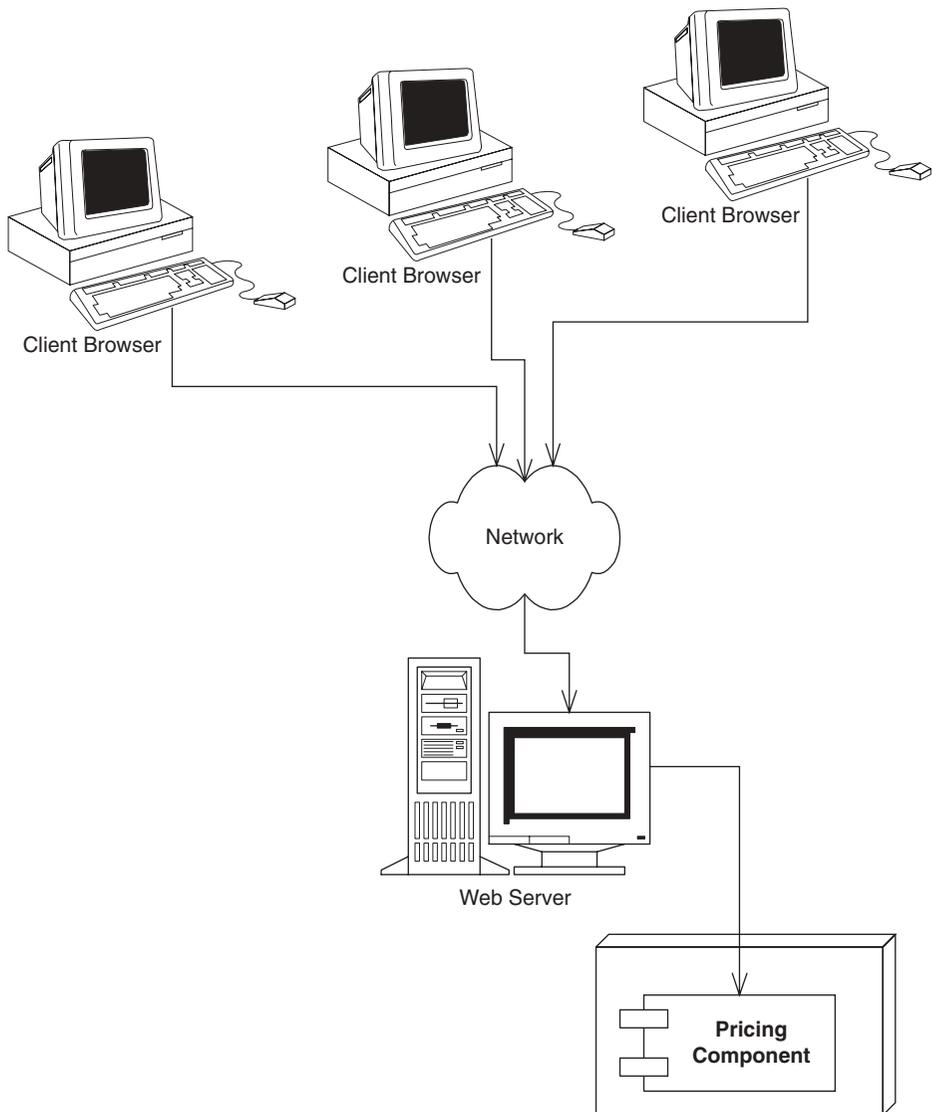


Figure 1.3 Reusing a pricing component for quoting car prices over the Internet.

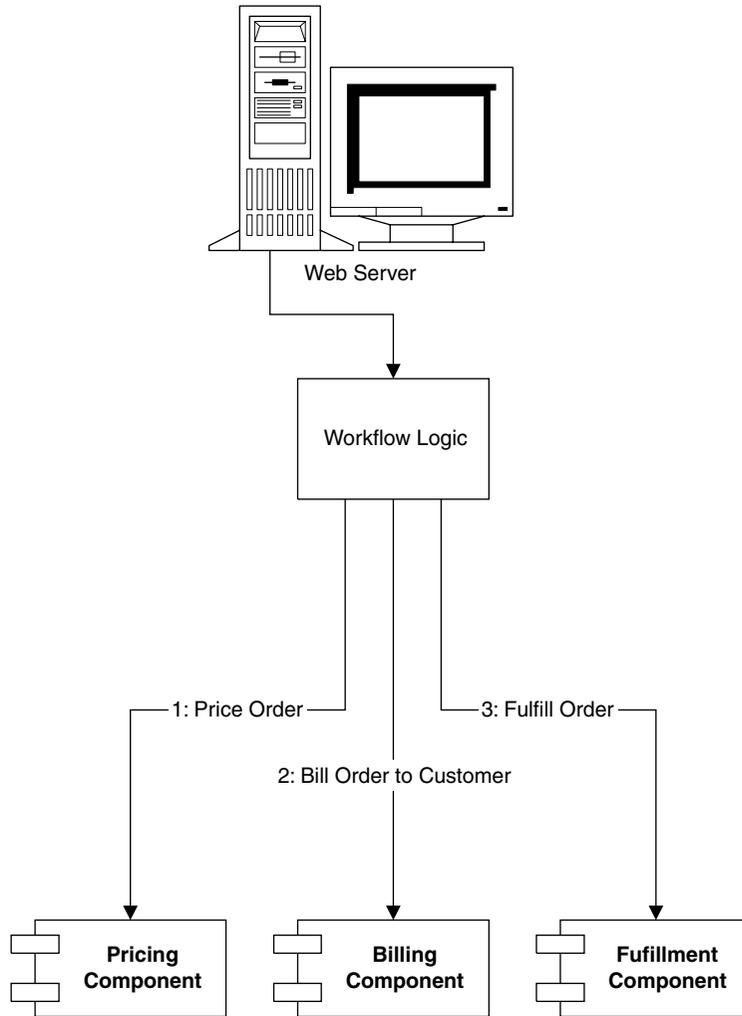


Figure 1.4 Reusing a pricing component as part of an e-commerce workflow solution.

The IT shop needs less in-house expertise. The IT shop can consider the pricing component to be a black box, and it does not need experts in complex pricing algorithms.

The application is assembled faster. The component vendor has already written the tough logic, and the IT shop can leverage that work, saving development time.

There is a lower total cost of ownership. The component vendor's cash cow is its components, and therefore it must provide top-notch documentation, support, and maintenance if it is to stay in business. Because the component vendor is an expert in its field, the component generally has fewer bugs and higher performance than an IT shop's home-grown solution. This reduces the IT shop's maintenance costs.

Once the rules of engagement have been laid down for how components should be written, a *component marketplace* is born, where vendors can sell reusable components to companies. The components are deployed within application servers, which provide the needed middleware.

Is a Component Marketplace a Myth?

There is a very small component marketplace today. For years we've been hoping that the marketplace will explode, but it is behind schedule. There are several reasons for Independent Software Vendors (ISVs) not shipping components:

Maturity. Because components live inside application servers, the application servers must be mature before we see components written to those servers.

Politics. Many ISVs have written their own application servers. Some (falsely) view this as a competitive advantage.

Questionable value. Most ISVs are customer-driven (meaning they prioritize what their customers are asking for). Since components are new to many customers, many of them are not asking for their ISVs to support components.

It is our opinion that the marketplace will eventually explode, and it's just a matter of time. If you represent an ISV, this could be a fantastic opportunity for you.

The good news is that the marketplace already beginning to emerge. Most packaged e-commerce ISVs (Ariba, Broadvision, Vignette, and so on) are shipping or have announced support for server-side Java technologies.

In the meantime, you'll have to build your own components from scratch within your organizations. Some of our customers at The Middleware Company are attempting this by having departments provide components to other departments. In effect, that department is acting as an internal ISV.

Component Architectures

It has been a number of years since the idea of multitier server-side deployments surfaced. Since then, well over 50 application servers have appeared on the market. At first, each application server provided component services in a nonstandard, proprietary way. This occurred because there was no agreed definition of what a component should be. The result? Once you bet on an application server, your code was locked into that vendor's solution. This greatly reduced portability and was an especially tough pill to swallow in the Java world, which promotes openness and portability. It also hampered the commerce of components, because a customer could not combine a component written to one application server with another component written to a different application server.

What we need is an *agreement*, or set of interfaces, between application servers and components. This agreement will enable any component to run within any application server. This will allow components to be switched in and out of various application servers without having to change code or potentially even recompile the components themselves. Such an agreement is called *component architecture* and is shown in Figure 1.5.



If you're trying to explain components to a nontechie, try these analogies:

- **Any CD player can play any compact disc because of the CD standard. Think of an application server as a CD player and components as compact discs.**
- **In the United States, any TV set can tune into any broadcast because of the NTSC standard. Think of an application server as a TV set and components as television broadcasts.**

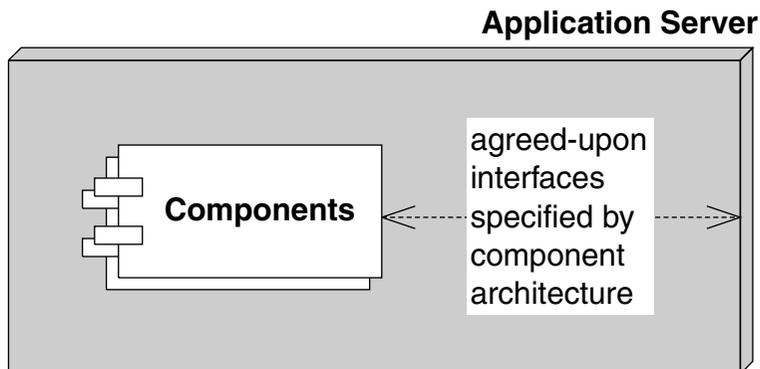


Figure 1.5 A component architecture.

Introducing Enterprise JavaBeans

The *Enterprise JavaBeans* (EJB) standard is a component architecture for deployable server-side components in Java. It is an agreement between components and application servers that enable any component to run in any application server. EJB components (called *enterprise beans*) are deployable, and can be imported and loaded into an application server, which hosts those components.

The top three values of EJB are as follows:

1. It is agreed upon by the industry. Those who use EJB will benefit from its widespread use. Because everyone will be on the same page, in the future it will be easier to hire employees who understand your systems (since they may have prior EJB experience), learn best practices to improve your system (by reading books like this one), partner with businesses (since technology will be compatible), and sell software (since customers will accept your solution). The concept of “train once, code anywhere” applies.
2. Portability is easier. The EJB specification is published and available freely to all. Since EJB is a standard, you do not need to gamble on a single, proprietary vendor’s architecture. And although portability will never be free, it is cheaper than without a standard.
3. Rapid application development. Your application can be constructed faster because you get middleware from the application server. There’s also less of a mess to maintain.

Note that while EJB does have these virtues, there are also scenarios where EJB is inappropriate. See Chapter 15 for a complete discussion of when to (and when not to) use EJB.



Physically, EJB is actually two things in one:

A specification. This is a 500-plus-page Adobe Acrobat PDF file, freely downloadable from <http://java.sun.com>. This specification lays out the rules of engagement between components and application servers. It constricts how you program so that you can interoperate.

A set of Java interfaces. Components and application servers must conform to these interfaces. Since all components are written to the same interfaces, they all look the same to the application server. The application server therefore can manage anyone’s components. You can freely download these interfaces from <http://java.sun.com>.

Why Java?

EJB components must be written in Java only and require dedication to Java. This is indeed a serious restriction. The good news, however, is that Java is an ideal language to build components, for many reasons.

Interface/implementation separation. We need a clean interface/implementation separation to ship components. After all, customers who purchase components shouldn't be messing with implementation. Upgrades and support will become horrendous. Java supports this at a syntactic level via the *interface* keyword and *class* keyword.

Safe and secure. The Java architecture is much safer than traditional programming languages. In Java, if a thread dies, the application stays up. Pointers are no longer an issue. Memory leaks occur much less often. Java also has a rich library set, so that Java is not just the syntax of a language but a whole set of prewritten, debugged libraries that enable developers to avoid reinventing the wheel in a buggy way. This safety is extremely important for mission-critical applications. Sure, the overhead required to achieve this level of safety might make your application slower, but 90 percent of all business programs are glorified Graphical User Interfaces (GUIs) to databases. That database is going to be your number one bottleneck, not Java.

Cross-platform. Java runs on any platform. Since EJB is an application of Java, this means EJB should also easily run on any platform. This is valuable for customers who have invested in a variety of powerful hardware, such as Win32, UNIX, and mainframes. They do not want to throw away these investments.



If you don't want to go the EJB route, you have two other choices as well:

- Microsoft's .NET managed components, part of the Microsoft.NET platform
- The Object Management Group (OMG's) Common Object Request Broker Architecture (CORBA)



Note that many EJB servers are based upon and can interoperate with CORBA (see Appendix B for strategies for achieving this).

EJB as a Business Solution

EJB is specifically used to help solve *business problems*. EJB components (enterprise beans) might perform any of the following tasks.

Perform business logic. Examples include computing the taxes on the shopping cart, ensuring that the manager has authority to approve the purchase order, or sending an order confirmation email using the *JavaMail API*.

Access a database. Examples include submitting an order for books, transferring money between two bank accounts, or calling a stored procedure to retrieve a trouble ticket in a customer support system. Enterprise beans achieve database access using the *Java Database Connectivity (JDBC) API*.

Access another system. Examples include calling a high-performing *CICS* legacy system written in COBOL that computes the risk factor for a new insurance account, calling a legacy *VSAM* data store, or calling *SAP R/3*. Enterprise beans achieve existing application integration via the *Java Connector Architecture (JCA)*.

EJB components are not GUI components; rather, enterprise beans sit behind the GUIs and do all the hard work. Examples of GUIs that can connect to enterprise beans include the following:

Thick clients. Thick clients execute on a user's desktop. They could connect via the network with EJB components that live on a server. These EJB components may perform any of the tasks listed above (business logic, database logic, or accessing other systems). Thick clients in Java include applets and applications.

Dynamically generated web pages. Web sites that are complex need their Web pages generated specifically for each request. For example, the homepage for Amazon.com is completely different for each user, depending on the user's profile. Java servlets and JavaServer Pages (JSPs) are used to generate such specific pages. Both servlets and JSPs live within a Web server and can connect to EJB components, generating pages differently based upon the values returned from the EJB layer.

XML-based Web Service wrappers. Some business applications require no user interface at all. They exist to interconnect with other business partners' applications that may provide their own user interface. For example, Dell Computer Corporation needs to purchase Intel chips to manufacture desktop computers. Intel could expose a Web Service that enables Dell's software to connect and order chips. In this case, Intel's system does not have a user interface of its own, but rather acts as a Web Service. Possible technologies used here include SOAP, UDDI, ebXML, and WSDL. This is shown in Figure 1.6.

The real difference between GUI components (thick clients, dynamically generated Web pages, and Web Service wrappers) and enterprise beans is the domain that each component type is intended to be part of. GUI components are well suited to handle *client-side* operations, such as rendering GUIs (although they don't necessarily need to have one), performing other presentation-related logic, and lightweight business logic operations. They deal directly with the end-user or business partner.

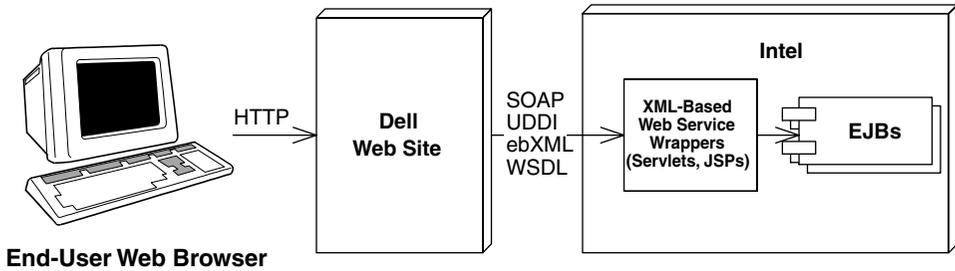


Figure 1.6 EJBs as the back-end to Web services.

Enterprise beans, on the other hand, are not intended for the client side; they are *server-side* components. They are meant to perform server-side operations, such as executing complex algorithms or performing high-volume business transactions. The server side has different kinds of needs from a rich GUI environment. Server-side components need to run in a highly available (24×7), fault-tolerant, transactional, and multiuser secure environment. The application server provides this high-end server-side environment for the enterprise beans, and it provides the runtime containment necessary to manage enterprise beans.

The EJB Ecosystem

To get an EJB deployment up and running successfully, you need more than just an application server and components. In fact, EJB encourages collaboration of *more than six* different parties. Each of these parties is an expert in its own field and is responsible for a key part of a successful EJB deployment. Because each party is a specialist, the total time required to build an enterprise-class deployment is significantly reduced. Together, these players form the *EJB Ecosystem*.

Let's discuss who the players are in the EJB Ecosystem. As you read on, think about your company's business model to determine which role you fill. If you're not sure, ask yourself what the core competency of your business is. Also think about what roles you might play in upcoming projects.



The EJB Ecosystem is not for everyone. At my company, we've heard ghastly stories of businesses choosing EJB because everyone else is using it, or because it is new and exciting. Those are the wrong reasons to use EJB and can result in disappointing results. For a complete discussion of when and when not to use EJB, see Chapter 15.

JavaBeans. Enterprise JavaBeans

You may have heard of another standard called *JavaBeans*. JavaBeans are completely different from Enterprise JavaBeans.

In a nutshell, JavaBeans are Java classes that have get/set methods on them. They are reusable Java components with properties, events, and methods (similar to Microsoft's *ActiveX controls*) that can be easily wired together to create (often visual) Java applications.

JavaBeans are much smaller than Enterprise JavaBeans. You can use JavaBeans to assemble larger components or to build entire applications. JavaBeans, however, are *development components* and are not *deployable components*. You typically do not deploy a JavaBean; rather, JavaBeans help you construct larger software that *is* deployable. And because they cannot be deployed, JavaBeans do not need to live in a runtime environment. Since JavaBeans are just Java classes, they do not need an application server to instantiate them, to destroy them, and to provide other services to them. The application itself is made up of JavaBeans.

The Bean Provider

The *bean provider* supplies business components, or enterprise beans. Enterprise beans are not complete applications, but rather are deployable components that can be assembled into complete solutions. The bean provider could be an ISV selling components or an internal department providing components to other departments.

Many vendors ship reusable components today. You can get the complete list from www.componentsource.com or www.flashline.com. In the future, traditional enterprise software vendors (such as sales force automation vendors, enterprise resource planning vendors, financial services vendors, and e-commerce vendors) will offer their software as enterprise beans or provide connectors to their current technology.

The Application Assembler

The application assembler is the overall application architect. This party is responsible for understanding how various components fit together and writes the applications that combine components. An application assembler may even author a few components along the way. His or her job is to build an application from those components that can be deployed in a number of

settings. The application assembler is the *consumer* of the beans supplied by the bean provider.

The application assembler could perform any or all of the following tasks:

- From knowledge of the business problem, decide which combination of existing components and new enterprise beans are needed to provide an effective solution; in essence, plan the application assembly.
- Supply a user interface (perhaps Swing, servlet/JSP, application/applet, or Web Service wrapper).
- Write new enterprise beans to solve some problems specific to your business problem.
- Write the code that calls on components supplied by bean providers.
- Write integration code that maps data between components supplied by different bean providers. After all, components won't magically work together to solve a business problem, especially if different vendors write the components.

An example of an application assembler is a systems integrator, a consulting firm, or an in-house programmer.

The EJB Deployer

After the application assembler builds the application, the application must be *deployed* (and go live) in a running operational environment. Some challenges faced here include the following:

- Securing the deployment with a firewall and other protective measures
- Integrating with an LDAP server for security lists, such as Lotus Notes or Microsoft Active Directory
- Choosing hardware that provides the required level of performance
- Providing redundant hardware and other resources for reliability and fault tolerance
- Performance-tuning the system

Frequently the application assembler (who is usually a developer or systems analyst) is not familiar with these issues. This is where the EJB deployer comes into play. EJB deployers are aware of specific operational requirements and perform the tasks above. They understand how to deploy beans within servers and how to customize the beans for a specific environment. The EJB deployer

has the freedom to adapt the beans, as well as the server, to the environment in which the beans are to be deployed.

An EJB deployer can be a staff person, an outside consultant, or a vendor. Examples of EJB deployers include Loudcloud and HostJ2EE.com, which both offer hosting solutions for EJB deployments.

The System Administrator

Once the deployment goes live, the system administrator steps in to oversee the stability of the operational solution. The system administrator is responsible for the upkeep and monitoring of the deployed system and may make use of runtime monitoring and management tools that the EJB server provides.

For example, a sophisticated EJB server might page a system administrator if a serious error occurs that requires immediate attention. Some EJB servers achieve this by developing hooks into professional monitoring products, such as Tivoli and Computer Associates. Others are providing their own systems management by supporting the *Java Management Extension* (JMX).

The Container and Server Provider

The container provider supplies an *EJB container* (the application server). This is the runtime environment in which beans live. The container supplies middleware services to the beans and manages them. Examples of EJB containers

Qualities of Service in EJB

Monitoring of EJB deployments is not specified in the EJB specification. It is an optional service that advanced EJB servers can provide. This means that each EJB server could provide the service differently.

At first blush you might think this hampers application portability. However, in reality this service should be provided *transparently* behind the scenes, and should not affect your application code. It is a quality of service that lies beneath the application level and exists at the systems level. Changing application servers should not affect your EJB code.

Other transparent qualities of service not specified in the EJB specification include load balancing, transparent fail-over, caching, clustering, and connection pooling algorithms.

are BEA's WebLogic, iPlanet's iPlanet Application Server, IBM's WebSphere, Oracle's Oracle 9i, Macromedia's JRun, Persistence's PowerTier, Brokat's Gemstone/J, HP's Bluestone, IONA's iPortal, Borland's AppServer, and the JBoss open source code application server.

The server provider is the same as the container provider. Sun has not yet differentiated these (and they may never do so). We will use the terms *EJB container* and *EJB server* interchangeably in this book.

The Tool Vendors

To facilitate the component development process, there should be a standardized way to build, manage, and maintain components. In the EJB Ecosystem, there are several *Integrated Development Environments* (IDEs) assist you in rapidly building and debugging components. Examples are Webgain's Visual Cafe, IBM's VisualAge for Java, or Borland's JBuilder.

Other tools enable you to model components in the Unified Modeling Language (UML), which is the diagram style used in this book. You can then auto-generate EJB code from that UML. Examples of products in this space are Togethersoftware's Together/J and Rational's Rational Rose.

There are other tools as well, such as tools to organize components (Flashline, ComponentSource), testing tools (JUnit, RSW Software), and build tools (Ant).

Summary of Roles

Figure 1.7 summarizes the interaction of the different parties in EJB.

You may be wondering why so many different participants are needed to provide an EJB deployment. The answer is that EJB enables companies or individuals to become experts in certain roles, and division of labor leads to best-of-breed deployments.

The EJB specification makes each role clear and distinct, enabling experts in different areas to participate in a deployment without loss of interoperability. Note that some of these roles could be combined as well. For example, the EJB server and EJB container today come from the same vendor. Or at a small startup company, the bean provider, application assembler, and deployer could all be the same person who is trying to build a business solution using EJB from scratch. What roles do you see yourself playing?

For some of the parties EJB merely suggests possible duties, such as the system administrator overseeing the well-being of a deployed system. For other parties, such as the bean provider and container provider, EJB defines a set of

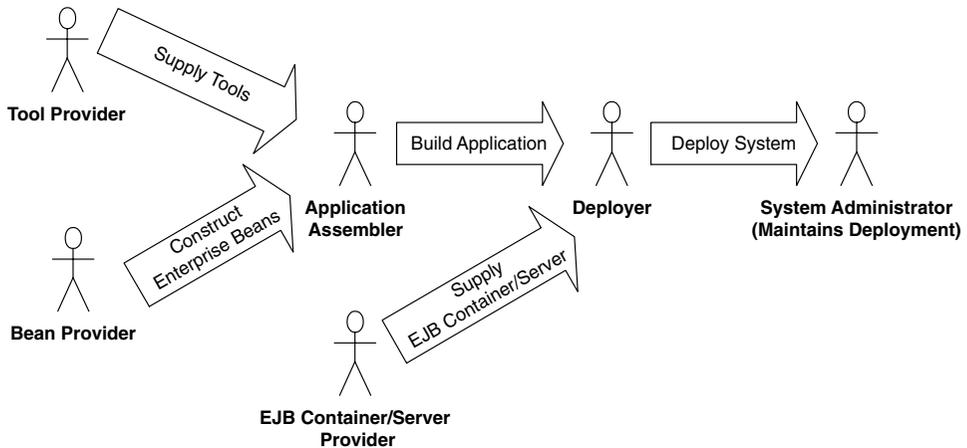


Figure 1.7 The parties of EJB.

strict interfaces and guidelines that must be followed or the entire ecosystem will break down. By clearly defining the roles of each party, EJB lays a foundation for a distributed, scalable component architecture where multiple vendors' products can interoperate.



A future EJB specification will define a new role, called the *persistence manager*, which plugs into an application server. Your components harness the persistence manager to map your business data into storage, such as mapping objects into relational databases.

The persistence manager may be written to understand how to persist business data to any storage type. Examples include legacy systems, flat file systems, relational databases, object databases, or a proprietary system.

The persistence manager provider may be the same as the container/server vendor, such as the case with IBM's WebSphere, which includes built-in persistence capabilities. Examples of ISV persistence manager providers include WebGain's TOPLink and Thought Inc's Cocobase.

Unfortunately, the persistence manager provider role is not explicitly defined in the EJB 2.0 specification. Due to time constraints, a standard for plugging persistence managers into application servers won't exist until a future version of EJB. The good news is this won't affect the portability of your code, because your application doesn't care whether it's being persisted by the container or by some persistence manager that happens to plug into the container. The bad news is that you'll need to rely on proprietary agreements between persistence manager providers and application server vendors, which means that not every persistence manager may work in every application server – for now.

The Java 2 Platform, Enterprise Edition (J2EE)

EJB is only a portion of a larger offering from Sun Microsystems called the Java 2 Platform, Enterprise Edition (J2EE). The mission of J2EE is to provide a platform-independent, portable, multiuser, secure, and standard enterprise-class platform for server-side deployments written in the Java language.

J2EE is a specification, not a product. J2EE specifies the rules of engagement that people must agree on when writing enterprise software. Vendors then implement the J2EE specifications with their J2EE-compliant products.

Because J2EE is a specification (meant to address the needs of many companies), it is inherently not tied to one vendor; it also supports cross-platform development. This encourages vendors to compete, yielding best-of-breed products. It also has its downside, which is that incompatibilities between vendor products will arise—some problems due to ambiguities with specifications, other problems due to the human nature of competition.

J2EE is one of *three different* Java platforms. Each platform is a conceptual superset of the next smaller platform.

The Java 2 Platform, Micro Edition (J2ME) is a development platform for Java-enabled devices, such as Palm Pilots, pagers, watches, and so on. This is a restricted form of the Java language due to the inherent performance and capacity limitations of small devices.

The Java 2 Platform, Standard Edition (J2SE) contains standard Java services for applets and applications, such as input/output facilities, graphical user interface facilities, and more. This platform contains what most people use in standard Java Development Kit (JDK) programming.

The Java 2 Platform, Enterprise Edition (J2EE) takes Java's Enterprise APIs and bundles them together in a complete development platform for enterprise-class server-side deployments in Java.

The arrival of J2EE is significant because it creates a unified platform for server-side Java development. J2EE consists of the following deliverables from Sun Microsystems.

Specifications. Each enterprise API within J2EE has its own specification, which is a PDF file downloadable from <http://java.sun.com>. Each time there is a new version of J2EE, Sun locks-down the versions of each Enterprise API specification and bundles them together as the de facto versions to use when developing with J2EE. This increases code portability across vendors' products because each vendor supports exactly the same API revision. This is analogous to a company such as Microsoft releasing a new

version of Windows every few years: Every time a new version of Windows comes out, Microsoft locks-down the versions of the technologies bundled with Windows and releases them together.

Test suite. Sun provides a test suite for J2EE server vendors to test their implementations against. If a server passes the tests, Sun issues a compliance brand, alerting customers that the vendor's product is indeed J2EE-compliant. There are numerous J2EE-certified vendors, and you can read reviews of their products for free on TheServerSide.com.

Reference implementation. To enable developers to write code against J2EE as they have with the JDK, Sun provides its own free reference implementation of J2EE. Sun is positioning it as a low-end reference platform, as it is not intended for commercial use.

BluePrints Document. Each of the Enterprise APIs has a clear role in J2EE, as defined by Sun's *J2EE BluePrints* document. This document is a downloadable PDF file that describes how to use the J2EE technologies together.

The J2EE Technologies

The Java 2 Platform, Enterprise Edition is a robust suite of middleware services that make life very easy for server-side application developers. J2EE builds on the existing technologies in the J2SE. J2SE includes the base Java support and the various libraries (.awt, .net, .io) with support for both applets and applications. Because J2EE builds on J2SE, a J2EE-compliant product must not only implement all of J2EE, but must also implement all of J2SE. This means that building a J2EE product is an absolutely *huge* undertaking. This barrier to entry has resulted in significant industry consolidation in the Enterprise Java space, with a few players emerging from the pack as leaders.

We will discuss version 1.3 of J2EE, which supports EJB 2.0. Some of the major J2EE technologies are shown working together in Figure 1.8.

To understand more about the real value of J2EE, here is each API that a J2EE 1.3-compliant implementation must provide for you.

Enterprise JavaBeans (EJB). EJB defines how server-side components are written and provides a standard contract between components and the application servers that manage them. EJB is the cornerstone for J2EE and uses several other J2EE technologies.

Java Remote Method Invocation (RMI) and RMI-IIOP. RMI is the Java language's native way to communicate between distributed objects, such as two different objects running on different machines. RMI-IIOP is an extension of RMI that can be used for CORBA integration. RMI-IIOP is the official API that we use in J2EE (not RMI). We cover RMI-IIOP in Appendix A.

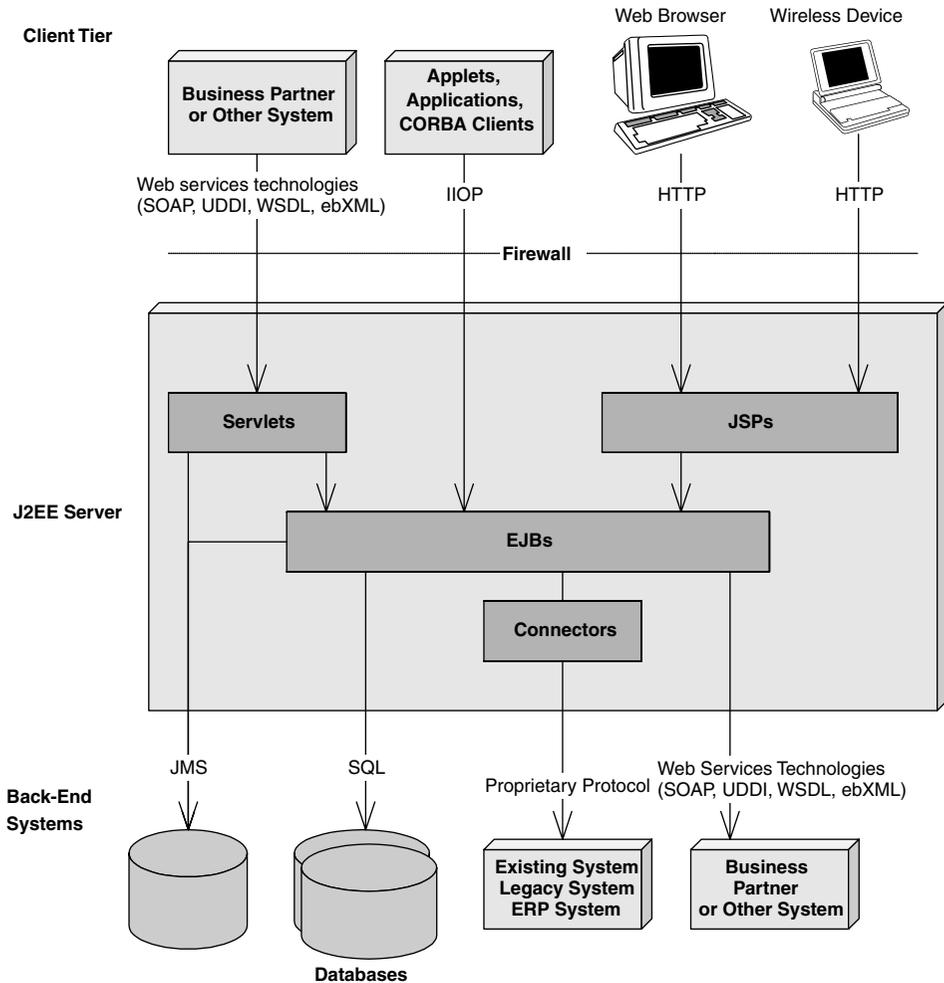


Figure 1.8 A Java 2 Platform, Enterprise Edition deployment.

Java Naming and Directory Interface (JNDI). JNDI is used to access naming and directory systems. You use JNDI from your application code for a variety of purposes, such as connecting to EJB components or other resources across the network, or accessing user data stored in a naming service such as Microsoft Exchange or Lotus Notes. JNDI is covered in Appendix A.

Java Database Connectivity (JDBC). JDBC is an API for accessing relational databases. The value of JDBC is that you can access any relational database using the same API. JDBC is used in Chapter 6.

Java Transaction API (JTA) Java Transaction Service (JTS). The JTA and JTS specifications allow for components to be bolstered with reliable transaction support. JTA and JTS are explained in Chapter 10.

Java Messaging Service (JMS). JMS allows for your J2EE deployment to communicate using messaging. You can use messaging to communicate within your J2EE system as well as outside your J2EE system. For example, you can connect to existing message-oriented middleware (MOM) systems such as IBM MQSeries or Microsoft Message Queue (MSMQ). Messaging is an alternative paradigm to RMI-IIOP, and has its advantages and disadvantages. We explain JMS in Chapter 8.

Java Servlets. Servlets are networked components that you can use to extend the functionality of a Web server. Servlets are request/response oriented in that they take requests from some client host (such as a Web browser) and issue a response back to that host. This makes servlets ideal for performing Web tasks, such as rendering an HTML interface. Servlets differ from EJB components in that the breadth of server-side component features that EJB offers is not readily available to servlets. Servlets are much better suited to handling simple request/response needs, and they do not require sophisticated management by an application server. We illustrate using Servlets with EJB in Chapter 17.

Java Pages (JSPs). JSPs are very similar to servlets. In fact, JSP scripts are compiled into servlets. The largest difference between JSP scripts and servlets is that JSP scripts are not pure Java code; they are much more centered around look-and-feel issues. You would use JSP when you want the look and feel of your deployment to be physically separate and easily maintainable from the rest of your deployment. JSPs are perfect for this, and they can be easily written and maintained by non-Java savvy staff members (JSPs do not require a Java compiler). We illustrate using JSPs with EJB in Chapter 17.

Java IDL. Java IDL is Sun Microsystems' Java-based implementation of CORBA. Java IDL allows for integration with other languages. Java IDL also allows for distributed objects to leverage CORBA's full range of services. J2EE is thus fully compatible with CORBA, completing the Java 2 Platform, Enterprise Edition. We discuss CORBA integration in Appendix B.

JavaMail. The JavaMail service allows you to send email messages in a platform-independent, protocol-independent manner from your Java programs. For example, in a server-side J2EE deployment, you can use JavaMail to confirm a purchase made on your Internet e-commerce site by sending an email to the customer. Note that JavaMail depends on the

JavaBeans Activation Framework (JAF), which makes JAF part of J2EE as well. We do not cover JavaMail in this book.

J2EE Connector Architecture (JCA). Connectors allow you to access existing enterprise information systems from a J2EE deployment. This could include *any* existing system, such as a mainframe systems running high-end transactions (such as those deployed with IBM's CICS or BEA's TUXEDO), Enterprise Resource Planning (ERP) systems, or your own proprietary systems. Connectors are useful because they automatically manage the details of middleware navigation to existing systems, such as handling transaction and security concerns. Another value of the JCA is that you can write a driver to access an existing system once, and then deploy that driver into any J2EE-compliant server. This is important because you only need to learn how to access any given existing system once. Furthermore, the driver needs to be developed only once and can be reused in any J2EE server. This is extremely useful for independent software vendors (ISVs) who want their software to be accessible from within application servers. Rather than write a custom driver for each server, the ISV can write a single driver. We discuss legacy integration more in Chapters 12 and 13.

The Java API for XML Parsing (JAXP). There are many applications of XML in a J2EE deployment. For example, you might need to parse XML if you are performing B2B interactions (such as through Web services), if you are accessing legacy systems and mapping data to and from XML, or if you are persisting XML documents to a database. JAXP is the de facto API for parsing XML documents in a J2EE deployment and is an implementation-neutral interface to XML parsers. You typically use the JAXP API from within servlets, JSPs, or EJB components. There is a free whitepaper available on TheServerSide.com that describes how to build Web services with J2EE.

The Java Authentication and Authorization Service (JAAS). JAAS is a standard API for performing security-related operations in J2EE. Conceptually, JAAS also enables you to plug in a security system to a J2EE deployment. See Chapter 9 for more details on security and EJB.

Summary

We've achieved a great deal in this chapter. First, we brainstormed a list of issues involved in a large, multitier deployment. We then understood that a server-side component architecture allows us to write complex business applications without understanding tricky middleware services. We then dove into

the EJB standard and fleshed out its value proposition. We investigated the different players involved in an EJB deployment and wrapped up by exploring J2EE.

The good news is that we're just getting started, and many more interesting and advanced topics lie ahead. The next chapter delves into the concept of *request interception*, which is the mental leap you need to make to understand EJB. Let's go!

EJB Fundamentals

Chapter 1 introduced the motivation behind EJB. In this chapter, we'll dive into EJB in detail. After reading this chapter, you will understand the different types of enterprise beans. You'll also understand what an enterprise bean component is comprised of, including the enterprise bean class, the remote interface, the local interface, the EJB object, the local object, the home interface, the home object, the deployment descriptor, and the Ejb-jar file.



EJB technology is based on two other technologies: Java RMI-IIOP and JNDI. Understanding these technologies is mandatory before continuing.

We have provided tutorials on each of these technologies in the appendices of this book. If you don't yet know RMI-IIOP or JNDI, go ahead and read Appendix A now.

Enterprise Beans

An *enterprise bean* is a server-side software component that can be deployed in a distributed multitier environment. An enterprise bean can compose one or more Java objects because a component may be more than just a simple object. Regardless of an enterprise bean's composition, the clients of the bean deal with a single exposed component interface. This interface, as well as the enterprise bean itself, must conform to the EJB specification. The specification requires that your beans expose a few required methods; these required

methods allow the EJB container to manage beans uniformly, regardless of which container your bean is running in.

Note that the client of an enterprise bean could be anything—a servlet, an applet, or even another enterprise bean. In the latter case, a client request to a bean can result in a whole chain of beans being called. This is a very powerful idea because you can subdivide a complex bean task, allowing one bean to call on a variety of prewritten beans to handle the subtasks. This hierarchical concept is quite extensible.

As a real-world example, imagine you go to a music store to purchase a compact disc. The cashier takes your credit card and runs it through a scanner. The scanner has a small Java Virtual Machine running within it, which acts as a client of enterprise beans running on a central server. The central server enterprise beans perform the following tasks:

1. Contact American Express, a Web service that itself has an EJB-compliant application server containing a number of beans. The beans are responsible for conducting the credit card transaction on behalf of that client.
2. Call a product catalog bean, which updates inventory and subtracts the quantity the customer purchased.
3. Call an order entry bean, which enters the record for the customer and returns that record locator to the scanner to give to the customer on a receipt.

As you can see, this is a powerful, flexible model, which can be extended as needed.

Types of Beans

EJB 2.0 defines three different kinds of enterprise beans:

Session beans. Session beans model business processes. They are like *verbs* because they are actions. The action could be anything, such as adding numbers, accessing a database, calling a legacy system, or calling other enterprise beans. Examples include a pricing engine, a workflow engine, a catalog engine, a credit card authorizer, or a stock-trading engine.

Entity beans. Entity beans model business data. They are like *nouns* because they are data objects—that is, Java objects that cache database information. Examples include a product, an order, an employee, a credit card, or a stock. Session beans typically harness entity beans to achieve business goals, such as a stock-trading engine (session bean) that deals with stocks (entity beans). For more examples of this, see Table 2.1.

Table 2.1 Session Beans Calling Entity Beans

SESSION BEAN	ENTITY BEAN
Bank teller	Bank account
Credit card authorizer	Credit card
DNA sequencer	DNA strand
Order entry system	Order, Line item
Catalog engine	Product
Auction broker	Bid, Item
Purchase order Approval router	Purchase order

Message-driven beans. Message-driven beans are similar to session beans in that they are actions. The difference is that you can call message-driven beans only by sending *messages* to those beans (fully described in Chapter 8). Examples of message-driven beans include beans that receive stock trade messages, credit card authorization messages, or workflow messages. These message-driven beans might call other enterprise beans as well.

You may be wondering why the EJB paradigm is so robust in offering the various kinds of beans. Why couldn't Sun come up with a simpler model? Microsoft's *n*-tier vision, for example, does not include the equivalent of entity beans—components that represent data in permanent storage.

The answer is that Sun is not the only company involved in constructing the EJB standard. Many companies have been involved, each with customers that have different kinds of distributed systems. To accommodate the needs of different enterprise applications, Sun allowed users the flexibility of each kind of bean.

Admittedly this increases the ramp-up time to learn EJB. It also adds an element of danger because some developers may misuse the intentions of each bean type. But it pays off in the long run with increased functionality. By including session beans, Sun provides a mechanism to model business processes in a distributed multitier environment. By including entity beans in the EJB specification, Sun has taken the first steps toward persistent, distributed objects usable by those business processes. And with message-driven beans, you can use messaging to access your EJB system.

See Figure 2.1 for a diagram showing some of the many possibilities of clients interacting with an EJB component system.

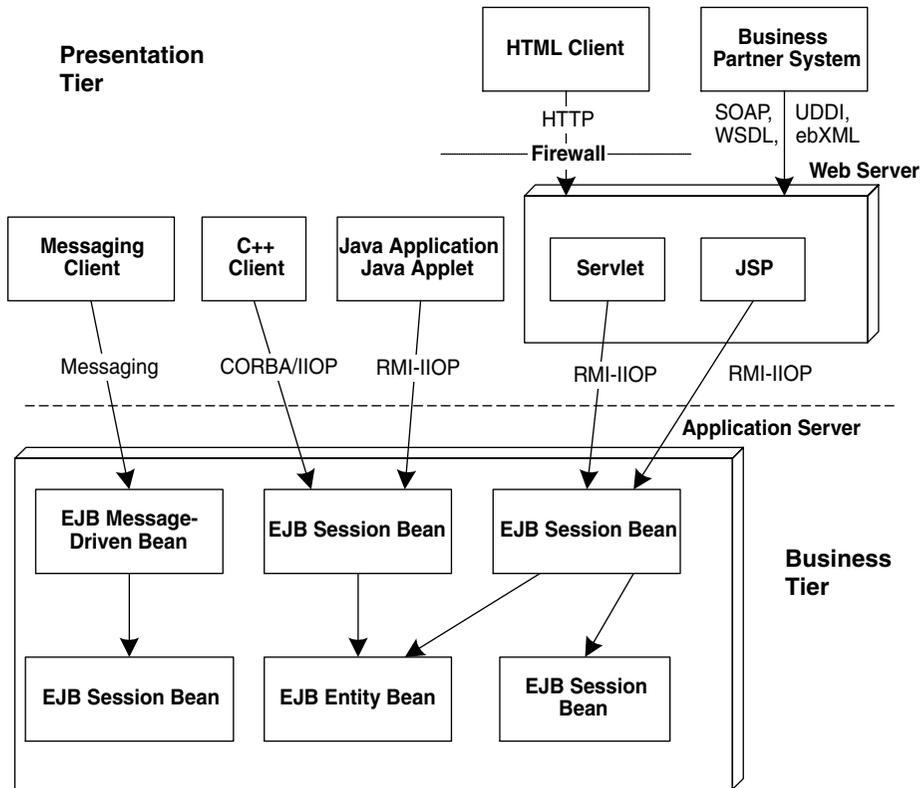


Figure 2.1 Clients interacting with an EJB component system.

Distributed Objects: The Foundation for EJB

Now that you've seen the different types of beans, let's dive into the technology behind them. EJB components are based on *distributed objects*. A distributed object is an object that is callable from a remote system. It can be called from an in-process client, an out-of-process client, or a client located elsewhere on the network.

Figure 2.2 shows how a client can call a distributed object. The following is an explanation of the diagram:

1. The client calls a *stub*, which is a *client-side proxy object*. This stub is responsible for masking network communications from the client. The stub

knows how to call over the network using sockets, massaging parameters as necessary into their network representation.

2. The stub calls over the network to a *skeleton*, which is a *server-side proxy object*. The skeleton masks network communication from the distributed object. The skeleton understands how to receive calls on a socket. It also knows how to massage parameters from their network representations to their Java representations.
3. The skeleton delegates the call to the distributed object. The distributed object does its work, and then returns control to the skeleton, which returns to the stub, which then returns control to the client.

A key point here is that both the stub and the distributed object implement the same interface (called the *remote interface*). This means the stub clones the distributed object's method signatures. A client who calls a method on the stub *thinks* he is calling the distributed object directly; in reality, the client is calling an empty stub that knows how to go over the network. This is called *local/remote transparency*.

You can achieve distributed objects using many technologies, including the OMG's CORBA, Microsoft's DCOM, and Sun's Java RMI-IIOP.

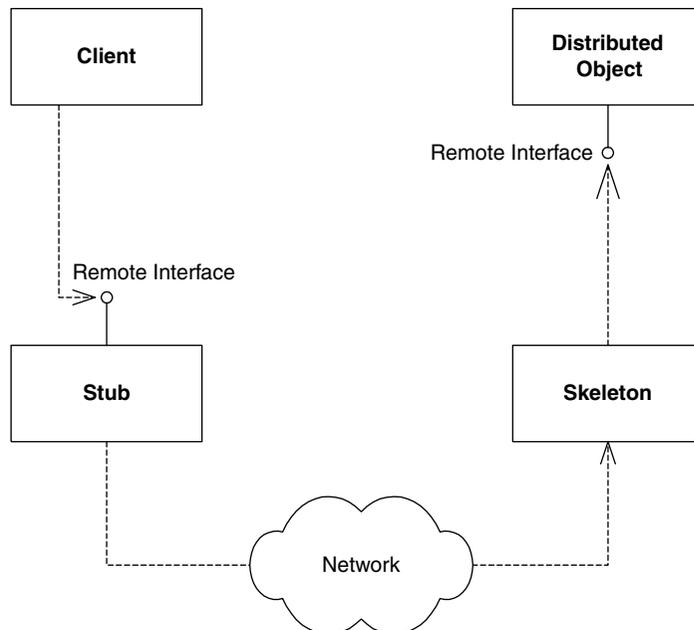


Figure 2.2 Distributed objects.

Distributed Objects and Middleware

Distributed objects are great because they allow you to break up an application across a network. However, as a distributed object application gets larger, you'll need help from middleware services, such as transactions and security. There are two ways to get middleware: explicitly and implicitly. Let's investigate both approaches.

Explicit Middleware

In traditional distributed object programming (such as traditional CORBA), you can harness middleware by purchasing that middleware off the shelf and writing code that calls that middleware API. For example, you could gain transactions by writing to a transaction API. We call this *explicit middleware* because you need to write to an API to gain that middleware. This is shown in Figure 2.3

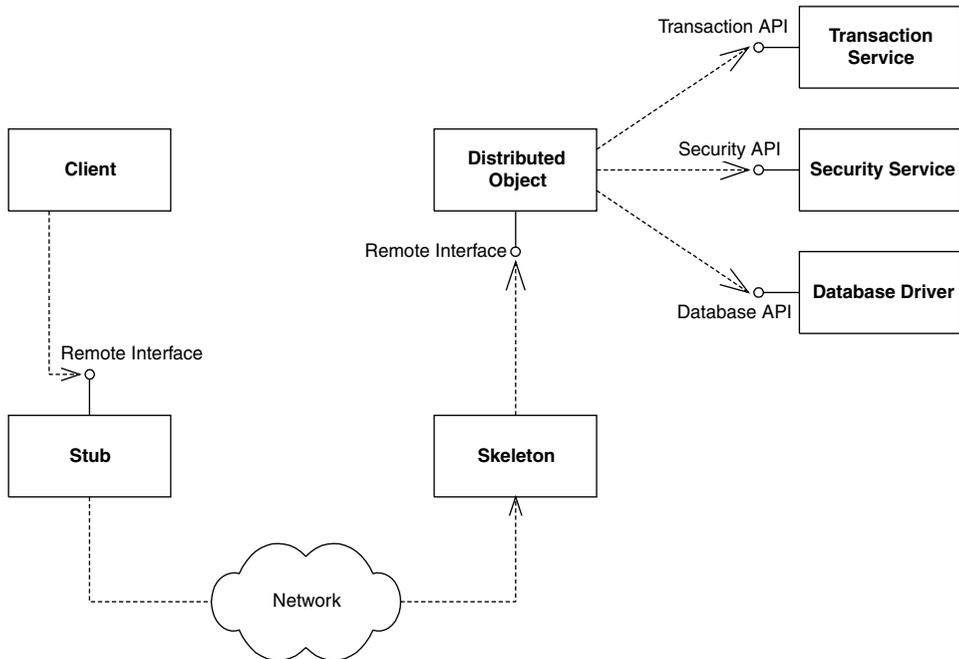


Figure 2.3 Explicit middleware (gained through APIs).

Here's a bank account distributed object that knows how to transfer funds from one account to another. It is filled with pseudo-code that illustrates explicit middleware.

```
transfer(Account account1, Account account2, long amount) {  
    // 1: Call middleware API to perform a security check  
    // 2: Call middleware API to start a transaction  
    // 3: Call middleware API to load rows from the database  
    // 4: Subtract the balance from one account, add to the other  
    // 5: Call middleware API to store rows in the database  
    // 6: Call middleware API to end the transaction  
}
```

As you can see, we are gaining middleware, but our business logic is intertwined with the logic to call these middleware APIs. The downsides to this approach are

Difficult to write. The code is bloated. We simply want to perform a transfer, but it requires a large amount of code.

Difficult to maintain. If you want to change how you do middleware, you need to rewrite your code.

Difficult to support. If you are an Independent Software Vendor (ISV) selling an application, or an internal department providing code to another department, you are unlikely to provide source code to your customers. This is because the source code is your intellectual property, and also because upgrading your customers to the next version of your software is difficult if those customers modify source code. Thus, your customers cannot change their middleware (such as changing how security works).

Implicit Middleware

The crucial difference between systems of the past (transaction processing monitors such as TUXEDO or CICS, or traditional distributed object technologies such as CORBA, DCOM, or RMI) and the newer, component-based technologies (EJB, CORBA Component Model, and Microsoft.NET) is that in this new world, you can harness complex middleware in your enterprise applications without writing to middleware APIs. This is shown in Figure 2.4, and works as follows:

1. Write your distributed object to contain *only business logic*. Do not write to complex middleware APIs. For example, this is the code that would run inside the distributed object:

```
transfer(Account account1, Account account2, long amount) {  
    // 1: Subtract the balance from one account, add to the other  
}
```

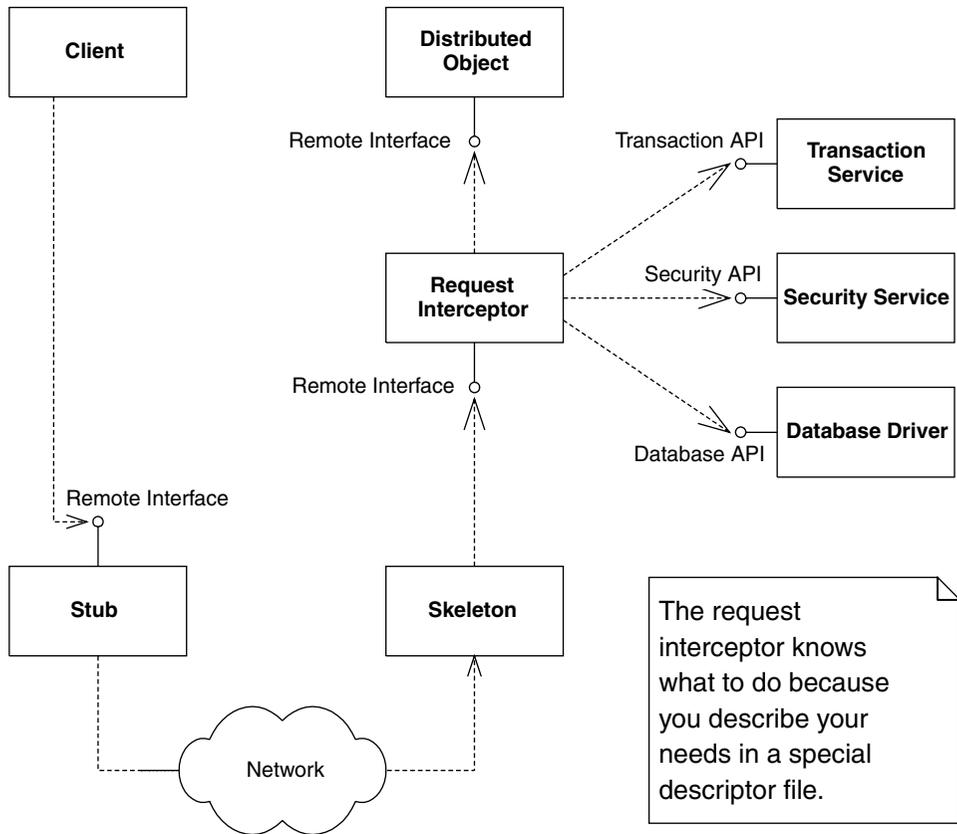


Figure 2.4 Implicit middleware (gained through declarations).

2. *Declare* the middleware services that your distributed object needs in a separate descriptor file, such as a plain text file. For example, you might declare that you need transactions, persistence, and a security check.
3. Run a command-line tool provided for you by the middleware vendor. This tool takes your descriptor file as input and generates an object that we'll call the *request interceptor*.
4. The request interceptor intercepts requests from the client, performs the middleware that your distributed object needs (such as transactions, security, and persistence), and then delegates the call to the distributed object.

The values of *implicit middleware* (also called *declarative middleware*) are:

Easy to write. You don't actually write any code to middleware APIs; rather, you declare what you need in a simple text file. The request interceptor

provides the middleware logic for you *transparently*. You focus away from the middleware and concentrate on your application's business code. This is truly divide and conquer!

Easy to maintain. The separation of business logic and middleware logic is clean and maintainable. It is less code, which makes things simpler. Furthermore, changing middleware does not require changing application code.

Easy to support. Customers can change the middleware they need by tweaking the descriptor file. For example, they can change how a security check is done without modifying source code. This avoids upgrade headaches and intellectual property issues.

What Constitutes an Enterprise Bean?

Now that we understand request interception, we can dive in and see exactly what constitutes an enterprise bean. As we will see, an enterprise bean component is not a single monolithic file—a number of files work together to make up an enterprise bean.

The Enterprise Bean Class

The first part of your bean is the implementation itself, which contains the guts of your logic, called the *enterprise bean class*. This is simply a Java class that conforms to a well-defined interface and obeys certain rules. The rules are necessary for your beans to run in any EJB container.

An enterprise bean class contains implementation details of your component. Although there are no hard-and-fast rules in EJB, session bean, entity bean, and message-driven bean implementations are all very different from each other.

For session beans, an enterprise bean class typically contains business-process-related logic, such as logic to compute prices, transfer funds between bank accounts, or perform order entry.

For entity beans, an enterprise bean class typically contains data-related logic, such as logic to change the name of a customer, reduce the balance of a bank account, or modify a purchase order.

For message-driven beans, an enterprise bean class typically contains message-oriented logic, such as logic to receive a stock trade message and call a session bean that knows how to perform stock trading.

The EJB specification defines a few standard interfaces that your bean class can implement. These interfaces force your bean class to expose certain methods that all beans must provide, as defined by the EJB component model. The EJB container calls these required methods to manage your bean and alert your bean to significant events.

The most basic interface that *all* bean classes (session, entity, and message-driven) must implement is the `javax.ejb.EnterpriseBean` interface, shown in Source 2.1.

This interface serves as a *marker* interface; implementing this interface indicates that your class is indeed an enterprise bean class. The interesting aspect of `javax.ejb.EnterpriseBean` is that it extends `java.io.Serializable`. This means that all enterprise beans can be converted to a bit-blob and share all the properties of serializable objects (described in Appendix A). This will become important later.

Session beans, entity beans, and message-driven beans each have more specific interfaces that *extend* the `javax.ejb.EnterpriseBean` interface. All session beans must implement `javax.ejb.SessionBean`; all entity beans must implement `javax.ejb.EntityBean`; and all message-driven beans must implement `javax.ejb.MessageDrivenBean`. We'll see the details of these interfaces a bit later. For now, know that your enterprise bean class never needs to implement the `javax.ejb.EnterpriseBean` interface directly; rather, your bean class implements the interface corresponding to its bean type.

The EJB Object

Enterprise beans are not full-fledged remote objects. When a client wants to use an instance of an enterprise bean class, the client never invokes the method directly on an actual bean instance. Rather, the invocation is *intercepted* by the EJB container and then *delegated* to the bean instance. This is the concept of request interception that we touched on earlier. By intercepting requests, the EJB container can automatically perform implicit middleware. As a component developer, this means your life is simplified greatly because you can rapidly develop components without writing, debugging, or maintaining

```
public interface javax.ejb.EnterpriseBean extends java.io.Serializable
{
}
```

Source 2.1 The `javax.ejb.EnterpriseBean` interface.

code that calls middleware APIs. Some of the services that you get at the point of interception include

Implicit distributed transaction management. Transactions allow for you to perform robust, deterministic operations in a distributed environment by setting attributes on your enterprise beans. We'll get into the details of transactions and how you can use them effectively in Chapter 10. For now, know that the EJB container provides a *transaction service*—a low-level implementation of transaction management and coordination. The transaction service must be exposed through the Java Transaction API (JTA). The JTA is a high-level interface that you can use to control transactions, which we also cover in Chapter 10.

Implicit security. Security is a major consideration for multitier deployments. The Java 2 Platform, Standard Edition yields a robust security service that can authorize and authenticate users, securing deployments from unwanted visitors. EJB adds to this the notion of transparent security, allowing components to reap the benefits of a secure deployment without necessarily coding to a security API.

Implicit resource management and component life cycle. The EJB container implicitly manages resources for your enterprise beans, such as threads, sockets, and database connections. The life cycle of the enterprise beans themselves is also managed, allowing the EJB container to reuse the enterprise bean instances as necessary.

Implicit persistence. Persistence is a natural requirement of any deployment that requires permanent storage. EJB offers assistance here by automatically saving persistent object data to an underlying storage and retrieving that data at a later time.

Implicit remote accessibility. Your enterprise bean class cannot be called across the network directly because an enterprise bean class is not network enabled. Your EJB container handles networking for you by wrapping your bean in a network-enabled object. The network-enabled object receives calls from clients and delegates these calls to instances of your bean class. This saves you from having to worry about networking issues (the container provides networking as a service to you). Thus EJB products automatically convert your stand-alone, networkless components into distributed, network-aware beings.

Implicit support. EJB containers automatically handle concurrent requests from clients. EJB containers provide built-in thread support, instantiating multiple copies of your component as necessary by instantiating lots of instances of your enterprise bean and pushing one thread through each instance. If multiple clients simultaneously invoke methods on a bean, the invocations are *serialized*, or performed lock step. The container will only

allow one client to call a bean at once. The other clients are routed to other bean instances of the same class, or are forced to wait. (Behind the scenes, the container might use Java thread synchronization to aid with this. The actual algorithm used is container-specific.) The value of threading is obvious—who enjoys writing multithreaded code?

Implicit component location transparency. Clients of components are decoupled from the specific whereabouts of the component being used.

Implicit monitoring. The EJB container can track which methods are invoked, display a real-time usage graph on a system administrator’s user interface, gather data for intelligent load balancing, and more. An EJB container is not required to perform these tasks; however, high-end EJB containers perform these tasks at the point of interception.

Thus, the EJB container acts as a layer of indirection between the client code and the bean. This layer of indirection manifests itself as a single network-aware object called the *EJB object*. The EJB object is the request interceptor we alluded to earlier. As the old saying goes, a layer of indirection solves every problem in computer science.

The EJB object is a surrogate object that knows about networking, transactions, security, and more. It is an intelligent object that knows how to perform intermediate logic that the EJB container requires before a method call is serviced by a bean class instance. An EJB object is the request interceptor, or the glue, between the client and the bean. EJB objects replicate and expose every business method that the bean itself exposes. EJB objects delegate all client requests to beans. We depict EJB objects in Figure 2.5.

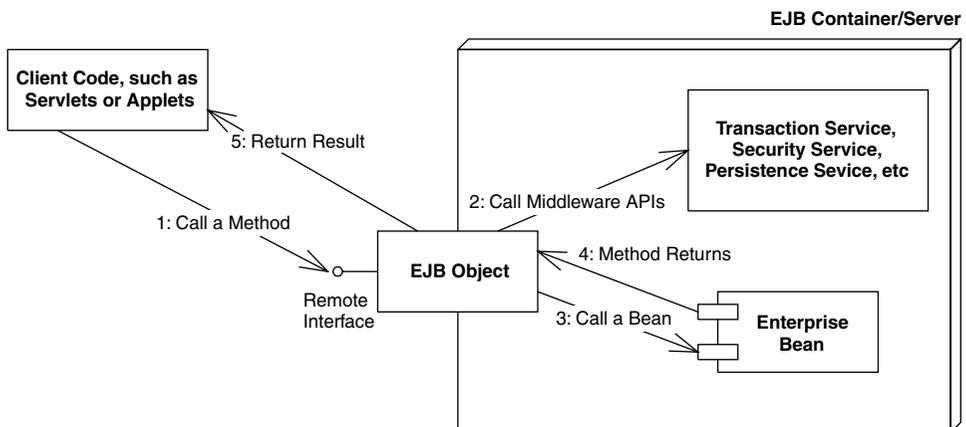


Figure 2.5 EJB objects.

You should think of EJB objects as physical parts of the container; all EJB objects have container-specific code inside of them. (Each container handles middleware differently and provides different qualities of service.) Because each bean's EJB object is different, your container vendor *generates* the class file for your EJB objects automatically.

Each EJB container ships with a suite of *glue-code tools*. These tools are meant to integrate beans into the EJB container's environment. The tools generate helper Java code—stubs, skeletons, data access classes, and other classes that this specific container requires. Bean providers do not have to think about the specifics of how each EJB container works because the container's tools generate its own proprietary Java code automatically.

The container's glue-code tools are responsible for transforming an enterprise bean into a fully managed, distributed server-side component. This involves logic to handle resource management, life cycle, state management, transactions, security, persistence, remote accessibility, and many

The EJB Container: Your Silent Partner

EJB containers are responsible for managing your beans. Containers can interact with your beans by calling your beans' required management methods as necessary. These management methods are your beans' callback methods that the container, and only the container, invokes. The management methods allow the container to alert your beans when middleware events take place, such as when an entity bean is about to be persisted to storage.

The most important responsibility of an EJB container is to provide an environment in which enterprise beans can run. EJB containers house the enterprise beans and make them available for clients to invoke remotely. In essence, EJB containers act as invisible middlemen between the client and the beans. They are responsible for connecting clients to beans, performing transaction coordination, providing persistence, managing a bean's life cycle, and other tasks.

The key to understanding EJB containers is to realize that they are *abstract entities*. Neither the beans nor the clients that call beans ever explicitly code to the API of an EJB container. Rather, the container implicitly manages the overhead of a distributed component architecture. The container is analogous to a behind-the-scenes stage manager in a theater, providing the lighting and backdrop necessary for a successful stage performance by the actors on stage. Neither the actors nor the audience interact directly with the stage manager. The same is true for EJB containers. Clients that call the beans never code directly to an EJB container API.

other services. The generated code handles these services in the container's proprietary way.

The Remote Interface

As mentioned previously, bean clients invoke methods on EJB objects, rather than the beans themselves. Therefore, EJB objects must clone every business method that your bean classes expose. But how do the tools that autogenerate EJB objects know which methods to clone? The answer is in a special interface that a bean provider writes. This interface duplicates all the business logic methods that the corresponding bean class exposes. This interface is called the *remote interface*.

Remote interfaces must comply with special rules that the EJB specification defines. For example, all remote interfaces must derive from a common interface supplied by Sun Microsystems. This interface is called *javax.ejb.EJBObject*, and it is shown in Source 2.2.

javax.ejb.EJBObject lists a number of interesting methods. For now, don't worry about fully understanding the meanings—just know that these are required methods that all EJB objects must implement. And remember that *you* don't implement the methods—the EJB container does when it autogenerates the EJB objects for you.

```
public interface javax.ejb.EJBObject
extends java.rmi.Remote
{
    public javax.ejb.EJBHome getEJBHome()
        throws java.rmi.RemoteException;

    public java.lang.Object getPrimaryKey()
        throws java.rmi.RemoteException;

    public void remove()
        throws java.rmi.RemoteException,
        javax.ejb.RemoveException;

    public javax.ejb.Handle getHandle()
        throws java.rmi.RemoteException;

    public boolean isIdentical(javax.ejb.EJBObject)
        throws java.rmi.RemoteException;
}
```

Source 2.2 A preview of the *javax.ejb.EJBObject* interface.

The client code that wants to work with your beans calls the methods in *javax.ejb.EJBObject*. This client code could be stand-alone applications, applets, servlets, or anything at all—even other enterprise beans.

In addition to the methods listed in Source 2.2, your remote interface duplicates your beans' business methods. When a bean's client invokes any of these business methods, the EJB object delegates the method to its corresponding implementation, which resides in the bean itself.

The Instance-Pooling Concept

A multitier architecture's overall scalability is enhanced when an application server intelligently manages needed resources across a variety of deployed components. The resources could be threads, socket connections, database connections, and more. For example, database connections could be pooled by application servers and reused across heterogeneous components. In the EJB realm, the container is responsible for providing all resource management services behind the scenes.

In addition to resource management, the EJB container is responsible for controlling the life cycle of the deployed enterprise bean components. As bean client requests arrive, the EJB container dynamically instantiates, destroys, and reuses beans as appropriate. For example, if a client requests a certain type of bean that does not yet exist in memory, the EJB container may instantiate a new in-memory instance on behalf of the client. On the other hand, if a bean already exists in memory, it may not be appropriate to instantiate a new bean, especially if the system is low on memory. It might make more sense to reassign a bean from one client to another instead. It might also make sense to destroy some beans that are not being used anymore. This is called *instance pooling*.

The benefit of bean instance pooling is that the pool of beans can be much smaller than the actual number of clients connecting. This is due to client think time, such as network lag or human decision time on the client side. The classic example of this is an HTML (Web) client interacting with a human being. Web users often click a button that executes some business logic in a component, but then read text before initiating another action. While the user is waiting and reading, the application server could reuse that component to service other clients. While the client is thinking, the container can use the bean instances to service other clients, saving previous system resources.

The take-away point here is that the EJB container is responsible for coordinating the entire effort of resource management as well as managing the deployed beans' life cycle. Note that the exact scheme used is EJB container-specific.

Java RMI-IIOP and EJB Objects

You may have noticed that *javax.ejb.EJBObject* extends *java.rmi.Remote*. The *java.rmi.Remote* interface is part of Java Remote Method Invocation over the Internet Inter-ORB Protocol (RMI-IIOP). Any object that implements *java.rmi.Remote* is a *remote object* and is callable from a different Java Virtual Machine. This is how remote method invocations are performed in Java. (We fully describe this in Appendix A).

Because the EJB object provided by the container implements your remote interface, it also indirectly implements *java.rmi.Remote*. Your EJB objects are fully networked RMI-IIOP objects, able to be called from other Java Virtual Machines or physical machines located elsewhere on the network. Thus, EJB remote interfaces are really just RMI-IIOP remote interfaces—except that EJB remote interfaces must also be built to conform to the EJB specification.

EJB remote interfaces must conform to RMI-IIOP's remote interface rules. For example, any method that is part of a remote object callable across virtual machines must throw a special *remote exception*. A remote exception is a *java.rmi.RemoteException*, or (technically) a subclass of it. A remote exception indicates that something unexpected happened on the network while you were invoking across virtual machines, such as a network, process, or machine failure. Every method shown in Source 2.2 for *javax.ejb.EJBObject* throws a *java.rmi.RemoteException*.

Remote interfaces must conform to RMI-IIOP's parameter-passing conventions as well. Not everything can be passed over the network in a cross-VM method call. The parameters you pass in methods must be valid types for RMI-IIOP. This includes primitives, serializable objects, and RMI-IIOP remote objects. The full details of what you can pass are in Appendix A.

The Home Object

As we've seen, client code deals with EJB objects and never with beans directly. The next logical question is, how do clients acquire references to EJB objects?

The client cannot instantiate an EJB object directly because the EJB object can exist on a different machine than the one the client is on. Similarly, EJB promotes location transparency, so clients should never be aware of exactly where an EJB object resides.

To acquire a reference to an EJB object, your client code asks for an EJB object from an EJB object *factory*. This factory is responsible for instantiating (and destroying) EJB objects. The EJB specification calls such a factory a *home object*. The chief responsibilities of home objects are the following:

Location Transparency

EJB inherits a significant benefit from RMI-IIOP. In RMI-IIOP, the physical location of the remote object you're invoking on is masked from you. This feature spills over to EJB. Your client code is unaware of whether the EJB object it is using is located on a machine next door or a machine across the Internet. It also means the EJB object could be located on the *same* Java VM as the client. This is called *location transparency*.

Why is location transparency beneficial? For one thing, you aren't writing your bean's client code to take advantage of a particular deployment configuration because you're not hard-coding machine locations. This is an essential part of reusable components that can be deployed in a wide variety of multitier situations.

Location transparency also enables container vendors to provide additional value-adds, such as the ability to take down a machine on the network temporarily to perform system maintenance, install new software, or upgrade components on that machine. During maintenance, location transparency allows another machine on the network to serve up components for a component's client because that client is not dependent on the hard locations of any components. If a machine that has components on it crashes due to hardware or software error, you may be able to reroute client invocations to other machines without the client even knowing about the crash, allowing for an enhanced level of fault tolerance.

- Create EJB objects
- Find existing EJB objects (for entity beans, which we'll learn about in Chapter 5)
- Remove EJB objects

Just like EJB objects, home objects are proprietary and specific to each EJB container. They contain interesting container-specific logic, such as load-balancing logic, logic to track information on a graphical administrative console, and more. And just like EJB objects, home objects are physically part of the container and are autogenerated by the container vendor's tools.

The Home Interface

We've seen that home objects are factories for EJB objects. But how does a home object know how you'd like your EJB object to be initialized? For example, one

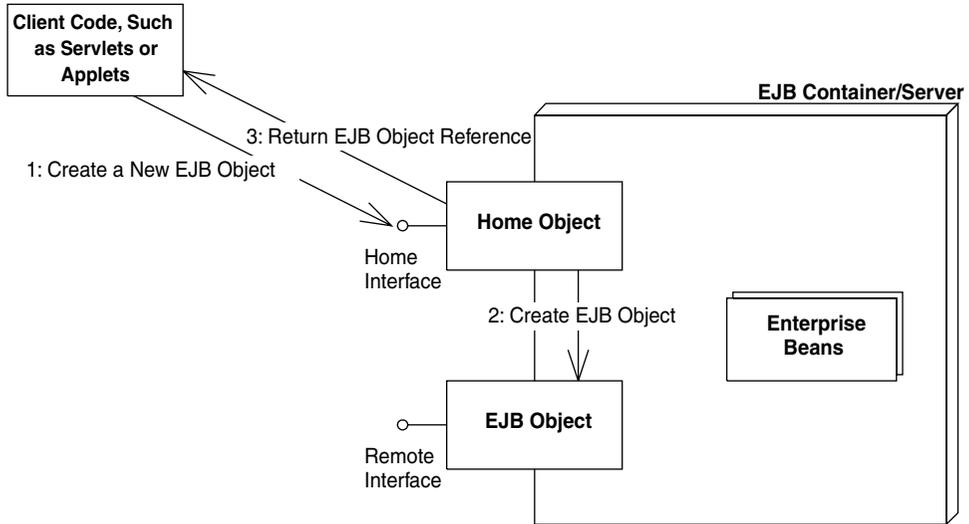


Figure 2.6 Home interfaces and objects.

EJB object might expose an initialization method that takes an integer as a parameter, and another EJB object might take a string instead. The container needs to know this information to generate home objects. You provide this information to the container by specifying a *home interface*. Home interfaces simply define methods for creating, destroying, and finding EJB objects. The container's home object *implements* your home interface. We show this in Figure 2.6.

As usual, EJB defines some required methods that all home interfaces must support. These required methods are defined in the `javax.ejb.EJBHome` interface—an interface that your home interfaces must extend. We show `javax.ejb.EJBHome` in Source 2.3. We will learn about these methods later.

Notice that the parent `javax.ejb.EJBHome` derives from `java.rmi.Remote`. This means your home interfaces do as well, implying that home objects are also fully networked Java RMI remote objects, which can be called across VMs. The types of parameters passed in the home interface's methods must be valid types for Java RMI-IIOP.

The Local Interfaces

One problem with the home interface is that creating beans through that interface is very slow. The same is true for calling beans through the remote interface. Just to give you an idea of what happens when you call an EJB object, the following steps may occur:

1. The client calls a local stub.
2. The stub marshals parameters into a form suitable for the network.
3. The stub goes over a network connection to the skeleton.
4. The skeleton demarshals parameters into a form suitable for Java.
5. The skeleton calls the EJB object.
6. The EJB object performs needed middleware, such as connection pooling, transactions, security, and lifecycle services.
7. Once the EJB object calls the enterprise bean instance, and the bean does its work, each of the preceding steps must be repeated for the return trip home.

Ouch! That's a lot of overhead. Figure 2.4 shows this process.

New to EJB 2.0, you can now call enterprise beans in a fast, efficient way by calling them through their *local objects* rather than *EJB objects*. Local objects implement a *local interface* rather than a *remote interface*. The local objects are speed demons that allow you to make high-performance enterprise beans. The process works as follows:

1. The client calls a local object.
2. The local object performs needed middleware, such as connection pooling, transactions, security, and lifecycle services.
3. Once the enterprise bean instance does its work, it returns control to the local object, which then returns control to the client.

As you can see, we avoid the steps of the stub, skeleton, network, and marshaling/demarshaling of parameters. This empowers us to write smaller beans that perform more fine-grained tasks, without fear of a performance hit at each and every cross-bean method call.

You can create beans in a fast way as well. Rather than using the home interface and home object, you can call a special *local home interface*, which is implemented by the container as the *local home object*.

These local interfaces are entirely optional; you can use them as a replacement or as a complement to the remote interfaces. For simplicity, in the remainder of this book, we will use the word EJB object to mean the request interceptor, the remote interface to mean the interface to the request interceptor, the home object to mean the factory, and the home interface to mean the factory interface. Unless it's pointed out explicitly, all information that applies to these remote interfaces and remote objects also apply to their local counterparts. Also note that the EJB specification has defined the term *component interface* to

```
public interface javax.ejb.EJBHome extends java.rmi.Remote
{
    public EJBMetaData getEJBMetaData()
        throws java.rmi.RemoteException;

    public javax.ejb.HomeHandle getHomeHandle()
        throws java.rmi.RemoteException;

    public void remove(javax.ejb.Handle handle)
        throws java.rmi.RemoteException,
            javax.ejb.RemoveException;

    public void remove(Object primaryKey)
        throws java.rmi.RemoteException,
            javax.ejb.RemoveException;
}
```

Source 2.3 A preview of the `javax.ejb.EJBHome` interface.

Relationship between Home Objects, EJB Objects, and Bean Instances

One question we frequently are asked in our EJB training courses is “How many home objects are there for each bean?” The answer to this question is vendor-specific. Most containers will have a 1:N relationship between home objects and bean instances. This means that all clients use the same home object instance to create EJB objects. The home object will probably be written to be thread-safe so that it can service many client requests concurrently. It is perfectly fine for the container to do this because the container itself is multithreaded (only your beans are single-threaded).

Another question we typically get is “How many EJB object instances are there for each bean instance?” Some containers can have a 1:N relationship, where each EJB object is multithreaded (just like home objects). Other containers might have an M:N relationship, where M represents the number of EJB objects instantiated (and corresponds exactly to the number of clients currently connected), and N represents the number of bean instances in the pool. In this case, each EJB object is single-threaded.

None of this really matters to you as a bean provider because you should think of the container as a black box. However, it’s sometimes fun to know what’s going on behind the scenes in case low-level debugging is required.

mean either the remote interface or local interface. We will occasionally use this term in this book.

When you write a local interface, you extend *javax.ejb.EJBLocalObject*, and when you write a local home interface, you extend *javax.ejb.EJBLocalHome*. Those interfaces are previewed in the following code, and are fully explained in Appendix E.

```
public interface javax.ejb.EJBLocalObject {
    public javax.ejb.EJBLocalHome getEJBLocalHome()
        throws javax.ejb.EJBException;

    public Object getPrimaryKey()
        throws javax.ejb.EJBException;

    public boolean isIdentical(javax.ejb.EJBLocalObject)
        throws javax.ejb.EJBException;

    public void remove()
        throws javax.ejb.RemoveException, javax.ejb.EJBException;
}

public interface javax.ejb.EJBLocalHome {
    public void remove(java.lang.Object)
        throws javax.ejb.RemoveException, javax.ejb.EJBException;
}
```



Local interfaces have two important side effects:

1. They only work when you're calling beans in the same process—for example, if you have a bank teller session bean that calls a bank account entity bean in the same application server. But there lies the rub. You cannot call a bean remotely if your code relies on the local interface. If you decide to switch between a local or remote call, you must change your code from using the local interface to using the remote interface. This is an inherent drawback to local interfaces.
2. They marshal parameters by reference rather than by value. While this may speed up your application because parameters are not copied, it also changes the semantics of your application. Be sure that you're aware of this when coding your clients and beans.

For a while, the primary author of this book (Ed Roman) has been pushing for Sun to adopt some kind of flag that enables you to switch between local and remote access to beans without changing code. The idea is that this flag would determine whether the container-generated interceptor object would behave as a local object or remote object. We think this is the best approach because (in reality) many developers will misjudge whether to use remote or local interfaces when designing their object models, and will have to rewrite parts of their code later in their projects.

The response so far from Sun is that this approach would not work because the semantics of the application change when switching between local interfaces and remote interfaces, due to the differences in pass-by-value versus pass-by-reference. It would be error-prone to allow developers to “flip a switch” in this respect.

Personally, we don't agree with Sun. We think developers are smart enough to avoid these mistakes, and the potential benefits outweigh the drawbacks. Many EJB server vendors disagree as well. They actually support this local/remote flag idea through proprietary container tools or vendor-specific files that are separate from your bean. Thus, if you want to, you may be able to still take advantage of these flags without sacrificing portability.

Deployment Descriptors

To inform the container about your middleware needs, you as a bean provider must *declare* your components' middleware service requirements in a *deployment descriptor* file. For example, you can use a deployment descriptor to declare how the container should perform lifecycle management, persistence, transaction control, and security services. The container inspects the deployment descriptor to fulfill the requirements that you lay out. The deployment descriptor is the key to implicit middleware.

For example, you can use a deployment descriptor to specify the following requirements of your bean.

Bean management and lifecycle requirements. These deployment descriptor settings indicate how the container should manage your beans. For example, you specify the name of the bean's class, whether the bean is a session, entity, or message-driven bean, and the home interface that generates the beans.

Persistence requirements (entity beans only). Authors of entity beans use the deployment descriptors to inform the container about whether the bean handles its persistence on its own or delegates the persistence to the EJB container in which it's deployed.

Transaction requirements. You can also specify transaction settings for beans in deployment descriptors. These settings specify the bean requirements for running in a transaction, such as a transaction must start whenever anyone calls this bean, and the transaction must end after my bean completes the method call.

Security requirements. Deployment descriptors contain *access control entries*, which the beans and container use to enforce access to certain operations. For example, you can specify who is allowed to use which beans, and even who is allowed to use each method on a particular bean. You can also spec-

ify what security roles the beans themselves should run in, which is useful if the beans need to perform secure operations. For example only bank executives can call the method to create new bank accounts.

In EJB 2.0, a deployment descriptor is an XML file. You can write these XML files by hand, or (if you're lucky) your Integrated Development Environment (IDE) or EJB container will supply tools to generate the XML deployment descriptor. In the latter case, you simply might need to step through a wizard in a Java IDE to generate a deployment descriptor.

As a bean provider, you are responsible for creating a deployment descriptor. Once your bean is used, other parties can modify its deployment descriptor settings. For example, an application assembler who is piecing together an application from beans can tune your deployment descriptor. Similarly, a deployer who is installing your beans in a container in preparation for a deployment to go live can tune your deployment descriptor settings as well. This is all possible because deployment descriptors *declare* how your beans should use middleware, rather than you writing code that uses middleware. Declaring rather than programming enables people without Java knowledge and without source code access to tweak your components at a later time. This paradigm becomes an absolute necessity when purchasing EJB components from a third party because third-party source code is typically not available. By having a separate, customizable deployment descriptor, you can easily fine-tune components to a specific deployment environment without changing source code.

Vendor-Specific Files

Since all EJB server vendors are different, they each have some proprietary value-added features. The EJB specification does not touch these features, such as how to configure load-balancing, clustering, monitoring, and so on. Therefore, each EJB server vendor may require that you include additional files specific to that vendor, such as XML files, text files, or binary files.

Ejb-Jar File

Once you've generated your bean classes, your home interfaces, your remote interfaces, and your deployment descriptor, it's time to package them into an *Ejb-jar file*. An Ejb-jar file is a compressed file that contains everything we have described, and it follows the .ZIP compression format. Jar files are convenient, compact modules for shipping your Java software. The Ejb-jar file creation process is shown in Figure 2.7.

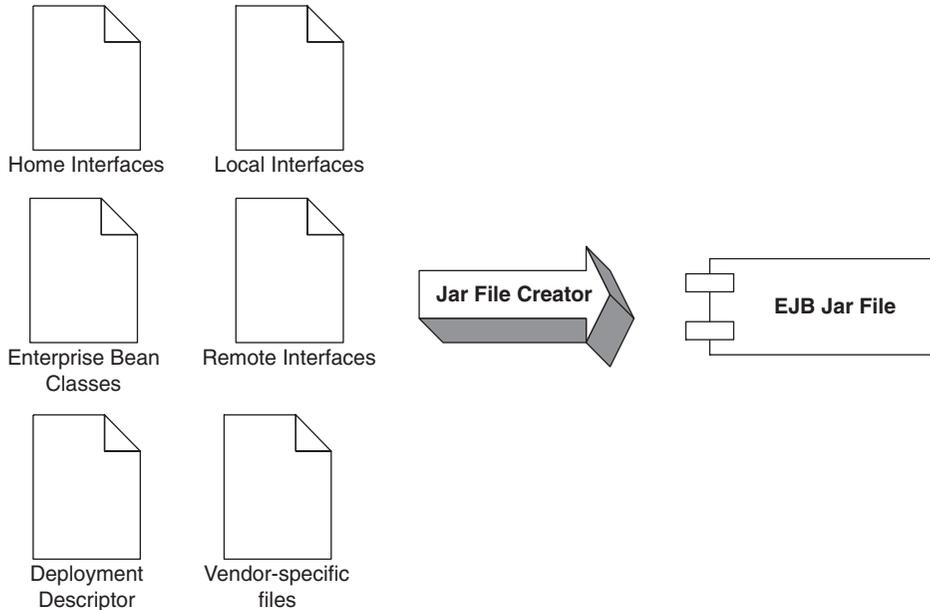


Figure 2.7 Creating an Ejb-jar file.

There are already a number of tools available to autogenerate Ejb-jar files, such as Java IDEs. You can also generate these files yourself—we'll show you how in Chapter 3.

Once you've made your Ejb-jar file, your enterprise bean is complete, and it is a deployable unit within an application server. When they are deployed (perhaps after being purchased), the tools that EJB container vendors supply are responsible for decompressing, reading, and extracting the information contained within the Ejb-jar file. From there, the deployer has to perform vendor-specific tasks, such as generating EJB objects, generating home objects, importing your bean into the container, and tuning the bean. Support for Ejb-jar files is a standard, required feature for all EJB tools.

Note that you can have more than one bean in an Ejb-jar file, allowing you to ship an entire product set of beans in a single jar file.

Summary of Terms

For your convenience, we now list the definitions of each term we've described so far. As you read future chapters, refer to these definitions whenever you need clarification. You may want to bookmark this page.

The enterprise bean instance is a Java object instance of an enterprise bean class. It contains business method implementations of the methods defined in the remote and/or local interface. The enterprise bean instance is networkless in that it contains no networked logic.

The remote interface is a Java interface that enumerates the business methods exposed by the enterprise bean class. In EJB, client code always goes through the remote interface and never interacts with the enterprise bean instance. The remote interface is network-aware in that the interface obeys the rules for Java RMI-IIOP.

The local interface is the high-performing version of the remote interface. Use the local interface when you are calling enterprise beans that live in the same process. Your calls will not undergo stubs, skeletons, network calls, or marshaling/demarshaling of parameters.

The EJB object is the container-generated implementation of the remote interface. The EJB object is a network-aware intermediary between the client and the bean instance, handling necessary middleware issues. All client invocations go through the EJB object. The EJB object delegates calls to enterprise bean instances and implements the remote interface.

The local object is the high-performing version of the EJB object. The local object implements the local interface.

The home interface is a Java interface that serves as a factory for EJB objects. Client code that wants to work with EJB objects must use the home interface to generate them. The home interface is network-aware because clients use it across the network.

The local home interface is the high-performing version of the home interface.

The home object is the container-generated implementation of the home interface. The home object is also network-aware, and it obeys RMI-IIOP's rules.

The local home object is the high-performing version of the home object. The local home object implements the local home interface.

The deployment descriptor is an XML file that specifies the middleware requirements of your bean. You use the deployment descriptor to inform the container about the implicit middleware you want, such as how to manage your bean, your bean's lifecycle needs, your transactional needs, your persistence needs, and your security needs.

The vendor-specific files allow you to take advantage of vendor-specific features. These files are not portable between application servers.

The **Ejb-jar file** is the finished, complete .ZIP file that contains the above files. It is the unit of deployment and is given to the application server. The application server unpacks the Ejb-jar file and loads the bean.

Summary

In this chapter, we've taken a whirlwind tour of EJB. We started by looking at what a bean is, and then discussed the different kinds of beans, including session, entity, and message-driven beans.

We then took a bean apart into its constituent pieces, and examined each part: the enterprise bean class, remote interface, local interface, EJB object, local object, home interface, home object, deployment descriptor, and Ejb-jar file.

Congratulations are in order—you've made it to the end of Part II! In these chapters, you've taken the first steps necessary to understanding and mastering EJB. Now that you understand the high-level concepts, let's learn how to write and use each type of EJB component, starting with a simple Hello World example.

Writing Your First Bean

In this chapter, we'll get down-and-dirty and write a real working EJB component. Our stateless session bean will be responsible for the mighty task of returning the string "Hello, World!" to the client. We'll see how to write each of the files that make up this bean and how to access it from clients.

This chapter is great for you if you want to discover how to get up and running with EJB quickly. While this may not be the most functional demonstration of the power of EJB, it illustrates the basics of EJB programming and is a useful template for building more complex beans. This will give you the necessary foundation to understand later chapters on entity beans, session beans, and message-driven beans.

How to Develop an EJB Component

When building an EJB component, the following is a typical order of operations:

1. Write the .java files that compose your bean: the component interfaces, home interfaces, enterprise bean class file, and any helper classes you might need.
2. Write the deployment descriptor.
3. Compile the .java files from step 1 into .class files.

4. Using the *jar* utility, create an Ejb-jar file containing the deployment descriptor and .class files.
5. Deploy the Ejb-jar file into your container in a vendor-specific manner, perhaps by running a vendor-specific tool or perhaps by copying your Ejb-jar file into a folder where your container looks to load Ejb-jar files.
6. Configure your EJB server so that it is properly configured to host your Ejb-jar file. You might tune things such as database connections, thread pools, and so on. This step is vendor-specific and might be done through a Web-based console or by editing a configuration file.
7. Start your EJB container and confirm that it has loaded your Ejb-jar file.
8. Optionally, write a standalone test client .java file. Compile that test client into a .class file. Run the test client from the command line and have it exercise your bean's APIs.

We will apply the above process to our Hello World example. The complete build scripts are available with the book's accompanying source code.

The object model for our Hello World example is shown in Figure 3.1.

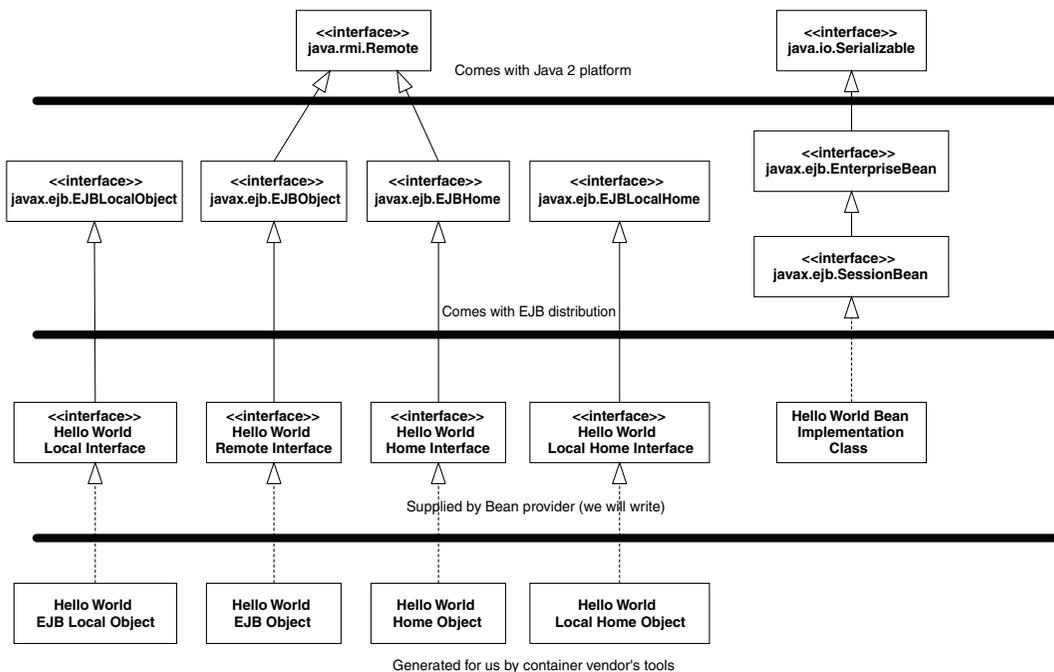


Figure 3.1 Our Hello World object model.

The Remote Interface

First, let's code up the remote interface. The remote interface duplicates every business method that our beans expose. The code is shown in Source 3.1.

Things to notice about our remote interface include the following:

- We extend *javax.ejb.EJBObject*. This means the container-generated EJB object, which implements the remote interface, will contain every method that the *javax.ejb.EJBObject* interface defines. This includes a method to compare two EJB objects, a method to remove an EJB object, and so on.
- We have one business method—*hello()*—which returns the String "Hello, World!" back to the client. We need to implement this method in our enterprise bean class. Because the remote interface is an RMI-IIOP remote interface (it extends *java.rmi.Remote*), it must throw a remote exception. This is the only difference between the remote interface's *hello()* signature and our bean's *hello()* signature. The exception indicates a networking or other critical problem.

```
package examples;

/**
 * This is the HelloBean remote interface.
 *
 * This interface is what clients operate on when
 * they interact with EJB objects. The container
 * vendor will implement this interface; the
 * implemented object is the EJB object, which
 * delegates invocations to the actual bean.
 */
public interface Hello extends javax.ejb.EJBObject
{

    /**
     * The one method - hello - returns a greeting to the client.
     */
    public String hello() throws java.rmi.RemoteException;
}
```

Source 3.1 Hello.java.

The Local Interface

Local clients will use our local interface, rather than remote interface, to call our beans' methods. It is shown in Source 3.2.

As you can see, there are trivial differences between the local interface and the remote interface. We extend a different interface, and we don't throw remote exceptions.

The Home Interface

Next, let's put together the home interface. The home interface has methods to create and destroy EJB objects. The implementation of the home interface is the home object, which is generated by the container tools.

The code for our home interface is shown in Source 3.3.

Notice the following about our home interface:

- The single *create()* is a factory method that clients use to get a reference to an EJB object. The *create()* method is also used to initialize a bean.

```
package examples;

/**
 * This is the HelloBean local interface.
 *
 * This interface is what local clients operate
 * on when they interact with EJB local objects.
 * The container vendor will implement this
 * interface; the implemented object is the
 * EJB local object, which delegates invocations
 * to the actual bean.
 */
public interface HelloLocal extends javax.ejb.EJBLocalObject
{

    /**
     * The one method - hello - returns a greeting to the client.
     */
    public String hello();
}
```

Source 3.2 HelloLocal.java.

```
package examples;

/**
 * This is the home interface for HelloBean. This interface
 * is implemented by the EJB Server's tools - the
 * implemented object is called the Home Object, and serves
 * as a factory for EJB Objects.
 *
 * One create() method is in this Home Interface, which
 * corresponds to the ejbCreate() method in HelloBean.
 */
public interface HelloHome extends javax.ejb.EJBHome
{
    /**
     * This method creates the EJB Object.
     *
     * @return The newly created EJB Object.
     */
    Hello create() throws java.rmi.RemoteException,
        javax.ejb.CreateException;
}
```

Source 3.3 HelloHome.java.

- The *create()* method throws a *java.rmi.RemoteException* and *javax.ejb.CreateException*. Remote exceptions are necessary side effects of RMI-IIOP because the home object is a networked RMI-IIOP remote object. The *CreateException* is also required in all *create()* methods. We explain this further in the following sidebar.
- Our home interface extends *javax.ejb.EJBHome*. This is required for all home interfaces. *EJBHome* defines a way to destroy an EJB object, so we don't need to write that method signature.

The Local Home Interface

Our local home interface, the higher-performing home interface used by local clients, is in Source 3.4

The differences between the remote interface and local interface are as follows:

- The local home interface extends *EJBLocalHome* rather than *EJBHome*. The *EJBLocalHome* interface does not extend *java.rmi.Remote*. This means that the generated implementation will not be a remote object.
- The local home interface does not throw *RemoteExceptions*.

```
package examples;

/**
 * This is the local home interface for HelloBean.
 * This interface is implemented by the EJB Server's
 * tools - the implemented object is called the
 * local home object, and serves as a factory for
 * EJB local objects.
 */
public interface HelloLocalHome extends javax.ejb.EJBLocalHome
{

    /**
     * This method creates the EJB Object.
     *
     * @return The newly created EJB Object.
     */
    HelloLocal create() throws javax.ejb.CreateException;
}
```

Source 3.4 HelloLocalHome.java.

Exceptions and EJB

Every networked object in EJB conforms to the RMI-IIOP standard and must throw a remote exception. Thus, every method in an EJB object and home object (such as our *hello()* method) must throw a remote exception. When such an exception is thrown, it indicates a special error condition—a network failure, machine failure, or other catastrophic failure.

But how can your beans throw exceptions that indicate regular, run-of-the-mill problems, such as bad parameters passed to a business method? EJB comes with some built-in exceptions to handle this, and it also allows you to define your own exception types.

More formally, EJB defines the following exception types:

1. A *system-level exception* is a serious error that involves some critical failure, such as a database malfunction.
2. An *application-level exception* is a more routine exception, such as an indication of bad parameters to a method or a warning of an insufficient bank account balance to make a withdrawal. For example, in our “Hello, World!” home interface, we throw a standard *javax.ejb.CreateException* from home

interface's *create()* method. This is an example of a required application-level exception, indicating that some ordinary problem occurred during bean initialization.

Why must we separate the concepts of system-level and application-level exceptions? The chief reason is that system-level exceptions are handled quite differently from application-level exceptions.

For example, system-level exceptions are not necessarily thrown back to the client. Remember that EJB objects—the container-generated wrappers for beans—are middlemen between a bean's client and the bean itself. EJB objects have the ability to *intercept* any exceptions that beans may throw. This allows EJB objects to pick and choose which exceptions the client should see. In some cases, if a bean fails, it may be possible to salvage the client's invocation and redirect it to another bean. This is known as *transparent fail-over*, a quality of service that some EJB container/server vendors provide. This is an easy service to provide for stateless beans because there is no lost state when a bean crashes. Some high-end EJB products even provide transparent fail-over for stateful beans by routinely checkpointing the stateful bean's conversational state (see Chapter 14 for more). In case of a critical, unrecoverable problem, your EJB container may support professional monitoring systems, alerting a system administrator if a catastrophic error occurs.

By way of comparison, application-level exceptions should always be thrown back to the client. Application-level exceptions indicate a routine problem, and the exception itself is valuable data that the client needs. For example, we could notify a client of insufficient funds in a bank account by throwing an application-level exception. The client would always want to know about this because it is an application-level problem, not a system-level problem.

Besides correctly routing system-level and application-level exceptions, the EJB object is responsible for catching all *unchecked* exceptions (flavors of *java.lang.RuntimeException*) that your bean may throw, such as a *NullPointerException*. These are typically not caught by code. Exceptions that are unchecked in the bean could leave the bean in an abnormal state because the bean is not expecting to handle such an exception. In this scenario, the EJB container intercepts the exception and performs some action, such as throwing the exception back to the client as a remote exception. It also probably stops using that bean because the bean is in an undefined state.

The following two rules of thumb should help you with exceptions.

1. Application-level exceptions are always thrown back to the client. This includes any exception the bean defines. It also includes the *javax.ejb.CreateException* for creating beans (and the *javax.ejb.FindException* for entity beans, which we'll see in Chapters 5 through 7).

2. When system-level exceptions occur, the EJB container can do anything it wants to: page a system administrator with an alert, send an email to a third party, or throw the exception back to the client. Your bean can throw a system-level exception as either an RMI-IIOP remote exception or an unchecked *RuntimeException*. If the exception is thrown to the client, it is always thrown as a remote exception or a subclass of it.

Exceptions also have an impact on transactions. We'll learn more about this effect in Chapter 10.

What Happens During *create()* and *remove()*

As we've learned, the container, rather than a client, creates and destroys your beans. But if the container is responsible for bean life cycle, then why does the home interface and local home interface specify *create()* and *remove()* methods? What you must remember is that these methods are for creating and destroying *EJB objects*. This may not correspond to the actual creation and destruction of beans. The client shouldn't care whether the actual bean is created or destroyed—all the client code cares about is that the client has an EJB object to invoke. The fact that beans are pooled and reused behind the EJB object is irrelevant.

So when debugging your EJB applications, don't be alarmed if your bean isn't being created or destroyed when you call *create()* or *remove()* on the home object or local home object. Depending on your container's policy, your beans may be pooled and reused, with the container creating and destroying at will.

The Bean Class

Now let's look at the bean class itself. The code is shown in Source 3.5.

This is just about the most basic bean class possible. Notice the following:

- Our bean implements the *javax.ejb.SessionBean* interface, which makes it a session bean. This interface defines a few required methods that you must fill in. The container uses these management methods to interact with the bean, calling them periodically to alert the bean to important events. For example, the container will alert the bean when it is being initialized and when it is being destroyed. These callbacks are not intended for client use, so you will never call them directly—only your EJB container will. We'll learn about the specifics of these management methods in the pages to come.

```
package examples;

/**
 * Demonstration stateless session bean.
 */
public class HelloBean implements javax.ejb.SessionBean {

    private SessionContext ctx;

    //
    // EJB-required methods
    //
    public void ejbCreate() {
        System.out.println("ejbCreate()");
    }

    public void ejbRemove() {
        System.out.println("ejbRemove()");
    }

    public void ejbActivate() {
        System.out.println("ejbActivate()");
    }

    public void ejbPassivate() {
        System.out.println("ejbPassivate()");
    }

    public void setSessionContext(javax.ejb.SessionContext ctx) {
        this.ctx = ctx;
    }

    //
    // Business methods
    //
    public String hello() {
        System.out.println("hello()");
        return "Hello, World!";
    }
}
```

Source 3.5 HelloBean.java.

- The bean has an *ejbCreate()* method which matches the home object's *create()* method, and takes no parameters.
- We have one business method, *hello()*. It returns Hello, World! to the client.
- The *ejbActivate()* and *ejbPassivate()* methods do not apply to stateless session beans, and so we leave these methods empty. We'll see what these methods mean and what to use them for later in this chapter.
- When we destroy the bean, there's nothing to clean up, so we have a very simple *ejbRemove()* method.

We also have a method called *setSessionContext()*. This method is explained in the following sidebar.

EJBContexts: Your Gateway to the Container

Since your enterprise beans live in a managed container, the container is free to call your EJB components' methods at its leisure. But what if your bean needs to query the container for information about its current status? For example, inside your bean, you may want to access the security credentials of the user currently calling your bean's method.

The container houses all of this information in one object, called an *EJB context object*. An EJB context object is your gateway to the container. EJB contexts are physical parts containers and can be accessed from within your beans. Thus, a context represents a way for beans to perform callbacks to the container. These callbacks help beans both ascertain their current status and modify their current status. This is shown in Figure 3.2.

The motivation behind a context is to encapsulate the bean's domain in one compact object. Note that a bean's status may change over the bean's life cycle, and thus this context object can dynamically change over time as well. At runtime, the container is responsible for changing the context to reflect any status changes, such as the bean becoming involved in a new transaction.

Here is what the *javax.ejb.EJBContext* interface looks like (thrown exceptions omitted):

```
public interface javax.ejb.EJBContext
{
    /*
     * Call these from within your bean to access
     * your own home object or local home object.
     *
     * You can use them to create, destroy, or
     * find EJB objects and EJB local objects
```

```

    * of your own bean class type.
    */
    public javax.ejb.EJBHome getEJBHome();
    public javax.ejb.EJBLocalHome getEJBLocalHome();
    /*
    * These are transaction methods - see Chapter 10
    */
    public boolean getRollbackOnly();
    public void setRollbackOnly();
    public javax.transaction.UserTransaction getUserTransaction();
    /*
    * These are security methods - see Chapter 9
    */
    public boolean isCallerInRole(java.lang.String);
    public java.security.Principal getCallerPrincipal();
}

```

An EJB context contains callbacks useful for session beans, entity beans, and message-driven beans. In comparison, a *session context*, *entity context*, and *message-driven context* are specific EJB contexts used only for session beans, entity beans, and message-driven beans.

The container associates your bean with a context by calling *setSessionContext*, *setEntityContext*, or *setMessageDrivenContext*, depending on your bean type. When you define each of these methods, you should store the context away in a member variable so the context can be queried later, as shown in Source 3.5.

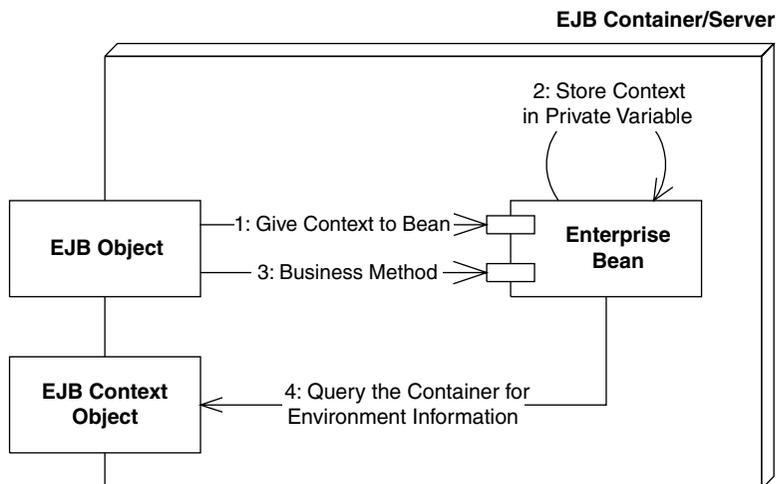


Figure 3.2 EJB Contexts.

The Deployment Descriptor

Next, we need to generate a *deployment descriptor*, which describes our bean's middleware requirements to the container. Deployment descriptors are one of the key features of EJB because they allow you to *declaratively* specify attributes on your beans, rather than programming this functionality into the bean itself.

Physically, a deployment descriptor is an XML document. Your EJB container, IDE environment, or other tool (such as a UML editor that can generate EJB code) should supply tools to help you generate such a deployment descriptor.

Our deployment descriptor is shown in Source 3.6.

Many different settings make up a deployment descriptor. For a full deployment descriptor reference, see Appendix C. For now, here is an explanation of our session bean descriptor:

<ejb-name> The nickname for this particular bean. Can be used later in the deployment descriptor to refer back to this bean to set additional settings.

<home> The fully qualified name of the home interface.

<remote> The fully qualified name of the remote interface.

<local-home> The fully qualified name of the local home interface.

```
<!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>
  <enterprise-beans>
    <session>
      <ejb-name>Hello</ejb-name>
      <home>examples.HelloHome</home>
      <remote>examples.Hello</remote>
      <local-home>examples.HelloLocalHome</local-home>
      <local>examples.HelloLocal</local>
      <ejb-class>examples.HelloBean</ejb-class>
      <session-type>Stateless</session-type>
      <transaction-type>Container</transaction-type>
    </session>
  </enterprise-beans>
</ejb-jar>
```

Source 3.6 ejb-jar.xml.

- <local>** The fully qualified name of the local interface.
- <ejb-class>** The fully qualified name of the enterprise bean class.
- <session-type>** Whether the session bean is a stateful or stateless session bean.
- <transaction-type>** Ignore for now—see Chapter 10 for more details on transactions.

The Vendor-Specific Files

Next in our stateless session bean are vendor-specific files. These files exist because the EJB specification can't cover everything; vendors differentiate their products in areas such as instance pooling algorithms, clustering algorithms, and so on. The vendor-specific files are not portable and can use any file format, including XML, flat file, or binary. In fact, it may not even exist as files—the settings could be stored in some database with a GUI on top of it.

The source code that accompanies this book shows an example of a vendor-specific file.

The Ejb-jar File

Now that we've written all the necessary files for our component, we need to package all the files together in an Ejb-jar file. If you're using a development environment supporting EJB, the development environment may contain an automated way to generate the Ejb-jar file for you. We can generate it manually as follows:

```
jar cf HelloWorld.jar *
```

The asterisk indicates the files to include in the jar—the bean class, home interface, local home interface, remote interface, local interface, deployment descriptor, and possibly vendor-specific files (depending on your container's policy).

The following is the folder structure within the Ejb-jar file:

```
META-INF/MANIFEST.MF
META-INF/ejb-jar.xml
examples/HelloBean.class
examples/HelloLocalHome.class
examples/HelloLocal.class
examples/Hello.class
```

The files must be in properly named subdirectories of the current directory. For example, our *Hello.class* file is located in *examples\Hello.class*, below the current directory. You must store your classes in a directory corresponding to the package that the class belongs to, or the JVM will be unable to locate your classes when it searches your jar. The *ejb-jar.xml* file must be placed in the META-INF subfolder. The container consults that file first when opening the Ejb-jar file to figure out what beans are inside the jar.

The *MANIFEST.MF* file is a listing of the files within the Ejb-jar file. It is auto-generated by the *jar* utility. You don't need to worry about this file.

Deploying the Bean

Finally, we're ready to deploy our bean in an EJB container. This step varies from container to container. When you reach this point, consult your container's documentation on how to deploy a bean. This could be anything from running a command-line tool on your Ejb-jar file to copying your Ejb-jar file into a well-known folder where your application server detects its presence. For an example of deploying a bean, see the source code accompanying this book.

When deploying an Ejb-jar file into a container, the following steps are usually performed:

- The Ejb-jar file is verified. The container checks that the enterprise bean class, the remote interface, and other items are valid. Any commercial tool should report intelligent errors back to you, such as, "You need to define an *ejbCreate()* method in your bean."
- The container tool generates an EJB object and home object for you.
- The container tool generates any necessary RMI-IIOP stubs and skeletons. (See Appendix A for more information about stubs and skeletons.)

Once you've performed these steps, start up your EJB container (if it isn't already running). Most products output a server log or have a GUI to view the beans that are deployed. Make sure that your container is indeed making your bean available. It should tell you it did so.

The Optional EJB Client JAR file

One common question deployers ask is, "Which classes do I need to deploy with my *client* applications that call enterprise beans?" EJB allows you to spec-

ify the exact classes you need with an *Ejb-client JAR file*. An Ejb-client JAR file is an archive of classes that must be deployed for any clients of a particular Ejb-jar file. You specify the name of the Ejb-client jar file in your XML deployment descriptor, as shown in Source 3.7.

When you build an Ejb-client jar file, you should bundle only the files needed by the client. This typically includes interfaces, helper classes, and stubs.

You might find Ejb-client jar files useful for saving hard disk space, so you can avoid copying the entire Ejb-jar file onto the client machine. This might be useful if you're in an applet environment.

However, Ejb-client jar files are completely optional and most deployments will not make use of them. This is because hard disk space is usually not a problem, especially if the client of your application server is a Web server. Laziness will usually prevail.

Understanding How to Call Beans

We now take a look at the other half of the world—the client side. We are now customers of the beans' business logic, and we are trying to solve some real-world problem by using one or more beans together. There are two different kinds of clients.

```
...
<ejb-jar>

    <enterprise-beans>
        ...
    </enterprise-beans>

    <!--
    This is an optional instruction to the deployer that
    he must make the this jar file accessible to
    clients of these beans.  If this instruction does not
    exist, the deployer must make the entire Ejb-jar file
    accessible to clients.
    -->
    <ejb-client-jar>HelloClient.jar</ejb-client-jar>

</ejb-jar>
```

Source 3.7 Declaring an Ejb-client jar file within a deployment descriptor.

Java RMI-IIOP based clients. These clients use the *Java Naming and Directory Interface* (JNDI) to look up objects over a network, and they use the *Java Transaction API* (JTA) to control transactions.

CORBA clients. Clients can also be written to the CORBA standard. This would primarily be useful if you want to call your EJB components using another language, such as C++. CORBA clients use the *CORBA Naming Service* (COS Naming) to look up objects over the network, and they use the CORBA's *Object Transaction Service* (OTS) to control transactions.

Whether you're using CORBA or RMI-IIOP, your client code typically looks like this:

1. Look up a home object.
2. Use the home object to create an EJB object.
3. Call business methods on the EJB object.
4. Remove the EJB object.

You're about to see how to call EJB components from RMI-IIOP clients. This is the paradigm we'll use throughout this book. If you're interested in CORBA clients, see Appendix B.

Looking up a Home Object

One of the goals of EJB is that your application code should be "write once, run anywhere." If you deploy a bean onto one machine and then switch it for a different machine, your code should not change because it is location transparent.

EJB achieves location transparency by leveraging *naming and directory services*. Naming and directory services are products that store and look up resources across a network. Some examples of directory service products are the iPlanet Directory Server, Microsoft's Active Directory, and IBM's Lotus Notes Domino Server.

Corporations traditionally have used naming and directory services to store usernames, passwords, machine locations, printer locations, and so on. EJB servers exploit naming services to store location information for resources that your application code uses in an enterprise deployment. These resources could be EJB home objects, enterprise bean environment properties, database drivers, message service drivers, and other resources. By using naming services, you can write application code that does not depend on specific machine names or locations. This is all part of EJB's location transparency, and it keeps your code portable. If you decide later that resources should be located elsewhere, your code does not need to be rebuilt because the naming service can

simply be updated to reflect the new resource locations. This greatly enhances maintenance of a multitier deployment that may evolve over time. This becomes absolutely necessary when purchasing prewritten software (such as enterprise beans), because your purchased components' source code will likely not be made available to you to change.



While naming and directory servers have typically run standalone, they can also run in the same process as the application server. Many containers are written in Java, and so their naming and directory services are just bunches of Java classes that run inside of the container.

Unless you're using CORBA, the de facto API used to access naming and directory services is the Java Naming and Directory Interface (JNDI), which we explain in Appendix A. JNDI adds value to your enterprise deployments by providing a standard interface for locating users, machines, networks, objects, and services. For example, you can use the JNDI to locate a printer on your corporate intranet. You can also use it to locate a Java object or to connect with a database. In EJB, JNDI is used to lookup home objects. JNDI is also useful for locating resources across an enterprise deployment, including environment properties, database resources, and more; we'll show you how to leverage JNDI for these purposes in Chapter 9.

How to Use JNDI to Locate Home Objects

To achieve location transparency, EJB containers mask the specific locations of home objects from your enterprise beans' client code. Clients do not hard-code the machine names that home objects reside on; rather, they use JNDI to *lookup* home objects. Home objects are physically located somewhere on the network—perhaps in the address space of an EJB container residing on machine #1, or perhaps on a container residing on machine #2. As a developer who writes client code to use beans, you don't care.

For clients to locate a home object, you must provide a *nickname* for your bean's home object. Clients will use this nickname to identify the home object it wants. For example, our Hello World example might have a nickname *HelloHome*. You specify this nickname using the proprietary vendor-specific files that are bundled with your bean.

When you deploy your bean into the container, the container automatically *binds* the nickname *HelloHome* to the home object. Then any client on any machine across a multitier deployment can use that nickname to find home objects, without regard to physical machine locations. Clients use the JNDI API to do this. JNDI goes over the network to some naming service, or JNDI

tree, to look for the home object, perhaps contacting one or more naming services in the process. Eventually the home object is found, and a reference to it is returned to the client. This is shown in Figure 3.3.

The complete client source code is shown in Source 3.8.

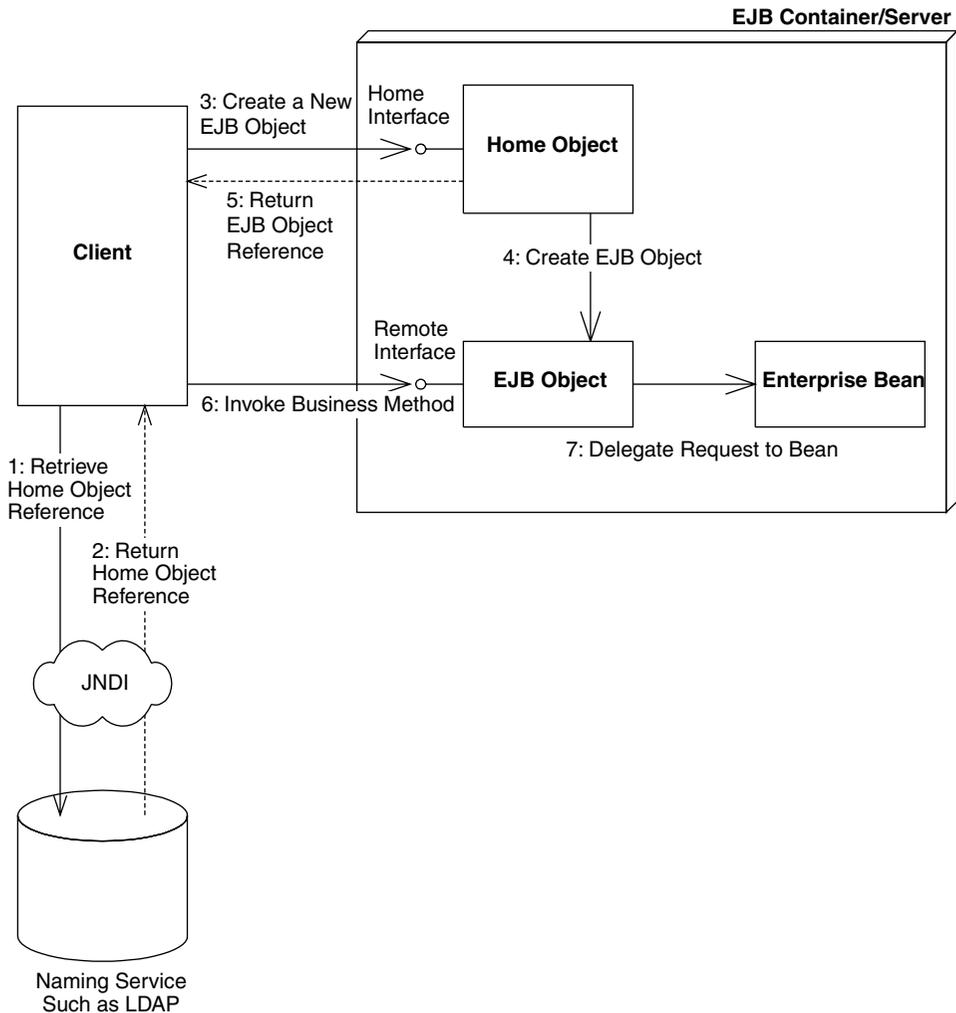


Figure 3.3 Acquiring a reference to a home object.

```
package examples;

import javax.naming.Context;
import javax.naming.InitialContext;
import java.util.Properties;

/**
 * This class is an example of client code that invokes
 * methods on a simple stateless session bean.
 */
public class HelloClient {

    public static void main(String[] args) throws Exception {
        /*
         * Setup properties for JNDI initialization.
         *
         * These properties will be read-in from
         * the command-line.
         */
        Properties props = System.getProperties();

        /*
         * Obtain the JNDI initial context.
         *
         * The initial context is a starting point for
         * connecting to a JNDI tree. We choose our JNDI
         * driver, the network location of the server, etc.
         * by passing in the environment properties.
         */
        Context ctx = new InitialContext(props);

        /*
         * Get a reference to the home object - the
         * factory for Hello EJB Objects
         */
        Object obj = ctx.lookup("HelloHome");

        /*
         * Home objects are RMI-IIOP objects, and so
         * they must be cast into RMI-IIOP objects
         * using a special RMI-IIOP cast.
         *
         * See Appendix A for more details on this.
         */
        HelloHome home = (HelloHome)
            javax.rmi.PortableRemoteObject.narrow(
                obj, HelloHome.class);
    }
}
```

Source 3.8 HelloClient.java.

```
    /*
     * Use the factory to create the Hello EJB Object
     */
    Hello hello = home.create();

    /*
     * Call the hello() method on the EJB object. The
     * EJB object will delegate the call to the bean,
     * receive the result, and return it to us.
     *
     * We then print the result to the screen.
     */
    System.out.println(hello.hello());

    /*
     * Done with EJB Object, so remove it.
     * The container will destroy the EJB object.
     */
    hello.remove();
}
}
```

Source 3.8 HelloClient.java. (continued)

The client code is self-explanatory.

Running the System

To try the deployment, you first must bring up the application server. This step varies depending on your vendor. Again, since we want to keep this book vendor-neutral, please see the book's accompanying source code for an example.

Next, run the client application. When running the client, you need to supply the client with JNDI environment information. As we explain in Appendix A, JNDI requires a minimum of two properties to retrieve an initial context:

- The name of the initial context factory. An example is *com.sun.jndi.ldap.LdapCtxFactory*.
- The provider URL, indicating the location of the JNDI tree to use. An example is *ldap://lowvre:389/o=Airius.com*.

The actual parameters you need should be part of your EJB container's documentation. See the book's accompanying source code for examples of this.



For your EJB client code to work, you must take care to distribute the correct class files on the right machines. If remote client code uses home interfaces and remote interfaces, then you must deploy those class files in your client environment. And because clients never directly access your bean implementation, you should not deploy your bean classes in your client environment.

The Server-Side Output

When we run the client, our container shows the following debug log. (Debug logs are great for seeing what your enterprise beans are doing.)

```
setSessionContext()  
ejbCreate()  
hello()  
ejbRemove()
```

As you can see, the container associated our bean with a session context, called *create()*, delegated a business method to the bean, and then called *remove()*. Note that some containers may give slightly different output than others—it's all implementation-specific and part of EJB product differentiation. Keep this in mind when debugging your beans.

The Client-Side Output

After running the client, you should see the following output:

```
Hello, World!
```

Implementing Component Interfaces

We wrap up this chapter with a quick design strategy. As you probably noticed, our enterprise bean class does not implement its own component interface (either remote interface or local interface). But why not? Doesn't the component interface seem like a natural fit for the interface to your bean? After all, the component interface defines every business method of the bean. Implementing your component interface would be a nifty way to perform compile-time checking to make sure your bean's method signature matches your component interface's signature.

There are two good reasons not to implement your bean's component interface:

Reason 1. Component interfaces extend interfaces defined by Sun, such as *javax.ejb.EJBObject* or *javax.ejb.EJBLocalObject*. These superinterfaces define additional methods intended for client use, and you'd therefore have provide no-op implementations of those methods in your bean. Those methods have no place in your bean class.

Reason 2. Let's assume your enterprise bean wants to call a method on a *different* enterprise bean, and you want to pass a reference to your bean as a parameter to the other bean's method (similar to passing the *this* parameter in Java). How can you do this in EJB?

Remember that all clients call methods on EJB objects, not beans. Thus, if your bean calls another bean, you must pass a reference to your bean's EJB object, rather than a reference to your bean. The other bean should operate on your EJB object, and not your bean, because the other bean is a client, just like any other client, and all clients must go through EJB objects.

The danger here is if your enterprise bean class implements your EJB object's remote interface. You could accidentally pass a reference to the bean itself, rather than pass a reference to the bean's EJB object. Because your bean implements the same interface as the EJB object, the compiler would let you pass the bean itself as a *this* parameter, which is an error.

A Solution

There is an alternative way to preserve compile-time checks of your method signatures. The approach is to contain your bean's business method signatures within a common superinterface that your remote interface extends and your bean implements. You can think of this superinterface as a *business interface* that defines your business methods and is independent of EJB. The following example illustrates this concept:

```
// Business interface
public interface HelloBusinessMethods {
    public String hello() throws java.rmi.RemoteException;
}
// EJB remote interface
public interface HelloRemote extends javax.ejb.EJBObject,
HelloBusinessMethods {
}
// EJB local interface
public interface HelloLocal extends javax.ejb.EJBLocalObject,
HelloBusinessMethods {
}
// Bean implementation
```

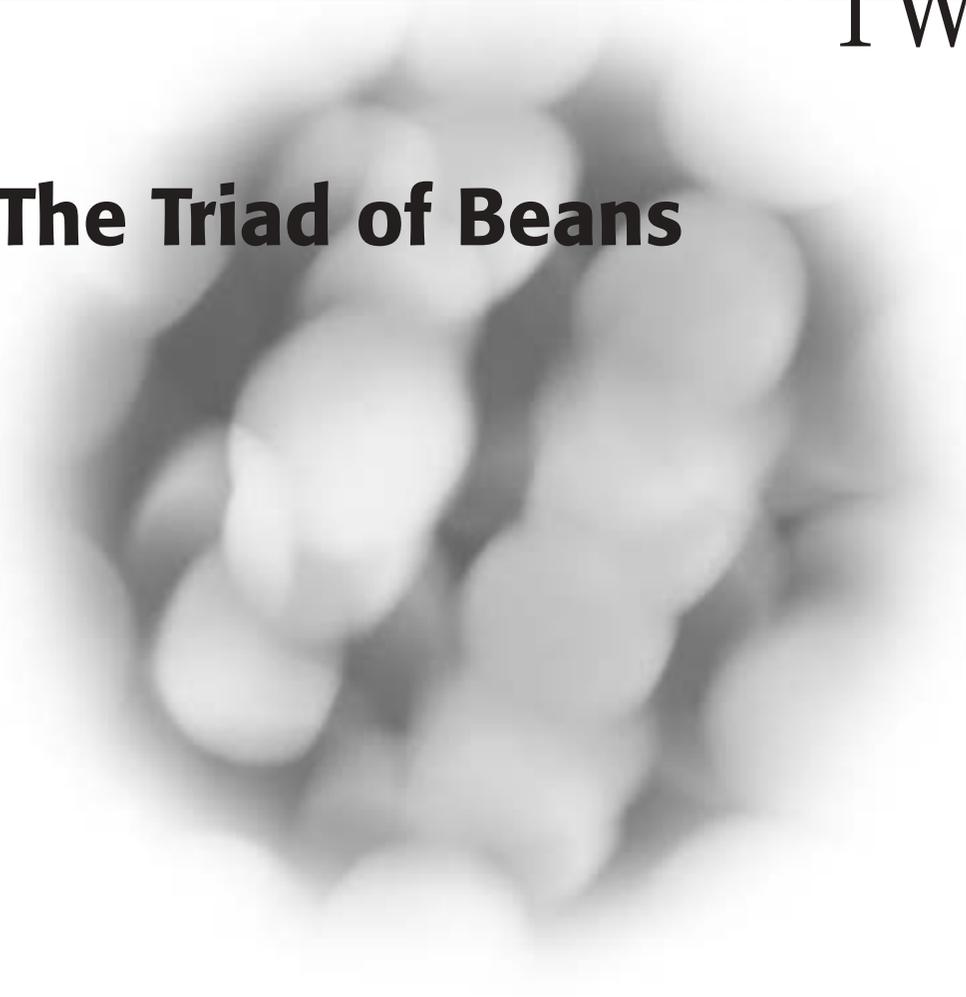
```
public class HelloBean implements SessionBean, HelloBusinessMethods {
    public String hello() {
        return "Hello, World!";
    }
    < . . . define other required callbacks . . . >
}
```

The only problem with this approach is that the local interface throws *remote exceptions*. If you can live with that, then this design strategy works.

Summary

In this chapter, we learned how to write the component interfaces, home interface, enterprise bean class, deployment descriptor, and Ejb-jar file. You also saw how to call beans using JNDI and RMI-IIOP. Congratulations are in order: It took awhile, but you've successfully completed your first Enterprise JavaBeans deployment!

The Triad of Beans



In Part 2, we'll focus on the development details for implementing an EJB application. We'll learn about the three types of enterprise beans: session beans (Chapter 4), entity beans (Chapter 5), and message-driven beans (Chapter 8). We'll also explore their subtypes: stateless session beans, stateful session beans, bean-managed persistent entity beans (Chapter 6), and container-managed persistent entity beans (Chapter 7). Not only will we see each of these conceptually, but we'll also write an example for each bean type. We'll end Part 2 with a discussion of container-provided services (Chapter 9), such as security, the environment, and calling beans from other beans.

Part 2 is essential for those of you who are ready to delve into EJB programming fundamentals. It is essential groundwork to prepare yourself for the more advanced topics, such as transactions and EJB design strategies, which are coming in Part 3.

Introduction to Session Beans

A *session bean* represents work being performed for client code that is calling it. Session beans are business process objects. They implement business logic, business rules, algorithms, and workflow. For example, a session bean can perform price quoting, order entry, video compression, banking transactions, stock trades, database operations, complex calculations, and more. They are reusable components that contain logic for business processes.

Let's examine the characteristics of session beans in detail and then code up a stateful session bean.

Session Bean Lifetime

A chief difference between session beans and entity beans is the scope of their lives. A session bean is a relatively short-lived component. It has roughly the lifetime equivalent of a *session* or lifetime of the client code that is calling the session bean.

For example, if the client code contacted a session bean to perform order entry logic, the EJB container is responsible for creating an instance of that session bean component. When the client later disconnects, the application server may destroy the session bean instance.

A client's session duration could be as long as a browser window is open, perhaps connecting to an e-commerce site with deployed session beans. It could

also be as long as your Java applet is running, as long as a standalone application is open, or as long as another bean is using your bean.

The length of the client's *session* generally determines how long a session bean is in use—that is where the term *session bean* originated. The EJB container is empowered to destroy session beans if clients time out. If your client code is using your beans for 10 minutes, your session beans might live for minutes or hours, but probably not weeks, months, or years. Typically session beans do not survive application server crashes, nor do they survive machine crashes. They are in-memory objects that live and die with their surrounding environments.

In contrast, entity beans can live for months or even years because entity beans are *persistent objects*. Entity beans are part of a durable, permanent storage, such as a database. Entity beans can be constructed in memory from database data, and they can survive for long periods of time.

Session beans are *nonpersistent*. This means that session beans are not saved to permanent storage, whereas entity beans are. Note that session beans *can* perform database operations, but the session bean *itself* is not a persistent object.

Session Bean Subtypes

All enterprise beans hold *conversations* with clients at some level. A conversation is an interaction between a client and a bean, and it is composed of a number of method calls between the client and the bean. A conversation spans a business process for the client, such as configuring a frame-relay switch, purchasing goods over the Internet, or entering information about a new customer.

The two subtypes of session beans are *stateful session beans* and *stateless session beans*. Each is used to model different types of these conversations.

Stateful Session Beans

Some business processes are naturally drawn-out conversations over several requests. An example is an e-commerce Web store. As a user peruses an online e-commerce Web site, the user can add products to the online shopping cart. Each time the user adds a product, we perform another request. The consequence of such a business process is that the components must track the user's state (such as a shopping cart state) from request to request.

Another example of a drawn-out business process is a banking application. You may have code representing a bank teller who deals with a particular

client for a long period of time. That teller may perform a number of banking transactions on behalf of the client, such as checking the account balance, depositing funds, and making a withdrawal.

A *stateful session bean* is a bean that is designed to service business processes that span multiple method requests or transactions. To accomplish this, stateful session beans *retain state* on behalf of an individual client. If a stateful session bean's state is changed during a method invocation, that same state will be available to that same client upon the following invocation.

Stateless Session Beans

Some business processes naturally lend themselves to a single request conversation. A single request business process is one that does not require state to be maintained across method invocations.

A *stateless session bean* is a bean that holds conversations that span a single method call. They are stateless because they do not hold multimethod conversations with their clients. After each method call, the container may choose to destroy a stateless session bean, or recreate it, clearing itself out of all information pertaining to past invocations. It also may choose to keep your instance around, perhaps reusing it for all clients who want to use the same session bean class. The exact algorithm is container specific. The takeaway point is this: Expect your bean to forget everything after each method call, and thus retain no conversational state from method to method. If your bean happens to hang around longer, then great—but that's your container's decision, and you shouldn't rely on it.

For a stateless session bean to be useful to a client, the client must pass all client data that the bean needs as parameters to business logic methods. Alternatively, the bean can retrieve the data it needs from an external source, such as a database.



Stateless really means no conversational state. Stateless session beans can contain state that is not specific to any one client, such as a database connection factory that all clients would use. You can keep this around in a private variable. So long as you're willing to lose the data in your private variable at any time, you'll be fine.

An example of a stateless session bean is a high-performance engine that solves complex mathematical operations on a given input, such as compression of audio or video data. The client could pass in a buffer of uncompressed data, as well as a compression factor. The bean returns a compressed buffer and is then available to service a different client. The business process spanned

one method request. The bean does not retain any state from previous requests.

Another example of a stateless session bean is a credit card verification component. The verifier bean takes a credit card number, expiration date, cardholder's name, and dollar amount as input. The verifier then returns a yes or no answer, depending on whether the card holder's credit is valid. Once the bean completes this task, it is available to service a different client and retains no past knowledge from the original client.

Because stateless session beans hold no conversational state, all instances of the same stateless session bean class are equivalent and indistinguishable to a client. It does not matter who has called a stateless session bean in the past, since a stateless session bean retains no state knowledge about its history. This means that *any* stateless session bean can service *any* client request because they are all exactly the same. In fact, stateless session beans can be pooled, reused, and swapped from one client to another client on *each method call!* We show this in Figure 4.1.

Special Characteristics of Stateful Session Beans

So far, we've seen session beans in general. We also coded up a simple stateless session bean in Chapter 3. Now let's look at the trickier flavor, stateful session beans.

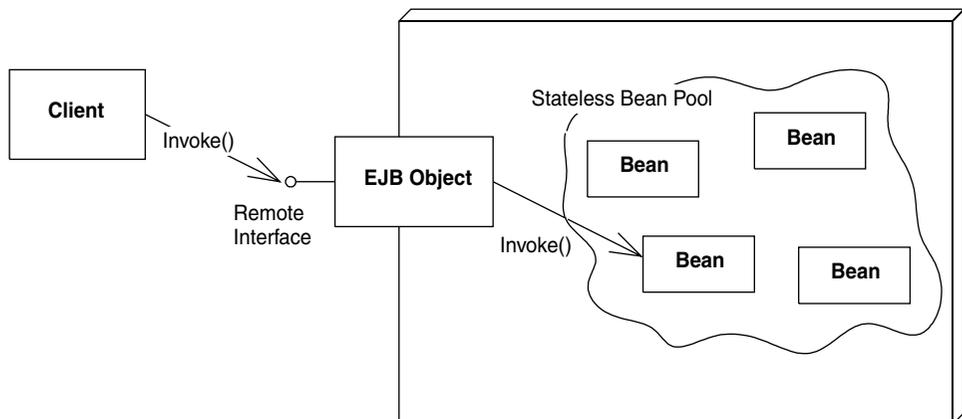


Figure 4.1 Stateless session bean pooling.

Achieving the Effect of Pooling with Stateful Beans

With stateful session beans, pooling is not as simple with stateful session beans. When a client invokes a method on a bean, the client is starting a *conversation* with the bean, and the conversational state stored in the bean must be available for that same client's next method request. Therefore, the container cannot easily pool beans and dynamically assign them to handle arbitrary client method requests, since each bean is storing state on behalf of a particular client. But we still need to achieve the effect of pooling for stateful session beans so that we can conserve resources and enhance the overall scalability of the system. After all, we only have a finite amount of resources available, such as memory, database connections, and socket connections. If the conversational state that the beans are holding is large, the EJB server could easily run out of resources. This was not a problem with stateless session beans because the container could pool only a few beans to service thousands of clients.

This problem should sound quite familiar to operating systems gurus. Whenever you run an application on a computer, you have only a fixed amount of physical memory in which to run. The operating system still must provide a way for many applications to run, even if the applications take up more aggregate memory than is available physically. To provide for this, operating systems use your hard disk as an extension of physical memory. This effectively extends your system's amount of *virtual memory*. When an application goes idle, its memory can be *swapped out* from physical memory and onto the hard disk. When the application becomes active again, any needed data is *swapped in* from the hard disk and into physical memory. This type of swapping happens often when switching between applications (called *context switching*).

EJB containers exploit this very paradigm to conserve stateful session bean resources. To limit the number of stateful session bean instances in memory, the container can *swap out* a stateful bean, saving its conversational state to a hard disk or other storage. This is called *passivation*. After passivating a stateful bean, the conversational state is safely stored away, allowing resources like memory to be reclaimed. When the original client invokes a method, the passivated conversational state is *swapped in* to a bean. This is called *activation*. This bean now resumes the conversation with the original client. Note that the bean that receives the activated state may not be the original bean instance. But that's all right because the new instance resumes its conversation from the point where the original instance was passivated.

Thus, EJB does indeed support the *effect* of pooling stateful session beans. Only a few instances can be in memory when there are actually many clients. But this pooling effect does not come for free—the passivation/activation steps

could entail an input/output bottleneck. Contrast this to stateless session beans, which are easily pooled because there is no state to save.

How does the container decide which beans to activate and which beans to passivate? The answer is specific to each container. Most containers employ a *Least Recently Used* (LRU) passivation strategy, which simply means to passivate the bean that has been called the least recently. This is a good algorithm because remote clients have the habit of disconnecting from the network, leaving beans stranded without a client, ready to be passivated. If a bean hasn't been invoked in a while, the container writes it to disk.

Passivation can occur at any time, as long as a bean is not involved in a method call. It's up to the container to decide when passivation makes sense. There is one exception to this rule: Any bean involved in a *transaction* (see Chapter 10) cannot be passivated until the transaction completes.

To activate beans, most containers usually use a *just-in-time* algorithm. Just in time means that beans should be activated on demand, as client requests come in. If a client request comes in, but that client's conversation has been passivated, the container activates the bean on demand, reading the passivated state back into memory.

In general, passivation and activation are not useful for stateless session beans. Stateless beans do not have any state to passivate/activate, so the container can simply destroy stateless beans arbitrarily.

The Rules Governing Conversational State

More rigorously, the *conversational state* of a bean follows the rules laid out by *Java object serialization*. At passivation time the container uses object serialization (or an equivalent protocol) to convert the bean's conversational state to a bit-blob and write the state out to disk. This safely tucks the state away. The bean instance (which still exists) can be reassigned to a different client, and can hold a brand-new conversation with that new client.

Activation reverses the process: A serialized blob that had been written to storage is read back into memory and converted to in-memory bean data. What makes this whole process work is the *javax.ejb.EnterpriseBean* interface extends *java.io.Serializable*, and every enterprise bean class indirectly implements this interface.

For every Java object that is part of a bean's conversational state, the previous algorithm is reapplied recursively on those objects. Thus, object serialization constructs an entire graph of data referred to by the main bean. Note that while your beans must follow the rules for object serialization, the EJB container

itself does not necessarily need to use the default serialization protocol; it could use a custom protocol to allow for flexibility and differentiation between container vendors.

More concretely, every member variable in a bean is considered to be part of the bean's conversational state if the following apply:

- The member variable is a nontransient primitive type, or
- The member variable is a nontransient Java object (extends *java.lang.Object*)

Your bean might also hold references to container-implemented objects. The container must preserve each of the following upon passivation/activation:

- EJB object references
- Home object references
- EJB context references (see Chapter 9)
- JNDI naming contexts

For example, let's say you have the following stateful session bean code:

```
public class MySessionBean implements javax.ejb.SessionBean
{
    // State variables
    private Long myLong;
    private MySessionBeanRemoteInterface ejbObject;
    private MySessionBeanHomeInterface homeObject;
    private javax.ejb.SessionContext mySessionContext;
    private javax.naming.Context envContext;
    // EJB-required methods (fill in as necessary)
    public void setSessionContext(SessionContext ctx) {}
    public void ejbCreate() {}
    public void ejbPassivate() {}
    public void ejbActivate() {}
    public void ejbRemove() {}
    // Business methods
    . . .
}
```

The container must retain the values of the preceding member variables across passivation and activation operations.

Activation and Passivation Callbacks

Let's now look at what actually happens to your bean during passivation and activation. When an EJB container passivates a bean, the container writes the bean's conversational state to secondary storage, such as a file or database. The

container informs the bean that it's about to perform passivation by calling the bean's required *ejbPassivate()* callback method. *ejbPassivate()* is a warning to the bean that its held conversational state is about to be swapped out.

It's important that the container inform the bean using *ejbPassivate()* so that the bean can relinquish held resources. These held resources include database connections, open sockets, open files, or other resources that do not make sense to be saved to disk or cannot be transparently saved using object serialization. The EJB container calls the *ejbPassivate()* method to give the bean a chance to release these resources or deal with the resources as the bean sees fit. Once the container's *ejbPassivate()* callback method into your bean is complete, your bean must be in a state suitable for passivation. For example:

```
import javax.ejb.*;
public class MyBean implements SessionBean {
    public void ejbPassivate() {
        <close socket connections, etc . . . >
    }
    . . .
}
```

The passivation process is shown in Figure 4.2. This is a typical stateful bean passivation scenario. The client has invoked a method on an EJB object that does not have a bean tied to it in memory. The container's pool size of beans has been reached. Thus, the container needs to passivate a bean before handling this client's request.

Exactly the opposite process occurs during the activation process. The serialized conversational state is read back into memory, and the container reconstructs the in-memory state using object serialization or the equivalent. The container then calls the bean's required *ejbActivate()* method. *ejbActivate()* gives the bean a chance to restore the open resources it released during *ejbPassivate()*. For example:

```
import javax.ejb.*;
public class MyBean implements SessionBean {
    public void ejbActivate() {
        <open socket connections, etc . . . >
    }
    . . .
}
```

The activation process is shown in Figure 4.3. This is a typical just-in-time stateful bean activation scenario. The client has invoked a method on an EJB object whose stateful bean had been passivated.

You probably don't need to worry about implementing *ejbPassivate()* and *ejbActivate()* unless you are using open resources, such as socket connections or

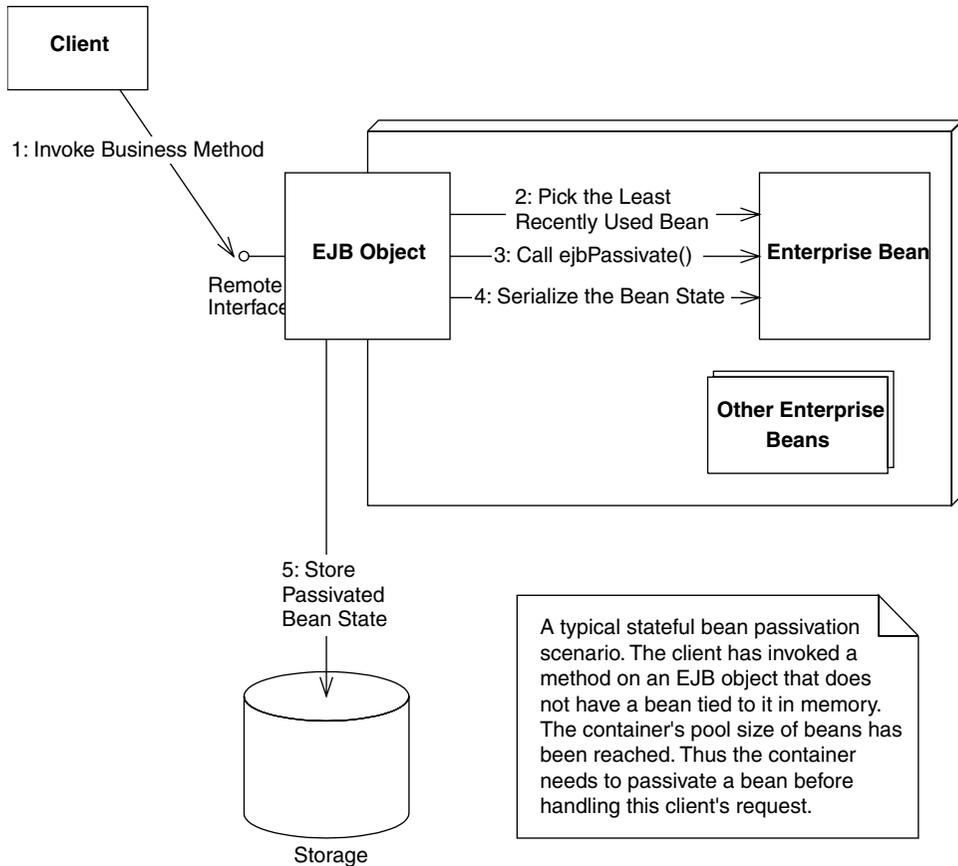


Figure 4.2 Passivation of a stateful bean.

database connections, that must be reestablished after activation. In most cases, you can simply leave these methods empty.

Method Implementation Summary

Table 4.1 summarizes how to develop session bean classes.

A Simple Stateful Session Bean

Let's put our stateful session bean knowledge to use by programming a simple stateful bean. Our bean will be a counter bean, and it will be responsible for simply counting up one by one. The current count will be stored within the bean and will increment as client requests arrive. Thus, our bean will be stateful and will hold a multimethod conversation with a particular client.

Table 4.1 Required Methods for Session Bean Classes

METHOD	DESCRIPTION	TYPICAL IMPLEMENTATION (STATELESS SESSION BEANS)	TYPICAL IMPLEMENTATION (STATEFUL SESSION BEANS)
setSessionContext (SessionContext ctx)	Associates your bean with a session context. Your bean can query the context about its current transactional state, its current security state, and more.	Store the context away in a member variable so the context can be queried later.	Store the context away in a member variable so the context can be queried later.
ejbCreate... (...)	Initializes your session bean.	Perform any initialization your bean needs, such as setting member variables to the argument values passed in. Note: You can define several <i>ejbCreate... (...)</i> methods, and each can take different arguments. You must provide at least one <i>ejbCreate... (...)</i> method in your session bean.	Perform any initialization your bean needs, such as setting member variables to the argument values passed in. Note: You can define only a single empty <i>ejbCreate()</i> method with no parameters. If it had parameters, and the bean initialized itself to those parameters, the bean would never remember what it initialized itself to upon subsequent calls, since it is stateless!
ejbPassivate()	Called immediately before your bean is passivated (swapped out to disk because there are too many instantiated beans).	Release any resources your bean may be holding.	Unused because there is no conversational state; leave empty.
ejbActivate()	Called immediately before your bean is activated (swapped in from disk because a client needs your bean).	Acquire any resources your bean needs, such as those released during <i>ejbPassivate()</i> .	Unused because there is no conversational state; leave empty.
ejbRemove()	Called by the container immediately before your bean is removed from memory.	Prepare your bean for destruction. Free all resources you may have allocated.	Prepare your bean for destruction. Free all resources you may have allocated.

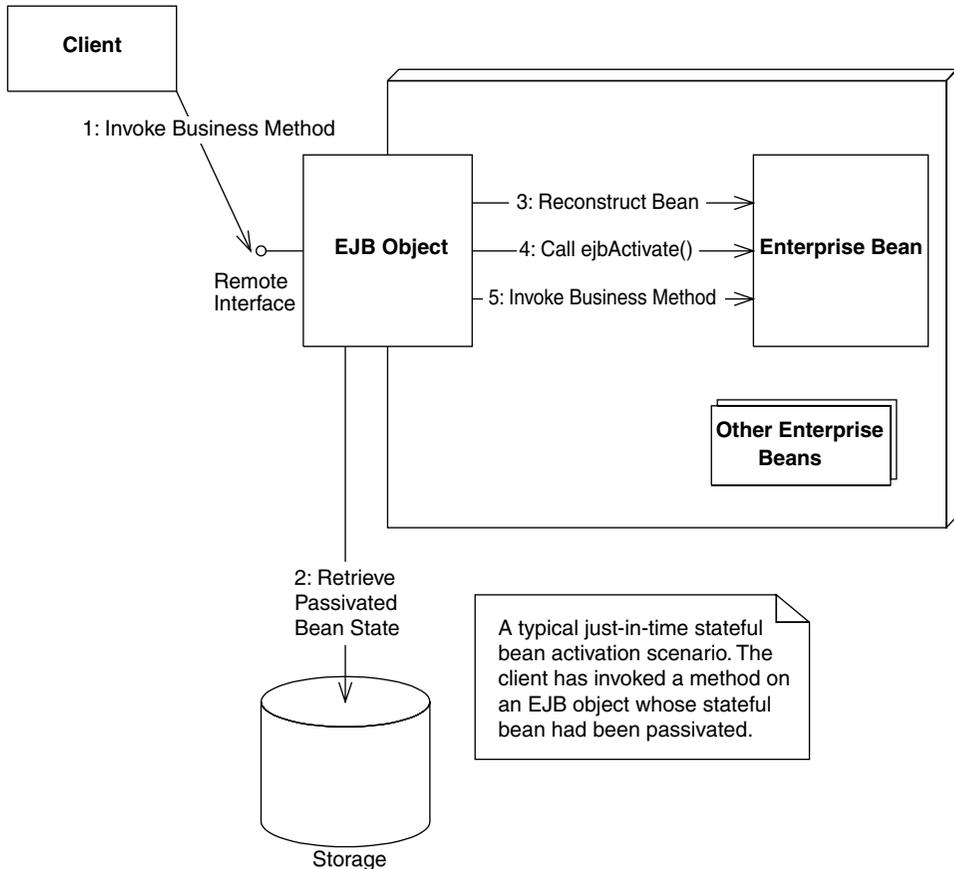


Figure 4.3 Activation of a stateful bean.

The Count Bean's Remote Interface

First let's define our bean's remote interface. The code is shown in Source 4.1.

Our remote interface defines a single business method, *count()*, which we will implement in the enterprise bean class.

The Count Bean

Our bean implementation has one business method, *count()*, which is responsible for incrementing an integer member variable, called *val*. The conversational state is the *val* member variable. We show the code for our counter bean in Source 4.2.

```
package examples;

import javax.ejb.*;
import java.rmi.RemoteException;

/**
 * These are CountBean's business logic methods.
 *
 * This interface is what clients operate on when they
 * interact with EJB objects. The container vendor will
 * implement this interface; the implemented object is
 * the EJB object, which delegates invocations to the
 * actual bean.
 */
public interface Count extends EJBObject {

    /**
     * Increments the int stored as conversational state
     */
    public int count() throws RemoteException;
}
```

Source 4.1 Count.java.

```
package examples;

import javax.ejb.*;

/**
 * Demonstration Stateful Session Bean. This Bean is initialized
 * to some integer value, and has a business method which
 * increments the value.
 *
 * This example shows the basics of how to write a stateful
 * session bean, and how passivation/activation works.
 */
public class CountBean implements SessionBean {

    // The current counter is our conversational state.
    public int val;

    //
    // Business methods
    //
}
```

Source 4.2 CountBean.java.

```
/**
 * Counts up
 */
public int count() {
    System.out.println("count()");
    return ++val;
}

//
// EJB-required methods
//

public void ejbCreate(int val) throws CreateException {
    this.val = val;
    System.out.println("ejbCreate()");
}

public void ejbRemove() {
    System.out.println("ejbRemove()");
}

public void ejbActivate() {
    System.out.println("ejbActivate()");
}

public void ejbPassivate() {
    System.out.println("ejbPassivate()");
}

public void setSessionContext(SessionContext ctx) {
}
}
```

Source 4.2 CountBean.java (continued).

Note the following about our bean:

- The bean implements *javax.ejb.SessionBean* (described fully in Appendix E). This means the bean must define all methods in the *SessionBean* interface. By looking at the bean, you can see we've defined them but kept them fairly trivial.
- Our *ejbCreate()* initialization method takes a parameter, *val*. This method customizes our bean to the client's needs. Our *ejbCreate()* method is

responsible for beginning a conversation with the client. It uses *val* as the starting state of the counter.

- The *val* member variable obeys the rules for conversational state because it is serializable. Thus, it lasts across method calls and is automatically preserved during passivation/activation.

Notice, too, that our code has a *setSessionContext()* method. This associates our bean with a *session context*, which is a specific EJB context used only for session beans. Our bean can callback to the container through this object. The session context interface looks like this:

```
public interface javax.ejb.SessionContext
    extends javax.ejb.EJBContext
{
    public javax.ejb.EJBLocalObject getEJBLocalObject();
    public javax.ejb.EJBObject getEJBObject();
}
```

Notice that the *SessionContext* interface extends the *EJBContext* interface, giving session beans access to all the methods defined in *EJBContext* (see Chapter 3 or Appendix E).

Specific to session beans, the *getEJBObject()* and *getEJBLocalObject()* methods are useful if your bean needs to call another bean and if you want to pass a reference to your own bean. In Java, an object can obtain a reference to itself with the *this* keyword. In EJB, however, a bean cannot use the *this* keyword and pass it to other beans because all clients invoke methods on beans indirectly through a bean's EJB object. Thus, a bean can refer to itself by using a reference to its EJB object, rather than the *this* keyword.

The Count Bean's Home Interface

To complete our stateful bean code, we must define a home interface. The home interface details how to create and destroy our Count EJB object. The code for our home interface is in Source 4.3.

Because we implement *javax.ejb.EJBHome*, our home interface gets the *remove()* destroy method for free.

The Count Bean's Deployment Descriptor

Now that we've got all our Java files for our bean, we need to define the deployment descriptor to identify the bean's settings to the container. The deployment descriptor settings we use are listed in Source 4.4.

```
package examples;

import javax.ejb.*;
import java.rmi.RemoteException;

/**
 * This is the home interface for CountBean. This interface
 * is implemented by the EJB Server's glue-code tools - the
 * implemented object is called the Home Object, and serves
 * as a factory for EJB Objects.
 *
 * One create() method is in this Home Interface, which
 * corresponds to the ejbCreate() method in the CountBean file.
 */
public interface CountHome extends EJBHome {

    /*
     * This method creates the EJB Object.
     *
     * @param val Value to initialize counter to
     *
     * @return The newly created EJB Object.
     */
    Count create(int val) throws RemoteException, CreateException;
}
```

Source 4.3 CountHome.java.

```
<!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>
  <enterprise-beans>
    <session>
      <ejb-name>Count</ejb-name>
      <home>examples.CountHome</home>
      <remote>examples.Count</remote>
      <ejb-class>examples.CountBean</ejb-class>
      <session-type>Stateful</session-type>
      <transaction-type>Container</transaction-type>
    </session>
  </enterprise-beans>
</ejb-jar>
```

Source 4.4 ejb-jar.xml.

Notice that our bean's stateful nature is defined declaratively in the deployment descriptor. We never introduce the notion of a bean being stateful in the bean code itself. This allows us to easily switch from the stateful to stateless paradigm and back.

The Count Bean's Proprietary Descriptor and Ejb-jar File

To complete our component, we need to write any proprietary files that our application server may require and package those files and our bean together into an Ejb-jar file. These steps are similar to our Hello, World! example.

One special setting we will make (which is vendor specific) will force the container to have a maximum pool of two beans. We will then create three beans and observe how the container must passivate instances to service requests.

To save space, in future examples we'll consider that the proprietary descriptors, the Ejb-jar file, and the deployment itself are implied steps. If you're really curious about how this is achieved, take a look at the source code accompanying the book.

The Count Bean's Client Code

Now that our bean is deployed, we can write some Java code to test our beans. Our client code performs the following steps:

1. We acquire a JNDI initial context.
2. We locate the home object using JNDI.
3. We use the home object to create three different Count EJB objects. Thus, we are creating three different conversations and are simulating three different clients.
4. We limited the bean pool size to two beans, so during the previous step some of the three beans must have been passivated. We print out a message during the *ejbPassivate()* callback to illustrate this.
5. Next, we call *count()* on each EJB object. This forces the container to activate the instances, restoring the conversations to memory once again. We print out a message during the *ejbActivate()* callback to illustrate this.
6. Finally, all the EJB objects are removed.

The code appears in Source 4.5.

```
package examples;

import javax.ejb.*;
import javax.naming.*;
import java.util.Properties;

/**
 * This class is a simple example of client code.
 *
 * We create 3 EJB Objects in this example, but we only allow
 * the container to have 2 in memory. This illustrates how
 * beans are passivated to storage.
 */
public class CountClient {

    public static void main(String[] args) {

        try {
            /*
             * Get System properties for JNDI initialization
             */
            Properties props = System.getProperties();

            /*
             * Get a reference to the Home Object - the
             * factory for EJB Objects
             */
            Context ctx = new InitialContext(props);
            CountHome home = (CountHome)
                javax.rmi.PortableRemoteObject.narrow(
                    ctx.lookup("CountHome"), CountHome.class);

            /*
             * An array to hold 3 Count EJB Objects
             */
            Count count[] = new Count[3];

            int countVal = 0;

            /*
             * Create and count() on each member of array
             */
            System.out.println("Instantiating beans . . . ");
            for (int i=0; i < 3; i++) {
                /*
                 * Create an EJB Object and initialize
                 * it to the current count value.
                */
            }
        }
    }
}
```

Source 4.5 CountClient.java.

```
        */
        count[i] = home.create(countVal);

        /*
         * Add 1 and print
         */
        countVal = count[i].count();

        System.out.println(countVal);

        /*
         * Sleep for 1/2 second
         */
        Thread.sleep(500);
    }

    /*
     * Let's call count() on each EJB Object to
     * make sure the beans were passivated and
     * activated properly.
     */
    System.out.println("Calling count() on beans . . . ");
    for (int i=0; i < 3; i++) {

        /*
         * Add 1 and print
         */
        countVal = count[i].count();

        System.out.println(countVal);

        /*
         * Sleep for 1/2 second
         */
        Thread.sleep(500);
    }

    /*
     * Done with EJB Objects, so remove them
     */
    for (int i=0; i < 3; i++) {
        count[i].remove();
    }
} catch (Exception e) {
    e.printStackTrace();
}
}
}
```

Source 4.5 CountClient.java (*continued*).

Running the Client

To run the client, you need to know the parameters your JNDI service provider uses. This should also be part of your container's documentation. See the book's accompanying source code for scripts.

Client-Side Output

After running the client, we see the following output:

```
Instantiating beans . . .
1
2
3
Calling count() on beans . . .
2
3
4
```

We first created three beans and then called *count()* on each. As expected, the beans incremented their values by one each during the second pass, so output is as expected. But were our beans really passivated and activated? Let's check the server log.

Server-Side Output

The container log yields the following results:

```
ejbCreate()
count()
ejbCreate()
count()
ejbCreate()
ejbPassivate()
count()
ejbPassivate()
ejbActivate()
ejbRemove()
ejbActivate()
ejbRemove()
ejbRemove()
```

As you can see from the passivation/activation messages in the log, the container is indeed passivating and activating beans to conserve system resources. Because the client-side output is correct, each of our beans' conversational state was retained properly.

Life Cycle Diagrams for Session Beans

Now that we've written a complete stateless session bean (in Chapter 3) and a complete stateful session bean (in this chapter), let's see what's happening behind the scenes.

Figure 4.4 shows the life cycle of a stateless session bean inside the container. Note that in this diagram, the client is not calling methods on the bean, since the client never accesses a bean directly. (The client always goes through the container.) In the diagram, the container (that is, the home object and EJB objects) is calling methods on our bean.

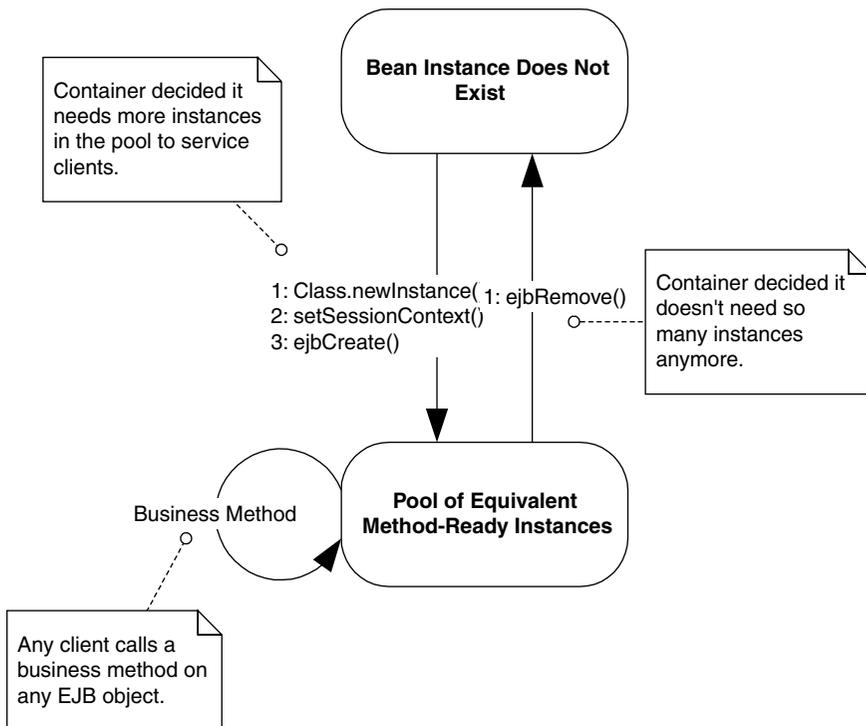


Figure 4.4 The life cycle of a stateless session bean.

Let's walk through this diagram.

First, the bean instance does not exist. Perhaps the application server has just started up.

Next, the container decides it wants to instantiate a new bean. When does the container decide it wants to instantiate a new bean? It depends on the container's policy for *pooling* beans. The container may decide to instantiate 10 beans all at once when the application server first starts because you told the container to do so using the vendor-specific files that you ship with your bean. Each of those beans are equivalent (because they are stateless) and they can be reused for many different clients.

Then the container instantiates your bean. The container calls *Class.newInstance("HelloBean.class")* on your session bean class, which is the dynamic equivalent of calling *new HelloBean()*. The container does this so that the container is not hard-coded to any specific bean name; the container is generic and works with any bean. This action calls your bean's default constructor, which can do any necessary initialization.

Next, the container calls *setSessionContext()*. This associates you with a context object, which enables you to make callbacks to the container (see Chapter 9 for some examples of these callbacks).

Then the container calls *ejbCreate()*. This initializes your bean. Note that because stateless session beans' *ejbCreate()* methods take no parameters, clients never supply any critical information that bean instances need to start up. EJB containers can exploit this and precreate instances of your stateless session beans. In general when a client creates or destroys a bean using the home object, that action might not necessarily correspond with literally creating or destroying in-memory bean objects, because the EJB container controls their life cycles to allow for pooling between heterogeneous clients.

Next, the container can call business methods on your bean. The container can call as many business methods as it wants to call. Each business method could originate from a completely different client because all bean instances are treated exactly the same. All stateless session beans think they are in the same state after a method call; they are effectively unaware that previous method calls happened. Therefore the container can dynamically reassign beans to client requests at the *per-method* level. A different stateless session bean can service *each* method call from a client. Of course, the actual implementation of reassigning beans to clients is container-specific.

Finally, the container calls *ejbRemove()*. When the container is about to remove your session bean instance, it calls your bean's *ejbRemove()* callback method. *ejbRemove()* is a clean-up method, alerting your bean that it is

about to be destroyed and allowing it to end its life gracefully. *ejbRemove()* is a required method of all beans, and it takes no parameters. Therefore there is only one *ejbRemove()* method per bean. This is in stark contrast to *ejbCreate()*, which has many forms. This makes perfect sense: Why should a destructive method be personalized for each client? (This is an analogous concept to destructors in C++.) Your implementation of *ejbRemove()* should prepare your bean for destruction. This means you need to free all resources you may have allocated.

Figure 4.5 shows the life cycle of a stateful session bean. Remember that in the diagram, the container (not the client) is calling methods on our bean instance.

The life cycle for stateful session beans is very similar to stateless session beans. The big differences are as follows:

- There is no pool of equivalent instances because each instance contains state.
- There are transitions for passivating and activating state.

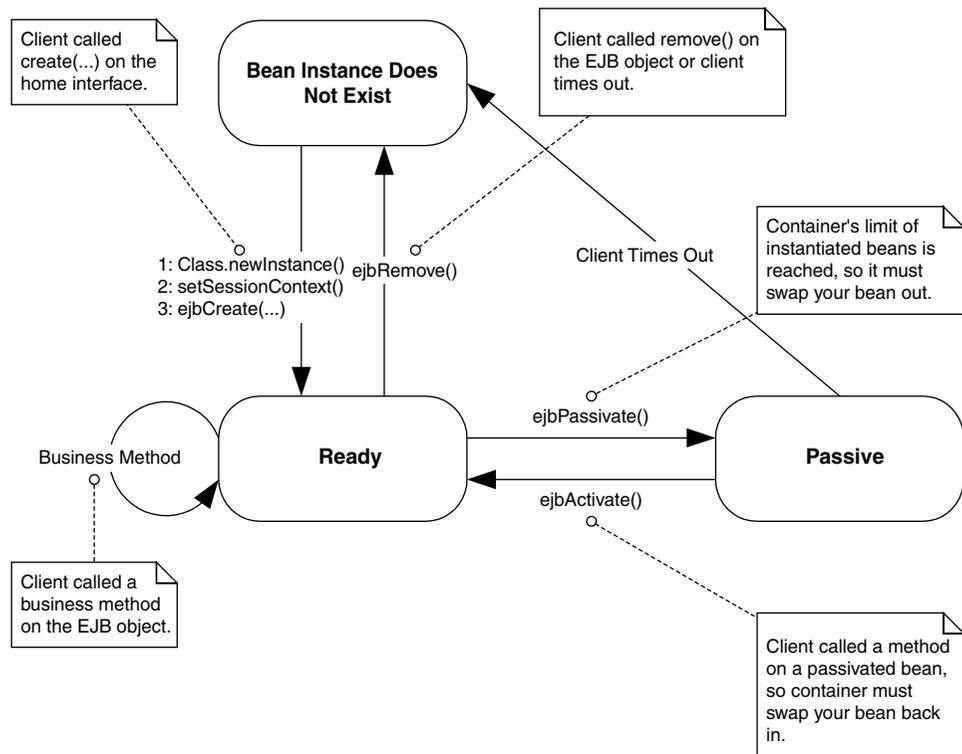


Figure 4.5 Life cycle of a stateful session bean.

Don't rely on *ejbRemove()*

Your container can call *ejbRemove()* at any time, even if the container decides that the bean's life has expired (perhaps due to a very long timeout). Note that the container may *never* call your bean's *ejbRemove()* method, such as if the container crashes or if a critical exception occurs. You must be prepared for this contingency. For example, if your bean performs shopping cart operations, it might store temporary shopping cart data in a database. Your application should provide a utility that runs periodically to remove any abandoned shopping carts from the database.

Summary

In this chapter, we learned the theoretical concepts behind session beans. We learned about achieving instance pooling with session beans, activation, and passivation. We wrote a stateful session bean that counted up and touched on session beans' life cycle.

This completes our introduction to session beans. In the next chapters, you'll learn about the more complex (and also quite interesting) entity bean. Turn the page and read on!

Introduction to Entity Beans

One of the key benefits of EJB is the power to create *entity beans*. Entity beans are *persistent objects* that can be stored in permanent storage. This means you can model your business's fundamental, underlying data as entity beans.

In this chapter, we'll cover these topics:

- The basic concepts of persistence
- A definition of entity beans, from a programmer's perspective
- The features that entity beans have to offer
- Entity bean programming concepts

This chapter is relatively theoretical, and it is meant to give you a deep foundation in entity bean programming concepts. For those of you with a traditional procedural programming background, entity beans can be a tough topic to grasp. You may need to reread this chapter a few times to really understand how things work. Make sure you've read and understood the previous chapters in this book; our discussion of entity beans will build on the knowledge you've acquired so far. We'll use these concepts with hands-on code in later chapters.

Persistence Concepts

Because entity beans are persistent objects, our discussion begins with a quick look at popular ways to persist objects.

Java Object Serialization

When you work with Java objects, in many cases you would like to capture the state of the object you're currently working with and save it to a permanent storage. One way to do this, as covered in Appendix A, is to use *object serialization*. Object serialization is an easy way to marshal an object graph into a compact representation. When you serialize an object graph, you convert the graph into a byte stream. You can then do anything you want to with that stream, such as push the data over the network (which is how Java RMI passes parameters over the network), or you can save the stream to storage, such as a file system, database, or JNDI tree. For sophisticated persistence, however, object serialization falls short in many areas.

For example, let's say we store a million serializable bank account objects onto a file system. We do this by converting the objects to their bit-blob representation and then storing the bytes on disk. Let's say we then want to retrieve all bank accounts that have balances over \$1,000. To do this with serialization, we'd have to load each and every bank account serialized bit-blob from the disk, construct the corresponding object, and then execute a method query on the object to determine if the balance is over \$1,000. We might want to perform more advanced queries as well, such as retrieving all checking accounts that have been inactive for six months. There is no efficient way to do this with object serialization.

In general querying objects stored using object serialization is expensive and cumbersome. Submitting queries against business data is an absolute necessity for large-scale applications, which makes simple object serialization unsuitable for persistent storage. Object serialization is best used in restricted domains—for network communications and simple persistence. For EJB we need a more robust persistence mechanism to address more complex querying operations.

Object-Relational Mapping

Another popular way to store Java objects is to use a traditional relational database, such as Oracle or Microsoft SQL Server. Rather than serialize each object, we could decompose each object into its constituent parts and store each part separately. For example, for a bank account object, the bank account number could be stored in one relational database field and the bank account balance in another field. When you save your Java objects, you would use JDBC or SQL/J to *map* the object data into a relational database. When you want to load your objects from the database, you would instantiate an object from that class, read the data in from the database, and then populate that object instance's fields with the relational data read in. This is shown in Figure 5.1

This mapping of objects to relational databases is a technology called *object-relational mapping*. It is the act of converting and unconverted in-memory objects to relational data. An object-relational (O/R) mapper may map your objects to any kind of relational database schema. For example, a simple object-relational mapping engine might map a Java class to a SQL table definition. An instance of that class would map to a row in that table, while fields in that instance would map to individual cells in that row. This is shown in Figure 5.2. You'll see more advanced cases of mapping data with *relationships* to other data in Chapter 11.

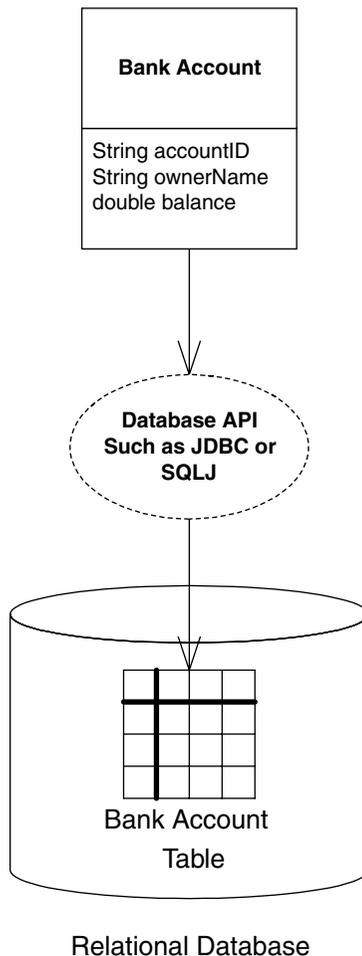


Figure 5.1 Object-relational mapping.

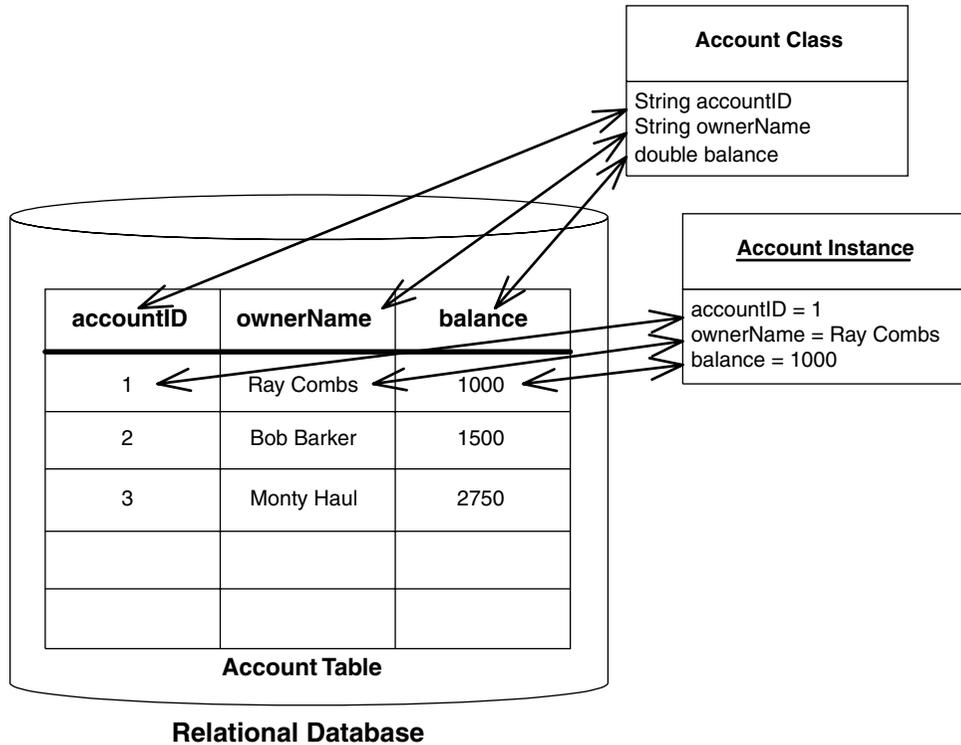


Figure 5.2 An example of object-relational mapping.

Object-relational mapping is a much more sophisticated mechanism of persisting objects than simple object serialization. By decomposing your Java objects as relational data, you can issue arbitrary queries for information. For example, you can search through all the database records that have an account balance entry greater than \$1,000 and load only the objects that fulfill this query. More advanced queries are also possible. You can also visually inspect the database data since it is not stored as bit-blobs, which is great for debugging or auditing.

Mapping of objects to relational data can be done in two ways. You can either hand-craft this mapping in your code or use an object-relational mapping product like WebGain's TOPLink or Sun's JavaBlend to automate or facilitate this mapping. Today, most users hand-craft the mapping using a database access API such as JDBC or SQL/J. Because the cost of developing and maintaining an object-relational mapping layer is significant, the object-relational mapping products are likely to be adopted as they mature.

Object Databases

An *object database management system* (ODBMS) is a persistent store that holds entire objects. In an object database, your objects are first-class citizens in the database. This means there is no O/R mapping layer—your Java objects themselves are stored as whole objects. Because of this, you don't need to program to a relational database API; rather, you program to the object database's API. This means you can sidestep object/relational mapping, resulting in simplified data access code.

Most object databases (and O/R mapping products) provide facilities to *query* persisted objects by using an *object query language* (OQL). OQL is a nice high-level interface that allows you to query object properties for arbitrary characteristics. It also adds a layer of abstraction from relational database queries.

In addition to OQL-based queries, object databases support *relationships* between objects. You can define a relationship between a Bank Account object and a Customer object and transparently navigate between them. The transparent navigation makes it easy to navigate the object model and provides excellent performance compared to SQL-based joins that are needed to perform equivalent operations in relational databases.

Object databases also have predictable performance and scalability. They offer strong integrity and security, and provide an excellent store for complex persistent objects. Certain applications go really well with object databases (geospatial or CAD/CAM, for example) are complete misfits for relational databases. Other applications map easily to relational databases, such as most business applications. For simple high-volume business transactions, relational databases typically scale better than object databases.

ObjectStore, Versant, and POET are a few of the current vendors who provide object database technology. Unfortunately, the industry has not yet fully embraced object database products. Although they are very useful for certain applications, object databases are currently limited because they do not have very many associated tools, such as reporting, tuning, and management tools.

Now that we've whetted your appetite with persistence mechanisms, let's take a look at how entity bean persistent objects are used in an EJB multitier environment.

What Is an Entity Bean?

In any sophisticated, object-oriented multitier deployment, we can draw a clear distinction between two different kinds of components deployed.

Application logic components. These components are method providers that perform common tasks. Their tasks might include the following:

- Computing the price of an order
- Billing a customer's credit card
- Computing the inverse of a matrix

Notice that these components represent actions (they're verbs). They are well suited to handle business processes.

Session beans model these application logic components very well. They often contain interesting algorithms and logic to perform application tasks. Session beans represent work being performed for a user. They represent the user session, which includes any workflow logic.

Persistent data components. These are objects (perhaps written in Java) that know how to render themselves into persistent storage. They use some persistence mechanism, such as serialization, O/R mapping to a relational database, or an object database. These kinds of objects represent *data*—simple or complex information that you'd like saved. Examples here include:

- Bank account information, such as account number and balance
- Human resources data, such as names, departments, and salaries of employees
- Lead tracking information, such as names, addresses, and phone numbers of prospective customers that you want to keep track of over time

Notice that these components represent people, places, and things (they're nouns). They are well suited to handle business data.

You might question the need for such persistent data components. Why should we deal with our business data as objects, rather than dealing with raw database data, such as relational rows? It is handy to treat data as objects because they can be easily handled and managed and because they are represented in a compact manner. We can group related data in a unified object. We associate some simple methods with that data, such as compression or other data-related activities. We can also gain implicit middleware services from an application server, such as relationships, transactions, network accessibility, and security. We can also cache that data for performance.

Entity beans are these persistent data components. Entity beans are enterprise beans that know how to persist themselves permanently to a durable storage like a database or legacy system. They are physical, storable parts of an enterprise. Entity beans store data as fields, such as bank account numbers and bank account balances. They also have methods associated with them, such as *getBankAccountNumber()* and *getAccountBalance()*. For a full discussion of when to (and when not to) use entity beans, see Chapter 12.

In some ways, entity beans are analogous to serializable Java objects. Serializable objects can be rendered into a bit-blob and then saved into a persistent store; entity beans can persist themselves in many ways, including serialization, O/R mapping, or object database persistence. Nothing in the EJB specification dictates any particular persistence mechanism.

Entity beans are very different from session beans. Session beans model a process or workflow (actions that are started by the user and that go away when the user goes away). Entity beans, on the other hand, contain core business data—product information, bank accounts, orders, lead tracking information, customer information, and more. An entity bean does not perform complex tasks or workflow logic, such as billing a customer. Rather, an entity bean *is* the customer itself. Entity beans represent persistent state objects (things that don't go away when the user goes away).

For example, you might want to read a bank account data into an entity bean instance, thus loading the stored database information into the in-memory entity bean instance's fields. You can then play with the Java object and modify its representation in memory because you're working with convenient Java objects, rather than bunches of database records. You can increase the bank account balance in-memory, thus updating the entity bean's in-memory bank account balance field. Then you can save the Java object, pushing the data back into the underlying store. This would effectively deposit money into the bank account.



The term *entity bean* is grossly overused. Sometimes it refers to an in-memory Java object instance of an entity bean class, and sometimes it refers to database data that an in-memory Java object instance represents. To make the distinction clear, we introduce two new terms:

The entity bean instance is the in-memory view into the database. It is an instance of your entity bean class.

The entity bean data (or data instance) is the physical set of data, such as a bank account record, stored in the database.

In summary, you should think of an entity bean instance as the following:

- An in-memory Java representation of persistent data
- Smart enough to know how to read itself from a storage and populate its fields with the stored data
- An object that can then be modified in-memory to change the values of data
- Persistable, so that it can be saved back into storage again, thus updating the database data

About the Files that Make up an Entity Bean

An entity bean contains the standard set of files that all EJB components have, including the remote and/or local interface, the home and/or local home interface, the enterprise bean class, and the deployment descriptor.

There are several noteworthy differences between entity bean files and other types of EJB components.

The entity bean class maps to an entity definition in a database schema. For example, an entity bean class could map to a relational table definition. In this case, an entity bean instance of that class would map to a row in that table. Your entity bean class can expose simple methods to manipulate or access that data, such as a method to decrease a bank account balance. Like a session bean class, EJB also requires that an entity bean class must fill in some standard callback methods. The EJB container will call these methods appropriately to manage the entity bean.

The primary key class makes every entity bean different. For example, if you have 1 million different bank account entity beans, each bank account needs to have a unique ID (such as a bank account ID string) that can never be repeated in any other bank account. A primary key is an object that may contain any number of attributes. This could be whatever data necessary to uniquely identify an entity bean data instance. In some advanced cases, when the entity bean represents a complex relationship, the primary key might be an entire object. EJB gives you the flexibility to define what your unique identifier is by including a primary key class with your entity bean. The one rule is that your primary key class must be serializable and follow the rules for Java object serialization. The rules for object serialization are in Appendix A.

Features of Entity Beans

Let's take a look at the features of entity beans.

Entity Beans Survive Failures

Entity beans are long lasting. They survive critical failures, such as application servers crashing, or even databases crashing. This is because entity beans are just representations of data in a permanent, fault-tolerant underlying storage. If a machine crashes, the entity bean can be reconstructed in memory. All we need to do is read the data back in from the permanent database and instanti-

ate an entity bean Java object instance whose fields contain the data read in from the database.

This is a huge difference between session and entity beans. Entity beans have a life cycle much longer than a client's session, perhaps years long, depending on how long the data sits in the database. In fact, the database records representing an object could have existed before the company even decided to go with a Java-based solution, because a database structure can be language-independent. This makes sense—you definitely would want your bank account to last for a few years, regardless of technology changes in your bank.

Entity Bean Instances Are a View into a Database

When you load entity bean data into an in-memory entity bean instance, you read in the data stored in a database so that you can manipulate the data within a Java Virtual Machine. However, *you should think of the in-memory object and the database itself as one and the same*. This means if you update the in-memory entity bean instance, the database should automatically be updated as well. You should *not* think of the in-memory entity bean as a separate version of the data in the database. The in-memory entity bean is simply a *view* or *lens* into the database.

Of course, in reality there are multiple physical copies of the same data: the in-memory entity bean instance and the entity bean data itself stored in the database. Therefore, there must be a mechanism to transfer information back and forth between the Java object and the database. This data transfer is accomplished with two special methods that your entity bean class must implement, called *ejbLoad()* and *ejbStore()*.

ejbLoad() reads the data in from the persistent storage into the entity bean's in-memory fields.

ejbStore() saves your bean instance's current fields to the underlying data storage. It is the complement of *ejbLoad()*.

So who decides when to transfer data back and forth between the in-memory bean and the database? That is, who calls *ejbLoad()* and *ejbStore()*? The answer is your EJB container. *ejbLoad()* and *ejbStore()* are callback methods that the container invokes. They are management methods required by EJB. The container worries about the proper time to call *ejbLoad()* and *ejbStore()*—this is one of the value-adds of the container. This is shown visually in Figure 5.3.

Your beans should be prepared to accept an *ejbLoad()* or *ejbStore()* call at almost any time (but not during a business method). The container automatically

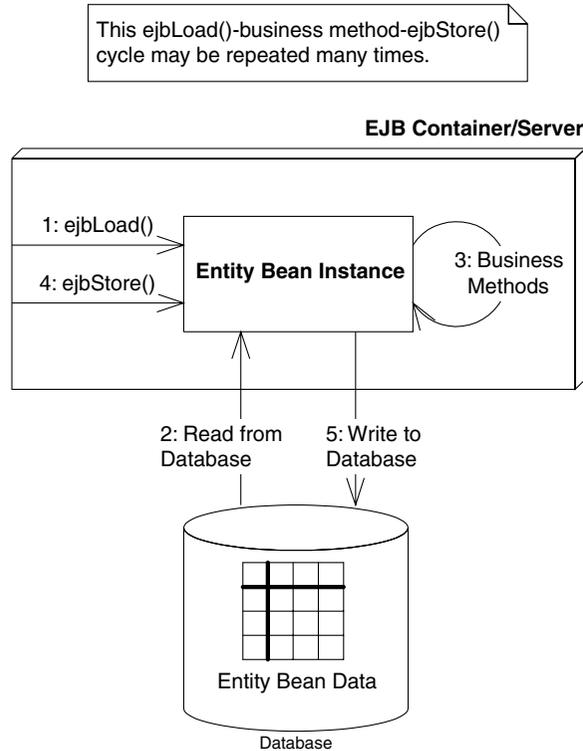


Figure 5.3 Loading and storing an entity bean.

figures out when each of your instances needs to be refreshed depending on the current transactional state (see Chapter 10). This means that you never explicitly call your own `ejbLoad()` or `ejbStore()` methods. This is one of the advantages of EJB: You don't have to worry about synchronizing your objects with the underlying database. The EJB black box handles it for you. That is why you can think of the entity bean and the database as the same; there should never be a time when the two are transactionally out of sync.

Several Entity Bean Instances May Represent the Same Underlying Data

Let's consider the scenario in which many threads of execution want to access the same database data simultaneously. In banking, interest might be applied to a bank account, while at the same time a company directly deposits a check into that same account. In e-commerce, many different client browsers may be simultaneously interacting with a catalog of products.

To facilitate many clients accessing the same data, we need to design a high-performance access system to our entity beans. One possibility is to allow many clients to share the same entity bean instance; that way, an entity bean could service many client requests simultaneously. While this is an interesting idea, it is not very appropriate for EJB, for two reasons. First, if we'd like an entity bean instance to service many concurrent clients, we'd need to make that instance thread-safe. Writing thread-safe code is difficult and error-prone. Remember that the EJB value proposition is rapid application development. Mandating that component vendors produce stable thread-safe code does not encourage this. Second, having multiple threads of execution makes transactions almost impossible to control by the underlying transaction system. For these reasons, EJB dictates that only a single thread can ever be running within a bean instance. With session beans and message-driven beans, as well as entity beans, all bean instances are single-threaded.

Mandating that each bean can service only one client at a time could result in performance bottlenecks. Because each instance is single-threaded, clients need to effectively run in lockstep, each waiting their turn to use a bean. This could easily grind performance to a halt in any large enterprise deployment.

To boost performance, we could allow containers to instantiate multiple instances of the same entity bean class. This would allow many clients to concurrently interact with separate instances, each representing the same underlying entity data. Indeed, this is exactly what EJB allows containers to do. Thus, client requests do not necessarily need to be processed sequentially, but rather concurrently.

Having multiple bean instances represent the same data now raises a new problem: data corruption. If many bean instances are representing the same underlying data via caching (see Chapter 14), we're dealing with multiple in-memory cached replicas. Some of these replicas could become stale, representing data that is not current.

To achieve entity bean instance cache consistency, each entity bean instance needs to be routinely synchronized with the underlying storage. The container synchronizes the bean with the underlying storage by calling the bean's *ejbLoad()* and *ejbStore()* callbacks, as described in the previous section.

The frequency with which beans are synchronized with an underlying storage is dictated by *transactions*, a topic we cover in Chapter 10. Transactions allow each client request to be isolated from every other request. They enable clients to *believe* they are dealing with a single in-memory bean instance, when in fact many instances are behind the scenes. Transactions give clients the illusion that they have exclusive access to data when in fact many clients are touching the same data.

Entity Bean Instances Can Be Pooled

Let's say you've decided to author your own EJB container/server. Your product is responsible for instantiating entity beans as necessary, with each bean representing data in an underlying storage. As clients connect and disconnect, you could create and destroy beans as necessary to service those clients.

Unfortunately this is not a scalable way to build an application server. Creation and destruction of objects is expensive, especially if client requests come frequently. How can we save on this overhead?

One thing to remember is that an entity bean class describes the fields and rules for your entity bean, but it does not dictate any specific data. For example, an entity bean class may specify that all bank accounts have the following fields:

- The name of the bank account owner
- An account ID
- An available balance

That bean class can then represent any distinct instance of database data, such as a particular bank account record. The class itself, though, is not specific to any particular bank account.

To save precious time instantiating objects, entity bean instances are therefore recyclable objects and may be pooled depending on your container's policy. The container may pool and reuse entity bean instances to represent different instances of the same type of data in an underlying storage. For example, a container could use a bank account entity bean instance to represent different bank account records. When you're done using an entity bean instance, that instance may be assigned to handle a different client's request and may represent different data. The container performs this by dynamically assigning the entity bean instance to different client-specific EJB objects. Not only does this save the container from unnecessarily instantiating bean instances, but this scheme also saves on the total amount of resources held by the system. We show this in Figure 5.4

Instance pooling is an interesting optimization that containers may provide, and it is not at all unique to entity beans. However, complications arise when reassigning entity bean instances to different EJB objects. When your entity bean is assigned to a particular EJB object, it may be holding resources such as socket connections. But when it's in the pool, it may not need that socket. Thus, to allow the bean to release and acquire resources, your entity bean class must implement two callback methods.

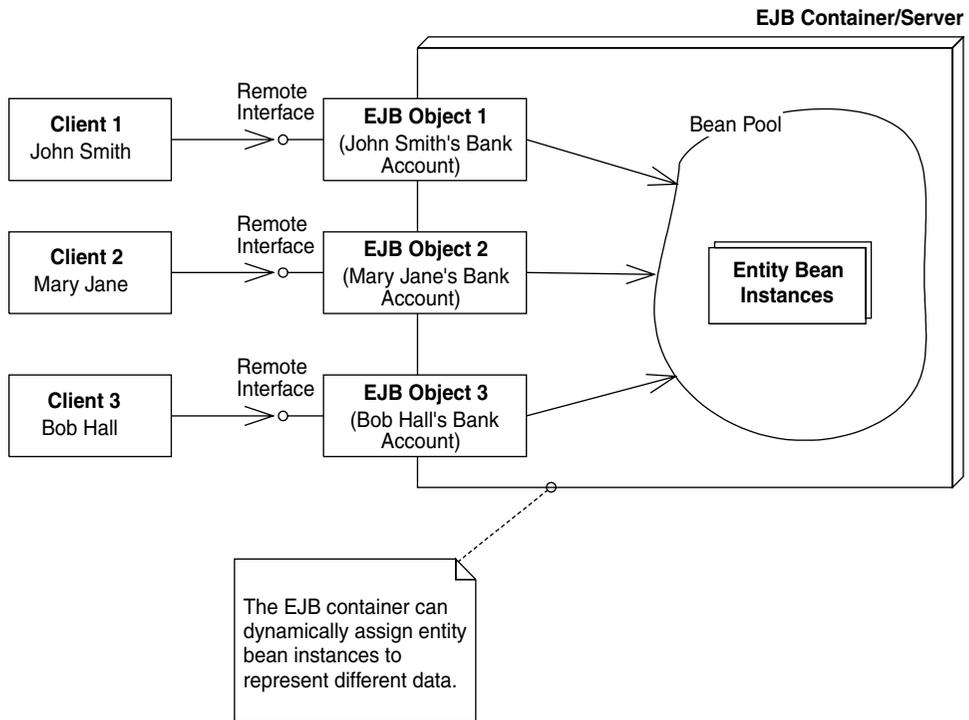


Figure 5.4 EJB container pooling of entity beans.

ejbActivate() is the callback that your container will invoke on your bean instance when transitioning your bean *out of* a generic instance pool.

This process is called *activation*, and it indicates that the container is associating your bean with a specific EJB object and a specific primary key. Your bean's *ejbActivate()* method should acquire resources, such as sockets, that your bean needs when assigned to a particular EJB object.

ejbPassivate() is the callback that your container will invoke when transitioning your bean *into* a generic instance pool. This process is called *passivation*, and it indicates that the container is disassociating your bean from a specific EJB object and a specific primary key. Your bean's *ejbPassivate()* method should release resources, such as sockets, that your bean acquired during *ejbActivate()*.

When an entity bean instance is passivated, it must not only release held resources but also save its state to the underlying storage; that way, the storage is updated to the latest entity bean instance state. To save the instance's fields

to the database, the container invokes the entity bean's *ejbStore()* method prior to passivation. Similarly, when the entity bean instance is activated, it must not only acquire any resources it needs but also load the most recent data from the database. To load data into the bean instance, the container invokes the entity bean's *ejbLoad()* method after activation. This is shown in Figure 5.5.

There Are Two Ways to Persist Entity Beans

Since entity beans map to a storage, someone needs to actually write the database access code.

A *bean-managed persistent entity bean* is an entity bean that must be persisted by hand. In other words, you as the component developer must write code to translate your in-memory fields into an underlying data store, such as a rela-

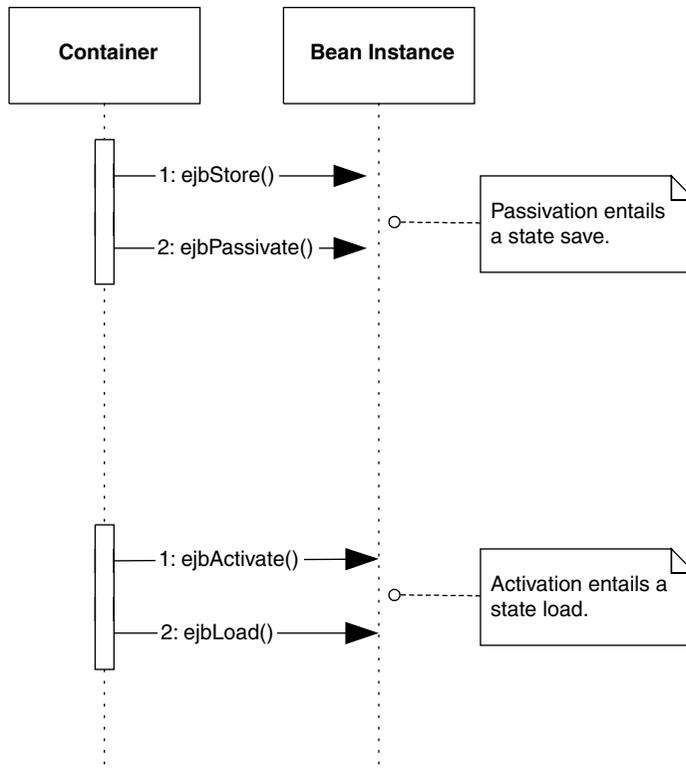


Figure 5.5 Passivation of entity beans entails a state save, and activation entails a state load.

tional database or an object database. You handle the persistent operations yourself—including saving, loading, and finding data—within the entity bean. Therefore, you must write to a persistence API, such as JDBC or SQL/J. For example, with a relational database, your entity bean could perform an SQL INSERT statement via JDBC to stick some data into a relational database. You could also perform an SQL DELETE statement via JDBC to remove data from the underlying store.

EJB offers an alternative to bean-managed persistence: You can have your EJB container perform your persistence for you. This is called *container-managed persistence*. In this case, you would usually strip your bean of any persistence logic. Then, you inform the container about how you'd like to be persisted by using the container's tools. The container then *generates* the data access code for you. For example, if you're using a relational database, the container may automatically perform SQL INSERT statements to create database data. Similarly, it will automatically perform SQL DELETE statements to remove database data, and it will handle any other necessary persistent operations. Even if you are not working with a relational database, you can have your container persist for you. If your container supports a nonrelational persistent store, such as an object database or a VSAM file, the container will generate the appropriate logic as necessary. In fact, you can wait until deployment time before you setup the O/R mapping, which is great because you can write storage-independent data objects, and reuse them in a variety of enterprise environments.

Container-managed persistence reduces the size of your beans tremendously because you don't need to write JDBC code—the container handles all the persistence for you. This is a huge value-add feature of EJB. Of course, it is still evolving technology. Once we've written a few entity beans, we'll review the trade-offs of bean-managed versus container-managed persistence (see Chapter 12).

Creation and Removal of Entity Beans

As we mentioned earlier, entity beans are a view into a database, and you should think of an entity bean instance and the underlying database as one and the same (they are routinely synchronized). Because they are one and the same, the initialization of an entity bean instance should entail initialization of database data. Thus, when an entity bean is initialized in memory during *ejbCreate()*, it makes sense to create some data in an underlying database that correlates with the in-memory instance. That is exactly what happens with entity beans. When a bean-managed persistent entity bean's *ejbCreate()* method is called, the *ejbCreate()* method is responsible for creating database data. Similarly, when a bean-managed persistent entity bean's *ejbRemove()* method is called, the *ejbRemove()* method is responsible for removing database

data. If container-managed persistence is used, the container will modify the database for you, and you can leave these methods empty of data access logic.

Let's look at this in more detail.

Understanding How Entity Beans Are Created and Destroyed

In EJB, remember that clients do not directly invoke on beans—they invoke an EJB object proxy. The EJB object is generated through the home object. Therefore, for each *ejbCreate()* method signature you define in your bean, you must define a corresponding *create()* in the home interface. The client calls the home object's *create()*, which delegates to your bean's *ejbCreate()*.

For example, let's say you have a bank account entity bean class called *AccountBean*, with a remote interface *Account*, home interface *AccountHome*, and primary key class *AccountPK*. Given the following *ejbCreate()* method in *AccountBean*:

```
public AccountPK ejbCreate(String accountID, String owner) throws . . .
```

you must have this *create()* in your home interface (notice there is no “*ejb*” prefix):

```
public Account create(String accountID, String owner) throws . . .
```

Notice that there are two different return values here. The bean instance returns a primary key (*AccountPK*), while the home object returns an EJB object (*Account*). This makes sense—the bean returns a primary key to the container (that is, to the home object) so that the container can identify the bean. Once the home object has this primary key, it can generate an EJB object and return that to the client. We show this process more rigorously with the sequence diagram in Figure 5.6.

To destroy an entity bean's data in a database, the client must call *remove()* on the EJB object or home object. This method causes the container to issue an *ejbRemove()* call on the bean. Figure 5.7 shows the relationship between *remove()* and *ejbRemove()*. Note that *remove()* can be called on either the home object or the EJB object. The figure happens to assume bean-managed persistence.

Note that *ejbRemove()* does not mean the in-memory entity bean instance is going to be destroyed; *ejbRemove()* destroys only database data. The bean instance can be recycled to handle a different database data instance, such as a bank account bean representing different bank accounts.

ejbRemove() is a required method of all entity beans, and it takes no parameters. There is only one form of *ejbRemove()*. With entity beans, *ejbRemove()* is not

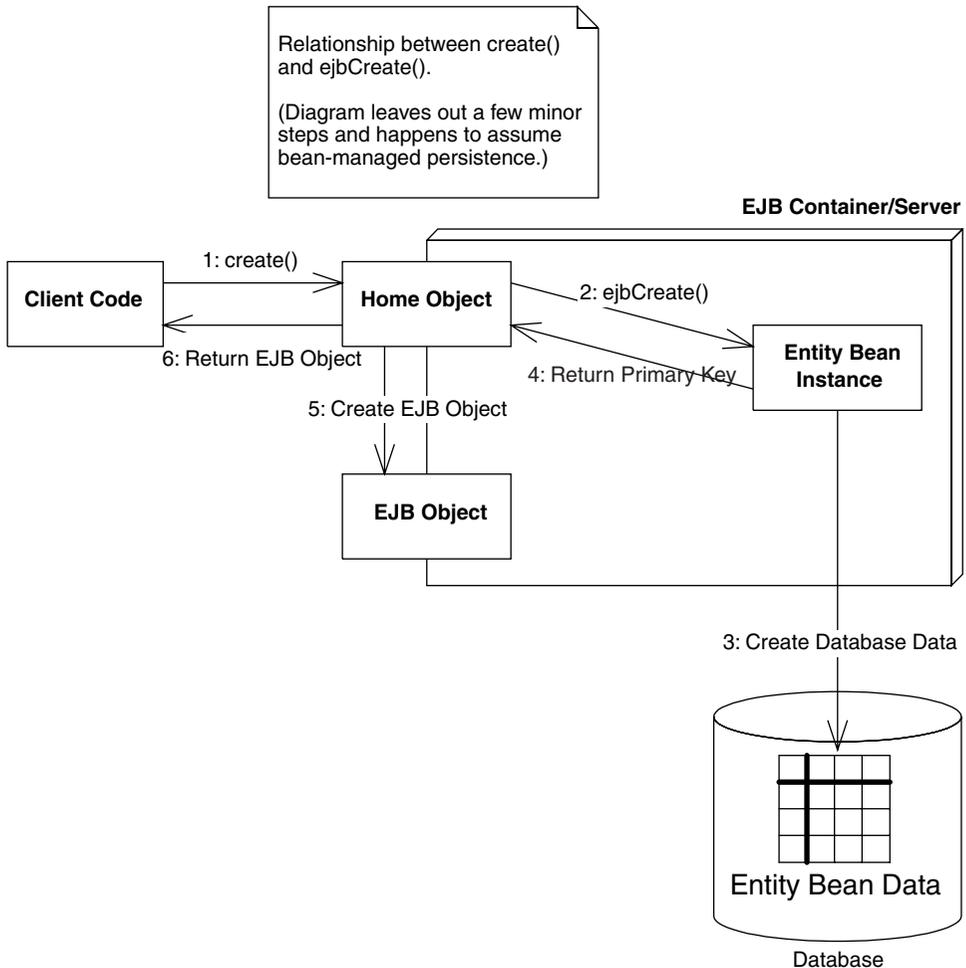


Figure 5.6 Creating an entity bean and EJB object.

called if the client times out because the lifetime of an entity bean is longer than the client's session.

Entity Beans Can Be Found

Because entity bean data is uniquely identified in an underlying storage, entity beans can also be *found* rather than created. Finding an entity bean is analogous to performing a SELECT statement in SQL. With a SELECT statement, you're searching for data from a relational database store. When you find an

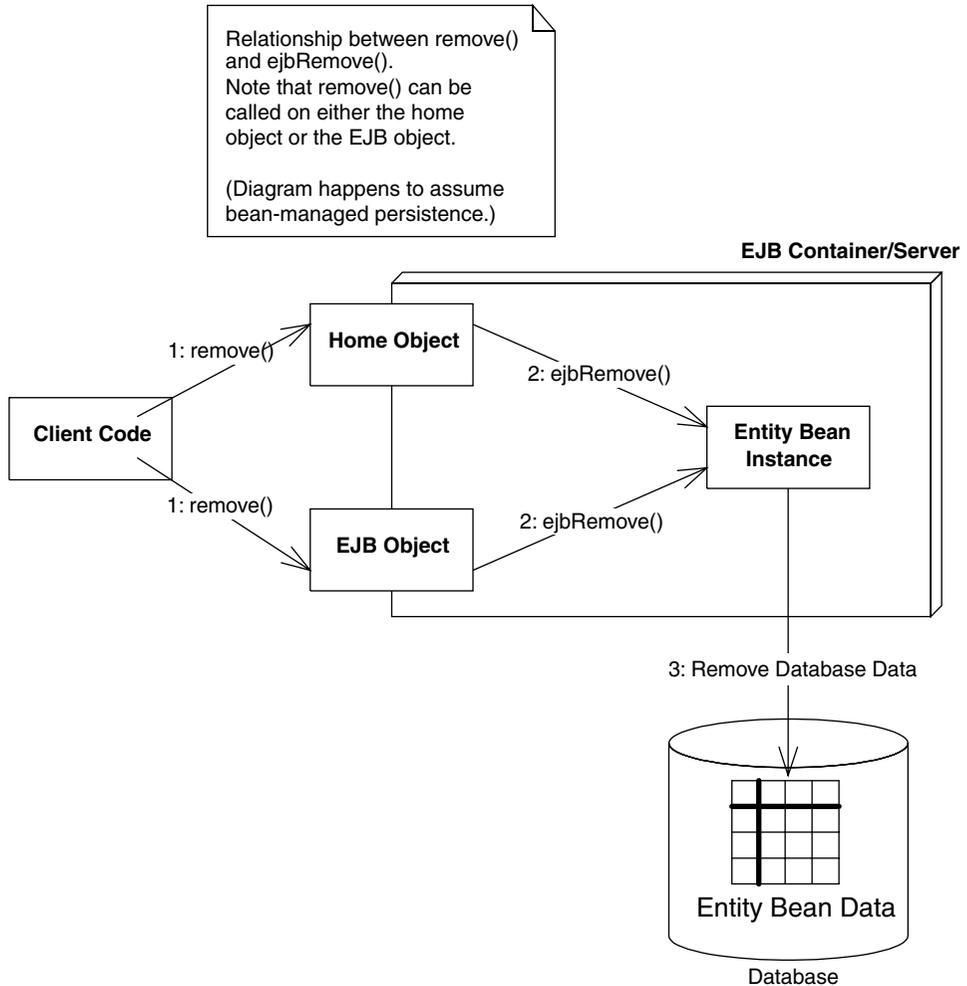


Figure 5.7 Destroying an entity bean's data representation.

entity bean, you're searching a persistent store for some entity bean data. This differs from session beans because session beans cannot be found: They are not permanent objects, and they live and die with the client's session.

You can define many ways to find an entity bean. You list these ways as methods in your entity bean home interface. These are called *finder* methods. Your home interface exposes finder methods in addition to methods for creating and destroying entity beans. This is the one big difference between an entity bean's home interface and other types of beans; the other bean types do not have finder methods.

You Can Modify Entity Bean Data without Using EJB

Usually you will create, destroy, and find entity bean data by using the entity bean's home object. But you can interact with entity beans another way, too: by directly modifying the underlying database where the bean data is stored. For example, if your entity bean instances are being mapped to a relational database, you can simply delete the rows of the database corresponding to an entity bean instance (see Figure 5.8). You can also create new entity bean data and modify existing data by directly touching the database. This may be necessary if you have an investment in an existing system that touches a database directly.



These external database updates could raise cache consistency issues if you're choosing to cache your entity beans. See Chapter 14 for more on this.

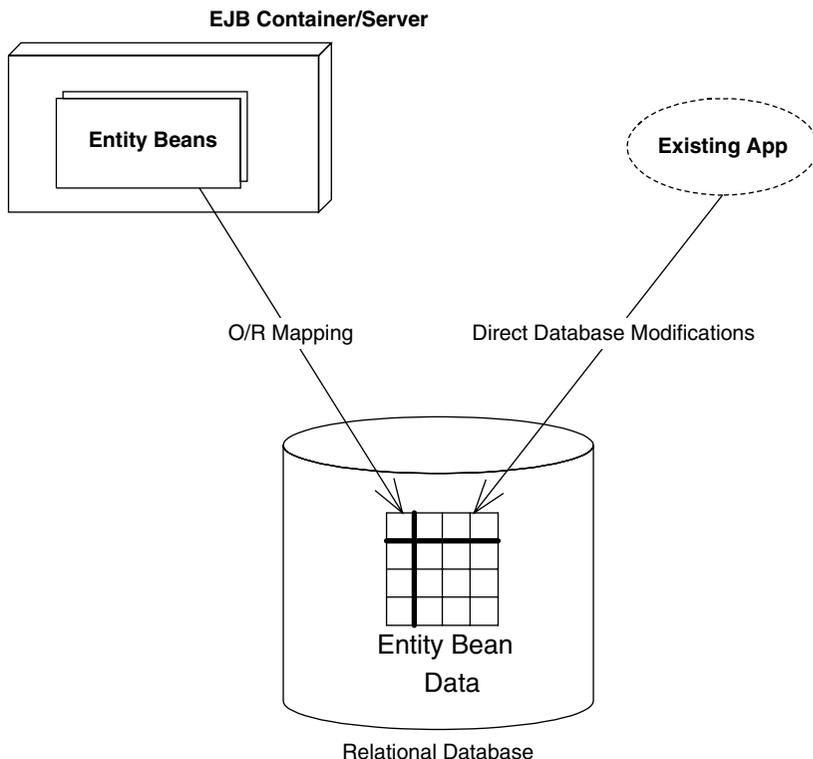


Figure 5.8 Modifying an entity bean's database representation manually.

Entity Contexts

As you learned in Chapter 3, all enterprise beans have a *context object* that identifies the environment of the bean. These context objects contain environment information that the EJB container sets. Your beans can access the context to retrieve all sorts of information, such as transaction and security information. For entity beans, the interface is *javax.ejb.EntityContext*.

We provide a refresher of the *javax.ejb.EJBContext* methods in Source 5.1. Appendix E explains the meaning of each method.

Entity contexts add the following methods on top of the generic EJB context, shown in Source 5.2.

Let's look at each of these methods in more detail.

getEJBLocalObject() / getEJBObject()

Call this to retrieve the current, client-specific EJB object that is associated with the entity bean. Remember that clients invoke on EJB objects, not on entity beans directly. Therefore, you can use the returned EJB object as a way to pass a reference to yourself, simulating the *this* argument in Java. *getEJBLocalObject()* is the same, except it gets the more optimized EJB local object.

```
public interface javax.ejb.EJBContext {
    public javax.ejb.EJBHome getEJBHome();
    public javax.ejb.EJBLocalHome getEJBLocalHome();
    public java.security.Principal getCallerPrincipal();
    public boolean isCallerInRole(java.lang.String);
    public void setRollbackOnly();
    public boolean getRollbackOnly();
}
```

Source 5.1 The *javax.ejb.EJBContext* interface.

```
public interface javax.ejb.EntityContext
    extends javax.ejb.EJBContext {
    public javax.ejb.EJBLocalObject getEJBLocalObject();
    public javax.ejb.EJBObject getEJBObject();
    public java.lang.Object getPrimaryKey();
}
```

Source 5.2 The *javax.ejb.EntityContext* interface.

getPrimaryKey()

getPrimaryKey() is new to entity beans. It retrieves the primary key that is currently associated with this entity bean instance. Primary keys uniquely identify an entity bean. When an entity bean is persisted in storage, the primary key can be used to uniquely retrieve the entity bean because no two entity bean database data instances can ever have the same primary key.

Why would you want to call *getPrimaryKey()*? You call it whenever you want to figure out with which database data your instance is associated. Remember that entity bean instances can be reused and pooled, as we saw in Figure 5.4. When the container wants to switch an entity bean instance from one data instance to another, the container needs to passivate and activate that entity bean instance. When this happens, your bean instance may switch to a different data instance and thus a different primary key. But your entity bean instance is never told this explicitly when it is activated. Rather, your entity bean must perform a *getPrimaryKey()* callback to the entity context to figure out what data it should be dealing with.

Thus, when you have an entity bean that's performing any persistent work (with bean-managed persistence), you should be calling *getPrimaryKey()* whenever you need to figure out what data you're bean is associated with. This is very useful, for example, in the following methods:

ejbLoad(). Recall that *ejbStore()* and *ejbLoad()* are bean callbacks to synchronize a bean instance with an underlying storage. *ejbStore()* saves data to storage, and *ejbLoad()* reads data from storage. When the container calls *ejbStore()*, your bean knows exactly what data to save because the bean instance has the data in memory. But when the container calls *ejbLoad()*, how does your bean know what data to load? After all, bean instances are pooled and can be dynamically assigned to different data. The answer is to use *getPrimaryKey()*; it will tell you what primary key you should be looking for in the underlying storage when loading database data.

ejbRemove(). Recall that *ejbCreate()* and *ejbRemove()* are callbacks for creating and removing data from an underlying storage, respectively. When the container calls *ejbCreate()*, your bean knows exactly what data to create in the database because your bean has received information in the parameters of *ejbCreate()*. But when the container calls *ejbRemove()*, how does your bean know what data to remove? Because bean instances are pooled and dynamically assigned to handle different data instances, you might be deleting the wrong data. Thus, you must call *getPrimaryKey()* to figure out what data, keyed on the primary key, your bean should remove from the database.

It is important to consider bean pooling when writing your enterprise beans, and *getPrimaryKey()* is the key to knowing what data your bean is representing.

Summary

In this chapter, we've taken the first steps toward developing with entity beans. We started by learning about various persistence mechanisms, including object serialization, object/relational mapping, and persistence to pure object databases. We then looked at exactly what an entity bean is, and we saw the files included with an entity bean component. After surveying their features, we took a look at entity contexts.

But the best is yet to come. In the coming chapters, we'll learn hands-on about entity bean programming. Chapter 6 explains bean-managed persistent entity beans and guides you through the steps in developing them using JDBC. Chapter 7 continues with container-managed persistent entity beans. In Chapter 11 we'll learn how to program entity beans that require relationships. By the time you're through, you'll be armed to create your own entity beans in enterprise deployments.

Writing Bean-Managed Persistent Entity Beans

In this chapter, we'll demonstrate how to program *bean-managed persistent entity beans*, the first of two flavors of entity beans. When you code these types of entity beans, you must provide your own data access logic. You are responsible for providing the implementation to map your entity bean instances to and from storage. To do this, you typically use a database API such as JDBC or SQL/J. This is in stark contrast to container-managed persistent entity beans, which have their data access handled for them by the EJB container. This chapter will teach you the basics of bean-managed persistence and show you how to build a simple bean-managed entity bean using JDBC.

Entity Bean Coding Basics

To write an entity bean class, you write a Java class that implements the `javax.ejb.EntityBean` interface. This interface defines a number of required methods that your entity bean class must implement. Most of these methods are management methods called by your EJB container. The following code (Source 6.1 and 6.2) details `javax.ejb.EntityBean`, as well as its parent, `javax.ejb.EnterpriseBean` (exceptions are omitted).

The `javax.ejb.EnterpriseBean` interface defines no methods—it is simply a marker interface. The `javax.ejb.EntityBean` interface defines callback methods that your bean must implement. The container will call these methods whenever it wishes.

```
public interface javax.ejb.EnterpriseBean implements java.io.
Serializable {
}
```

Source 6.1 The `javax.ejb.EnterpriseBean` interface.

```
public interface javax.ejb.EntityBean
extends javax.ejb.EnterpriseBean {
    public void setEntityContext(javax.ejb.EntityContext);
    public void unsetEntityContext();
    public void ejbRemove();
    public void ejbActivate();
    public void ejbPassivate();
    public void ejbLoad();
    public void ejbStore();
}
```

Source 6.2 The `javax.ejb.EntityBean` interface.

Java Database Connectivity (JDBC)

This chapter uses Java Database Connectivity (JDBC). JDBC is a standard Java extension that allows Java programmers to access relational databases. By using JDBC, Java programmers can represent database connections, issue SQL statements, process database results, and more in a relatively portable way. Clients program to the unified JDBC API, which is implemented by a *JDBC Driver*, an adapter that knows how to talk to a particular database in a proprietary way (see Figure 6.1). JDBC is similar to the Open Database Connectivity (ODBC) standard, and the two are quite interoperable through JDBC-ODBC bridges. JDBC contains built-in support for database connection pooling, further enhancing the database independence of your application code.

All entity bean classes, both bean-managed persistent and container-managed persistent, must implement the `javax.ejb.EntityBean` interface. This interface defines callback methods that the container invokes on your beans. There are additional methods you also may define, such as methods to create and find your entity beans.

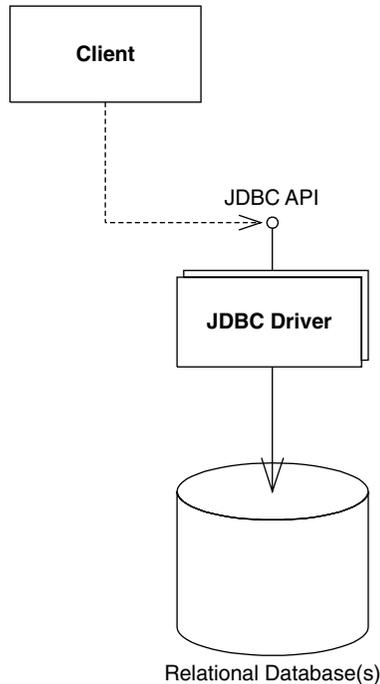


Figure 6.1 Java Database Connectivity.

Table 6.1 is a preview of what you should implement in each method in your entity bean, assuming your entity bean's persistence is bean-managed. For now, take a quick glance at the chart; you should refer back to it when reading through the code in this chapter or when programming your own entity bean classes.

Finding Existing Entity Beans: `ejbFind()`

Notice from Table 6.1 that we have methods labeled `ejbFind()`. Finder methods are used to find an existing entity bean in storage. Finder methods do not create new database data—they simply load old entity bean data.



You only define `ejbFind()` methods when you use bean-managed persistence. With container-managed persistence, these method implementations are generated for you.

Table 6.1 Descriptions and Implementation Guidelines for Bean-Managed Persistent Entities

METHOD	EXPLANATION	TYPICAL IMPLEMENTATION
setEntityContext()	<p>If the container wants to increase its pool size of bean instances, it will instantiate a new entity bean instance. Following this, the container calls the instance's <code>setEntityContext()</code>. This method associates a bean with <i>context information</i>—information about the bean's environment. Once this method is called, the bean can access information about its environment.</p>	<p>Stick the entity context somewhere, such as in a member variable. You can access the context later to acquire environment information, such as security information, from the container.</p> <p>You should also request any resources your instance will need regardless of what data the bean represents.</p> <p><i>The bean is now in a pool, does not have any specific database data inside of it, and is not bound to any particular EJB object.</i></p>
ejbFind<...>(<...>) (also called <i>finder</i> methods)	<p>While your bean instance is still in the pool, the container can use your bean to service a <i>finder</i> method. Finder methods locate one or more existing entity bean data instances in the underlying persistent store. You must define at least one finder method—<code>ejbFindByPrimaryKey()</code>.</p>	<p>Search through a data store using a storage API such as JDBC or SQL/J. For example, you might perform a relational query such as "SELECT id FROM accounts WHERE balance > 0."</p> <p>When you've found some data, return the primary keys for that data back to the container by creating one or more primary key Java object instances. The container will then create EJB objects for the client to invoke on and possibly associate some entity bean instances with those EJB objects.</p> <p><i>Those entity bean instances are no longer in the pool—they now have specific database data inside of them, and they are bound to particular EJB objects.</i></p>

METHOD

`ejbHome(<...>`
`(<...>)`

EXPLANATION

Sometimes you need methods on an entity bean that are not specific to any given data instance (or row). For example, counting the total number of accounts in a table. You can write *home methods* to perform these operations. The home methods are special business methods because they are called from a bean in the pool, before the bean is associated with any specific data. Clients call home methods from the home interface or local home interface.

TYPICAL IMPLEMENTATION

Perform your global operations, such as counting up the rows in a database via JDBC, and return the result to the client.

`ejbCreate(<...>)`

Note: You do not need to write any `ejbCreate()` methods if you don't want EJB clients to be able to create new database data.

Instead, you could mandate that all data is created through other means, such as via direct database inserts or through batch files.

When a client calls `create()` on a home object, the container calls `ejbCreate()` on a pooled bean instance. `ejbCreate()` methods are responsible for creating new database data and for initializing your bean. Each `ejbCreate()` method you define gives clients a different way to create your entity beans, such as methods to create a checking account and a savings account.

Make sure the client's initialization parameters are valid. Explicitly create the database representation of the data via a storage API like JDBC or SQL/J, typically through a SQL INSERT. Then return a primary key to the container, so that the container can identify which data your instance represents.

Your entity bean instance is no longer in the pool—it now has specific database data inside of it. The container will bind your instance to a particular EJB objects.

continues

Table 6.1 Descriptions and Implementation Guidelines for Bean-Managed Persistent Entities (*continued*)

METHOD	EXPLANATION	TYPICAL IMPLEMENTATION
ejbPostCreate(<...>)	Your bean class must define one <code>ejbPostCreate()</code> for each <code>ejbCreate()</code> . Each pair must accept the same parameters. The container calls <code>ejbPostCreate()</code> right after <code>ejbCreate()</code> .	The container calls <code>ejbPostCreate()</code> after it has associated your bean instance with an EJB object. You can now complete your initialization by doing anything you need to that requires that EJB object, such as passing your bean's EJB object reference to other beans.
ejbActivate()	When a client calls a business method on an EJB object, but no entity bean instance is bound to the EJB object, the container needs to take a bean from the pool and transition it into a ready state. This is called <i>activation</i> . Upon activation, the <code>ejbActivate()</code> method is called by the EJB container.	You might also use this method to reset certain transaction-related parameters. For example, you could keep a data status flag in the bean to indicate whether a field has been changed. Because the bean instance may have been used before, these fields might have dirty data.
ejbLoad()	The EJB container calls this to load database data into your bean instance (typically a SQL SELECT), based on the current transactional state.	Acquire any resources, such as socket connections, that your bean needs to service a particular client when it is moved into the ready state. Note that you should <i>not</i> read the entity bean data from the database in this method. That is handled by a separate method, <code>ejbLoad()</code> , which is called right after <code>ejbActivate()</code> .
ejbStore()	The EJB container calls this to update the database to the new values of your in-memory fields, thus synchronizing the database. The current transactional state dictates when this method is called. This method is also called during passivation, directly before <code>ejbPassivate()</code> .	First, your bean instance must figure out what data it should load. Call the <code>getPrimaryKey()</code> method on the entity context; that will tell your bean what data it should be loading. Next, read database data into your bean via a storage API such as JDBC or SQL/J. Explicitly update the database representation of the data via a storage API like JDBC. Typically, you'll write a number of your member variable fields out to disk through a SQL UPDATE.

METHOD	EXPLANATION	TYPICAL IMPLEMENTATION
<code>ejbPassivate()</code>	<p>The EJB container calls this method when it wants to return your entity bean to the pool. This is called <i>passivation</i> and is the opposite of activation. On <i>passivation</i>, the <code>ejbPassivate()</code> method is called by the EJB container.</p>	<p>Release any resources, such as socket connections, that you allocated in <code>ejbActivate()</code> and that your bean was holding during the ready state for a particular client.</p> <p>You should not save the entity bean data into the database in this method. That is handled by a separate method, <code>ejbStore()</code>, which is called right before <code>ejbPassivate()</code>.</p>
<code>ejbRemove()</code>	<p>Destroys database data. It is not used to destroy the Java object; the object can be pooled and reused for different data.</p>	<p>First, figure out what data you should destroy via <code>getPrimaryKey()</code> on the <code>EntityContext</code>. Then explicitly delete the database representation of the data via a storage API like JDBC, typically through a SQL DELETE.</p>
<code>unsetEntityContext()</code>	<p>This method disassociates a bean from its environment. The container calls this right before your entity bean instance is destroyed (when it wants to reduce the pool size).</p>	<p>Release any resources you allocated during <code>setEntityContext()</code>, and get ready to be garbage collected.</p>

You can have many different finder methods, all of which perform different operations. Here are some examples:

```

/**
 * Finds the unique bank account indexed by primary key
 */
public AccountPK ejbFindByPrimaryKey(AccountPK key)
    throws FinderException { . . . }

/**
 * Finds all the product entity beans. Returns a Collection
 * of primary keys.
 */
public Collection ejbFindAllProducts()
    throws FinderException { . . . }

/**
 * Finds all Bank Accounts that have at least a minimum balance.
 * Returns a Collection of primary keys.
 */
public Collection ejbFindBigAccounts(int minimum)
    throws FinderException { . . . }

/**
 * Finds the most recently placed order
 */
public OrderPK ejbFindMostRecentOrder()
    throws FinderException { . . . }

```

Here are some of the rules about finder methods.

All finder methods must begin with *ejbFind*. This is simply a syntactic rule.

You must have at least one finder method, called *ejbFindByPrimaryKey*.

This method finds one unique entity bean instance in the database based on its unique primary key. Because every entity bean has an associated primary key, it makes sense that every entity bean class supports this method.

You can have many different finder methods, each with different names and different parameters. This allows you to find using different semantics, as illustrated by the examples above.

A finder method must return either the primary key for the entity bean it finds or a collection of primary keys if it finds more than one. Because you could find more than one data instance in the database, finder methods can return collections of primary keys.

As with *ejbCreate()*, clients do not invoke your finder methods on the bean instance itself. A finder method is just like any other method on your entity bean class—clients never directly call any of your bean’s methods. Rather, clients invoke finder methods on home objects, implemented by

the EJB container, that delegate to your bean. Therefore, for each finder method you define in your bean class, you should define a corresponding finder in the local home interface. Clients call your local home object's finder methods, which delegate to your bean's finders.

For example, given the following finder method in the local home interface:

```
public AccountLocal findBigAccounts(int minimum) throws FinderException;
```

here is the finder implementation in your bean class (notice the `ejb` prefix):

```
public AccountPK ejbFindBigAccounts(int minimum)
    throws FinderException { . . . }
```

As with `ejbCreate()`, the home signature and the bean class signature have a couple of differences:

- The entity bean instance returns a primary key to the container, whereas the home object returns an EJB object to the client.
- The bean class signature is the same as the home signature, except for an extra, mandatory `ejb` prefix and that the first letter in the word *Find* is capitalized.

These signature differences between the home and bean are valid because the bean does not implement the local home interface. Rather, the local home object *delegates* to the bean, so strict signature matching is not needed.

Another interesting aspect of finders is that they can return collections. Your database search may turn up more than one result and therefore more than one entity bean. Here is the local home interface signature:

```
public Collection findAllProducts() throws FinderException;
```

And here is the bean implementation signature:

```
public Collection ejbFindAllProducts()
    throws FinderException { . . . }
```

The finder process works as follows:

- When the client invokes the home object's finder, the home object asks a bean to find all primary keys matching the client's criteria. The bean then returns a *collection* of those primary keys to the container.
- When the container receives the collection of keys from the entity bean instance, *it creates a collection of EJB objects*, one for each primary key, and returns those EJB objects in its own collection to the client. The client can then invoke methods on the EJB objects: Each EJB object represents its own instance of data within the entity bean's database storage.

Bean-Managed Persistence Example: A Bank Account

Our first example is a simple bank account entity bean. This bank account bean can be used to represent and manipulate real bank account data in an underlying relational database. The object model for our bank account is detailed in Figure 6.2.

Notice that we're developing both local and remote interfaces. When this bean is used in production, the local interfaces will be used, because this entity bean will be accessed by other beans that run in-process. However, for testing purposes, and to help you understand entity beans easily, we don't want to introduce other beans. Rather, we will connect to this bean from a standalone application. Since a standalone application is remote, we thus need to use its remote interface. This is a common issue with EJB programming—to test beans on an individual basis in this manner, you need to code its remote interface even though you only plan to use the local interface in production. The good news is that the code is almost identical for the local interface—see the book's accompanying source code (the e-commerce example) for examples of

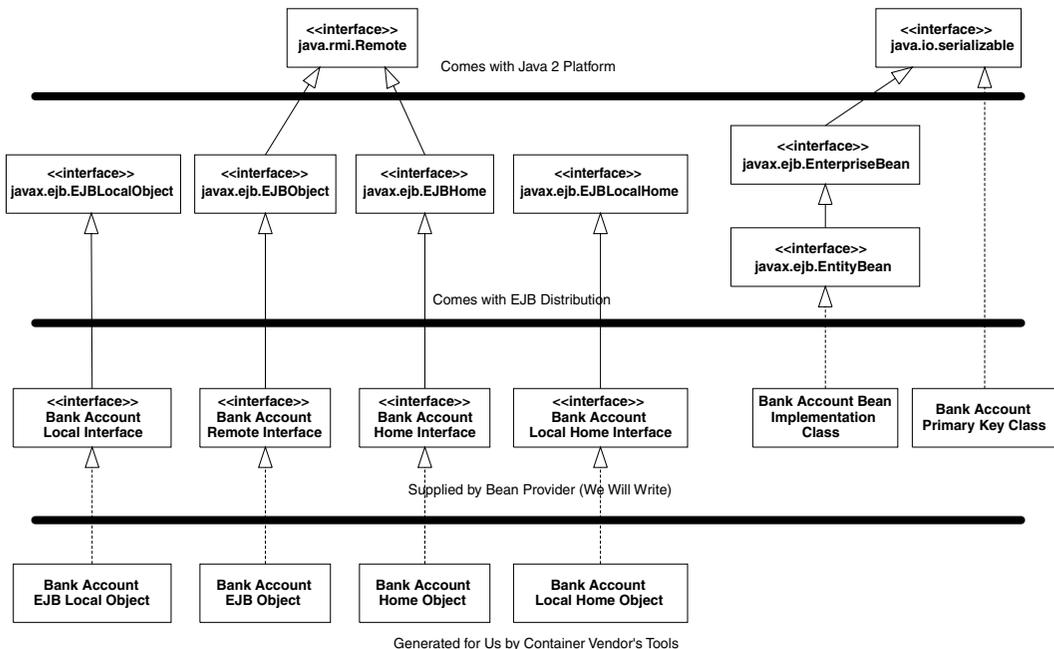


Figure 6.2 The bank account object model.

calling entity beans through their local interfaces. Now let's take a look at each of the files that we must create for our entity bean component.

Account.java

Account.java is our entity bean's remote interface—what remote clients use to call our bean's methods. It is shown in Source 6.3.

```
package examples;

import javax.ejb.*;
import java.rmi.RemoteException;

/**
 * This is the remote interface for AccountBean.
 *
 * This interface is what clients operate on when they interact with
 * beans. The container will implement this interface; the
 * implemented object is called the EJB object, which delegates
 * invocations to the actual bean.
 */
public interface Account extends EJBObject {

    /**
     * Deposits amt into account.
     */
    public void deposit(double amt) throws AccountException,
        RemoteException;

    /**
     * Withdraws amt from bank account.
     * @throws AccountException thrown in amt < available balance
     */
    public void withdraw(double amt) throws AccountException,
        RemoteException;

    // Getter/setter methods on Entity Bean fields

    public double getBalance() throws RemoteException;

    public String getOwnerName() throws RemoteException;
    public void setOwnerName(String name) throws RemoteException;

    public String getAccountID() throws RemoteException;
    public void setAccountID(String id) throws RemoteException;
}
```

Source 6.3 Account.java.

Notice that the account remote interface extends *javax.ejb.EJBObject*, which all remote interfaces must do. Our interface exposes a number of methods for manipulating entity beans, such as for making deposits and withdrawals. All of our methods throw remote exceptions to facilitate system-level catastrophic failures. Notice that in our withdrawal method, we also throw our own custom application-level exception, *AccountException*. We'll define that exception later.

AccountLocal.java

AccountLocal.java is our entity bean's local interface—what local clients use to call our bean's methods. It is shown in Source 6.4.

AccountHome.java

Our home interface is specified by *AccountHome.java*, shown in Source 6.5.

We provide one create method to create a new account. This will create new database data representing a bank account. It returns an EJB object to the client so the client can manipulate that newly created account. Notice that we throw the application-level *javax.ejb.CreateException*, which all *create()* methods must throw.

We also have two finder methods. *findByPrimaryKey()* searches the database for a bank account that already exists; it searches by the account ID, which we will define in *AccountPK.java*. We also have a custom finder method, *findByOwnerName()*, which searches the database for all bank accounts that have the same owner's name. Because we're using bean-managed persistence, we need to implement both of these finder methods in our entity bean implementation. (If we were using container-managed persistence, the container would search the database for us). As with our *create* method, both finders return EJB objects so the client can manipulate the newly found bank accounts. We throw the application-level *javax.ejb.FinderException*, which all finders must throw.

Finally, we have a business method, *getTotalBankValue()*. This business method is an operation applied to the entire table rather than to an individual row. Thus it is a global method that is independent of any particular entity bean instance. This business method will be implemented in the bean class as an *ejb-Home()* method, as previously described in Table 6.1.

AccountLocalHome.java

Our local home interface, the higher performing home interface used by local clients, is specified by *AccountLocalHome.java*, shown in Source 6.6.

```
package examples;

import javax.ejb.*;

/**
 * This is the local interface for AccountBean.
 *
 * This interface is what clients operate on when they interact with
 * beans. The container will implement this interface; the
 * implemented object is called the local object, which delegates
 * invocations to the actual bean.
 */
public interface AccountLocal extends EJBLocalObject {

    /**
     * Deposits amt into account.
     */
    public void deposit(double amt) throws AccountException;

    /**
     * Withdraws amt from bank account.
     * @throws AccountException thrown in amt < available balance
     */
    public void withdraw(double amt) throws AccountException;

    // Getter/setter methods on Entity Bean fields

    public double getBalance();

    public String getOwnerName();
    public void setOwnerName(String name);

    public String getAccountID();
    public void setAccountID(String id);
}
```

Source 6.4 AccountLocal.java.

The only differences between the local home interface and the home interface are that the local home interface does not throw remote exceptions, and the local home interface extends a different parent interface.

AccountPK.java

Our entity bean's primary key class is defined by *AccountPK.java*, detailed in Source 6.7.

```
package examples;

import javax.ejb.*;
import java.util.Collection;
import java.rmi.RemoteException;

/**
 * This is the home interface for Account. This
 * interface is implemented by the EJB container's tools - the
 * implemented object is called the home object, which
 * is a factory for EJB objects.
 */
public interface AccountHome extends EJBHome {

    /**
     * We define a single create() method in this home interface,
     * which corresponds to the ejbCreate() method in AccountBean.
     * This method creates the local EJB object.
     *
     * Notice that the local home interface returns a local interface,
     * whereas the bean returns a PK.
     *
     * @param accountID The number of the account (unique)
     * @param ownerName The name of the person who owns the account
     * @return The newly created local object.
     */
    Account create(String accountID, String ownerName) throws
    CreateException, RemoteException;

    /**
     * Finds a Account by its primary Key (Account ID)
     */
    public Account findByPrimaryKey(AccountPK key) throws
    FinderException, RemoteException;

    /**
     * Finds all Accounts under an owner name
     */
    public Collection findByOwnerName(String name) throws
    FinderException, RemoteException;

    /**
     * This home business method is independent of any particular
     * account. It returns the total of all accounts in the bank.
     */
    public double getTotalBankValue() throws AccountException,
    RemoteException;
}
```

Source 6.5 AccountHome.java.

```
package examples;

import javax.ejb.*;
import java.util.Collection;

/**
 * This is the local home interface for Account. This
 * interface is implemented by the EJB container's tools - the
 * implemented object is called the local home object, which
 * is a factory for local EJB objects.
 */
public interface AccountLocalHome extends EJBLocalHome {

    /**
     * We define a single create() method in this home interface,
     * which corresponds to the ejbCreate() method in AccountBean.
     * This method creates the local EJB object.
     *
     * Notice that the local home interface returns a
     * local interface, whereas the bean returns a PK.
     *
     * Notice we don't throw RemoteExceptions because we are
     * local not remote.
     *
     * @param accountID The number of the account (unique)
     * @param ownerName The name of the person who owns the account
     * @return The newly created local object.
     */
    public AccountLocal create(String accountID, String ownerName)
        throws CreateException;

    /**
     * Finds an Account by its primary Key (Account ID)
     */
    public AccountLocal findByPrimaryKey(AccountPK key) throws
        FinderException;

    /**
     * Finds all Accounts under an owner's name
     */
    public Collection findByOwnerName(String name) throws
        FinderException;

    /**
     * This home business method is independent of any particular
     * account instance. It returns the total of all the bank
     * accounts in the bank.
     */
    public double getTotalBankValue() throws AccountException;
}
```

Source 6.6 AccountLocalHome.java.

```
package examples;

import java.io.Serializable;

/**
 * Primary Key class for Account.
 */
public class AccountPK implements java.io.Serializable {
    public String accountID;

    public AccountPK(String id) {
        this.accountID = id;
    }

    public AccountPK() {
    }

    public String toString() {
        return accountID;
    }

    public int hashCode() {
        return accountID.hashCode();
    }

    public boolean equals(Object account) {
        return ((AccountPK) account).accountID.equals(accountID);
    }
}
```

Source 6.7 AccountPK.java.

Notice the following about Source 6.7:

- Our primary key contains a simple `String`—the account ID string. For example, an account ID string could be “ABC-123-0000.” This string must be unique to its bank account; we rely on the client code that constructs our account ID to make sure it is unique. The primary key is used to identify each bank account uniquely. More advanced entity beans that map to more than one table may have primary key classes that have several fields inside of them, each representing the primary key of a table in the database.
- There is a required `toString()` method. This container calls this method to retrieve a `String` value of this primary key. For simple primary keys, we

just return the stored field. For more advanced primary keys, we need to somehow combine the various fields in the primary key class to form a String.

- There is a required *hashCode()* method. By supplying this method, our primary key class can be stored in a Hashtable. The container needs this because inside of the container it may use a Hashtable or similar structure to store a list of all entity beans it has in memory, keyed on their primary keys.
- There is a required *equals()* method. The container calls this to compare this primary key to others when determining internally if two cached entity beans (which each have a primary key) are representing the same database data.

AccountBean.java

Next we have our entity bean implementation class, *AccountBean.java*. Our bean implementation code is quite lengthy and is divided into several sections.

Bean-managed state fields. These are the persistable fields of our entity bean class. Our bean instance will load and store the database data into these fields.

Business logic methods. These methods perform services for clients, such as withdrawing or depositing into an account. They are exposed by the remote interface, *Account*.

EJB-required methods. These are EJB-required methods that the container calls to manage our bean. They also include our creator and finder methods defined in the home interface.

The code is presented in Source 6.8 through Source 6.10. We divide the code into three parts because the code is extremely cumbersome, even for a simple bank account. This is an unfortunate drawback of bean-managed persistence because you must provide all data access code.

The first part of our bean is straightforward. We have our bean's fields (one of which is the primary key field), and a default constructor. We keep an *EntityContext* around so that we can query the container from our bean as necessary (However, the *EntityContext* is not a persistent field).

The next part of our bean is the business logic methods, shown in Source 6.9.

Our withdraw and deposit methods simply modify the in-memory fields of the entity bean instance. If the client tries to withdraw from a negative account, we throw our custom application-level exception, *AccountException*.

```
package examples;

import java.sql.*;
import javax.naming.*;
import javax.ejb.*;
import java.util.*;

/**
 * Demonstration Bean-Managed Persistent Entity Bean.
 * This Entity Bean represents a Bank Account.
 */
public class AccountBean implements EntityBean {

    protected EntityContext ctx;

    //
    // Bean-managed state fields
    //

    private String accountID;    // PK
    private String ownerName;
    private double balance;

    public AccountBean() {
        System.out.println("New Bank Account Entity Bean Java Object
        created by EJB Container.");
    }

    ... methods continue ...
}
```

Source 6.8 AccountBean.java (Part 1 of 3).

```
... continued ...

//
// Business Logic Methods
//

/**
 * Deposits amt into account.
 */
public void deposit(double amt) throws AccountException {
    System.out.println("deposit(" + amt + ") called.");
}
```

Source 6.9 AccountBean.java (Part 2 of 3).

```
        balance += amt;
    }

    /**
     * Withdraws amt from bank account.
     * @throws AccountException thrown in amt < available balance
     */
    public void withdraw(double amt) throws AccountException {
        System.out.println("withdraw(" + amt + ") called.");

        if (amt > balance) {
            throw new AccountException("Your balance is " +
                balance + "! You cannot withdraw "
                + amt + "!");
        }

        balance -= amt;
    }

    // Getter/setter methods on Entity Bean fields

    public double getBalance() {
        System.out.println("getBalance() called.");
        return balance;
    }

    public void setOwnerName(String name) {
        System.out.println("setOwnerName() called.");
        ownerName = name;
    }

    public String getOwnerName() {
        System.out.println("getOwnerName() called.");
        return ownerName;
    }

    public String getAccountID() {
        System.out.println("getAccountID() called.");
        return accountID;
    }

    public void setAccountID(String id) {
        System.out.println("setAccountID() called.");
        this.accountID = id;
    }

    /**
```

Source 6.9 AccountBean.java (Part 2 of 3) (*continued*).

```
* This home business method is independent of any
* particular account instance. It returns the total
* of all the bank accounts in the bank.
*/
public double.ejbHomeGetTotalBankValue() throws AccountException
{
    PreparedStatement pstmt = null;
    Connection conn = null;

    try {

System.out.println("ejbHomeGetTotalBankValue()");

        /*
         * Acquire DB connection
         */
        conn = getConnection();

        /*
         * Get the total of all accounts
         */
        pstmt = conn.prepareStatement(
            "select sum(balance) as total from accounts");
        ResultSet rs = pstmt.executeQuery();

        /*
         * Return the sum
         */
        if (rs.next()) {
            return rs.getDouble("total");
        }
    }
    catch (Exception e) {
        e.printStackTrace();
        throw new AccountException(e);
    }
    finally {
        /*
         * Release DB Connection for other beans
         */
        try { if (pstmt != null) pstmt.close(); }
        catch (Exception e) {}
        try { if (conn != null) conn.close(); }
        catch (Exception e) {}
    }

    throw new AccountException("Error!");
}
```

Source 6.9 AccountBean.java (Part 2 of 3) (continued).

```
    }

    /**
     * Gets JDBC connection from the connection pool.
     *
     * @return The JDBC connection
     */
    public Connection getConnection() throws Exception {
        try {
            Context ctx = new InitialContext();
            javax.sql.DataSource ds = (javax.sql.DataSource)
                ctx.lookup("java:comp/env/jdbc/ejbPool");
            return ds.getConnection();
        }
        catch (Exception e) {
            System.err.println("Couldn't get datasource!");
            e.printStackTrace();
            throw e;
        }
    }
}
```

Source 6.9 AccountBean.java (Part 2 of 3) (*continued*).

The *ejbHome()* business method implementation adds the total of all bank account balances in the database. It retrieves a JDBC connection via the *getConnection()* helper method. In that *getConnection()* method we lookup the database connection via JNDI (see Chapter 9 for a full description of this process).

Notice, too, that we close each connection after every method call. This allows our EJB container to pool JDBC connections. When the connection is not in use, another bean can use our connection. This is the standard, portable way for *connection pooling*. The connection pooling is built-into the JDBC 2.0 specification and happens automatically behind the scenes.

The final part of our bean is the various EJB callback methods, shown in Source 6.10.

Source 6.10 is quite long because of the enormous amount of JDBC coding required to write even a simple bean-managed persistent entity bean. The bulk of the code occurs in the methods that perform CRUD operations (Create, Read, Update, Delete). These are namely *ejbCreate()*, *ejbFind()* and *ejbLoad()*, *ejbStore()*, and *ejbRemove()*. The code is self-documenting and you should be able to understand it if you cross-reference Table 6.1. If you're still stuck, we will further explain these methods later in this chapter when we discuss the lifecycle of a bean-managed persistent entity bean.

```
... continued ...

//
// EJB-required methods
//

/**
 * Called by Container. Implementation can acquire
 * needed resources.
 */
public void ejbActivate() {
    System.out.println("ejbActivate() called.");
}

/**
 * Removes entity bean data from the database.
 * Corresponds to when client calls home.remove().
 */
public void ejbRemove() throws RemoveException {
    System.out.println("ejbRemove() called.");

    /*
     * Remember that an entity bean class can be used to
     * represent different data instances. So how does
     * this method know which instance in the database
     * to delete?
     *
     * The answer is to query the container by calling
     * the entity context object. By retrieving the
     * primary key from the entity context, we know
     * which data instance, keyed by the PK, that we
     * should delete from the DB.
     */
    AccountPK pk = (AccountPK) ctx.getPrimaryKey();
    String id = pk.accountID;

    PreparedStatement pstmt = null;
    Connection conn = null;
    try {
        /*
         * 1) Acquire a new JDBC Connection
         */
        conn = getConnection();

        /*
         * 2) Remove account from the DB
         */
    }
}
```

Source 6.10 AccountBean.java (Part 3 of 3).

```
        pstmt = conn.prepareStatement(
            "delete from accounts where id = ?");
        pstmt.setString(1, id);

        /*
         * 3) Throw a system-level exception if something
         * bad happened.
         */
        if (pstmt.executeUpdate() == 0) {
            throw new RemoveException(
                "Account " + pk +
                " failed to be removed from the database");
        }
    }
    catch (Exception ex) {
        throw new EJBException(ex.toString());
    }
    finally {
        /*
         * 4) Release the DB Connection
         */
        try { if (pstmt != null) pstmt.close(); }
        catch (Exception e) {}
        try { if (conn != null) conn.close(); }
        catch (Exception e) {}
    }
}

/**
 * Called by Container. Releases held resources for
 * passivation.
 */
public void ejbPassivate() {
    System.out.println("ejbPassivate () called.");
}

/**
 * Called by the container. Updates the in-memory entity
 * bean object to reflect the current value stored in
 * the database.
 */
public void ejbLoad() {
    System.out.println("ejbLoad() called.");

    /*
     * Again, query the Entity Context to get the current
     * Primary Key, so we know which instance to load.
     */
}
```

Source 6.10 AccountBean.java (Part 3 of 3) (*continued*).

```
    */
    AccountPK pk = (AccountPK) ctx.getPrimaryKey();
    String id = pk.accountID;

    PreparedStatement pstmt = null;
    Connection conn = null;
    try {
        /*
         * 1) Acquire a new DB Connection
         */
        conn = getConnection();

        /*
         * 2) Get account from the DB, querying
         *    by account ID
         */
        pstmt = conn.prepareStatement(
            "select ownerName, balance from accounts "
            + "where id = ?");
        pstmt.setString(1, id);
        ResultSet rs = pstmt.executeQuery();
        rs.next();
        ownerName = rs.getString("ownerName");
        balance = rs.getDouble("balance");
    }
    catch (Exception ex) {
        throw new EJBException(
            "Account " + pk
            + " failed to load from database", ex);
    }
    finally {
        /*
         * 3) Release the DB Connection
         */
        try { if (pstmt != null) pstmt.close(); }
        catch (Exception e) {}
        try { if (conn != null) conn.close(); }
        catch (Exception e) {}
    }
}

/**
 * Called from the Container. Updates the database
 * to reflect the current values of this in-memory
 * entity bean instance.
 */
```

Source 6.10 AccountBean.java (Part 3 of 3) (continued).

```
public void ejbStore() {
    System.out.println("ejbStore() called.");

    PreparedStatement pstmt = null;
    Connection conn = null;
    try {
        /*
         * 1) Acquire a new DB Connection
         */
        conn = getConnection();

        /*
         * 2) Store account in DB
         */
        pstmt = conn.prepareStatement(
            "update accounts set ownerName = ?, balance = ?"
            + " where id = ?");
        pstmt.setString(1, ownerName);
        pstmt.setDouble(2, balance);
        pstmt.setString(3, accountID);
        pstmt.executeUpdate();

    }
    catch (Exception ex) {
        throw new EJBException(
            "Account " + accountID
            + " failed to save to database", ex);
    }
    finally {
        /*
         * 3) Release the DB Connection
         */
        try { if (pstmt != null) pstmt.close(); }
        catch (Exception e) {}
        try { if (conn != null) conn.close(); }
        catch (Exception e) {}
    }
}

/**
 * Called by the container. Associates this bean
 * instance with a particular context. We can query
 * the bean properties that customize the bean here.
 */
public void setEntityContext(EntityContext ctx) {
    System.out.println("setEntityContext called");
    this.ctx = ctx;
}
```

Source 6.10 AccountBean.java (Part 3 of 3) (*continued*).

```
    }

    /**
     * Called by Container. Disassociates this bean
     * instance with a particular context environment.
     */
    public void unsetEntityContext() {
        System.out.println("unsetEntityContext called");
        this.ctx = null;
    }

    /**
     * Called after ejbCreate(). Now, the Bean can retrieve
     * its EJBObject from its context, and pass it as
     * a 'this' argument.
     */
    public void ejbPostCreate(String accountID, String ownerName) {
    }

    /**
     * This is the initialization method that corresponds to the
     * create() method in the Home Interface.
     *
     * When the client calls the Home Object's create() method,
     * the Home Object then calls this ejbCreate() method.
     *
     * @return The primary key for this account
     */
    public AccountPK ejbCreate(String accountID, String ownerName)
        throws CreateException {

        PreparedStatement pstmt = null;
        Connection conn = null;
        try {
            System.out.println("ejbCreate() called.");
            this.accountID = accountID;
            this.ownerName = ownerName;
            this.balance = 0;

            /*
             * Acquire DB connection
             */
            conn = getConnection();

            /*
             * Insert the account into the database
             */

```

Source 6.10 AccountBean.java (Part 3 of 3) (continued).

```
        pstmt = conn.prepareStatement(
            "insert into accounts (id, ownerName, balance)"
            + " values (?, ?, ?)");
        pstmt.setString(1, accountID);
        pstmt.setString(2, ownerName);
        pstmt.setDouble(3, balance);
        pstmt.executeUpdate();

        /*
         * Generate the Primary Key and return it
         */
        return new AccountPK(accountID);
    }
    catch (Exception e) {
        throw new CreateException(e.toString());
    }
    finally {
        /*
         * Release DB Connection for other beans
         */
        try { if (pstmt != null) pstmt.close(); }
        catch (Exception e) {}
        try { if (conn != null) conn.close(); }
        catch (Exception e) {}
    }
}

/**
 * Finds a Account by its primary Key
 */
public AccountPK ejbFindByPrimaryKey(AccountPK key)
    throws FinderException {
    PreparedStatement pstmt = null;
    Connection conn = null;
    try {
        System.out.println("ejbFindByPrimaryKey("
            + key + ") called");

        /*
         * Acquire DB connection
         */
        conn = getConnection();

        /*
         * Find the Entity in the DB
         */
        pstmt = conn.prepareStatement(
```

Source 6.10 AccountBean.java (Part 3 of 3) (*continued*).

```
        "select id from accounts where id = ?");
        pstmt.setString(1, key.toString());
        ResultSet rs = pstmt.executeQuery();
        rs.next();

        /*
         * No errors occurred, so return the Primary Key
         */
        return key;
    }
    catch (Exception e) {
        throw new FinderException(e.toString());
    }
    finally {
        /*
         * Release DB Connection for other beans
         */
        try { if (pstmt != null) pstmt.close(); }
        catch (Exception e) {}
        try { if (conn != null) conn.close(); }
        catch (Exception e) {}
    }
}

/**
 * Finds Accounts by name
 */
public Collection ejbFindByOwnerName(String name)
    throws FinderException {
    PreparedStatement pstmt = null;
    Connection conn = null;
    Vector v = new Vector();

    try {
        System.out.println(
            "ejbFindByOwnerName(" + name + ") called");

        /*
         * Acquire DB connection
         */
        conn = getConnection();

        /*
         * Find the primary keys in the DB
         */
        pstmt = conn.prepareStatement(
            "select id from accounts where ownerName = ?");
```

Source 6.10 AccountBean.java (Part 3 of 3) (continued).

```
        pstmt.setString(1, name);
        ResultSet rs = pstmt.executeQuery();

        /*
         * Insert every primary key found into a vector
         */
        while (rs.next()) {
            String id = rs.getString("id");
            v.addElement(new AccountPK(id));
        }

        /*
         * Return the vector of primary keys
         */
        return v;
    }
    catch (Exception e) {
        throw new FinderException(e.toString());
    }
    finally {
        /*
         * Release DB Connection for other beans
         */
        try { if (pstmt != null) pstmt.close(); }
        catch (Exception e) {}
        try { if (conn != null) conn.close(); }
        catch (Exception e) {}
    }
}
}
```

Source 6.10 AccountBean.java (Part 3 of 3) (*continued*).



When a statement is sent to a database, the container's installer JDBC driver parses it, determines the best way to execute the statement based on statistics that it maintains, and then executes the statement. Parsing and determining an execution strategy can be computationally expensive. The good news is that JDBC is smart—when an instance of `PreparedStatement` is executed on a connection, it first checks its cache to see if this statement has been executed previously; if so, it reuses the previously prepared version, thus improving performance. For more information, refer to Billy Newport's article, "How Prepared Statements Greatly Improve Performance," posted at www.ejbinfo.com.

AccountException.java

Our custom exception class is *AccountException.java*, displayed in Source 6.11. It simply delegates to the parent *java.lang.Exception* class. It is still useful to define our own custom exception class, however, so that we can distinguish between a problem with our bank account component and a problem with another part of a deployed system.

Client.java

Our last Java file is a simple test client to exercise our bean's methods. It is shown in Source 6.12.

The client code is fairly self-explanatory. We perform some bank account operations in the try block. We have a finally clause to make sure our bank account is properly deleted afterward, regardless of any exceptions that may have been thrown.

The Deployment Descriptor

Now, let's take a look at our deployment descriptor, shown in Source 6.13.

```
package examples;

/**
 * Exceptions thrown by Accounts
 */
public class AccountException extends Exception {

    public AccountException() {
        super();
    }

    public AccountException(Exception e) {
        super(e.toString());
    }

    public AccountException(String s) {
        super(s);
    }
}
```

Source 6.11 AccountException.java.

```
package examples;

import javax.ejb.*;
import javax.naming.*;
import java.rmi.*;
import javax.rmi.*;
import java.util.*;

/**
 * Sample client code that manipulates a Bank Account Entity Bean.
 */
public class AccountClient {

    public static void main(String[] args) throws Exception {

        Account account = null;

        try {
            /*
             * Get a reference to the Account Home Object - the
             * factory for Account EJB Objects
             */
            Context ctx =
                new InitialContext(System.getProperties());

            Object obj = ctx.lookup("AccountHome");
            AccountHome home = (AccountHome)
                PortableRemoteObject.narrow(
                    obj, AccountHome.class);

            System.err.println(
                "Total of all accounts in bank initially = "
                + home.getTotalBankValue());

            /*
             * Use the factory to create the Account EJB Object
             */
            home.create("123-456-7890", "John Smith");

            /*
             * Find an account
             */
            Iterator i = home.findByName(
                "John Smith").iterator();
            if (i.hasNext()) {
                account = (Account)
                    javax.rmi.PortableRemoteObject.narrow(
```

Source 6.12 Client.java.

```
        i.next(), Account.class);
    }
    else {
        throw new Exception("Could not find account");
    }

    /*
     * Call the balance() method, and print it
     */
    System.out.println(
        "Initial Balance = " + account.getBalance());

    /*
     * Deposit $100 into the account
     */
    account.deposit(100);

    /*
     * Retrieve the resulting balance.
     */
    System.out.println(
        "After depositing 100, account balance = "
        + account.getBalance());

    System.out.println(
        "Total of all accounts in bank now = "
        + home.getTotalBankValue());

    /*
     * Retrieve the Primary Key from the EJB Object
     */
    AccountPK pk = (AccountPK) account.getPrimaryKey();

    /*
     * Release our old EJB Object reference. Now call
     * find() again, this time querying on Account ID
     * (i.e. the Primary Key).
     */
    account = null;
    account = home.findByPrimaryKey(pk);

    /*
     * Print out current balance
     */
    System.out.println(
        "Found account with ID " + pk + ". Balance = "
        + account.getBalance());
```

Source 6.12 Client.java (continued).

```
        /*
         * Try to withdraw $150
         */
        System.out.println(
            "Now trying to withdraw $150, which is more "
            + "than is currently available. This should "
            + "generate an exception..");
        account.withdraw(150);

    }
    catch (Exception e) {
        System.out.println("Caught exception!");
        e.printStackTrace();
    }
    finally {
        /*
         * Destroy the Entity permanently
         */
        try {
            System.out.println("Destroying account..");
            if (account != null) {
                account.remove();
            }
        }
        catch (Exception e) {
            e.printStackTrace();
        }
    }
}
}
```

Source 6.12 Client.java (continued).

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

<!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>
  <enterprise-beans>
    <entity>
      <ejb-name>Account</ejb-name>
      <home>examples.AccountHome</home>
```

Source 6.13 The Account Bean's ejb-jar.xml deployment descriptor.

```

<remote>examples.Account</remote>
<local-home>examples.AccountLocalHome</local-home>
<local>examples.AccountLocal</local>
<ejb-class>examples.AccountBean</ejb-class>
<persistence-type>Bean</persistence-type>
<prim-key-class>examples.AccountPK</prim-key-class>
<reentrant>False</reentrant>

<resource-ref>
  <res-ref-name>jdbc/ejbPool</res-ref-name>
  <res-type>javax.sql.DataSource</res-type>
  <res-auth>Container</res-auth>
</resource-ref>
</entity>
</enterprise-beans>

<assembly-descriptor>
  <container-transaction>
    <method>
      <ejb-name>Account</ejb-name>
      <method-intf>Local</method-intf>
      <method-name>*</method-name>
    </method>
    <method>
      <ejb-name>Account</ejb-name>
      <method-intf>Remote</method-intf>
      <method-name>*</method-name>
    </method>
    <trans-attribute>Required</trans-attribute>
  </container-transaction>
</assembly-descriptor>
</ejb-jar>

```

Source 6.13 The Account Bean's `ejb-jar.xml` deployment descriptor (*continued*).

Notice the following features of our deployment descriptor that are different from session beans:

- The *persistence-type* element indicates whether we are bean-managed persistent (set it to "Bean") or container-managed persistent (set it to "Container").
- The *prim-key-class* element specifies our primary key class.
- The *reentrant* element dictates whether our bean can call itself through another bean. A given bean A is reentrant if bean A calls bean B, which calls back on bean A. This is a special case of multithreading because it is really only one path of execution that happens to loop back on itself. If we

would like to support this reentrant behavior, we should set this setting to True so that the container will allow two threads to run inside of bean A at once. Since our bean doesn't call itself through another bean, we set it to False, which is usually what you'll want to do to avoid unintended multi-threading issues.

- The *resource-ref* element sets up our JDBC driver and makes it available at the proper JNDI location (see Chapter 9 for a full description of this process).
- The *assembly-descriptor* associates transactions with our bean. We will describe transactions fully in Chapter 10.

The Container-Specific Deployment Descriptor

Finally, we have our container-specific deployment descriptor, which configures our bean in ways specific to a particular EJB server. We will not show this file because we wish the code in this book to remain vendor-neutral. Typically you would use this proprietary descriptor to associate the home interface, local home interface, and JDBC driver with JNDI locations. For an example descriptor, see the book's accompanying source code.

Setting up the Database

Lastly, you need to create the appropriate database table and columns for our bank accounts. You can do this through your database's GUI or command-line interface. The book's included source code comes with a preconfigured sample database that you can use right away. If you're using a different database, you should enter the following *SQL Data Definition Language* (DDL) statements in your database's SQL interface:

```
drop table accounts;  
create table accounts (id varchar(64), ownername varchar(64), balance  
numeric(18));
```

This creates an empty table of bank accounts. The first column is the bank account ID (the primary key), the second column is the bank account owner's name, and the third column is the bank account balance.

Running the Client Program

To run the client program, type a command similar to the following (depending on what your EJB container's Java Naming and Directory Interface (JNDI) connection parameters are—see your container's documentation):

```
java -Djava.naming.factory.initial=
    weblogic.jndi.WLInitialContextFactory
-Djava.naming.provider.url=
    t3://localhost:7001
examples.AccountClient
```

The initialization parameters are required by JNDI to find the home object, as we learned in Chapter 3.

Server-Side Output

When you run the client, you should see something *similar* to the following on the server side. Note that your particular output may vary, due to variances in EJB container behavior.

```
New Bank Account Entity Bean Java Object created by EJB Container.
setEntityContext called.
ejbHomeGetTotalBankValue() called.
ejbCreate() called.
ejbStore() called.
New Bank Account Entity Bean Java Object created by EJB Container.
setEntityContext called.
ejbFindByOwnerName(John Smith) called.
ejbLoad() called.
getBalance() called.
ejbStore() called.
ejbLoad() called.
deposit(100.0) called.
ejbStore() called.
ejbLoad() called.
getBalance() called.
ejbStore() called.
ejbHomeGetTotalBankValue() called.
ejbFindByPrimaryKey(123-456-7890) called.
ejbLoad() called.
getBalance() called.
ejbStore() called.
ejbLoad() called.
withdraw(150.0) called.
ejbStore() called.
ejbLoad() called.
ejbRemove() called.
```

Notice what's happening here:

- When our client code called *create()* on the home object, the container created an entity bean instance. The container first called *newInstance()* and *setEntityContext()* to get the entity bean into the available pool of entity beans. The container then serviced our client's home business method and used the bean in the pool. Then the client called *create()*, which caused the

container to take the bean out of the pool and call the bean's `ejbCreate()` method, which created some new database data, and returned control back to the container. Finally, the container associated the bean instance with a new EJB object and returned that EJB object to the client.

- To service our *finder* method, the container instantiated another entity bean. The container called `newInstance()` and then `setEntityContext()` to get that new bean instance into the available pool of entity beans. It then used the bean in the pool to service our finder method. Note that the bean instance is still in the pool and could service any number of finder methods.
- In addition to the methods that the client calls, our EJB container interleaved a few `ejbStore()` and `ejbLoad()` calls to keep the database in synch.

Client-Side Output

Running the client program yields the following client-side output:

```
Total of all accounts in bank initially = 1200000.0
Initial Balance = 0.0
After depositing 100, account balance = 100.0
Total of all accounts in bank now = 1200100.0
Found account with ID 123-456-7890. Balance = 100.0
Now trying to withdraw $150, which is more than is currently available.
This should generate an exception..
Caught exception!
examples.AccountException: Your balance is 100.0! You cannot withdraw
150.0!
Destroying account..
```

Our table already had \$1,200,000 from previous records in the database. We then created an entity bean, deposited into it, and tried to withdraw more than we had. The entity bean correctly threw an application-level exception back to us indicating that our balance had insufficient funds.

Putting It All Together: Walking through a BMP Entity Bean's Life Cycle

Let's wrap up this chapter by examining the big picture and understanding exactly how a container interacts with a BMP entity bean. The state machine diagram in Figure 6.3 illustrates the life cycle of a BMP entity bean.

Here is what's going on in this diagram.

1. The *does not exist* state represents entity bean instances that have not been instantiated yet.

The lifecycle of a bean-managed persistent entity bean. Each method call shown is an invocation from the container to the bean instance.

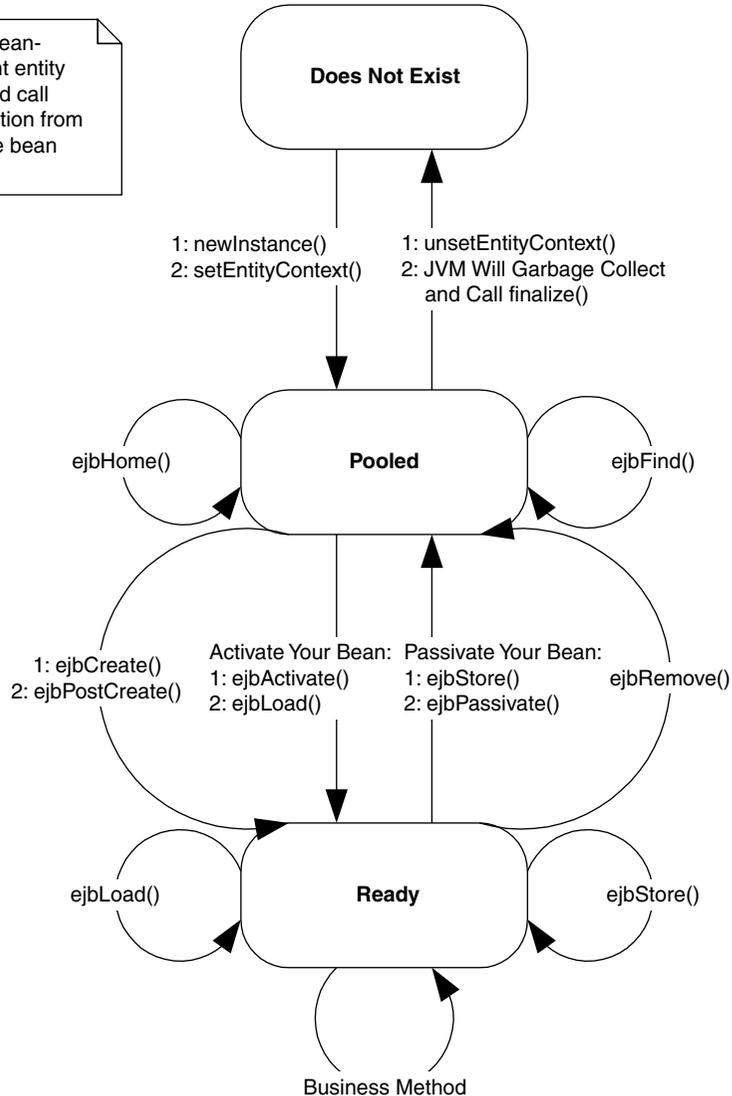


Figure 6.3 The BMP entity bean life cycle.

- To create a new instance, the container calls the `newInstance()` method on the entity bean class. This calls your entity bean's default constructor, bringing a new instance into memory. Next, the container associates your entity bean with an entity context object via a callback that you implement, called `setEntityContext(EntityContext ctx)`. Note that this step occurs only when the container wants to increase the available pool of entity bean instances, not necessarily when a client connects.

3. Next, your entity bean is in a pool of other entity beans. At this point your entity bean does not have any entity bean database data loaded into it, and it does not hold any bean-specific resources, such as socket connections. Your bean instance can be used in this mode to find entity data in the database, by servicing a *finder* method on behalf of a client. Your bean instance can also perform operations not dependent on a particular data instance by servicing an *ejbHome()* method on behalf of a client. If the container wants to reduce its pool size, it can destroy your bean. The container signals your bean instance that it is about to be destroyed by calling the *unsetEntityContext()* method on your bean. Once this is done, the container releases any references to your bean, and eventually, the Java garbage collector cleans up the memory your instance had been using. Therefore your *unsetEntityContext()* method should prepare your bean to be cleaned up, perhaps by releasing any resources your bean had claimed during *setEntityContext()*.
4. When the client wants to create some new database data (say, a new order for goods placed over the Internet), it calls a *create()* method on your entity bean's home object. The container then grabs an entity bean instance from the pool, and the instance's *ejbCreate()* method is called. *ejbCreate()* initializes the entity bean to a specific data set. For example, if a client calls a *create()* method to create a bank account, it might pass the bank account holder's name and the initial balance as parameters. Your entity bean's *ejbCreate()* method would populate its member variables with these parameters. It would also create the corresponding database representation (if you're using bean-managed persistence). Now your bean is in the "ready" state.
5. While your bean is in the ready state, it is tied to specific data and hence a specific EJB object. If there are other entity bean instances that are views into the same database data, the container may occasionally need to synchronize your bean instance with the underlying database, so that you will always be working with the most recent data. The *ejbLoad()* and *ejbStore()* methods do this; the container calls them as appropriate, based on how you define your transactions (see Chapter 10).
6. Your entity beans can be kicked back into the pool in two ways. If a client calls *remove()* on the home object, the container will call your instance's *ejbRemove()*. The underlying database data is destroyed and so, of course, your entity bean instance will become disassociated with the client's EJB object to which it was bound.
7. The second way your bean can return to the pool is if the EJB container decides that your client has timed out, if the container needs to use your bean to service a different client, or if the container is simply running out

of resources. At this point, your bean is passivated, and the container calls your *ejbStore()* method to ensure the database has the most recent version of your in-memory data. Next the container calls your *ejbPassivate()* method, allowing your bean instance to release held resources. Your bean instance then enters the pool.

8. When the container wants to assign you to an EJB object again, your bean instance must be activated. The container calls your bean's *ejbActivate()* method, allowing your bean to acquire resources. The container then calls your instance's *ejbLoad()* method to load the database data into your bean.

Note that there are a few other minor steps in this process, such as transactional synchronization. Overall, these stages are the essence of a BMP entity bean instance's life cycle. The next step is for you to look at this diagram again and make sure you fully grasp it. Do you understand how a single Java object instance can be pooled and reused, going back and forth through various transitions between the pooled and ready state, perhaps representing different database data each time? If so, congratulations. This is a crucial step towards fully understanding EJB.

Summary

In this chapter, you've seen how to write bean-managed persistent entity beans. Bean-managed persistent entity beans are useful if you need to control the underlying database operations yourself. But the real advantage of EJB comes from *container-managed persistent* entity beans. Container-managed persistent entity beans can be developed much more rapidly because the container handles all data access logic for you. The next chapter covers the new EJB 2.0 container-managed persistence model, an exciting and interesting new addition to EJB.

Writing Container-Managed Persistent Entity Beans

In the previous chapters, we learned the basics of entity beans and wrote a bean-managed persistent entity bean representing a bank account. In this chapter, we'll see how things change when we move to a container-managed persistent (CMP) model. With container-managed persistence, you don't implement any persistence logic (such as JDBC or SQL/J) in the entity bean itself; rather, the EJB container performs storage operations for you. As you will see, this greatly simplifies bean development.

Features of CMP Entity Beans

We'll kick things off by looking at the major differences between CMP and bean-managed persistence (BMP). Before reading this, you should be familiar with the entity bean concepts we covered in the last two chapters.

CMP Entity Beans Are Subclassed

Imagine that you are a bean provider who writes beans that others will consume, such as an independent software vendor (ISV) or a department that writes components that other departments reuse. You need to write your beans to be database-independent because you don't know what storage the consumers of your bean will use. You certainly don't want to allow the consumers of your beans to access your source code, because it violates your

intellectual property rights. Furthermore, if they modify the code, it makes future upgrades to new versions of your components difficult.

To answer this need, the authors of the EJB specification have tried to make CMP have a clean separation between an entity bean and its persistent representation—that is, a separation between the data logic methods (such as logic in your entity bean to add two fields together) and the JDBC. This separation is valuable because you can modify the persistent representation of an entity bean (such as changing from a relational database to an object database) without affecting the entity bean logic. This is a crucial feature for bean providers.

To achieve this clean separation, you write your CMP entity bean class to be devoid of any JDBC or other persistence logic. The container then *generates* the JDBC by *subclassing* your entity bean class. The generated subclass inherits from your entity bean class. Thus, all CMP entity beans are each broken up into two classes: the superclass, which you write and which contains the entity bean data logic; and the subclass, which the container generates and which contains the persistence logic. These two classes achieve a clean separation of entity bean logic and persistent representation. The actual entity bean is a combination of the superclass and the subclass. This is shown in Figure 7.1.



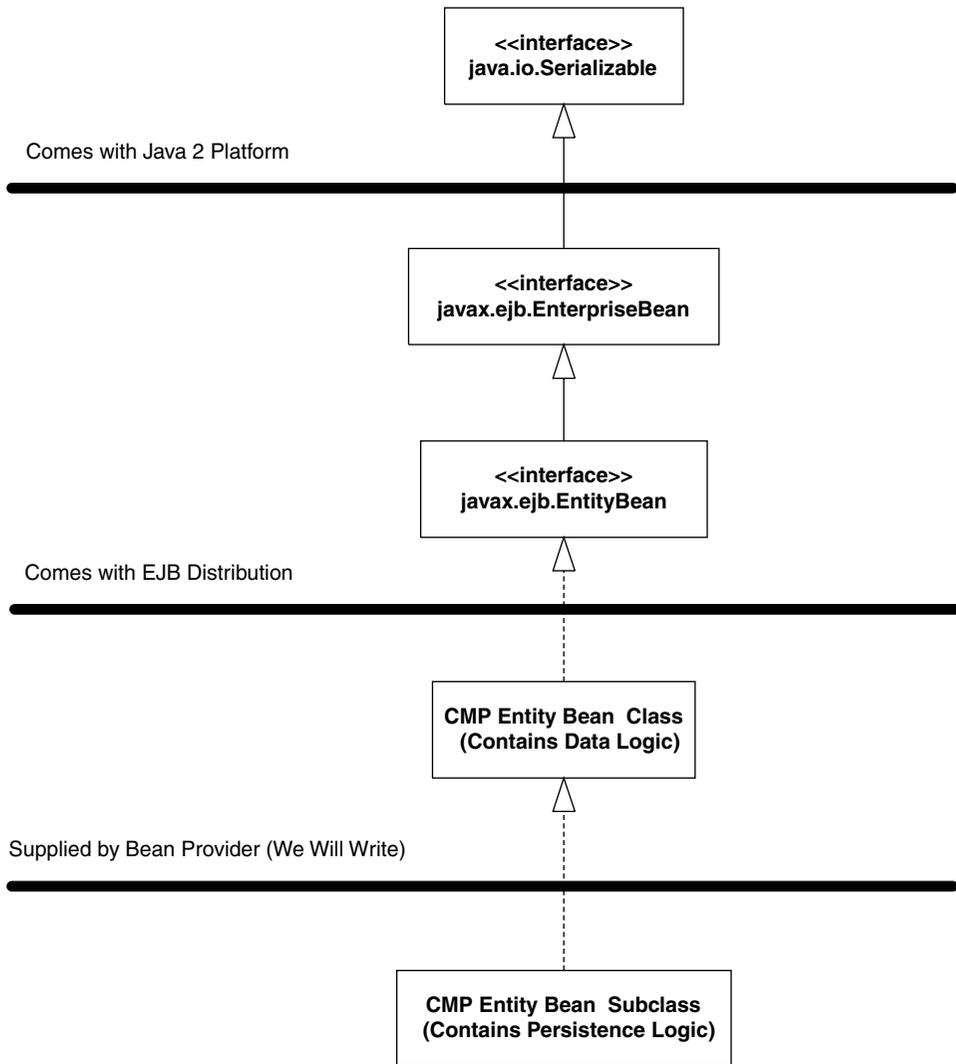
Entity beans are very different between EJB 1.1 and EJB 2.0. EJB 1.1 entity beans do not require the use of subclassing. EJB 2.0 containers must support both the old EJB 1.1 style and the new EJB 2.0 style of entity beans.

CMP Entity Beans Have No Declared Fields

Another issue with CMP is that the container might have additional fields or logic that are part of your persistent representation but are container-specific. As a bean developer, you should be oblivious to this information. Here are two examples:

- A container might keep around a bit vector that tracks which of your entity bean fields have been modified (that is, are dirty) and need to be written to storage. Then when your bean is stored, the container persists only the part of your bean that has changed.
- Your bean might hold references to other beans. The container must preserve referential integrity of those relationships, as described in Chapter 11.

Since every container has its own proprietary way of dealing with your persistent representation, your persistent fields are kept in the subclass, not the superclass. This is another paradigm shift with container-managed persistent



Generated for Us by Container Vendor's Tools

Figure 7.1 The subclassing concept.

entity beans: You don't declare any persistent fields in your bean. For example, take a look at the following is a snippet of code from a BMP bank account entity bean class that we wrote in Chapter 6:

```
// BMP
public class AccountBean implements EntityBean {
    public String accountID;    // PK
    public String ownerName;
```

```
        public double balance;

        ...methods...
    }
}
```

With CMP, the fields are not present. Rather, the container generates your persistent fields in the subclass. For example, the following subclass might be generated from the container tools:

```
// CMP Subclass
public class AccountBeanSubClass extends AccountBean {
    public String accountID;    // PK
    public String ownerName;
    public double balance;

    ...methods...
}
```

CMP Get/Set Methods Are Defined in the Subclass

One corollary of the subclass paradigm is that the subclass, not the superclass, implements the get/set methods. For example, here is that BMP bank account again:

```
// BMP
public class AccountBean implements EntityBean {
    public String accountID;    // PK
    public String ownerName;
    public double balance;

    public String getOwnerName() {
        return ownerName;
    }

    public void setOwnerName(String ownerName) {
        this.ownerName = ownerName;
    }

    ...other methods...
}
```

With CMP, the get/set methods would appear in the subclass, since that is where the fields exist and thus the only place they can be accessed. Here is what the container-generated subclass looks like.

```
// CMP subclass
public class AccountBeanSubClass extends AccountBean {
    public String accountID;    // PK
    public String ownerName;
```

```
public double balance;

public String getOwnerName() {
    return ownerName;
}

public void setOwnerName(String ownerName) {
    this.ownerName = ownerName;
}

...other methods...
}
```

So what does the superclass look like? First, realize that the superclass cannot possibly implement the get/set methods because it doesn't have access to the fields. However, the superclass *does* need to *call* those get/set methods. For example, let's say you have a shopping cart entity bean that contains a *subtotal* field and a *taxes* field on the contents in the shopping cart. One useful method you might want to write is a *getTotal()* method, which returns the subtotal + taxes. That is more than just a simple get/set method and thus cannot be generated automatically by the container in the subclass. Therefore you need to write that method in the superclass yourself. But what would that *getTotal()* method look like? With BMP, it could look like this:

```
// BMP
public class CartBean implements EntityBean {
    ...

    public float getTotal() {
        return this.getSubtotal() + this.getTaxes();
    }

    ...
}
```

This code works well with BMP because we can define the *getSubtotal()* and *getTaxes()* methods. But with CMP, the simple get/set methods *getSubtotal()* and *getTaxes()* are defined in the subclass, so how can we access those get/set methods? The answer is to declare your get/set methods as *abstract* methods in the superclass. An abstract method is a method whose implementation is deferred to a subclass; yet by defining a method as abstract you can call it from the superclass. For example, a CMP shopping cart bean would look like this:

```
// CMP superclass
public abstract class CartBean implements EntityBean {
    // no fields

    // abstract get/set methods
    public abstract float getSubTotal();
}
```

```

    public abstract float getTaxes();

    // other business methods
    public float getTotal() {
        return this.getSubtotal() + this.getTaxes();
    }

    // EJB required methods follow
}

```

The subclass for this bean is the subclass we showed earlier. As another example, a CMP account bean would look like this:

```

// CMP superclass
public abstract class AccountBean implements EntityBean {
    // no fields

    // abstract get/set methods
    public abstract String getOwnerName();
    public abstract void setOwnerName(String ownerName);

    // EJB required methods follow
}

```

CMP Entity Beans Have an Abstract Persistence Schema

So far, we've discussed how the container generates JDBC code, persistent fields, and get/set method implementations. One lurking question is how does the container know what to generate? The answer is that you *declare it* in your bean's deployment descriptors. The EJB container inspects the deployment descriptors to figure out what to generate. This definition of how you'd like to be persisted is called your *abstract persistence schema*. For example, here is a snippet from an Account deployment descriptor:

```

...
<cmp-version>2.x</cmp-version>

<abstract-schema-name>AccountBean</abstract-schema-name>

<cmp-field>
  <field-name>accountID</field-name>
</cmp-field>

<cmp-field>
  <field-name>ownerName</field-name>
</cmp-field>

<cmp-field>
  <field-name>balance</field-name>

```

```
</cmp-field>

<primkey-field>accountID</primkey-field>
...
```

Here is a brief explanation of this deployment descriptor snippet.

- The *cmp-version* must be 2.x if you want to take advantage of EJB 2.0 CMP. If you are on the older EJB 1.1 specification, you should define this to be 1.x. For an example of an EJB 1.1 CMP bean, see the book's accompanying source code.
- The *abstract-schema-name* is the nickname you want to give this abstract persistence schema. It can have any value you want. We recommend naming it after your bean. Later we will reference this nickname when doing queries.
- The *cmp-field* elements are your container-managed persistent fields. Each field is a persistent field that the container will generate in the subclass. The names of these fields must match the names of your abstract get/set methods, except the first letter is not capitalized. For example, if your abstract get/set methods are *getOwnerName()* and *setOwnerName()* then your *cmp-field* should be called *ownerName*. The container derives the types of these fields from the get/set methods as well.

We will see a complete example of an abstract persistence schema later in this chapter.

CMP Entity Beans Have a Query Language

Another piece of our CMP entity bean puzzle is addressing how to query entity beans. To enable clients of your bean to find you, you must define *finder methods*. For example, in BMP you'd define this method in your home interface:

```
public Collection findBigAccounts(int minimum);
```

The home object would delegate this call to your bean, whose implementation would be:

```
public Collection.ejbFindBigAccounts(int minimum) {
    // Perform JDBC, and return primary keys for
    // all accounts whose balance is greater
    // than the minimum passed in
}
```

With CMP, the container generates this JDBC for us. However, we need a way to tell the container how to generate that JDBC, because the container can't

magically know what *find big accounts* means. We want to specify how to generate the persistence code in a portable way so that we don't have to completely rewrite the definitions of these finder methods every time we port our bean to a new container.

The solution to this challenge is the *EJB Query Language* (EJB-QL). EJB-QL is an object-oriented SQL-like syntax for querying entity beans. It contains a SELECT clause, a FROM clause, and an optional WHERE clause. You write the EJB-QL code in the deployment descriptor, and the container should be able to generate the corresponding database logic (such as SQL), perhaps with some help from the container tools. This is a similar concept to the Object Query Language (OQL) described in Chapter 5.

Here is an example of EJB-QL that finds all accounts:

```
SELECT OBJECT(a)
FROM Account AS a
WHERE a.accountID IS NOT NULL
```

If you are using a relational database, at deployment time and with the help of the container's tools that you use, the container will inspect this code and generate the appropriate JDBC code.

Here is another example that satisfies the *findBigAccounts()* home method:

```
SELECT OBJECT(a)
FROM Account AS a
WHERE a.balance > ?1
```

In the above code, *?1* means the first parameter passed in, which in this case is the variable *minimum*.

We will see more EJB-QL in the example later in this chapter. There is also a complete reference in Appendix D.



Not all fields within the bean have to be managed by the container. You might be pulling data manually from a secondary source, or you might have calculated fields. The EJB container automatically notifies your bean class during persistent operations, allowing you to manage these fields.

In general, containers are not responsible for persisting any data in the superclass, such as entity context references or environment naming contexts used for JNDI lookups. You never store these persistently as container-managed fields because they contain runtime EJB-specific information, and they do not represent persistent business data.

The complete process of developing and deploying a CMP entity bean is shown in Figure 7.2.

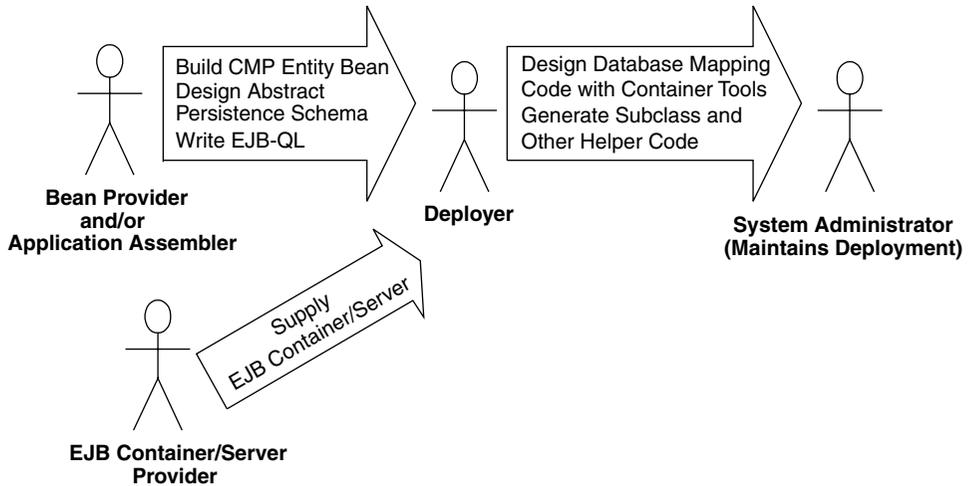


Figure 7.2 The process of developing and deploying a CMP entity bean.

CMP Entity Beans Can Have `ejbSelect()` Methods

The final major difference between BMP and CMP entity beans is that CMP entity beans can have special `ejbSelect()` methods. An `ejbSelect()` method is a query method (like a finder method) but is not directly exposed to the client in the home interface or component interface. Rather, `ejbSelect()` is used internally within an entity bean as a helper method to access a storage. `ejbSelect()` is useful when you have entity beans in relationships with external data, such as other entity beans.

For example, in our bank account example from the previous chapter, we defined a method called `ejbHomeGetTotalBankValue()`, which added up the total of all bank accounts in the bank table by performing a SQL SELECT statement using JDBC. With CMP, you shouldn't be writing this JDBC code—rather, the container should generate it for you in an `ejbSelect()` method, and you should call that `ejbSelect()` method from the `ejbHomeGetTotalBankValue()` method. You then tell the container how to write the `ejbSelect()` method just like you do a finder method—by using the EJB Query Language (EJB QL) described earlier.

For example, you might define the following method in your entity bean:

```
public abstract double.ejbSelectAllAccountBalances()
    throws FinderException;

public double.ejbHomeGetTotalBankValue() throws Exception {
```

```
// Get a collection of bank account balances
Collection c = this.ejbSelectAllAccountBalances();

// Loop through collection and return sum
}
```

ejbSelect() methods are not exposed to end clients via the remote interface or local interface. They *must* be called from within your bean, either from a business method or a home business method.

The value of *ejbSelect()* methods are threefold:

- Select methods can perform fine-grained database operations that your bean needs, but that you do not want to expose to end clients.
- Select methods can retrieve data from other entity beans that you have relationships with (see Chapter 11 to learn more about relationships).
- Like finder methods, select methods can return entity beans. But select methods are more powerful because they can also return container-managed fields, such as our example above—it returns a collection of *double* values.

You tell the container about how to implement your select method by defining an EJB-QL query string. For more details on how EJB-QL affects *ejbSelect()* methods, see Appendix D.



As you may have noticed by now, the major differences between CMP and BMP lie in the entity bean class and the deployment descriptors. The remote interface, local interface, home interface, local home interface, and primary key class remain basically the same. This means it is possible to switch between CMP and BMP without changing the clients who call your beans, which is a nice side effect.

Implementation Guidelines for Container-Managed Persistence

Now that we've explored CMP entity beans in theory, let's see how to build CMP entity beans. The method implementations of your BMP entity beans should be different for CMP. No longer are you controlling the routine persistent operations of your beans, and so many of the methods can be left empty—the container will do it for you. Table 7.1 summarizes what you should implement in each method, assuming your entity bean's persistence is container managed. Take a quick glance at the table for now. As you can see, many of the database-intensive operations have been reduced in scope significantly. You

Table 7.1 Descriptions and Implementation Guidelines for Container-Managed Persistent Entities

METHOD	EXPLANATION	TYPICAL IMPLEMENTATION
setEntityContext() (same as BMP)	If the container wants to increase its pool size of bean instances, it instantiates a new entity bean instance. Following this, the container calls the instance's <i>setEntityContext()</i> . This method associates a bean with <i>context information</i> —information about the bean's environment. Once this method is called, the bean can access information about its environment.	Stick the entity context somewhere, such as in a member variable. You can access the context later to acquire environment information, like security information, from the container. You should also request any resources your instance will need regardless of what data the bean represents. <i>The bean is now in a pool, does not have any specific database data inside of it, and is not bound to any particular EJB object.</i>
ejbFind<...>(<...>) (new for CMP)	<i>You do not write finder methods for container-managed beans.</i> The EJB container will handle <i>all</i> issues relating to finding data for you. But how does the EJB container know what kinds of finder methods you want in your bean? After all, an infinite variety of ways exist to find data in a database. The answer is that your EJB container ships with tools for this purpose. You use EJB-QL and the container tools to tell the container what logic to execute when the client performs a finder method on the home object.	You should not implement these methods for CMP entity beans.
ejbSelect<...>(<...>) (new for CMP)	ejbSelect() methods are helper methods that perform queries internally by your bean but are not accessible to clients of your bean.	Define this method as abstract. Then write EJB-QL in the deployment descriptor to set up the query.
ejbHome<...>(<...>) (new for CMP)	Sometimes you need methods on an entity bean that are not specific to any given data instance (or row)—for example, counting up the total number of accounts in a table. You can write <i>ejbHome</i> methods to perform these operations. The <i>ejbHome</i> methods are special business methods because they are called from a bean in the pool before the bean is associated with any specific data. Clients call these methods from the home interface or local home interface.	Perform your global operations, such as counting up the rows in a database and returning the results to the client. The fast-and-easy way to achieve this is to use JDBC. The cleaner (but lower-performing if you're not careful) way is to call <i>ejbSelect()</i> and perhaps other entity bean methods.

continues

Table 7.1 Descriptions and Implementation Guidelines for Container-Managed Persistent Entities (*continued*)

METHOD	EXPLANATION	TYPICAL IMPLEMENTATION
<p><code>ejbCreate(<... >)</code> (new for CMP)</p> <p><i>Note: You do not need to write any <code>ejbCreate()</code> methods if you don't want EJB clients to be able to create new database data. Some systems may allow creation of data to occur via direct database inserts through batch files or other means.</i></p>	<p>When a client calls <code>create()</code> on a home object, the container then calls <code>ejbCreate()</code> on a pooled bean instance. <code>ejbCreate()</code> methods are responsible for creating new database data and initializing your bean.</p>	<p><i>Do not create database data in this method.</i></p> <p>Rather, validate the client's initialization parameters. Call your <code>abstract set()</code> methods to initialize the generated bean subclass to the parameters passed in. The container will then use these values in the subclass to create the database data for you. Your entity bean instance is then no longer in the pool—it now has specific database data inside it. The container will bind your instance to a particular EJB objects.</p>
<p><code>ejbPostCreate(<... >)</code> (same as BMP)</p>	<p>There is one <code>ejbPostCreate(...)</code> for each <code>ejbCreate(...)</code>. Each pair has the same parameters. The container calls your bean instance's <code>ejbPostCreate(...)</code> method following <code>ejbCreate(...)</code>.</p>	<p>The container calls <code>ejbPostCreate()</code> after it has associated your bean instance with an EJB object. You can now complete your initialization by doing anything you need to that requires that EJB object, such as passing your bean's EJB object reference to other beans.</p> <p><i>Note: By now the EJB container will have created your primary key object, so you can retrieve and use it.</i></p>
<p><code>ejbActivate()</code> (same as BMP)</p>	<p>When a client calls a business method on an EJB object, but no entity bean instance is bound to the EJB object, the container needs to take a bean from the pool and transition it into a ready state. This is called <i>activation</i>. On activation, the <code>ejbActivate()</code> method is called by the EJB container.</p>	<p>Acquire any bean-specific resources, like socket connections, that your bean needs to service a particular client when it is moved into the ready state.</p>

METHOD	EXPLANATION	TYPICAL IMPLEMENTATION
<code>ejbLoad()</code> (new for CMP)	The EJB container calls this to load database data into your bean instance, based on the current transactional state.	<i>Do not read data from the database in this method.</i> Rather, the EJB container will read in data from the database for you automatically right before calling your <code>ejbLoad()</code> method. It does this by setting your container-managed fields to the data it reads from the database. In this method, you should perform any utilities you need to work with the read-in data, such as decompressing a text field.
<code>ejbStore()</code> (new for CMP)	The EJB container calls this to update the database to the new values of your in-memory fields, thus synchronizing the database. The current transactional state dictates when this method is called. This method is also called during passivation, directly before <code>ejbPassivate()</code> .	<i>Do not update the database in this method.</i> Rather, the EJB container will update the database for you automatically right after calling your <code>ejbStore()</code> method. It does this in the subclass by writing your container-managed fields to the database. In this method, you should prepare your container-managed fields to be written to the database. For example, you can compress the text of your fields if necessary by calling your own abstract <code>set()</code> methods.
<code>ejbPassivate()</code> (same as BMP)	The EJB container calls this method when it wants to return your entity bean to the pool. This is called <i>passivation</i> and is the opposite of activation. On passivation, the <code>ejbPassivate()</code> method is called by the EJB container.	Release any resources, such as socket connections, that you allocated in <code>ejbActivate()</code> and that your bean was holding during the ready state for a particular client.
<code>ejbRemove()</code> (new for CMP)	The client calls the home object's <code>remove()</code> method to destroy database data <code>remove()</code> , then calls your <code>ejbRemove()</code> . Note that this does not destroy the Java Object because the object can be pooled and reused for different data.	<i>Do not destroy database data in this method.</i> Rather, simply perform any operations that must be done before the data in the database is destroyed. The EJB container will destroy the data for you right after <code>ejbRemove()</code> is called.
<code>unsetEntityContext()</code> (same as BMP)	This method disassociates a bean from its environment. The container calls this right before your entity bean instance is destroyed (when it wants to reduce the pool size).	Release any resources you allocated during <code>setEntityContext()</code> , and get ready for garbage collection.

Let's take a look at each of the files that we must create for our entity bean component.

Product.java

Remote clients will call our remote interface. The only case in which a remote client should call an entity bean is when you are writing small test applications to exercise your entity bean's API, as we will do in this example. Otherwise you should use the local interface for performance reasons, and wrapper your entity beans with session beans (see Chapter 13). The remote interface is shown in Source 7.1.

Our remote interface is quite simple. It has methods to modify the entity bean instance's fields and throws remote exceptions to indicate system-level errors.

```
package examples;

import javax.ejb.*;
import java.rmi.RemoteException;

/**
 * These are the public business methods of ProductBean.
 *
 * This remote interface is what remote clients operate
 * on when they interact with beans. The EJB container
 * will implement this interface; the implemented object
 * is called the EJB Object, which delegates invocations
 * to instances of the entity bean class.
 */
public interface Product extends EJBObject {

    public String getName() throws RemoteException;
    public void setName(String name) throws RemoteException;

    public String getDescription() throws RemoteException;
    public void setDescription(String description) throws
        RemoteException;

    public double getBasePrice() throws RemoteException;
    public void setBasePrice(double price) throws RemoteException;

    public String getProductID() throws RemoteException;
}
```

Source 7.1 Product.java.

ProductLocal.java

Our local interface is our business interface called by local clients, such as session beans or other entity beans. It is shown in Source 7.2.

The local interface is trivially different than the remote interface. The only differences are the lack of thrown *RemoteExceptions* and the fact that we extend *EJBLocalObject* rather than *EJBObject*.

ProductHome.java

Next, we have the product's home interface, which is shown in Source 7.3. As with the remote interface, this home interface should be used only by remote clients, such as a standalone application.

Our home interface defines a single *create()* method to create a new product in the database. It returns a Product EJB object so the client can manipulate the

```
package examples;

import javax.ejb.*;

/**
 * These are the public business methods of ProductBean.
 *
 * This local interface is what local clients operate
 * on when they interact with our bean. The container
 * will implement this interface; the implemented object
 * is called the EJB local object, which delegates
 * invocations to instances of the entity bean class.
 */
public interface ProductLocal extends EJBLocalObject {

    public String getName();
    public void setName(String name);

    public String getDescription();
    public void setDescription(String description);

    public double getBasePrice();
    public void setBasePrice(double price);

    public String getProductID();
}
```

Source 7.2 ProductLocal.java.

```
package examples;

import javax.ejb.*;
import java.rmi.RemoteException;
import java.util.Collection;

/**
 * This is the home interface for Product. This interface
 * is implemented by the EJB container. The implemented
 * object is called the Home Object, and serves as a
 * factory for EJB Objects.
 *
 * One create() method is in this Home Interface, which
 * corresponds to the ejbCreate() method in the bean class.
 */
public interface ProductHome extends EJBHome {

    /**
     * Creates a product
     *
     * @param productID The number of the product (unique)
     * @param name The name of the product
     * @param description Product description
     * @param basePrice Base Price of product
     *
     * @return The newly created EJB Object.
     */
    Product create(String productID, String name, String description,
        double basePrice) throws CreateException, RemoteException;

    // Finder methods. These are implemented by the
    // container. You can customize the functionality of
    // these methods in the deployment descriptor through
    // EJB-QL and container tools.

    public Product findByPrimaryKey(ProductPK key) throws
        FinderException, RemoteException;

    public Collection findByName(String name) throws FinderException,
        RemoteException;

    public Collection findByDescription(String description) throws
        FinderException, RemoteException;

    public Collection findByBasePrice(double basePrice) throws
        FinderException, RemoteException;
}
```

Source 7.3 ProductHome.java.

```
    public Collection findExpensiveProducts(double minPrice) throws
        FinderException, RemoteException;

    public Collection findCheapProducts(double maxPrice) throws
        FinderException, RemoteException;

    public Collection findAllProducts() throws FinderException,
        RemoteException;
}
```

Source 7.3 ProductHome.java (continued).

entity bean data and throws a *javax.ejb.CreateException* to indicate an application-level problem.

We also expose all sorts of finder methods to find existing products. One of the finders returns a single EJB object, while others return a *java.util.Collection* of multiple EJB objects. This is needed if the finder methods find more than one matching object. Note that *findByPrimaryKey()* should never return a collection, because primary keys must be unique.

ProductLocalHome.java

Our entity bean's local home interface is the more optimized (see Chapter 2) home interface that session beans or other entity beans should use. The code is in Source 7.4.

ProductPK.java

Our primary key class is defined by *ProductPK.java*, shown in Source 7.5. This unique identifier uses a *productID* that could represent the product's SKU number.

As with BMP, CMP dictates that your primary key class must be serializable. Because the EJB container is persisting for you, it may need to query the primary key class and manipulate or compare its fields with the fields in your bean. Thus, an important restriction with CMP is that the fields you have in your primary key class must come from the container-managed fields defined in your deployment descriptor.

In our example, the *ProductPK* class is valid because it is serializable and because its public fields come from our container-managed fields, which we will define shortly in the deployment descriptor.

```
package examples;

import javax.ejb.*;
import java.util.Collection;

/**
 * This is the local home interface for Product.
 * This interface is implemented by the EJB container.
 * The implemented object is called the local home object,
 * and serves as a factory for EJB local objects.
 *
 * One create() method is in this Home Interface, which
 * corresponds to the ejbCreate() method in the bean class.
 */
public interface ProductLocalHome extends EJBLocalHome {

    /*
     * Creates a product
     *
     * @param productID The number of the product (unique)
     * @param name The name of the product
     * @param description Product description
     * @param basePrice Base Price of product
     *
     * @return The newly created EJB local Object.
     */
    ProductLocal create(String productID, String name, String
description, double basePrice) throws CreateException;

    // Finder methods. These are implemented by the
    // container. You can customize the functionality of
    // these methods in the deployment descriptor through
    // EJB-QL and container tools.

    public ProductLocal findByPrimaryKey(ProductPK key) throws
FinderException;

    public Collection findByName(String name) throws FinderException;

    public Collection findByDescription(String description) throws
FinderException;

    public Collection findByBasePrice(double basePrice) throws
FinderException;

    public Collection findExpensiveProducts(double minPrice) throws
FinderException;
}
```

Source 7.4 ProductLocalHome.java.

```
        public Collection findCheapProducts(double maxPrice) throws
FinderException;

        public Collection findAllProducts() throws FinderException;
    }
}
```

Source 7.4 ProductLocalHome.java (*continued*).

```
package examples;

import java.io.Serializable;

/**
 * Primary Key class for our 'Product' Container-Managed
 * Entity Bean
 */
public class ProductPK implements java.io.Serializable {

    /*
     * Note that the primary key fields must be a
     * subset of the the container-managed fields.
     * The fields we are marking as container-managed in
     * our Bean are productID, name, desc, and basePrice.
     * Therefore our PK fields need to be from that set.
     */
    public String productID;

    public ProductPK(String productID) {
        this.productID = productID;
    }

    public ProductPK() {}

    public String toString() {
        return productID.toString();
    }

    public int hashCode() {
        return productID.hashCode();
    }

    public boolean equals(Object prod) {
        return ((ProductPK)prod).productID.equals(productID);
    }
}
```

Source 7.5 ProductPK.java.

ProductBean.java

Next, we have our container-managed entity bean implementation, *ProductBean.java*, shown in Source 7.6.

```
package examples;

import javax.ejb.*;

/**
 * Entity Bean that demonstrates Container-Managed persistence.
 *
 * This is a product that's persistent. It has an ID #, a name,
 * a description, and a base price.
 */
public abstract class ProductBean implements EntityBean {

    protected EntityContext ctx;

    public ProductBean() {
    }

    //-----
    // Begin abstract get/set methods
    //-----

    public abstract String getName();
    public abstract void setName(String name);
    public abstract String getDescription();
    public abstract void setDescription(String description);
    public abstract double getBasePrice();
    public abstract void setBasePrice(double price);
    public abstract String getProductID();
    public abstract void setProductID(String productID);

    //-----
    // End abstract get/set methods
    //-----

    //-----
    // Begin EJB-required methods. The methods below
    // are called by the Container, and never called
    // by client code.
    //-----

    /**
```

Source 7.6 ProductBean.java.

```
* Called by Container.
* Implementation can acquire needed resources.
*/
public void ejbActivate() {
    System.out.println("ejbActivate() called.");
}

/**
 * EJB Container calls this method right before it
 * removes the Entity Bean from the database.
 * Corresponds to when client calls home.remove().
 */
public void ejbRemove() {
    System.out.println("ejbRemove() called.");
}

/**
 * Called by Container.
 * Releases held resources for passivation.
 */
public void ejbPassivate() {
    System.out.println("ejbPassivate () called.");
}

/**
 * Called from the Container. Updates the entity bean
 * instance to reflect the current value stored in
 * the database.
 *
 * Since we're using Container-Managed Persistence, we
 * can leave this method blank. The EJB Container will
 * automatically load us in the subclass.
 */
public void ejbLoad() {
    System.out.println("ejbLoad() called.");
}

/**
 * Called from the Container. Updates the database to
 * reflect the current values of this in-memory Entity Bean
 * instance representation.
 *
 * Since we're using Container-Managed Persistence, we can
 * leave this method blank. The EJB Container will
 * automatically save us in the subclass.
 */
public void ejbStore() {
```

Source 7.6 ProductBean.java (continued).

```
        System.out.println("ejbStore() called.");
    }

    /**
     * Called by Container. Associates this Bean instance with
     * a particular context. Once done, we can query the
     * Context for environment info
     */
    public void setEntityContext(EntityContext ctx) {
        System.out.println("setEntityContext called");
        this.ctx = ctx;
    }

    /**
     * Called by Container. Disassociates this Bean instance
     * with a particular context environment.
     */
    public void unsetEntityContext() {
        System.out.println("unsetEntityContext called");
        this.ctx = null;
    }

    /**
     * Called after ejbCreate(). Now, the Bean can retrieve
     * its EJBObject from its context, and pass it as a 'this'
     * argument.
     */
    public void ejbPostCreate(String productID, String name, String
description, double basePrice) {
        System.out.println("ejbPostCreate() called");
    }

    /**
     * This is the initialization method that corresponds to the
     * create() method in the Home Interface.
     *
     * When the client calls the Home Object's create() method,
     * the Home Object then calls this ejbCreate() method.
     *
     * We need to initialize our Bean's fields with the
     * parameters passed from the client, so that the Container
     * can create the corresponding database entries in the
     * subclass after this method completes.
     */
    public String ejbCreate(ProductPK productID, String name,
String description, double basePrice)
        throws CreateException {
```

Source 7.6 ProductBean.java (continued).

```

        System.out.println("ejbCreate() called");

        setProductID(productID);
        setName(name);
        setDescription(description);
        setBasePrice(basePrice);

        return new ProductPK(productID);
    }

    // No finder methods
    // (they are implemented by Container)

    //-----
    // End EJB-required methods
    //-----
}

```

Source 7.6 ProductBean.java (*continued*).

This bean is more complex than our bank account example. We've defined many finder methods, and we have more persistent fields. Yet even though we've added this complexity, our bean is less than 40 percent of the size of our Bank Account bean. This is an amazing reduction in code complexity. And because our bean has no database code in it, we have reduced the chance for bugs in our bean due to user error working with JDBC code. This is a huge savings in development and testing time.

We do not have any fields, since the container declares them in the subclass. We have a few abstract get/set methods, which the container also implements in the subclass. The only really interesting method is *ejbCreate()*, which takes the parameters passed in from the client and calls the bean's own abstract *set()* methods to populate the bean with the initialization data. The container then performs a SQL INSERT in the subclass once *ejbCreate()* concludes.

The rest of our bean is just empty EJB-required methods and comments. In fact, if we took the comments, whitespace, and *println*'s out, the bean would just be this:

```

package examples;
import javax.ejb.*;

public abstract class ProductBean implements EntityBean {
    protected EntityContext ctx;

    public abstract String getName();

```

```
public abstract void setName(String name);
public abstract String getDescription();
public abstract void setDescription(String description);
public abstract double getBasePrice();
public abstract void setBasePrice(double price);
public abstract String getProductID();
public abstract void setProductID(String productID);

public void ejbActivate() {}
public void ejbRemove() {}
public void ejbPassivate() {}
public void ejbLoad() {}
public void ejbStore() {}
public void setEntityContext(EntityContext ctx) {
    this.ctx = ctx;
}
public void unsetEntityContext() { this.ctx = null; }

public void ejbPostCreate(String productID, String name,
    String description, double basePrice) {}

public String ejbCreate(String productID, String name,
    String description, double basePrice) {
    setProductID(productID);
    setName(name);
    setDescription(description);
    setBasePrice(basePrice);
    return productID;
}
}
```

The Deployment Descriptor

We now need to inform our container about our entity bean, including our container-managed fields and our EJB-QL. The deployment descriptor is shown in Source 7.7.

Our deployment descriptor begins by identifying the name of the bean, then the bean class, and so on, which is the same as BMP. We then define the container-managed fields, which must match the abstract get/set methods in the enterprise bean class.

The bulk of the descriptor following this is the code for our queries. For example, the *findExpensiveProducts()* finder method locates all products that are more expensive than the *double* parameter passed in. To instruct the container on how to implement this finder functionality, we define our EJB-QL as follows:

```
<![CDATA[SELECT OBJECT(a) FROM ProductBean AS a WHERE basePrice > ?1]]>
```

```

<?xml version="1.0"?>

<!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>
  <enterprise-beans>
    <entity>
      <ejb-name>Product</ejb-name>
      <home>examples.ProductHome</home>
      <remote>examples.Product</remote>
      <local-home>examples.ProductLocalHome</local-home>
      <local>examples.ProductLocal</local>
      <ejb-class>examples.ProductBean</ejb-class>
      <persistence-type>Container</persistence-type>
      <prim-key-class>examples.ProductPK</prim-key-class>

      <reentrant>False</reentrant>

      <cmp-version>2.x</cmp-version>
      <abstract-schema-name>ProductBean</abstract-schema-name>

      <cmp-field>
        <field-name>productID</field-name>
      </cmp-field>
      <cmp-field>
        <field-name>name</field-name>
      </cmp-field>
      <cmp-field>
        <field-name>description</field-name>
      </cmp-field>
      <cmp-field>
        <field-name>basePrice</field-name>
      </cmp-field>

      <query>
        <query-method>

          <method-name>findByName</method-name>
          <method-params>
            <method-param>java.lang.String</method-param>
          </method-params>
        </query-method>
      <ejb-ql>
        <![CDATA[SELECT OBJECT(a) FROM ProductBean AS a WHERE name =
?1]]>
      </ejb-ql>
    </entity>
  </enterprise-beans>
</ejb-jar>

```

Source 7.7 ejb-jar.xml.

```
</query>

<query>
  <query-method>
    <method-name>findByDescription</method-name>
    <method-params>
      <method-param>java.lang.String</method-param>
    </method-params>
  </query-method>
  <ejb-ql>
    <![CDATA[SELECT OBJECT(a) FROM ProductBean AS a WHERE description
= ?1]]>
  </ejb-ql>
</query>

<query>
  <query-method>
    <method-name>findByBasePrice</method-name>
    <method-params>
      <method-param>double</method-param>
    </method-params>
  </query-method>
  <ejb-ql>
    <![CDATA[SELECT OBJECT(a) FROM ProductBean AS a WHERE basePrice =
?1]]>
  </ejb-ql>
</query>

<query>
  <query-method>
    <method-name>findExpensiveProducts</method-name>
    <method-params>
      <method-param>double</method-param>
    </method-params>
  </query-method>
  <ejb-ql>
    <![CDATA[SELECT OBJECT(a) FROM ProductBean AS a WHERE basePrice >
?1]]>
  </ejb-ql>
</query>

<query>
  <query-method>
    <method-name>findCheapProducts</method-name>
    <method-params>
      <method-param>double</method-param>
    </method-params>
```

Source 7.7 ejb-jar.xml (continued).

```

    </query-method>
    <ejb-ql>
      <![CDATA[SELECT OBJECT(a) FROM ProductBean AS a WHERE basePrice <
?1]]>
    </ejb-ql>
  </query>

  <query>
    <query-method>
      <method-name>findAllProducts</method-name>
      <method-params>
      </method-params>
    </query-method>
    <ejb-ql>
      <![CDATA[SELECT OBJECT(a) FROM ProductBean AS a WHERE productID
IS NOT NULL]]>
    </ejb-ql>
  </query>

</entity>
</enterprise-beans>

<assembly-descriptor>
  <container-transaction>
    <method>
      <ejb-name>Product</ejb-name>
      <method-intf>Remote</method-intf>
      <method-name>*</method-name>
    </method>
    <trans-attribute>Required</trans-attribute>
  </container-transaction>
</assembly-descriptor>

</ejb-jar>

```

Source 7.7 ejb-jar.xml (*continued*).

When the container interprets this EJB-QL, it generates database access code (such as JDBC) to find all of the expensive products whose *basePrice* column is greater in value than the double passed in, represented by the *?1*. Whenever a client wants to execute a finder method on the home object, the container automatically runs the database access code.

Notice also the word *CDATA*. This instructs the container's XML parser to ignore the text *SELECT OBJECT(a) FROM ProductBean AS a WHERE basePrice > ?1*. This is important because the container's XML parser may think that the

text inside the CDATA section does not comply with the XML standard; it may think the > character is actually the closing of an XML tag, rather than a less-than sign. Thus, all EJB-QL must be enclosed in CDATA sections.

The end of our descriptor associates transactions with our entity bean, which we'll learn about in Chapter 10.

The Container-Specific Deployment Descriptor

In addition to the deployment descriptor, we need to tell the container exactly how to perform persistent operations. This is one trade-off of CMP—you still need to declare persistent rules, rather than code them into your bean using JDBC or SQL/J.

If you're using a relational data store, you need to define exactly how your entity bean's public fields map to that database. Thus, we must define a series of object-relational mapping entries. These entries map entity bean fields to relational database column names. The EJB container uses this mapping when storing or retrieving our container-managed fields from the database. Note

When to Use Custom Primary Key Classes

In our bean we've declared a custom primary key class, *ProductPK*. We then have this element in our deployment descriptor:

```
<prim-key-class>examples.ProductPK</prim-key-class>
```

This is not strictly necessary, however. You can choose not to invent a custom primary key class and just use one of your container-managed fields as the primary key. For example, we could use the *productID* String field as the primary key, rather than wrapping it in another primary key wrapper class. Then we would declare the primary key class to be a *java.lang.String*, and we would have this element after we declare the container-managed fields:

```
<primkey-field>productID</primkey-field>
```

When should you use a custom primary key class, and when should you use one of your fields? In our opinion, you should avoid using your own fields as primary key classes. The reason is because having a primary key class wrapper isolates you from changes to how you'd like to be uniquely represented in an underlying storage. Having a primary key class wrapper makes it much easier to change how you'd like to be uniquely identified without breaking code.

that this is very EJB container-specific! Some EJB containers support object databases and thus do not have a mapping into a two-dimensional relational database. Consult your EJB container's documentation for more information. Our product line's persistent entries for a relational database are shown in Table 7.2. See the book's accompanying source code for the actual descriptor.

Client.java

Our client code is a simple suite of test cases to try out our bean, as shown Source 7.8.

Because this standalone application runs in a separate process from the application server, for testing purposes this client calls through the bean's remote interface rather than a local interface. However, in a real-world scenario, we would wrap this entity bean with a session bean and calling through its local interface.

The client performs a JNDI lookup to acquire the home object and create some entity bean data. We then try out a couple of finder methods. We can loop through the finders' returned collection and call business methods on each EJB object. We then destroy all the EJB objects we created in a *finally*{ } clause.

Running the Client Program

To run the client program, type a command similar to the following (depending on your EJB container's JNDI initialization parameters):

```
java -D java.naming.factory.initial=weblogic.jndi.  
WLInitialContextFactory -Djava.naming.provider.url=t3://localhost:7001  
examples.ProductClient
```

The initialization parameters are required by JNDI to find the home object, as we learned in Chapter 3.

Table 7.2 Sample Persistent Settings for ProductBean

OBJECT/RELATIONAL SETTING (ENTITY BEAN FIELD = RELATIONAL COLUMN NAME)
productID=id
name=name
description=description
basePrice=basePrice

```
package examples;

import javax.ejb.*;
import javax.naming.*;
import java.rmi.*;
import javax.rmi.PortableRemoteObject;
import java.util.*;

/**
 * Client test application on a CMP Entity Bean, Product.
 */
public class ProductClient {

    public static void main(String[] args) throws Exception {

        ProductHome home = null;

        try {
            /*
             * Get a reference to the Product Home Object - the
             * factory for Product EJB Objects
             */
            Context ctx = new InitialContext(System.getProperties());
            home = (ProductHome) PortableRemoteObject.narrow(
                ctx.lookup("ProductHome"), ProductHome.class);

            /*
             * Use the factory to create the Product EJB Object
             */
            home.create("123-456-7890", "P5-350", "350 Mhz Pentium", 200);
            home.create("123-456-7891", "P5-400", "400 Mhz Pentium", 300);
            home.create("123-456-7892", "P5-450", "450 Mhz Pentium", 400);
            home.create("123-456-7893", "SD-64", "64 MB SDRAM", 50);
            home.create("123-456-7894", "SD-128", "128 MB SDRAM", 100);
            home.create("123-456-7895", "SD-256", "256 MB SDRAM", 200);

            /*
             * Find a Product, and print out its description
             */
            Iterator i = home.findByName("SD-64").iterator();
            System.out.println("These products match the name SD-64:");
            while (i.hasNext()) {
                Product prod = (Product) PortableRemoteObject.narrow(
                    i.next(), Product.class);
                System.out.println(prod.getDescription());
            }
        }
    }
}
```

Source 7.8 Client.java.

```
    /*
     * Find all products that cost $200
     */
    System.out.println("Finding all products that cost $200");
    i = home.findByBasePrice(200).iterator();

    while (i.hasNext()) {
        Product prod = (Product) PortableRemoteObject.narrow(
            i.next(), Product.class);
        System.out.println(prod.getDescription());
    }
}
catch (Exception e) {
    e.printStackTrace();
}
finally {
    if (home != null) {
        System.out.println("Destroying products..");

        /*
         * Find all the products
         */
        Iterator i = home.findAllProducts().iterator();
        while (i.hasNext()) {
            try {
                Product prod = (Product) PortableRemoteObject.narrow(
                    i.next(), Product.class);
                if (prod.getProductID().startsWith("123")) {
                    prod.remove();
                }
            }
            catch (Exception e) {
                e.printStackTrace();
            }
        }
    }
}
}
```

Source 7.8 Client.java (continued).

The life cycle of a container-managed persistent entity bean. Each method call shown is an invocation from the container to the bean instance.

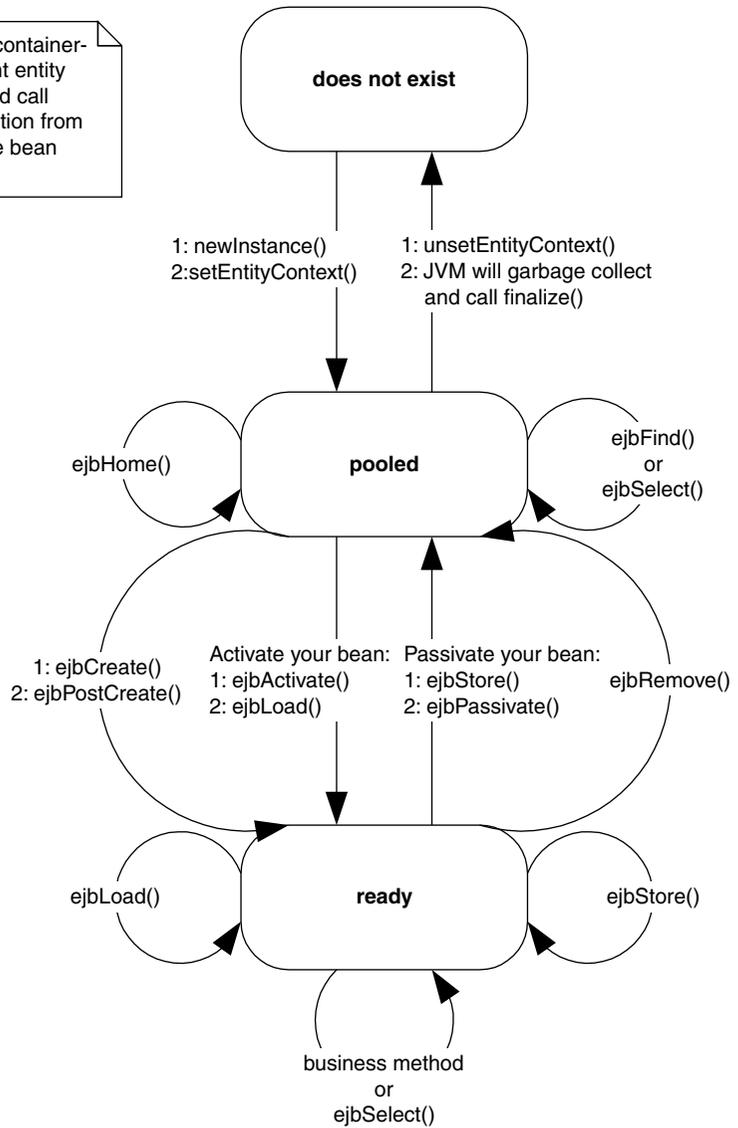


Figure 7.4 The CMP entity bean life cycle.

When we run the client, we first create a few products and then perform a *find* for all products that cost \$200. Indeed, multiple entity beans were returned in our collection, as shown below:

```
These products match the name SD-64:  
64 MB SDRAM  
Finding all products that cost $200  
350 Mhz Pentium  
256 MB SDRAM  
Destroying products..
```

The Life Cycle of a CMP Entity Bean

Now that we've seen a complete CMP entity bean example, let's fully understand how the container interacts with CMP entity beans. Figure 7.4 shows this.

The life cycle of a CMP entity bean is exactly the same as that of a BMP entity bean, which we fully described at the end of the previous chapter (refer back if you need to refresh your memory). The only differences are that *ejbSelect()* methods can be called from the pooled state or ready state.

Summary

In this chapter, you learned how to write CMP entity beans. We saw how the bean instance callback methods differ between BMP and CMP. We then went through an example that modeled a product line. Finally, we wrapped up with a look at the life cycle of a CMP entity bean.

In the next chapter, we'll look at the new EJB 2.0 bean type, message-driven beans.

Introduction to Message-Driven Beans

In this chapter, we will learn about *messaging*, which is a lightweight vehicle for communications. Messaging is more appropriate than RMI-IIOP in numerous scenarios. We'll also learn about *message-driven beans*, special beans that can be accessed via messaging and a new addition to the EJB 2.0 specification.

Specifically, you'll learn about the following:

- An introduction to messaging, including an overview of asynchronous behavior and message-oriented middleware
- A brief tutorial of the Java Message Service (JMS), which message-driven beans depend on
- Features of message-driven beans
- How message-driven beans compare with entity and session beans
- How to develop message-driven beans
- Advanced message-driven bean topics, gotchas, and possible solutions

Motivation to Use Message-Driven Beans

In previous chapters, you learned how to code session and entity beans—distributed components that are accessed using RMI-IIOP. RMI-IIOP is a traditional, heavyweight way to call components. While RMI-IIOP may be useful in many scenarios, several other areas are challenging for RMI-IIOP. Here are just three examples.

Performance. An RMI-IIOP client must wait (or block) while the server performs its processing. Only when the server completes its work does the client receive a return result, which allows it to continue processing.

Reliability. When an RMI-IIOP client calls the server, it has to be running. If the server crashes or the network crashes, the client cannot perform its intended operation.

Support for multiple senders and receivers. RMI-IIOP limits you to a single client talking to a single server at any given time. There is no built-in functionality for multiple clients to broadcast *events* to multiple servers.

Messaging is an alternative to remote method invocations and is shown in Figure 8.1. The idea behind messaging is that a *middleman* sits between the client and the server. (A layer of indirection solves every problem in computer science). This middleman receives messages from one or more *message producers* and broadcasts those messages to one or more *message consumers*. Because of this middleman, the producer can send a message and then continue processing. He can optionally be notified of the response later when the consumer finishes. This is called *asynchronous* programming.

Messaging addresses the three previous concerns with RMI-IIOP as follows.

Performance. A messaging client does not need to block when performing a request. As an example, when you purchase a book using Amazon.com's one-click order functionality, you can continue browsing the site without waiting to see if your credit card authorizes. Unless something goes wrong, Amazon.com sends you a confirmation email afterwards. This type of fire-and-forget system could easily be coded using messaging. When the user clicks to buy the book, a message is sent that results in credit card processing later. The user can continue to browse.

Remote Method Invocations:



Messaging:



Figure 8.1 Remote method invocations vs. messaging.

Reliability. If your message-oriented middleware supports *guaranteed delivery*, you can send a message and know for sure that it will reach its destination, even if the consumer is not available. You send the message to the MOM middleman, and that middleman routes the message to the consumer when he comes back alive again. With RMI-IIOP, this is not possible because there is no middleman: If the server is down, an exception is thrown.

Support for multiple senders and receivers. Most message-oriented middleware products can accept messages from many senders and broadcast them to many receivers. This allows you to have *n*-ary communications.

Note that messaging also has many disadvantages. Performance, for one, can be slower in many circumstances due to the overhead of having the messaging middleman. For a complete comparison of when to (and when not to) use messaging, see Chapter 13.

Message-oriented middleware (MOM) is a term used to refer to any infrastructure that supports messaging. A variety of products are considered to have a MOM-based architecture. Examples include Tibco Rendezvous, IBM MQSeries, BEA Tuxedo/Q, Microsoft MSMQ, Talarian SmartSockets, Progress SonicMQ, and Fiorano FioranoMQ. These products can give you a whole host of value-added services, such as guaranteed message delivery, fault tolerance, load balancing of destinations, subscriber throttling of message consumption, inactive subscribers, and much, much more. By allowing the MOM server to address these infrastructure issues, you can focus on the business task at hand.

The Java Message Service (JMS)

Over the years, MOM systems have evolved in a proprietary way. Each product has its own API, which creates vendor lock-in because code is not portable to other messaging systems. It also hurts developers, because they need to relearn each messaging product's proprietary API.

The Java Message Service (JMS) is a messaging standard, designed to eliminate many of the disadvantages that MOM-based products faced over past years. JMS has two parts: an API, which you write code to send and receive messages, and a Service Provider Interface (SPI) where you plug in JMS drivers. A JMS driver knows how to talk to a specific MOM implementation. The JMS promise is that you can learn the JMS API once and reuse your messaging code with different plug-and-play MOM implementations (an idea similar to the other J2EE APIs, such as JNDI or JDBC).

How Does Guaranteed Message Delivery Work?

With guaranteed message delivery, the MOM system persists your messages to a file, database, or other store. Your message resides in the persistent store until it's sent to a message consumer, *and* the message consumer acknowledges the consumption of the message. If the acknowledgement of a message is not received in a reasonable amount of time, the message remains on the persistent store and is redelivered.

This feature is beneficial when the message consumer is brought down on a regular basis for maintenance, and lost messages are unacceptable. This is especially true in industries such as financial services, where messages represent securities changing hands.

A variation on the guaranteed message delivery theme is certified message delivery. Certified message delivery not only ensures the delivery of a message from a producer to a consumer, but also generates a consumption receipt that is delivered to the message originator, indicating a successful consumption of the message. Certified message delivery is used by producers to better manage communication with consumers.

Another variation of guaranteed message delivery is called store and forward. Store and forward allows a message producer to successfully send a message to an inactive MOM system. The producer transparently spools the message to a local store until the MOM system is reactivated, at which point the message is delivered to the MOM system and forwarded to any available consumers. Guaranteed message delivery without the store-and-forward option requires producers to send messages to active MOM systems, but consumers do not have to be active. Store and forward with guaranteed message delivery allows messages to be sent whether MOM systems or consumers are active or inactive.

Let's explore the JMS API and see how to write a simple JMS program that publishes messages.

Messaging Domains

When you perform messaging, you need to choose a *domain*. A domain is a fancy word for style of messaging. The types of domains are:

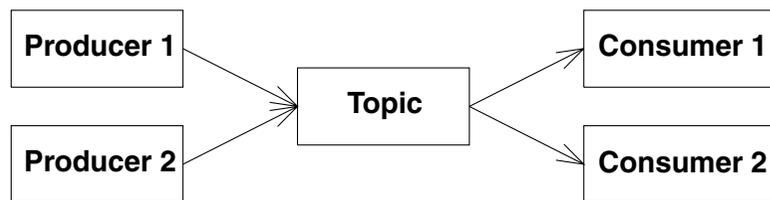
Publish/subscribe (pub/sub). Publish/subscribe is analogous to watching television. Many TV stations broadcast their signals, and many people listen to those broadcasts. Thus, with publish/subscribe, you can have *many*

message producers talking to *many* message consumers. In this sense, the pub/sub domain is an implementation of a distributed event-driven processing model. Subscribers (listeners) register their interest in a particular event *topic*. Publishers (event sources) create messages (events) that are distributed to all of the subscribers (listeners). Producers aren't hard-coded to use specific consumers; rather, the MOM system maintains the subscriber list.

Point-to-point (PTP). Point-to-point is analogous to calling a toll-free number and leaving a voice mail. Some person will listen to your voice mail and then delete it. Thus, with point-to-point, you can have only a single consumer for *each* message. Multiple consumers can grab messages off the queue, but any given message is consumed exactly once. In this sense, point-to-point is a degenerate case of publish/subscribe. Multiple producers can send messages to the queue, but each message is delivered only to a single consumer. The way this works is that publishers send messages directly to the consumer or to a centralized *queue*. Messages are typically distributed off the queue in a first-in, first-out (FIFO) order, but this isn't assured.

The difference between publish/subscribe and point-to-point is shown in Figure 8.2.

Publish/Subscribe:



Point-to-Point:

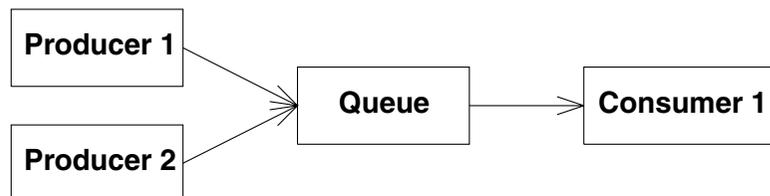


Figure 8.2 Publish/subscribe vs. point-to-point.



Another domain called **request/reply** is less broadly used than the others. The request/reply domain is analogous to RMI-IIOP. It requires any producer that generates a message to receive a reply message from the consumer at some later point in time. Typically, most MOM architectures implement a request/reply paradigm using the technologies supplied in the point-to-point and publish/subscribe domains.

The JMS API

The JMS API is more involved than RMI-IIOP. You need to become familiar with many different interfaces to get going. Despite the complexities involved with working with each of these interfaces, low-level topology issues such as networking protocol, message format and structure, and server location are mostly abstracted from the developer.

The JMS programming model is shown in Figure 8.3. It is explained as follows:

1. *Locate the JMS driver.* You first need to get access to the driver to the particular JMS product you're using. You do this by *looking up* the driver using JNDI, just like with JDBC. The driver is called a *ConnectionFactory*.
2. *Create a JMS connection.* A *JMS Connection* is an active connection to the JMS provider, managing the low-level network communications (similar to a JDBC connection). You use the *ConnectionFactory* to get a *Connection*. If you're in a large deployment, this connection might be load-balanced across a group of machines.
3. *Create a JMS session.* A *JMS Session* is a helper object that you use when sending and receiving messages. It serves as a factory for message consumers and producers, and also allows you to encapsulate your messages in transactions. You use the *Connection* to get a *Session*.
4. *Locate the JMS destination.* A *JMS Destination* is the channel to which you're sending or from which you're receiving messages. Locating the right destination is analogous to tuning into the right channel when watching television or answering the correct phone, so that you get the messages you desire. Your deployer typically sets up the destination in advance by using your JMS provider's tools, so that the destination is permanently setup. Your code *looks up* that destination using JNDI. This enables your programs to use the destination over and over again at runtime.
5. *Create a JMS producer or a JMS consumer.* If you want to send messages, you need to call a JMS object to pass it your messages. This object is called the *producer*. To receive messages, you call a JMS object and ask it for a message. This object is called the *Consumer*. You use the *Session* and *Destination* to make a *Producer* or a *Consumer*.

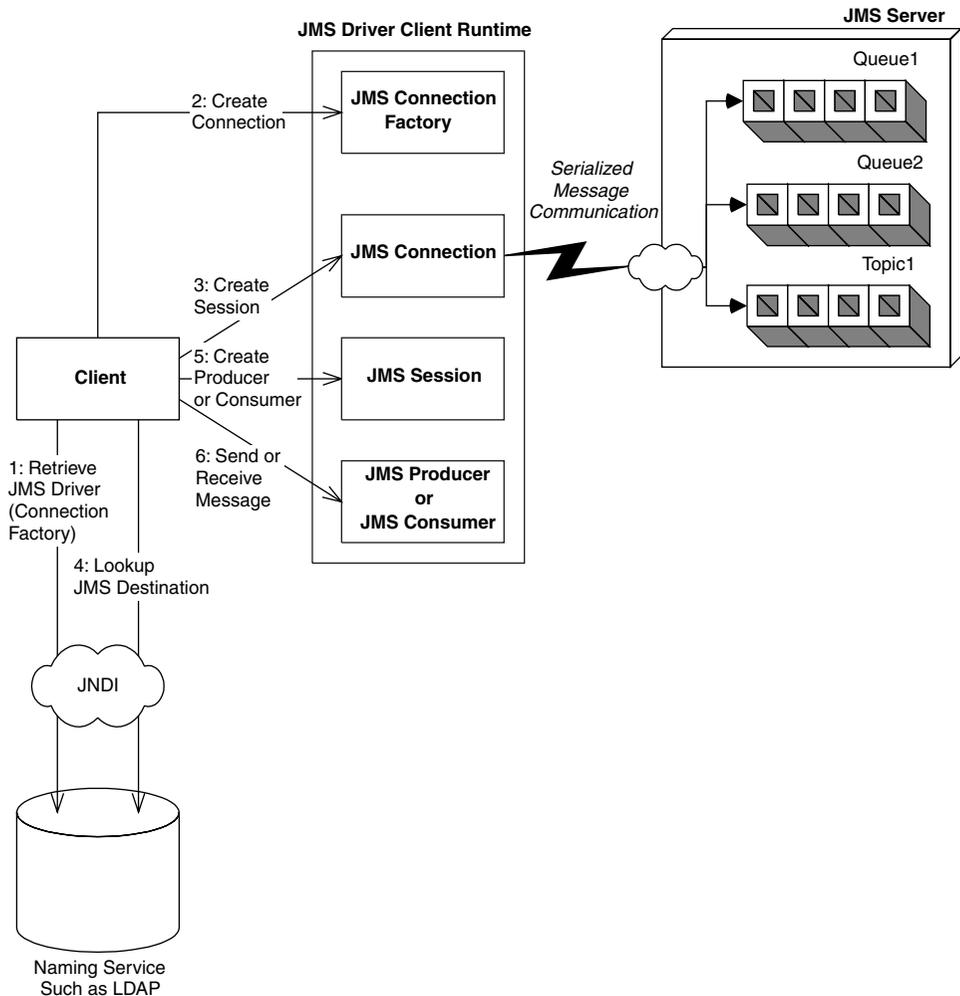


Figure 8.3 Client view of a JMS system.

6. *Send or receive your message.* If you're producing, you first need to put your message together. There are many different types of messages, such as text, bytes, streams, objects, and maps. After you instantiate your message, you send it using the *Producer*. If, on the other hand, you're receiving messages, you first receive a message using the *Consumer*, and then crack it open (depending on the message type) and see what it is.

Everything we just learned applies to both publish/subscribe and point-to-point messaging. The words in *italics* above represent actual JMS interface names. There are two different flavors of those interfaces, and the flavor you

use depends on if you're using publish/subscribe or point-to-point. See Table 8.1 for a list.



As you can see from Table 8.1, point-to-point has two types of message consumers: a receiver and a browser. What do you think these are for? And why does publish/subscribe have only one type of consumer?

As an example, the code for a client application that publishes a *TextMessage* to a topic using publish/subscribe is provided in Source 8.1.

Most of Source 8.1 is self-explanatory. Here are the answers to a few questions you might have.

- The parameters to *InitialContext* should be your JNDI driver information. If your JMS provider is integrated into your EJB server, the JNDI parameters should be the same as those when you lookup an EJB home. You specify this via the command-line using the `-D` switch to the *java* runtime. See the book's accompanying source code for example scripts.
- Our JNDI name for the *TopicConnectionFactory* is *javax.jms.TopicConnectionFactory* but it could be anything—it depends on your container's policy and also where you choose to place it using your container's tools.
- When we create a *Session*, we pass two parameters: *false*, which indicates that we don't want to use transactions (see Chapter 10 for more on transactions), and *Session.AUTO_ACKNOWLEDGE*, which indicates how we should acknowledge messages that we receive. Since our code is sending (not receiving) messages, this parameter doesn't matter. If you're curious about how message acknowledgement works, see Table 8.3 later in this chapter.

Note that this example does not illustrate point-to-point. The point-to-point code is basically the same, except we use the point-to-point interfaces listed in

Table 8.1 The Two Flavors of JMS Interfaces

PARENT INTERFACE	POINT-TO-POINT	PUB/SUB
ConnectionFactory	QueueConnectionFactory	TopicConnectionFactory
Connection	QueueConnection	TopicConnection
Destination	Queue	Topic
Session	QueueSession	TopicSession
MessageProducer	QueueSender	TopicPublisher
MessageConsumer	QueueReceiver, QueueBrowser	TopicSubscriber

```
import javax.naming.*;

import javax.jms.*;
import java.util.*;

public class Client {

    public static void main (String[] args) throws Exception {

        // Initialize JNDI
        Context ctx = new InitialContext(System.getProperties());

        // 1: Lookup ConnectionFactory via JNDI
        TopicConnectionFactory factory =
            (TopicConnectionFactory)
                ctx.lookup("javax.jms.TopicConnectionFactory");

        // 2: Use ConnectionFactory to create JMS connection
        TopicConnection connection =
            factory.createTopicConnection();

        // 3: Use Connection to create session
        TopicSession session = connection.createTopicSession(
            false, Session.AUTO_ACKNOWLEDGE);

        // 4: Lookup Destination (topic) via JNDI
        Topic topic = (Topic) ctx.lookup("testtopic");

        // 5: Create a Message Producer
        TopicPublisher publisher = session.createPublisher(topic);

        // 6: Create a text message, and publish it
        TextMessage msg = session.createTextMessage();
        msg.setText("This is a test message.");
        publisher.publish(msg);
    }
}
```

Source 8.1 TopicClient.java.

Table 8.1. We'll leave the point-to-point example as an exercise for you.

Note, too, that this example does not demonstrate any consumption logic. Although message consumption is an important concept, it's not relevant to our discussion, because message-driven beans effectively act as our message consumers.

You should now know enough about JMS to be productive with message-driven beans. If you want to learn more about JMS, a free tutorial is available on <http://java.sun.com>. Rather than repeating this free information, let's cover some more interesting topics—JMS-EJB integration, advanced message-driven bean topics, and gotchas.

Single-Threaded versus Multithreaded Beans

One great benefit of EJB is you don't need to write thread-safe code. You design your enterprise beans as single-threaded components and never need to worry about thread synchronization when concurrent clients access your component. Your EJB container automatically instantiates multiple instances of your component to service concurrent client requests.

The container's thread services can be both a benefit and a restriction. The benefit is that you don't need to worry about race conditions or deadlock in your application code. The restriction is that some problems lend themselves well to multithreaded programming, and that class of problems cannot be easily solved in an EJB environment.

So why doesn't the EJB specification allow for multithreaded beans? EJB is intended to relieve component developers' worry about threads or thread synchronization. The EJB container handles those issues for you by load-balancing client requests to multiple instances of a single-threaded component. An EJB server provides a highly scalable environment for single-threaded components.

If the EJB specification allowed for beans to control threads, then a Pandora's box of problems would result. For example, an EJB container would have a very hard time controlling transactions if beans are randomly starting and stopping threads, especially because transaction information is often associated with a thread.

One alternative to threading is to use a transactional messaging API, such as JMS, that allows for asynchronous actions to occur in a distributed object environment. JMS enables you to safely and reliably achieve multitasking without the beans themselves messing around with threads.

The bottom line is that EJB was not meant to be a Swiss army knife, solving every problem in existence. It was designed to assist with server-side *business problems*, which are largely single-threaded. For applications that absolutely must be multithreaded, EJB may not be the correct choice of distributed object architectures.

Integrating JMS with EJB

JMS-EJB integration is a compelling idea. It would allow EJB components to benefit from the value proposition of messaging, such as nonblocking clients and n -ary communications.

To help us figure out why Sun needed to make message-driven beans, let's imagine for a moment that we worked at Sun Microsystems. We'd have several different approaches to integrating JMS with EJB. They include:

Using a Java object that receives JMS messages to call EJB components.

Rather than coming up with a whole new type of bean, Sun could have promoted the idea of a Java object that knew how to receive messages as a wrapper for your other types of EJB components, such as session beans and entity beans. The problems with this approach are:

- You'd need to write special code to register yourself as a listener for JMS messages. This is a decent amount of code (as we saw previously).
- Your Java object would need some way of starting up, since it wrapped your other EJB components. If the class ran in-process to the container, you would need to use an EJB server-specific *startup class* to activate your Java object when the EJB server came up. This is not portable. If the class ran out-of-process, your application would not be as elegant and you'd need to deal with multiple processes context-switching.
- Your Java object wouldn't receive any services from an EJB container, such as automatic life cycle management, clustering, pooling, and transactions. You would need to hard-code this yourself, which is difficult and error-prone.
- You would need to hard-code the JMS destination name in your Java object. This hurts reusability, because you couldn't reuse that Java object with other destinations. If you read the destination from a disk (such as with property files), this is a bit clunky.

Reuse an existing type of EJB component somehow to receive JMS messages. Sun could have tried to shoehorn session beans or entity beans into receiving JMS messages. Problems with this approach include:

- **Threading.** If a message arrives for a bean while its processing other requests, how can it take that message, given that EJB does not allow components to be multithreaded?
- **Life cycle management.** If a JMS message arrives and there are no beans, how does the container know to create a bean?

- **Transactions.** If a bean error occurs, what happens? Does the message get put back on the queue?

What Is a Message-Driven Bean?

A *message-driven bean* is a special EJB component that can receive JMS messages. A message-driven bean consumes messages from queues or topics that are sent by any valid JMS client. Message-driven beans are new to EJB 2.0.

A message-driven bean is decoupled from any clients that send messages to it. A client cannot access a message-driven bean through a component interface. JMS is the API you use to send messages to message-driven beans. This is shown in Figure 8.4.

The following are some major characteristics of message-driven beans.

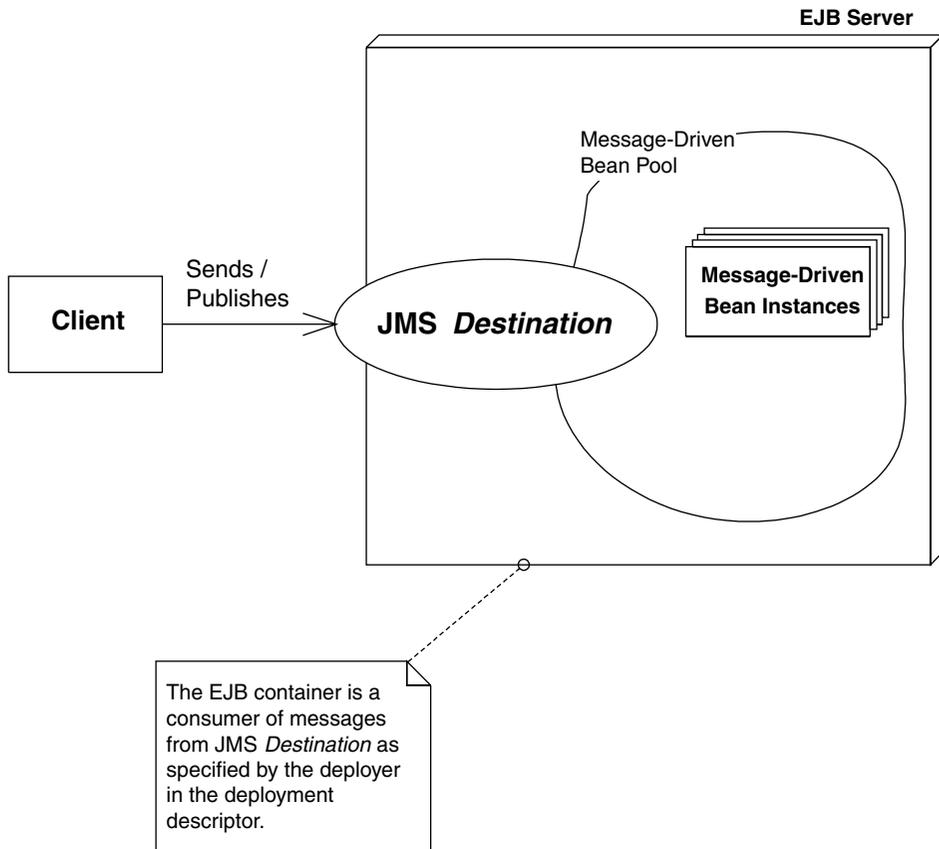


Figure 8.4 A client calling message-driven beans.

A message-driven bean does not have a home interface, local home interface, remote interface, or a local interface. You do not call message-driven beans using an object-oriented remote method invocation interface. The reason is that message-driven beans process messages, and those messages can come from any messaging client, such as an MQSeries client, an MSMQ client, or (most likely) a J2EE client using the JMS API. Message-driven beans can consume any valid JMS message from either a topic or a queue.

Message-driven beans have a single, weakly typed business method.

Message-driven beans are merely receiving messages from a JMS destination, and that JMS destination doesn't know anything about what's inside the messages. Therefore a message-driven bean has only *one* business method, called *onMessage()*. This method accepts a JMS *Message*, which could represent anything—a *BytesMessage*, *ObjectMessage*, *TextMessage*, *StreamMessage*, or *MapMessage*. You cannot provide lots of different business methods on your message-driven beans; rather, you need to crack open the message at runtime and figure out what to do with it, perhaps with a bunch of *if* statements. In formal terms, you don't get compile-time type-checking of messages that are consumed; rather, you need to use the *instanceof* operator to determine the exact type of a consumed message at runtime. This also means that you need to be careful to make sure the message you receive is intended for you. In comparison, session or entity beans can support lots of strongly typed business methods. Type checking can be performed at compile time to ensure that clients are properly using a given interface.

Message-driven beans do not have any return values. This is because message-driven beans are decoupled from message producers. The message producers don't wait for your message-driven bean to respond because they continue processing once the message is sent. The good news is that it is possible to send a response to a message producer using any number of design patterns. We discuss this later in this chapter.

Message-driven beans cannot send exceptions back to clients. Again, this is because message producers don't wait for your message-driven bean to receive a message so therefore can't receive any exceptions. In fact, the EJB specification prohibits application exceptions from being thrown by a message-driven bean. A message-driven bean, however, *is* allowed to generate system exceptions. The container (rather than the client) handles system exceptions.

Message-driven beans are stateless. Message-driven beans hold no conversational state. It would be impossible to spread messages across a cluster of message-driven beans if a message-driven bean held state. In this sense, they are similar to stateless session beans because the container can similarly treat each message-driven bean instance as equivalent to all other

instances. All instances are anonymous and do not have an identity that is visible to a client. Thus, multiple instances of the bean can process multiple messages from a JMS destination concurrently.

Message-driven beans can be durable or nondurable subscribers. A *durable* subscription to a topic means that a JMS subscriber receives all messages, even if the subscriber is inactive. If a message is sent to a topic that has an inactive durable subscriber, the message is persisted and delivered when the durable subscriber is once again active. A *nondurable* subscription to a topic means the subscriber receives only messages that are published while the subscriber is active. Any messages delivered while the subscriber is inactive are lost. Since message-driven bean containers are JMS consumers, the container can register itself as a durable or nondurable subscriber to messages published to a topic. Durability allows persistent messages to be sent to a topic even though the application server hosting the message-driven bean consumers has crashed. The messages will persist until the crashed application server restarts and the durable subscriber message-driven bean container positively acknowledges consumption all of the stored messages.

Developing Message-Driven Beans

Let's now take a look at what's involved with developing message-driven beans.

The Semantics

Message-driven beans are classes that implement two interfaces: *javax.jms.MessageListener* and *javax.ejb.MessageDrivenBean*. Additionally, every message-driven bean implementation class must provide an *ejbCreate()* method that returns *void* and accepts no arguments. Here is what the *javax.jms.MessageListener* interface looks like:

```
public interface javax.jms.MessageListener {
    public void onMessage(Message message);
}
```

Here is what the *javax.ejb.MessageDrivenBean* interface looks like:

```
public interface javax.ejb.MessageDrivenBean
    extends javax.ejb.EnterpriseBean {

    public void ejbRemove()
```

```
throws EJBException;  
  
public void setMessageDrivenContext(MessageDrivenContext ctx)  
    throws EJBException;  
}
```

We summarize the methods that must be provided in every message-driven bean implementation class in Table 8.2.

Given this simple description, you can see that developing message-driven beans is significantly less complicated than developing session or entity beans. The number of methods that have to be implemented is less than with session or entity beans.

The life cycle of a message-driven bean is also very straightforward. See Figure 8.5 for a diagram of the life cycle of a message-driven bean. A message-driven bean is either in the *does not exist* state or in the *pooled* state. When a container decides to add another instance to its pool, it creates a new instance, passes the instance its *MessageDrivenContext* object describing the domain, and then calls *ejbCreate()* allowing the bean to initialize itself. That application server will likely create an initial pool of beans at boot time and then increase the size of the pool as the quantity of messages increases. A container will remove an instance from the pool and destroy it at system shutdown or when the container decides it needs to decrease the size of the pool to conserve cache space. If the container decides to take an instance out of the bean pool, it calls the bean's *ejbRemove()* method.

A Simple Example

Now that we've learned the theory behind message-driven beans, let's apply our knowledge to construct a simple bean that logs text messages to the screen. In the future, you could generalize this bean and make it into a generic logging facility, where you have different log levels depending on the urgency of the log.

This is a trivial example and not demonstrative of real-world systems. It is, however, a good template to use when writing your own beans. If you want to see a real-world message-driven bean in action that uses other EJB components, see Chapter 17, along with the book's accompanying source code.

As we will see when writing this bean, the rules for writing message-driven beans are trivial. Part of the reason is that message-driven beans are brand new to the EJB 2.0 specification. As the technology matures, we're sure that Sun will come up with a bunch of special rules and restrictions for message-driven beans. For now, we get to bask in the simplicity.

Table 8.2 Methods to Be Implemented in Message-Driven Beans

METHOD	DESCRIPTION
onMessage(Message)	This method is invoked for each message that is consumed by the bean. The input parameter of the method is the incoming message that is being consumed. The container is responsible for serializing messages to a single message-driven bean. A single message-driven bean can process only one message at a time. It is the container's responsibility to provide concurrent message consumption by pooling multiple message-driven bean instances. A single instance cannot concurrently process messages, but a container can. This method does not have to be coded for reentrancy and should not have any thread synchronization code contained within.
ejbCreate()	This method is invoked when a message-driven bean is first created and added to a pool. Application server vendors can implement an arbitrary algorithm that decides when to add message-driven bean instances from the pool. Beans are typically added to the pool when the component is first deployed or when message throughput increases. Bean developers should initialize variables and references to resources needed by the bean, such as other EJBs or database connections. Bean developers should initialize only references to resources that are needed for every message that is consumed by the bean, as opposed to gaining access and releasing the resource every time a message is consumed.
ejbRemove()	This method is invoked when a message-driven bean is being removed from a pool. Application server vendors can implement an arbitrary algorithm that decides when to remove message-driven bean instances from the pool. Beans are typically removed from the pool when the component is being undeployed or when message throughput decreases and idle instances are wasting system resources. Bean developers should use this method to clean up any dangling resources that are used by the bean.
setMessageDrivenContext(MessageDrivenContext)	This method is called as part of the event transition that a message-driven bean goes through when it is being added to a pool. This method is called before the <i>ejbCreate()</i> method is invoked. The input parameter for this method is an instance of the <i>MessageDrivenContext</i> interface. The input parameter gives the bean access to information about the environment that it executes within. The only methods on the <i>MessageDrivenContext</i> that are accessible by the message-driven bean are transacted-related methods. Other methods, such as <i>getCallerPrincipal()</i> , cannot be invoked in this method because message-driven beans do not have home, local home, remote, or local interface and have no client-visible security context.

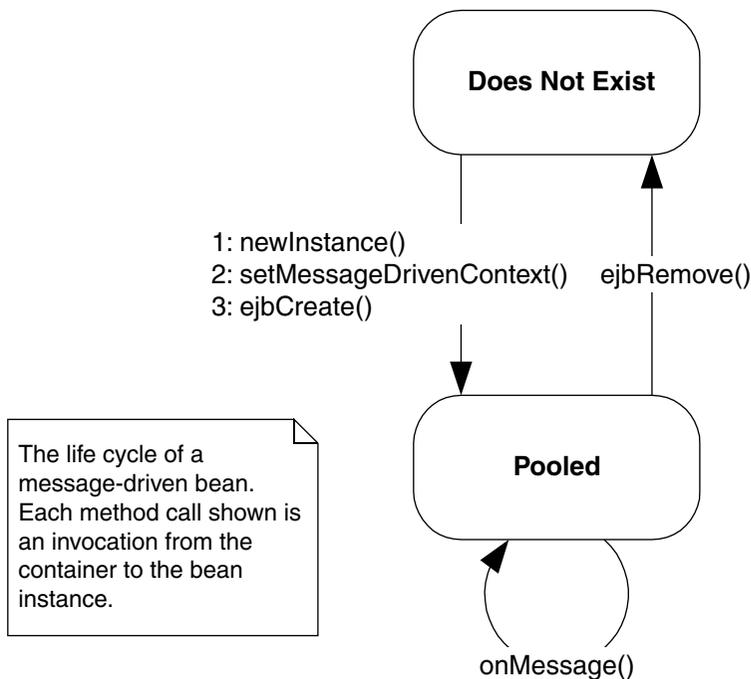


Figure 8.5 Life cycle of a message-driven bean.

The Bean Implementation Class

Since message-driven beans do not have home, component, local home, or local interfaces associated with them, we can completely skip designing the public interface to our bean. We can get right to the heart of development of this bean and write the implementation class. The code for the implementation class is shown in Source 8.2.

This is the most basic message-driven bean. Notice the following:

- Our bean implements the *javax.ejb.MessageDrivenBean* interface that makes it a message-driven bean.
- Our bean implements the *javax.jms.MessageListener* interface that provides the methods necessary for message consumption.
- The *setMessageDrivenContext(...)* method associates a bean with an environment. We store the context as a member of the implementation class so that other methods of the bean can make use of it.

```
package examples;

import javax.ejb.*;
import javax.jms.*;

/**
 * Sample Message-Driven Bean
 */
public class LogBean implements MessageDrivenBean, MessageListener {

    protected MessageDrivenContext ctx;

    /**
     * Associates this Bean instance with a particular context.
     */
    public void setMessageDrivenContext(MessageDrivenContext ctx) {
        this.ctx = ctx;
    }

    /**
     * Initializes the bean
     */
    public void ejbCreate() {
        System.err.println("ejbCreate()");
    }

    /**
     * Our one business method
     */
    public void onMessage(Message msg) {

        if (msg instanceof TextMessage) {
            TextMessage tm = (TextMessage) msg;

            try {
                String text = tm.getText();
                System.err.println("Received new message : " + text);
            }
            catch(JMSEException e) {
                e.printStackTrace();
            }
        }
    }

    /**
     * Destroys the bean
     */
}
```

Source 8.2 LogBean.java.

```
*/  
public void ejbRemove() {  
    System.err.println("ejbRemove()");  
}  
}
```

Source 8.2 LogBean.java (continued).

- The bean is stateless and does not contain any client-specific state that spans messages. Therefore each bean is identical and has an identical initialization method—a simple *ejbCreate()* that takes no arguments.
- The *onMessage(...)* method receives a message, checks to make sure that the passed-in message is of type *TextMessage* by using the *instanceof* operator, and then downcasts appropriately. If the passed in message is not a *TextMessage*, the method just returns. *TextMessage* is a particular type of JMS message that has methods for getting and setting the text as the body of the message. After downcasting the input parameter, the method prints out the content of the message, if any exists.
- When the bean is being destroyed, there is nothing to clean up so we have a very simple *ejbRemove()* method.

Notice that you don't hard-code message-driven beans for a specific queue or topic. Your message-driven bean code is independent of destination. The deployment descriptor determines whether a topic or a queue is consumed, as we will see.

The Deployment Descriptor

Message-driven beans have only a couple of deployment descriptor tags applicable to them. The portion of the deployment descriptor relevant to our simple message-driven bean is shown in Source 8.3.

Table 8.3 contains definitions for additional deployment descriptor tags that are unique to message-driven beans. All of these tags are optional. Just glance over it now—it's not important to fully understand them if you're just starting to learn message-driven beans. See Appendix C for a complete deployment descriptor 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>
  <enterprise-beans>

    <!--
      For each message-driven bean that is located in an
      ejb-jar file, you have to define a <message-driven> entry
      in the deployment descriptor.
    -->
    <message-driven>

      <!-- The nickname for the bean could be used later in DD -->
      <ejb-name>Log</ejb-name>

      <!-- The fully qualified package name of the bean class -->
      <ejb-class>examples.LogBean</ejb-class>

      <!-- The type of transaction supported (see Chapter 10) -->
      <transaction-type>Container</transaction-type>

      <!-- Whether I'm listening to a topic or a queue -->
      <message-driven-destination>
        <destination-type>javax.jms.Topic</destination-type>
      </message-driven-destination>
    </message-driven>
  </enterprise-beans>
</ejb-jar>
```

Source 8.3 ejb-jar.xml for the simple bean.

As you can see, developing the deployment descriptor for message-driven beans is simple. In addition to the characteristics that are definable for all message-driven beans, application server vendors can provide value-add extensions in an application server-specific deployment descriptor. For example, an application server vendor may provide a deployment descriptor parameter that defines the maximum size of the message-driven bean pool and/or another parameter that defines its initial size.

A question that you may be wondering now is, “Exactly how does the application server bind a message-driven bean container to a specific topic or queue?” If you look carefully at the deployment descriptor provided in Source 8.3, the `<message-driven-destination>` tag specifies whether the bean should consume queue or topic messages; however, it never indicates which topic or

Table 8.3 Optional Sub-Elements for the <message-driven> Tag

METHOD	DESCRIPTION	EXAMPLE
<message-selector>	<p>A message selector filters, or limits, which messages are sent to your bean. Message selectors are very powerful—they increase overall performance by reducing the number of messages delivered to clients that have no interest in the message. To use message selectors, first your JMS client sets up header fields on JMS messages using the JMS API. For example, the JMS client might call <code>message.setStringProperty("logLevel", "severe")</code> before sending the message. When the JMS destination receives the message, the container applies the message selector criteria defined in the deployment descriptor. Only messages with headers that match the selector are delivered.</p>	<pre><message-selector> JMSType = 'log' AND logLevel = 'severe' </message-selector></pre> <p><i>Note:</i> You can use more complicated SQL-like functionality here as well, such as arithmetic, logical operators (AND/OR/NOT), and more. If you use greater than (>) or less than (<) signs then you need to wrap this in a CDATA section, to avoid XML parsing confusion, as we described in Chapter 7. See the JMS specification, downloadable from http://java.sun.com, for the complete rules for message selector syntax, which is a subset of the SQL 92 standard.</p>
<acknowledge-mode>	<p>If you let the container handle transactions for you (called <i>container-managed transactions</i> described in Chapter 10), and the container delivers the message to you in a transaction. There is no need for message acknowledgement then, because if the transaction rolls back, the message is automatically put back on the queue. If you program your own transactions (called <i>bean-managed transactions</i>), the transaction occurs within your bean, and begins and ends <i>after</i> the message has been delivered to your bean; thus the consumption of the message occurs outside the transaction. Therefore, if you're using bean-managed transactions, you need to tell the container to</p>	<pre><acknowledge-mode> Auto-acknowledge </acknowledge-mode></pre>

continues

Table 8.3 Optional Sub-Elements for the `<message-driven>` Tag (continued)

METHOD	DESCRIPTION	EXAMPLE
<code><message-driven-destination></code> <code><destination-type></code> <code><subscription-durability></code>	<p><i>acknowledge</i> messages. Setting this tag to <i>Auto-acknowledge</i> forces the container to acknowledge a message when the message-driven bean's <i>onMessage()</i> method has successfully returned. Setting this tag to <i>Dups-ok-acknowledge</i> allows the container to acknowledge the message when it feels like doing so to save resources and processing time. Since it may not acknowledge the message fast enough, you run the risk of the JMS destination sending you a duplicate message. You should use this only if you can tolerate duplicate messages.</p> <p>This tag provides advice to the deployer as to whether a message-driven bean is intended for consumption by a queue or a topic. The bean developer is responsible for providing initial values to these tags even though the bean deployer may override them. This element has two sub-elements: <code><destination-type></code> and <code><subscription-durability></code>. The valid values for <code><destination-type></code> are <i>javax.jms.Queue</i> and <i>javax.jms.Topic</i>. If the <code><destination-type></code> tag is <i>javax.jms.Topic</i>, the <code><subscription-durability></code> element may be included to indicate whether this message-driven bean should be a durable topic subscriber of messages. The valid values for <code><subscription-durability></code> are <i>durable</i> and <i>non-durable</i>.</p>	

queue the message-driven bean container should bind to. This is done purposely to make message-driven beans portable across application servers. Since the names of actual topics and queues deployed into a JMS server are application server-specific, the mapping of a bean's container to a specific JMS server destination has to be done in an application server-specific deployment descriptor. Most EJB vendors are expected to have a custom deployment descriptor that binds the bean to a specific destination.

The Client Program

The client application for our simple message-driven bean example is the JMS client we developed earlier in this chapter in Source 8.1. This shows you the power of message-driven beans—our client is solely a JMS client, and the application is never the wiser that a message-driven bean is consuming the messages.

If you'd like to try this example yourself, see the book's accompanying source code for compilation and deployment scripts.

Advanced Concepts

So far, we have discussed the mechanics of with developing message-driven beans. Now let's take a deeper look at the support containers can give for message-driven beans. We'll see how they might integrate with transactions, provide advanced JMS features, and behave in a clustered environment.

Transactions

Message-driven beans do not run in the same transaction as the producer who sends the message, because there are typically two transactions associated with every durable JMS message (one transaction for the producer to put the message on the queue, and another transaction for the message-driven bean to get the message off the queue). It is theoretically impossible for the message-driven bean to participate in the same transaction (and hence the same unit of work) as the producer, because until the producer commits the transaction, the message wouldn't even appear on the queue!

For a complete discussion of transactions and how they apply to message-driven beans, see Chapter 10.

Security

Message-driven beans do not receive the security identity of the producer who sends the message, because there is no standard way to stick security information into a JMS message. Therefore you cannot perform EJB security operations (described in Chapter 9) with message-driven beans.

Load-balancing

Clustering message-driven beans is quite different than clustering session or entity beans (see Chapter 14). With session and entity beans, your requests are *load-balanced* across a group of containers. The load-balancing algorithm *guesses* which server is the least-burdened server and pushes requests out to that server. It's guessing because the client's RMI-IIOP runtime can never know for sure which server is the least burdened, because all load-balancing algorithms are approximation algorithms based on imperfect historical data. This is called a *push model* because we are pushing requests out to the server, and the server has no say about what requests it receives.

With message-driven beans, producers put messages onto a destination. The messages reside in the destination until a consumer takes the messages off of the destination, or (if the messages are nondurable) the server hosting the destination crashes. This is a *pull model*, since the message resides on the destination until a consumer asks for it. The containers contend (fight) to get the next available message on the destination.

Thus, message-driven beans feature an *ideal* load-balancing paradigm and distribute load more smoothly than session or entity beans. The server that is the least burdened and asks for a message gets the message. The tradeoff for this optimal load-balancing is that messaging has extra overhead because a destination "middleman" sits between the client and the server.

Duplicate Consumption in a Cluster

Since JMS topics use the publish/subscribe model, it's possible that a message sent to a JMS topic will be delivered to more than one consumer. Many containers will create a pool of many message-driven beans instances to concurrently process multiple messages, so some concern can arise around message-driven bean containers that subscribe to JMS topics.

In particular, if a message-driven bean container has pooled five instances of its message-driven bean type and is subscribed to the *DogTopic*, how many consumers will consume a message sent to the *DogTopic* topic? Will the message be consumed by each message-driven bean instance in the container or just once by a single message-driven bean? The answer is simple: A container

that subscribes to a topic consumes any given message only once. This means that for the five instances that the container created to concurrently process messages, only one of the instances will receive any particular message freeing up the other instances to process other messages that have been sent to the *DogTopic*.

Be careful, though. *Each* container that binds to a particular topic will consume a message sent to that topic. The JMS subsystem will treat each message-driven bean container as a separate subscriber to the message. This means that if the same message-driven bean is deployed to many containers in a cluster, then *each* deployment of the message-driven bean will consume a message from the topic it subscribes to. If this is not the behavior you want, and you need to consume messages exactly once, you should consider deploying a queue instead of a topic.

For message-driven beans that bind to a queue, the JMS server will deliver any message on the queue to only one consumer. Each container registers as a consumer to the queue, and the JMS server load-balances messages to consumers based upon availability. Message-driven beans that bind to queues that are deployed in a cluster are ideal for scalable processing of messages. For example, if you have two servers in your cluster and 50 messages on a queue, each server will consume on average 25 messages—as opposed to a single server responsible for consuming 50 messages.

Message-driven beans in a cluster are shown in Figure 8.6. Notice that many message-driven beans process the same message from Topic #1. Also notice that only a single bean processes any given message from Queue #1.

Message-Driven Bean Gotchas

Although developing message-driven beans is a straightforward process, many dark corners and caveats can be encountered unknowingly. In this section we uncover some of these message-driven demons and suggest solutions to help speed you on your way to successful implementation.

Message Ordering

A JMS server is not guaranteed to deliver messages to a pool of message-driven beans in any particular order. The container likely attempts to deliver messages in an order that doesn't impact the concurrency of message processing, but there is no guarantee as to the order that the beans actually process the message. Therefore message-driven beans should be prepared to process messages that are not in sequence. For example, a message adding a second hamburger to a fast food order might be processed before the message indicating

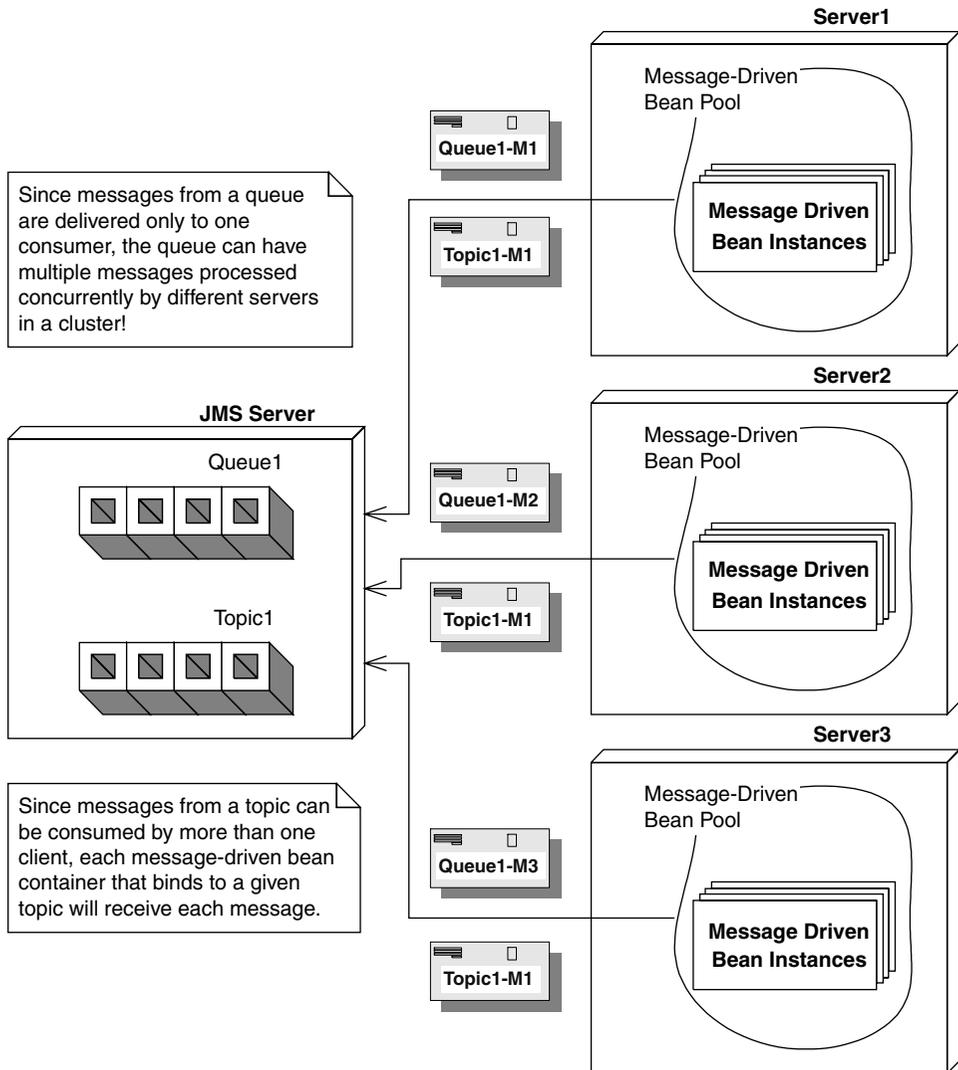


Figure 8.6 Message-driven beans in a cluster.

that a new fast food order with a hamburger should be created. Bean developers must take these scenarios into account and handle them appropriately.

Missed `ejbRemove()` Calls

As with session and entity beans, you are not guaranteed that the container will call your `ejbRemove()` method when your bean is destroyed. In particular,

Using Queues to Partition Business Processing in a Cluster

Suppose you have two clusters of machines: One cluster is configured for a development and test environment, and the other cluster is configured for a production environment. You need to make sure that traffic coming from test clients are sent to the development cluster, while traffic coming from real clients is sent to the production cluster.

As one solution, you could setup your JMS server with two queues: *DevelopmentQueue* and *ProductionQueue*. You could deploy a series of JSPs or front-end stateless session beans that analyze each incoming request, format it into a JMS message, and then place requests onto one of the queues. Requests that come from an internal development machine could be placed onto the *DevelopmentQueue*, and all other requests could be placed on the *ProductionQueue*.

On the back end, you could configure two clusters: One cluster has message-driven beans bound to the *DevelopmentQueue*, and the other cluster has message-driven beans bound to the *ProductionQueue*. The logic for each of these beans can vary based upon the needs of the system. For example, the behavior of the message-driven beans bound to the *DevelopmentQueue* can mimic those bound to the *ProductionQueue* but add on debugging statements. You can also tune each cluster independently based upon load to the system. Since the *ProductionQueue* will likely have more throughput than the *DevelopmentQueue*, you could independently grow the size of the cluster servicing the *ProductionQueue* without impacting the cluster servicing the *DevelopmentQueue*.

This illustrates a general paradigm of using queues to partition business logic processing. Rather than the servers pulling messages off a single queue, you pre-choose which machines get the messages by splitting the queue into two queues. This is an artificial way to achieve control load-balancing in a JMS system.

if there is a system crash or a crash from within the EJB container, any active message-driven bean instances are destroyed without going through the proper life cycle shutdown. Additionally, for any method that throws a system exception, such as *EJBException*, the *ejbRemove()* method is not invoked. Developers should be alert to this fact and perform any relevant cleanup before throwing a system exception.

Developers should also be aware that the *ejbRemove()* method is invoked by the container only when the container no longer needs that instance. Many containers pool the necessary number of message-driven bean instances needed to concurrently handle multiple messages. The boundaries on the minimum and maximum size of the message-driven bean pool is typically

set in an application-server specific deployment descriptor. A container adds and removes message-driven bean instances to and from the pool as appropriate. However, since message-driven beans are extremely lightweight objects, a container generally destroys a message-driven bean instance *only* when the EJB itself is being undeployed (the whole EJB component is being undeployed). For most systems, the only time container undeployment occurs is at system shutdown or when an administrator decides to undeploy the component. The important point here is that message-driven bean containers are rarely undeployed and therefore message-driven instances are rarely destroyed. As a general rule of thumb, the `ejbRemove()` method is rarely invoked.

Poison Messages

When using container-managed transactions (see Chapter 10) with a message-driven bean, it is easy to code yourself into a situation that causes the generation of *poison messages*. A poison message is a message that is continually retransmitted by a JMS destination to consumer because the consumer continuously fails to acknowledge the consumption of the message. Any time your message-driven bean does not acknowledge messages to the JMS destination, you have a situation with potential to create poison messages. See Figure 8.7 to see a diagram indicating how poison messages can inadvertently be generated.

For example, suppose you have a stock-quoting message-driven bean that accepts a text message, which represents the stock ticker symbol to be quoted. Your bean cracks open that message. If the string contained within the message matches a stock symbol, the bean retrieves the value of that symbol and

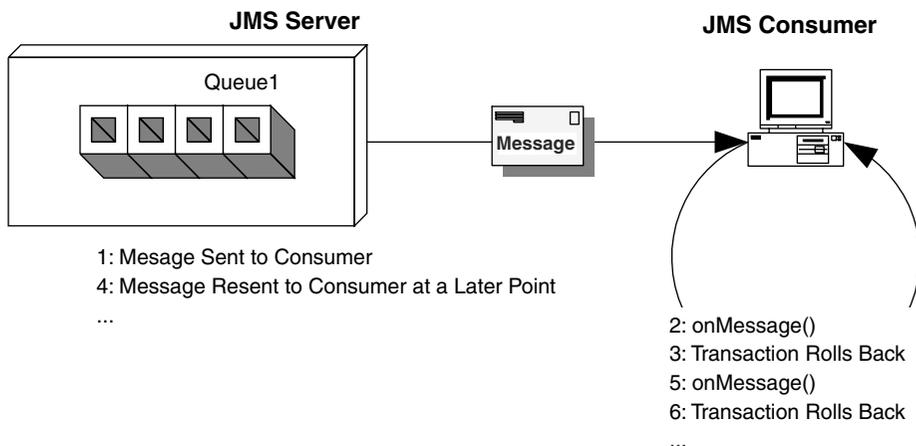


Figure 8.7 How message-driven beans can cause poison messages.

sends a response message. Otherwise, the bean throws a system exception or calls *MessageDrivenContext.setRollbackOnly()*. This causes the transaction to be rolled back, which means the message acknowledgement never to be sent to the JMS destination. The JMS destination eventually resends the same message to the container, causing this same process to occur.

See Source 8.4 for an example of a message-driven bean implementation class that will cause a poison message scenario. Note that our abuse of threading is for illustrative purposes only!

```
package examples;

import javax.ejb.*;
import javax.jms.*;

public class PoisonBean
    implements MessageDrivenBean, MessageListener {

    private MessageDrivenContext ctx;

    public void setMessageDrivenContext(MessageDrivenContext ctx) {
        this.ctx = ctx;
    }

    public void ejbCreate() {}

    public void ejbRemove() {}

    public void onMessage(Message msg) {
        try {
            System.out.println("Received msg " + msg.getJMSMessageID());

            // Let's sleep a little bit so that we don't
            // see rapid fire re-sends of the message.
            Thread.sleep(3000);

            // We could either throw a system exception here or
            // manually force a rollback of the transaction.
            ctx.setRollbackOnly();
        }
        catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Source 8.4 PoisonBean.java.

Several strategies can resolve poison messages:

- Make sure to not throw any system exceptions for any business logic-related error conditions. System exceptions like *EJBException* are intended to indicate system and/or container failure. If this were a session or entity bean, the ideal solution would be to generate an application exception and throw it (especially since application exceptions do not force transactions to be rolled back). However, the EJB specification prohibits application exceptions from being thrown from the *onMessage()* method of a message-driven bean. The ideal solution to this problem would likely involve logging the business error message and then quietly returning.
- Consider using bean-managed transactions instead of container-managed transactions. Message consumption and acknowledgement is not part of the transaction if bean-managed transactions are used. A bean-managed transaction can be rolled back and the message is acknowledged anyway.
- Some application servers allow you to configure a poison message queue. Messages that are redelivered a certain number of times is flagged as poison messages, removed from their primary queue, and placed into a poison message queue. Typically, any message that is redelivered from three to five times can be considered a poison message. You can then bind special consumers or message-driven beans to the poison message queue to handle any unexpected error conditions.
- Some application servers place a retry count value as a property of any redelivered messages. Each redelivery of a message incrementally increases the retry count. Your message-driven bean could check the value of a retry count (if it exists) to see if it has repeatedly consumed the same message.
- Some application server vendors provide a *redelivery delay* feature that administrators can configure to determine how long the JMS destination delays the redelivery of a message after it receives a negative acknowledgement. This way, your system doesn't grind to a halt in case of rapid-fire poison messages.

How to Return Results Back to Message Producers

The EJB specification does not outline any mechanism that allows a message-driven bean to propagate a response back to the client that originally generated the message. So we need to build those facilities ourselves. Figure 8.8 shows how this could be accomplished.

Here is an explanation of Figure 8.8:

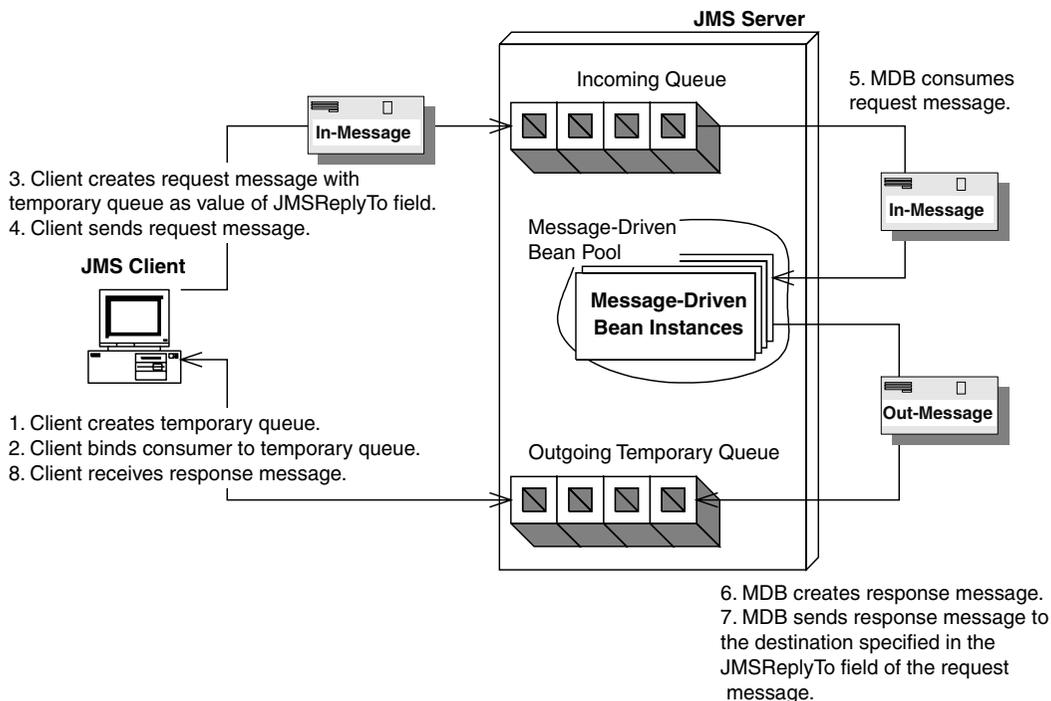


Figure 8.8 A simple request/response paradigm solution.

- The client that generates a JMS message for consumption creates a *temporary destination* associated with its *Connection*. The JMS server temporarily creates a *Topic* or *Queue* and that object exists for the lifetime of the *Connection*.
- The request message that the client sends contains extra information, so the receiving message-driven bean knows how to reply correctly. Specifically, the client sticks the name of the temporary queue in the `JMSReplyTo` header field of the request message. The message-driven bean can harness this field to reply on the correct queue. The client also has a unique identifier of the original message in the `JMSCorrelationID` header field of the original message. When the message-driven bean replies, it embeds this original identifier, so the client knows to which original message he's receiving a reply.
- The client creates a new *Session* and registers a *MessageListener* to consume messages sent to the temporary destination that was just created.
- The client sends the message.

- After consuming the message, the message-driven bean formats a response and sends it using the *JMSReplyTo* and *JMSCorrelationID* attribute of the received message.
- The client's *MessageListener* class asynchronously consumes the message that is sent to the temporary destination, recognizes that it is a response to the original message, and processes it.

Even though this scenario seems like a straightforward solution for responding to clients from within a message-driven bean, it could potentially lead to some unexpected results. The problem arises if the *client itself* is an EJB component, such as a stateful session bean. When your stateful session bean creates the temporary destination, that temporary destination has a lifespan equal to the lifespan of the JMS connection that your bean currently holds. If your bean is *passivated* (meaning swapped out of memory), then you need to release that connection. The temporary destination then goes away, and you've lost all messages delivered to that temporary destination while you were passivated, even if you recreate the destination after you are swapped into memory again.

We propose two possible solutions to this problem:

1. Don't use a stateful session bean. Instead the end client, such as a servlet, application, or JSP tag library (rather than the stateful session bean), creates a temporary queue that all response messages are sent to. The stateful session bean is therefore not holding onto a connection, eliminating any danger of the destination going away because of passivation. See the book's accompanying source code for an implementation of this solution.

The advantages of using this architecture include:

Ease of implementation. Creating temporary queues doesn't require any extra configuration from an administrator, whereas setting up a dedicated response topic requires management on the part of the administrator and your application.

Security. Since temporary queues are bound to a particular connection, malicious clients cannot bind to a temporary queue and intercept response messages.

Immediate client notification. Since the remote client creates and manages the receiving logic for the temporary queue, the client is notified immediately when a response *message* is generated, rather than having to wait for a middleman session bean to respond.

The disadvantages of this architecture include:

No persistent messages. Temporary queues cannot have persistent stores associated with them and therefore cannot support guaranteed message delivery. If the system fails while a response message is located on the temporary queue, the message will be lost.

Poor abstraction. Since temporary queues are associated with a *Connection* object, a stateful session EJB cannot perform middle-tier management of the request/response process. It might be more natural to abstract away the JMS request/response logic from the client.

2. A *permanent* response topic is configured and deployed in the JMS server. All response messages are delivered to the same response topic for all clients. Clients filter out the messages that belong to them by registering a message selector with the JMS server. Any request message that is sent has a custom application property called *ClientName=MyID* where *MyID* varies for each client. The message-driven bean that consumes the request message takes the application property from the request message and inserts the same property in the response message. All response messages are sent to the same response topic irrespective of the client. Figure 8.9 illustrates this scenario, and the book's accompanying source code has its implementation.

The advantages of using this architecture include:

Better fault tolerance. Because this architecture proposes that a permanent topic be set up for all outgoing messages, the response topic could be associated with a persistent store. All outgoing messages could then be sent persistently with guaranteed message delivery. Temporary topics and queues cannot have persistent messages delivered to them. This could be ideal for a data retrieval system. For example, suppose you had a remote client that randomly connected to the central server requesting a download of the latest market data as it pertains to that client. The data could be anywhere from 1K to 1MB. Let's also suppose that for situations where a large amount of data needs to be retrieved for the client, you want to break up the data chunks into 100K messages. If the client needed to retrieve 1MB of data, you would need to send 10 response messages. All of the response messages could be sent with guaranteed message delivery. If the remote client application were to fail during the download process, it could easily resume from the last response message that it received instead of having to restart the entire download process.

Better filtering. You can add on additional filtering of response messages through the message selector that the client registers with the JMS server. In the example provided with this book, the client registers to receive messages that have an application property *ClientName=MyID*. You could

conceivably add on application properties about the response message that the client filters on. These properties could be message size, message importance, and so on.

The main disadvantage of this architecture is lack of security. Since the JMS specification does not have any security restrictions on which clients can bind which message selectors, any client can register any message selector. This presents the opportunity for a malicious client to register for consumption of response messages that are destined for another client. This malicious behavior is not possible with temporary destinations. Of course, if you're secured by a firewall, security probably isn't an issue. Also, it would take a pretty snazzy developer to actually figure out that you're sending messages and register a message listener.

As a final note, it is important to mention that this approach allows a session EJB to act as a mediator between the client and the back-end system, as mentioned in the actual description of the problem. By using an intermediary session EJB, security can be improved, because the topic that response messages are delivered to can be made available only internally by simply not exposing it to a client or blocking the message server using a firewall or

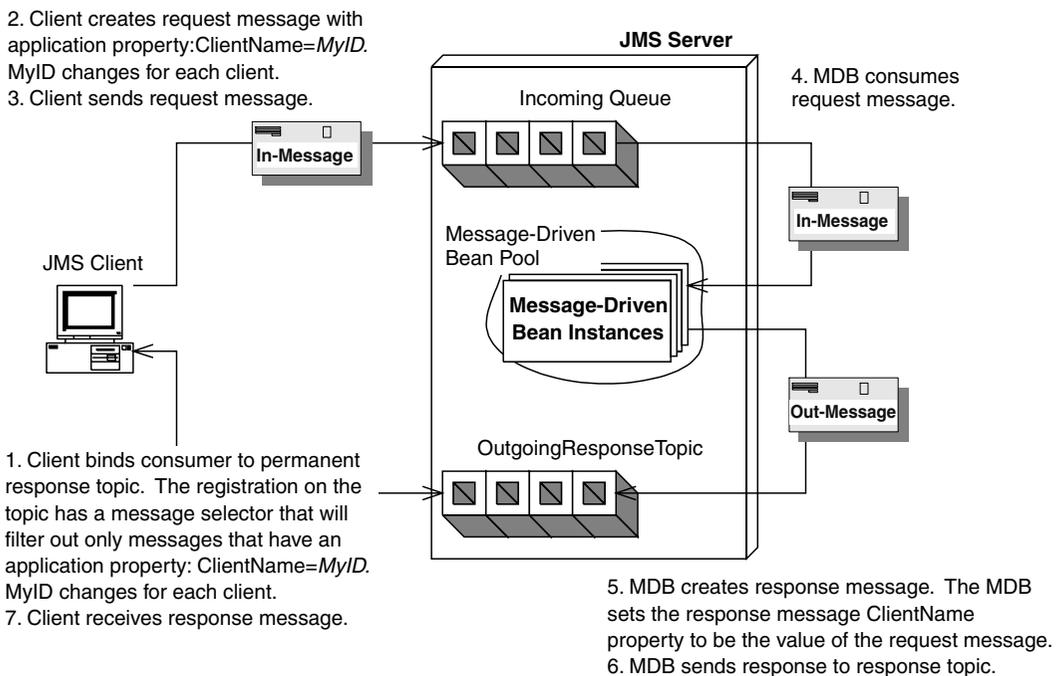


Figure 8.9 Another request/response paradigm solution.

other security measure. The session EJB can be coded to filter out messages based upon the logged-in user name.

An alternative request/response paradigm.

If you don't feel like writing your own request/response code as we've just described, you can tap into JMS's facilities to help you. JMS has two special classes, *javax.jms.QueueRequestor* and *javax.jms.TopicRequestor*, that implement a simple request/response paradigm. You call a method called *request()* that takes as input the request message and returns the response message. This is implemented in the book's accompanying source code.

The downsides to this approach are:

You need to block when waiting for a response. You can't continue processing and do other things, which is one of the major advantages of messaging in the first place.

You can't use transactions. If you did, the outgoing message would be *buffered* until the transaction committed. Since the *QueueRequestor* class doesn't commit right away, but instead blocks until it receives a response message, it will block indefinitely. The outgoing request message will wait forever to be flushed from the buffer. See Chapter 10 for more on transactions.

The Future: Asynchronous Method Invocations

One of the downsides to message-driven beans is that you need to learn a whole new API, JMS, to call them. This API is highly procedural in nature, because you are not invoking lots of different business methods on your message-driven bean; rather, you are sending messages using the JMS API, and the server has a single method to crack the message open and then call the intended method using a giant *if* statement.

An *asynchronous method invocation* is a real method invocation executed in an asynchronous fashion. You are actually calling business methods on the server, such as *logMessage()* or *quoteStock()*. You can choose whether you want to block and wait for an asynchronous response or to immediately return and to not wait for a response. Furthermore, the server can take on the context information of the client.

Asynchronous RMI and Microsoft's Queued Components are asynchronous method invocation infrastructures. CORBA also has some support for this,

with a slightly different definition of deferred synchronous invocations: “A request where the client does not wait for completion of the request, but does intend to accept results later.”

We hope a future EJB specification supports asynchronous method invocations. Until then, you’ll have to build such facilities on top of JMS yourself, perhaps by writing a code generator.

Summary

In this chapter, we’ve learned about developing message-driven beans and the pitfalls associated with doing asynchronous development with EJBs. We started by learning about the various benefits of developing asynchronous components and how message-driven beans compare to their session and entity bean counterparts. We looked at how to build a message-driven bean and deploy it. Next we looked at how a message-driven bean behaves in its environment, including how it interacts with transactions. Finally, we took a look at the common pitfalls of using message-driven beans and proposed some solutions.

Adding Functionality to Your Beans

In previous chapters, you learned the fundamentals of EJB programming. In this chapter, we'll build on that knowledge and cover a slew of essential topics, including:

- How to call beans from other beans
- How to use environment properties to customize your beans and access those environment properties at runtime
- How to access resource factories (such as JDBC or JMS drivers) from your bean
- How to use the EJB security model
- How to use EJB object handles and EJB home handles

This knowledge is key for building nontrivial EJB deployments. So let's get to it!

Calling Beans from Other Beans

Any nontrivial EJB object model has *layers* of beans calling other beans. For example, a bank teller bean might call a bank account bean, or a customer bean might call a credit card bean. In this chapter, we'll use the example of:

- **A pricing engine** that computes prices of products, using all sorts of interesting rules, such as discounts, taxes, and shipping costs.

- **A catalog engine** that is a catalog for products, retrieving products from the database as necessary.

The pricing engine calls the catalog engine. For simplicity, we'll assume that both of these beans are stateless session beans, since that's what you've learned so far.

Default JNDI Lookups

For your bean to call another bean, you must go through the same process that any other client would go through. Your bean might:

1. Look up the other bean's home object via JNDI
2. Call `create()` on the home object
3. Call business methods on the EJB object
4. Call `remove()` on the EJB object

As we learned about earlier, to lookup a home via JNDI, you first need to supply *JNDI initialization parameters*, such as the JNDI driver you're using, which differs from container to container. But if you're writing a bean that calls another bean, how do you know what JNDI service provider to use? After all, your beans should be container-independent. Hard-coding that JNDI information into your bean would destroy portability.

The good news is that if you're looking up a bean from another bean, you don't need to supply *any* JNDI initialization parameters. You simply acquire a *default* JNDI initial context. The container sets the default JNDI initial context before your bean ever runs. For example, the following code snippet is taken from a bean calling another bean:

```
// Obtain the DEFAULT JNDI initial context by calling the
// no-argument constructor
Context ctx = new InitialContext();

// Look up the home interface
Object result = ctx.lookup("java:comp/env/ejb/CatalogHome");

// Convert the result to the proper type, RMI-IIOP style
CatalogHome home = (CatalogHome)
    javax.rmi.PortableRemoteObject.narrow(
        result, CatalogHome.class);

// Create a bean
Catalog c = home.create(...);
```

The preceding code is portable because nobody ever needs to supply container-specific JNDI initialization parameters.

Understanding EJB References

Notice from the previous section that we looked up a bean in `java:comp/env/ejb`. This is the JNDI location that the EJB specification recommends (but does not require) you put beans that are referenced from other beans.

Unfortunately, you cannot guarantee that the JNDI location you've specified will be available. This could happen if your bean has a conflict with another bean or if the deployer has a funky JNDI tree that is spread out across multiple domain boundaries.

Thus, your code will break if the JNDI location changes at deployment time. And often, the deployer is unable to modify your code, because it comes to him as `.class` files only. This could happen for example, if you are an independent software vendor that ships beans, and you want to protect your intellectual property and make future upgrades easier by preventing customers from seeing source code.

EJB resolves this situation with *EJB references*. An EJB reference is a *nickname* for the JNDI location that you want to lookup a bean. This nickname may not correspond to the actual JNDI location the deployer sticks your bean into. Your code looks up a home via its nickname, and the deployer then binds that nickname to the JNDI location of his choice, perhaps using symbolic links (an advanced JNDI feature not covered in this book—see the JNDI specification for more). Once again, a layer of indirection solves every problem in computer science.

EJB references are declared in the deployment descriptor. Source 9.1 illustrates references.

Programming with EJB references is straightforward. Our pricer bean is using a catalog bean, so inside the pricer bean we simply list all the necessary information about the catalog bean in an EJB reference. The deployer then knows that our pricer bean uses exactly one other enterprise bean—catalog—and no other. This is useful, because the deployer now knows which class files pricer depends on and what JNDI location needs to be bound. Similarly, the container's tools can easily inspect the deployment descriptor and verify that the deployer has done his job.

Note that while this example declares the catalog bean within our deployment descriptor, we didn't have to do this. The catalog bean could have been in its own Ejb-jar file with its own deployment descriptor.

```

. . .
<enterprise-beans>

    <!--
    Here, we define our Catalog bean.  Notice we use the
    "Catalog" ejb-name.  We will use this below.
    -->
    <session>
        <ejb-name>Catalog</ejb-name>
        <home>examples.CatalogHome</home>
        . . .
    </session>

    <session>

        <ejb-name>Pricer</ejb-name>
        <home>examples.PricerHome</home>
        . . .

        <ejb-ref>
            <description>
This EJB reference says that the Pricing Engine
session bean (Pricer) uses the Catalog Engine
session bean (Catalog)
            </description>

            <!--
            The nickname that Pricer uses to look
            up Catalog.  We declare it so the deployer
            knows to bind the Catalog home in
            java:comp/env/ejb/CatalogHome.  This may not
            correspond to the actual location to which the
            deployer binds the object via the container
            tools.  The deployer may set up some kind of
            symbolic link to have the nickname point to the
            real JNDI location.
            -->
            <ejb-ref-name>ejb/CatalogHome</ejb-ref-name>

            <!-- Catalog is a Session bean -->
            <ejb-ref-type>Session</ejb-ref-type>

            <!-- The Catalog home interface class -->
            <home>examples.CatalogHome</home>

            <!-- The Catalog remote interface class -->

```

Source 9.1 Declaring an EJB reference.

```
<remote>examples.Catalog</remote>

<!-- (Optional) the Catalog ejb-name -->
<ejb-link>Catalog</ejb-link>
</ejb-ref>
</session>

</enterprise-beans>
. . .
```

Source 9.1 Declaring an EJB reference (*continued*).

You can also access EJB components from other EJB components through their local interfaces rather than their remote interfaces. To do this, our deployment descriptor would be almost exactly the same—except instead of calling the element `<ejb-ref>` we would call it `<ejb-local-ref>`, instead of `<home>` we would use `<local-home>`, and instead of `<remote>` we would use `<local>`. The JNDI code to lookup the bean would change as well; it would lookup the local home interface rather than the home interface, and call the local interface rather than the remote interface:

```
// Obtain the DEFAULT JNDI initial context by calling the
// no-argument constructor
Context ctx = new InitialContext();

// Look up the home interface
Object result = ctx.lookup("java:comp/env/ejb/CatalogLocalHome");

// Convert the result to the proper type. No RMI-IIOP cast
// required since local interfaces are being used.
CatalogLocalHome home = (CatalogLocalHome) result;

// Create a bean
CatalogLocal c = home.create(...);
```

Resource Factories

Our next topic is how to perform callouts to external resources from an EJB component. A *resource factory* is a provider of resources. Examples include a Java Database Connectivity (JDBC) driver, a Java Message Service (JMS) driver, or a J2EE Connector Architecture (JCA) resource adapter. A resource factory is the driver that gives you connections, such as a JDBC driver giving you a database connection.

Connection Pooling

Connection pooling is the reuse of sockets. If a client isn't using a socket, a different client can harness the socket. This increases the scalability of a system. Connection pooling is built into most containers. JDBC specifies standard interfaces for connection pooling, further enhancing your code portability. The connection pooling typically happens completely behind the scenes, and your bean code is oblivious to it.

To begin using a resource factory, you need to locate it. EJB mandates that you use JNDI to look up a resource factory. This is very nice, because you merely need to learn a single API—JNDI—and you can lookup JDBC drivers, JMS drivers, JCA drivers, and so on. In fact, you already know how to perform this lookup. It's the same JNDI code as looking up an EJB home object:

```
// Obtain the initial JNDI context
Context initCtx = new InitialContext();

// Perform JNDI lookup to obtain resource factory
javax.sql.DataSource ds = (javax.sql.DataSource)
    initCtx.lookup("java:comp/env/jdbc/ejbPool");
```

Notice that we're using *java:comp/env/jdbc*. While this is the EJB-suggested location for your JDBC resources, you must specify your resource factory's JNDI location in the deployment descriptor. When your bean is deployed, the deployer binds a real resource factory to that JNDI location. The corresponding deployment descriptor is shown in Source 9.2.

Source 9.2 is fairly self-explanatory, except for the *res-auth* entry. To understand it, realize that when you acquire a connection to a database or other resource, that resource may require authorization. For example, you may need to specify a username and password when obtaining a JDBC connection. EJB gives you two choices for authenticating yourself to a resource:

Perform the authentication yourself in the bean code. Call the resource factory with the appropriate sign-on information, such as a login name and password. In this case, set the deployment descriptor's *res-auth* element to *Application*.

Let the deployer handle authentication for you. The deployer specifies all sign-on information in the deployment descriptor. In this case, set the deployment descriptor's *res-auth* element to *Container*.

```
. . .
<enterprise-beans>

  <session>

    <ejb-name>Catalog</ejb-name>
    <home>examples.CatalogHome</home>
    . . .

    <!--
    This element indicates a resource factory reference
    -->
    <resource-ref>

      <description>
      This is a reference to a JDBC driver used within
      the Catalog bean.
      </description>

      <!--
      The JNDI location that Catalog uses to look up
      the JDBC driver.
      We declare it so the deployer knows to bind the
      JDBC driver in java:comp/env/jdbc/ejbPool.
      -->
      <res-ref-name>jdbc/ejbPool</res-ref-name>

      <!--
      The resource factory class
      -->
      <res-type>javax.sql.DataSource</res-type>

      <!--
      Security for accessing the resource factory.
      Can either be "Container" or "Application".
      -->
      <res-auth>Container</res-auth>

      <!--
      Whether connections should be shared with other
      clients in the different transactions
      -->
      <res-sharing-scope>Sharable</res-sharing-scope>
    </resource-ref>

  </session>
</enterprise-beans>

. . .
```

Source 9.2 Declaring a resource factory reference within a deployment descriptor.

The second choice is the most useful, especially when you are writing beans for resale or reuse by other companies, because only the deployer will know what sign-on credentials are needed to access a particular resource.

Environment Properties

Our next tidbit of essential EJB knowledge is how to *customize* our beans at runtime. What does customization mean? Well, our pricing bean might have several different pricing algorithms it could apply. We'd like the consumers of our bean to be able to select their preferred algorithm.

Your bean's *environment properties* are application-specific properties that your beans read in at runtime. These properties can be used to customize your bean and make your beans data-driven. It's a quick-and-dirty alternative to storing information in a database.

The first step to using environment properties is to declare them in the deployment descriptor. The container reads in this deployment descriptor and makes the environment properties available for your bean to access at runtime. An example is shown in Source 9.3.

The environment property declared in Source 9.3 tells our pricing engine to use an algorithm that gives all customers no taxes, due to the Internet tax moratorium that we all love.

You use JNDI to access the environment from your bean. The following code illustrates this.

```
// 1: Acquire the initial context
Context initCtx = new InitialContext();

// 2: Use the initial context to look up
//     the environment properties
String taxAlgorithm = (String)
initCtx.lookup("java:comp/env/Pricer/algorithm");

// 3: Do what you want with the properties
if (!taxAlgorithm.equals("NoTaxes")) {
    // add tax
}
```

Notice that we lookup environment properties under the JNDI name *java:comp/env*. All EJB environment properties *must* be somewhere beneath this naming context.

```
. . .
<enterprise-beans>

  <session>

    <ejb-name>Pricer</ejb-name>
    <home>examples.PricerHome</home>
    . . .

    <!--
    This element contains a single environment property.
    The property is only accessible from the Pricer.
    -->
    <env-entry>

      <description>
      The algorithm for this pricing engine.
      </description>

      <!--
      The JNDI location that Pricer uses to look up
      the environment property. We declare it so the
      container knows to bind the property in
      java:comp/env/PricerProperties/algorithm.
      -->
      <env-entry-name>Pricer/algorithm</env-entry-name>

      <!-- The type for this environment property -->
      <env-entry-type>java.lang.String</env-entry-type>

      <!-- The environment property value -->
      <env-entry-value>NoTaxes</env-entry-value>
    </env-entry>
  </session>
</enterprise-beans>
. . .
```

Source 9.3 Declaring environment properties within an EJB deployment descriptor.

Understanding EJB Security

The next topic is adding security to your enterprise beans. So let's get right down to the meat: There are two security measures that clients must pass when you add security to an EJB system.

First, the client must be *authenticated*. Authentication verifies that the client is who he claims to be. For instance, the client may enter a username/password in an application or Web browser, and those credentials are checked against a permanent client profile stored in a database or LDAP server. Once the client is authenticated, he is associated with a *security identity* for the remainder of his session.

Then the client must be *authorized*. Once the client has been authenticated, he must have permission to perform desired operations. For example, in a procurement application, you want to ensure that while anyone can submit purchase orders, only supervisors can approve purchase orders.

There is an important difference here—*authentication* verifies that the client is who he claims to be, whereas *authorization* checks to see if an already authenticated client is allowed to perform a task. Authentication must be performed sometime before an EJB method is called. If the client has an identity, then it has been authenticated. Authorization, on the other hand, occurs during an EJB method call.

Security Step 1: Authentication

In earlier versions of EJB (1.0 and 1.1), there was no portable way to achieve authentication. The specific way your client code became associated with a security identity was left to the discretion of your application and your EJB container. This meant each EJB container may handle authentication differently.

The good news is that in EJB 2.0, authentication is now portable and robust. You perform authentication through the *Java Authentication and Authorization Service* (JAAS), a separate J2EE API. Let's now take a minitutorial of JAAS and see how it can be used in an EJB environment.

JAAS Overview

JAAS is a portable interface that enables you to authenticate and authorize users in Java. In a nutshell, it allows you to log into a system without knowing about the underlying security system being used. Behind the scenes in JAAS, the implementation (such as an application server) then determines if your credentials are authentic.

The power of JAAS lies in its ability to use almost any underlying security system. Some application servers allow you to set up usernames and passwords in the application server's properties, which the application server reads in at runtime. More advanced servers support complex integration with existing security systems, such as a list of usernames and passwords stored in an LDAP

server, database, or custom security system. Other systems support certificate-based authentication. Regardless, the integration should be performed behind the scenes by your container and should not affect your application code.

There are two likely candidate scenarios when you may want to use JAAS from your code, shown in Figure 9.1.

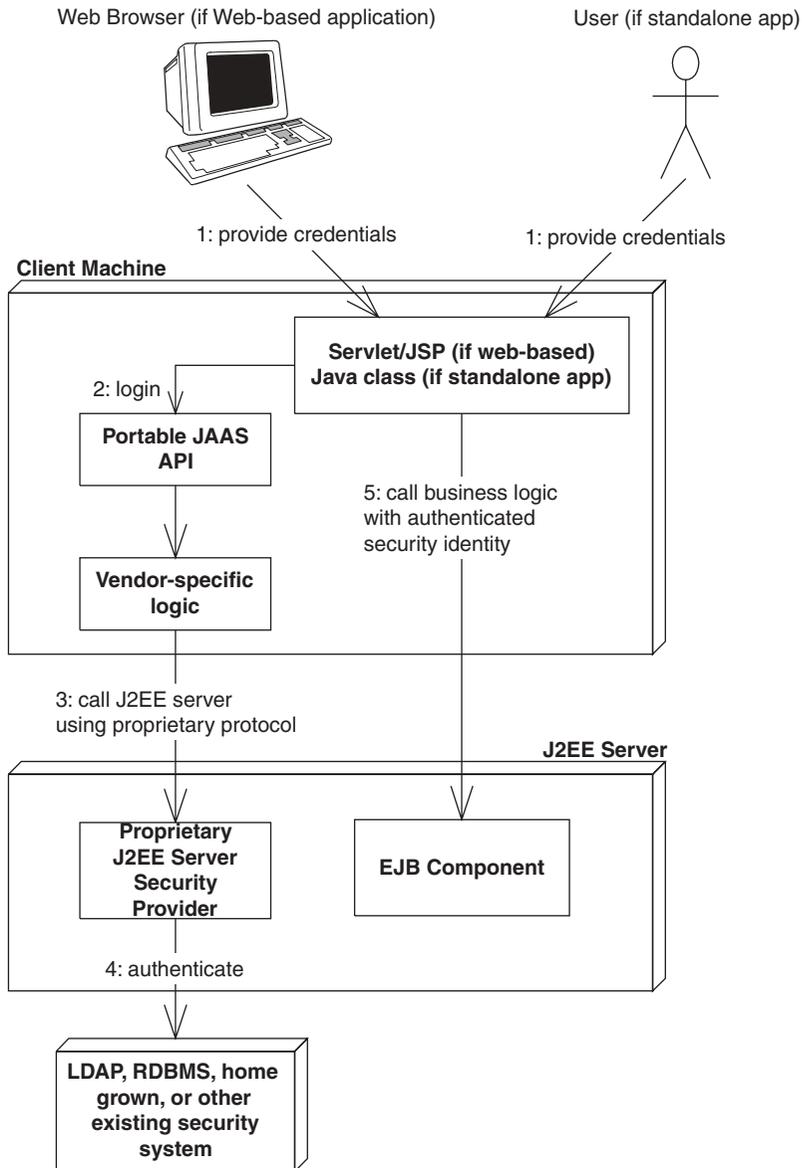


Figure 9.1 JAAS overview.

1. When you have a standalone application connecting to a remote EJB system, the user would supply credentials to the application (or perhaps the application would retrieve the credentials from a file or other system). The standalone application would then use the JAAS API to authenticate the user prior to calling the EJB components residing within the application server. The application server would verify the user's credentials. Once the user has been authenticated via JAAS, the client can call EJB methods securely, and the user's security identity will be propagated to the server upon method invocations.
2. When you have a Web browser client connecting to a servlet/JSP layer, the web browser user supplies credentials to a servlet/JSP layer, and the servlet/JSP layer could use JAAS to authenticate the user. The Web browser could supply the credentials in one of four ways:

Basic authentication. The Web client supplies a username and password to the Web server. The server checks these credentials against a permanent storage of usernames and passwords. Note that while this is a simple approach, this is not a secure form of authentication because the password is sent in clear-text to the server. Some J2EE servers allow you to use secure socket layer (SSL) to encrypt this data.

Form-based authentication. This is just like basic authentication, except the application uses a customizable form, such as a special login screen.

Digest authentication. The Web client supplies a special message digest to the Web server. This message digest is a mathematical transformation on both the user's password and the HTTP message itself. The password itself is not sent to the Web server. The Web server then tries to reproduce the message digest by performing the same mathematical transformation, except this time the Web server uses a secure copy of the user's password kept in permanent storage. If the digests match, the user is authenticated.

Certificate authentication. The client can establish an identity with X.509 certificates. The client can also (optionally) ensure that a third party is not impersonating the server by receiving X.509 certificates that authenticate the server.

As with standalone applications, once the user has been authenticated via JAAS, the client can call EJB methods securely, and the user's security identity will be propagated to the server upon method invocations.



The JAAS ideas and JAAS code we're about to present are useful for both Web clients (servlets and JSPs) as well as standalone clients. However, if you're developing a Web-based system, it behooves you to learn more about the four methods of Web

authentication. We recommend that you take a look at the *J2EE BluePrints* document, as well as the *J2EE Platform Specification*, both freely downloadable from <http://java.sun.com>. Note that, unfortunately, J2EE does not require that all servers support security interoperability between the Web layer and the EJB layer. Check your J2EE server's documentation before embarking down this path.

The JAAS Architecture

Sun has built JAAS to be a very robust. It has a powerful design, and is surprisingly complicated for what you think would be a simple subject. We have distilled JAAS down into a simple procedure to make it easier for you to understand.

Figure 9.2 shows the basics of a JAAS authentication procedure.

The JAAS authentication procedure breaks down as follows. Follow along with the picture as we review each step:

1. The client instantiates a new login context. This is a container-provided class (you don't write it). It's responsible for coordinating the authentication process.

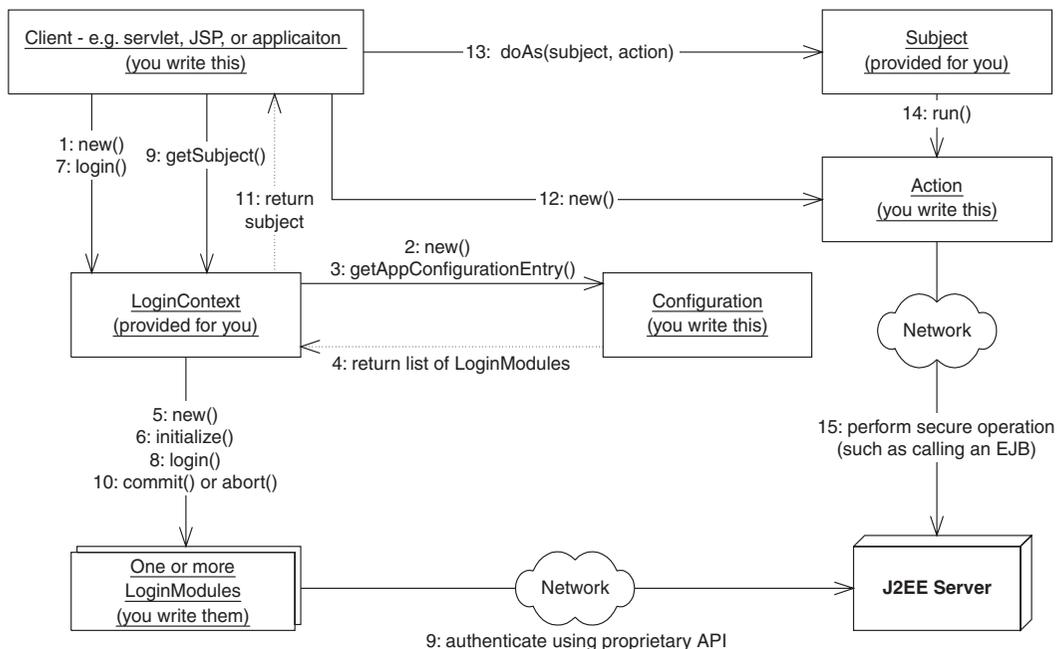


Figure 9.2 JAAS authentication in detail.

2. The login context instantiates a new configuration object, which you must have written beforehand. This configuration object knows about the type of authentication you want to achieve. For example, your configuration object might know that you want to perform both password-based authentication and certificate-based authentication.
3. The login context asks the configuration object for the list of authentication mechanisms that we're going to use (such as password-based and certificate-based).
4. The configuration object returns a list of authentication mechanisms. Each one is called a login module. A login module knows how to contact a specific security provider and authenticate in some proprietary way.
5. The login context instantiates your login modules. You can have many login modules if you want to authenticate across several different security providers. In the example we're about to show, we will use only one login module, and it will know how to authenticate using a username/password to a J2EE server.
6. The login context initializes the login modules.
7. The client code tries to log in by calling the *login()* method on the login context.
8. The login context delegates the *login()* call to the login modules, since only the login modules know how to perform the actual authentication.
9. The login modules (written by you) authenticate you using a proprietary means. In the example we're about to show, our username/password login module will try to contact a J2EE server and tell the J2EE server our username and password. That J2EE server will verify the credentials against a permanent record of usernames and passwords, such as ones stored in a flat file, database, or LDAP server.
10. If the logins succeed, then the login modules are told to *commit()*. They can also *abort()* if the login process fails. This is not a very critical step to understand—read the JAAS docs if you're curious to understand more.
11. A new subject is returned to the client code. This subject represents someone (or something) that has been authenticated. You can use this subject to perform secure operations.
12. Your client code instantiates a new action. An action is an object that you write. It knows how to perform an operation you want to perform securely, such as calling an EJB component, using a database, and so on.
13. You tell the subject to *do* the action *as* the subject—hence the *doAs()* method name.
14. The subject calls the action's *run()* method.

15. The action performs its operation (such as calling an EJB component) and the logged-in security context is automatically propagated along with the method call. This completes authentication. Since the security context is sent to the server, the server can now perform authorization.

That's it—a mere 15 steps. You've got to hand it to Sun for being elegant.



What's neat about JAAS is that the login modules are separate from the configuration, which means you can chain together different login modules in interesting combinations by using different configuration objects. You can choose the configuration class that you want typically via system properties, although this may vary from container to container. You can also theoretically make an entry called `login.configuration.provider=<class name of your configuration object>` in the `java.security` file located within your J2SE SDK install folder, but this did not work in testing at the time of this writing.

JAAS Sample Code

Let's show a simple JAAS example. The code will authenticate and then call a "Hello, World" method on a bean. If the password is right, then the invocation succeeds. If not, then an exception is thrown while trying to log in.

The code is in Source 9.4 through 9.7, and is fairly self-documenting. By reviewing Figure 9.2, this sample code, and the process we laid out earlier, you should be able to get a feeling for what this code is doing.

```
package examples;

import javax.naming.*;
import javax.security.auth.*;
import javax.security.auth.callback.*;
import javax.security.auth.login.*;
import javax.rmi.PortableRemoteObject;

public class HelloClient {

    public static void main(String[] args) throws Exception {

        /*
         * Authenticate via JAAS
         */
    }
}
```

Source 9.4 HelloClient.java.

```
LoginContext loginContext = new LoginContext("Hello Client");
loginContext.login();

/*
 * Retrieve the logged-in subject
 */
Subject subject = loginContext.getSubject();

/*
 * Perform business logic while impersonating the
 * authenticated subject
 */
CallHelloWorld action = new CallHelloWorld();
String result = (String) Subject.doAs(subject, action);

/*
 * Print the return result from the business logic
 */
System.out.println(result);
}
}
```

Source 9.4 HelloClient.java (*continued*).

```
package examples;

import java.util.Hashtable;
import javax.security.auth.login.*;

/**
 * Sample configuration class for JAAS user authentication.
 * This class is useful because it can be rewritten to use
 * different login modules without affecting client code.
 *
 * For example, we could have a login module that did
 * username/password authentication, and another that did
 * public/private key certificate authentication.
 */
public class PasswordConfig extends Configuration {

/**
 * A configuration class must have a no-argument constructor
 */
}
```

Source 9.5 PasswordConfig.java.

```
public PasswordConfig() {}

/**
 * This method chooses the proper login module.
 */
public AppConfigurationEntry[]
    getAppConfigurationEntry(String applicationName)
{
    /**
     * Return the one login module we've written, which uses
     * username/password authentication.
     *
     * - The "REQUIRED" flag says that we require that this
     * login module succeed for authentication.
     * - The new hashtable is a hashtable of options that
     * our login module will receive. For example, we might
     * define an option that turns debugging on. Our login
     * module would inspect this hashtable and start logging
     * output.
     */
    AppConfigurationEntry[] loginModules
        = new AppConfigurationEntry[1];
    loginModules[0] = new AppConfigurationEntry(
        "examples.PasswordLoginModule",
        AppConfigurationEntry.LoginModuleControlFlag.REQUIRED,
        new Hashtable());
    return loginModules;
}

/**
 * Refresh and reload the Configuration object by reading
 * all of the login configurations again.
 */
public void refresh() {}
}
```

Source 9.5 PasswordConfig.java (*continued*).

```
package examples;

import java.util.*;
import javax.naming.Context;
import javax.security.auth.*;
import javax.security.auth.callback.*;
```

Source 9.6 PasswordLoginModule.java.

```
import javax.security.auth.login.*;
import javax.security.auth.spi.*;

/**
 * Sample login module that performs password authentication.
 *
 * The purpose of this class is to actually go out and perform
 * the authentication.
 */
public class PasswordLoginModule implements LoginModule {
    private Subject subject = null;

    /**
     * Initializes us. We set ourselves to the particular
     * subject which we will later authenticate.
     */
    public void initialize(Subject subject,
                          CallbackHandler callbackHandler,
                          Map sharedState,
                          Map options)
    {
        this.subject = subject;
    }

    /**
     * This method authenticates the user. It is called when
     * the client tries to login in.
     *
     * Our method implementation contains the vendor-specific way
     * to access our permanent storage of usernames and passwords.
     *
     * Note that while this code is not portable, it is 100%
     * hidden from your application code behind the LoginModule.
     * The intention is that you develop a different LoginModule
     * for each J2EE server.
     *
     * In this case, BEA has provided us with a helper class that
     * talks JNDI to the Weblogic server, and the server then goes
     * to whatever the currently configured security realm is,
     * such as a file, RDBMS, or LDAP server.
     */
    public boolean login() throws LoginException
    {
        try {
            /**
             * Authenticate the user's credentials, populating Subject
             */
        }
    }
}
```

Source 9.6 PasswordLoginModule.java (*continued*).

```
* Note: In a real application, we would not hardcode the
* username and password. Rather, we would write a reusable
* LoginModule that would work with any username and password.
* We would then write a special callback handler that knows
* how to interact with the user, such as prompting the user
* for a password. We would then call that callback handler
* here.
*/
weblogic.jndi.Environment env =
    new weblogic.jndi.Environment(System.getProperties());
env.setSecurityPrincipal("guest");
env.setSecurityCredentials("guest");

weblogic.security.auth.Authenticate.authenticate(
    env, subject);

/*
 * Return that we have successfully authenticated
 * the subject
 */
return true;
}
catch (Exception e) {
    throw new LoginException(e.toString());
}
}

/**
 * This method is called if the overall authentication
 * succeeded (even if this particular login module
 * failed). This could happen if there are other login
 * modules involved with the authentication process.
 *
 * This is our chance to perform additional operations,
 * but since we are so simple, we don't do anything.
 *
 * @return true if this method executes properly
 */
public boolean commit() throws LoginException {
    return true;
}

/**
 * This method is called if the overall authentication
 * failed (even if this particular login module
 * succeeded). This could happen if there are other
 * login modules involved with the authentication
```

Source 9.6 PasswordLoginModule.java (*continued*).

```
* process.  
*  
* This is our chance to perform additional operations,  
* but since we are so simple, we don't do anything.  
*  
* @return true if this method executes properly  
*/  
public boolean abort() throws LoginException {  
    return true;  
}  
  
/**  
 * Logout the user.  
 *  
 * @return true if this method executes properly  
 */  
public boolean logout() throws LoginException {  
    return true;  
}  
}
```

Source 9.6 PasswordLoginModule.java (*continued*).

```
package examples;  
  
import java.security.*;  
import javax.naming.*;  
import java.util.Hashtable;  
import javax.rmi.PortableRemoteObject;  
  
/**  
 * This is a helper class that knows how to call a  
 * "Hello, World!" bean. It does so in a secure manner,  
 * automatically propagating the logged in security context  
 * to the J2EE server.  
 */  
public class CallHelloWorld implements PrivilegedAction {  
  
    /*  
     * This is our one business method. It performs an action  
     * securely, and returns application-specific results.  
     */  
    public Object run() {
```

Source 9.7 CallHelloWorld.java.

```
String result = "Error";
try {
    /*
     * Make a bean
     */
    Context ctx = new InitialContext(System.getProperties());
    Object obj = ctx.lookup("HelloHome");
    HelloHome home = (HelloHome)
        PortableRemoteObject.narrow(obj, HelloHome.class);
    Hello hello = home.create();

    /*
     * Call a business method, propagating the security context
     */
    result = hello.hello();
}
catch (Exception e) {
    e.printStackTrace();
}

/*
 * Return the result to the client
 */
return result;
}
```

Source 9.7 CallHelloWorld.java (*continued*).

Security Step 2: Authorization

Once the client has been authenticated, it must pass an authorization test to call methods on your beans. You enforce authorization by defining *security policies* for your beans. There are two ways to perform authorization with EJB:

With *programmatically authorization*, you hard-code security checks into your bean code. Your business logic is interlaced with security checks.

With *declarative authorization*, the container performs all authorization checks for you. You declare how you'd like authorization to be achieved through the deployment descriptor, and the container *generates* all necessary security checks. You are effectively delegating authorization to the EJB container.

Security Roles

Regardless of whether you're performing programmatic or declarative authorization, you need to understand the concept of *security roles*. A security role is a collection of client identities. For a client to be authorized to perform an operation, its security identity must be in the correct security role for that operation. The EJB deployer is responsible for associating the identities with the correct security roles *after* you write your beans.

The advantage to using security roles is you do not hard-code specific identities into your beans. This is necessary when you are developing beans for deployment in a wide variety of security environments, because each environment will have its own list of identities. This also allows you to modify access control without recompiling your bean code.

Specifying security roles in EJB is application server-specific but should not affect portability of your code. Table 9.1 shows some sample mappings.

Performing Programmatic Authorization

Let's see how to authorize programmatically. Then we'll see how to authorize declaratively and compare the two approaches.

Step 1: Write the Programmatic Security Logic

To perform explicit security authorization checks in your enterprise beans, you must first get information about who is calling your bean's method. You can get this information by querying the container through the EJB context object. We first learned about the EJB context in Chapter 3; feel free to refer back if you need to.

The EJB context object has the following relevant security methods:

```
public interface javax.ejb.EJBContext
{
    ...
    public java.security.Principal getCallerPrincipal();
    public boolean isCallerInRole(String roleName);
    ...
}
```

Table 9.1 Sample Security Roles

SECURITY ROLE	VALID IDENTITIES
employees	EmployeeA, EmployeeB
managers	ManagerA
administrators	AdminA

isCallerInRole(String role) checks whether the current caller is in a particular security role. When you call this method, you pass the security role that you want the caller compared against. For example:

```
public class EmployeeManagementBean implements SessionBean {

    private SessionContext ctx;

    ...

    public void modifyEmployee(String employeeID)
    throws SecurityException {
        /*
         * If the caller is not in the 'administrators'
         * security role, throw an exception.
         */
        if (!ctx.isCallerInRole("administrators")) {
            throw new SecurityException( . . . );
        }

        // else, allow the administrator to modify the
        // employee records
        // . . .
    }
}
```

The preceding code demonstrates how to perform different actions based on the security role of the client. Only if the caller is in the *administrators* role (defined in Table 9.1, and setup using your container's tools) does the caller have administrator access.

The other programmatic security method, *getCallerPrincipal()*, retrieves the current caller's security principal. You can use that principal for many purposes, such as using the caller's distinguished name in a database query. This might be handy if you're storing your security information in a database. Here is sample code that uses *getCallerPrincipal()*:

```
import java.security.Principal;

...

public class EmployeeManagementBean implements SessionBean {

    private SessionContext ctx;

    ...

    public void modifyEmployee() {
        Principal id = ctx.getCallerIdentity();
        String name = id.getName();
        // Query a database based on the name
    }
}
```

```

        // to determine if the user is authorized
    }
}

```

Step 2: Declare the Abstract Security Roles Your Bean Uses

Next you must declare all the security roles that your bean code uses, such as an *administrators* role, in your deployment descriptor. This signals to others (like application assemblers and deployers) that your bean makes the security check *isCallerInRole (administrators)*. That is important information for the deployer, because the deployer needs to fulfill that role, just like the deployer fulfills EJB references, as mentioned earlier. Source 9.8 demonstrates this.

```

. . .
<enterprise-beans>

    <session>

        <ejb-name>EmployeeManagement</ejb-name>
        <home>examples.EmployeeManagementHome</home>
        . . .

        <!--
        This declares that our bean code relies on
        the administrators role; we must declare it here
        to inform the application assembler and deployer.
        -->
        <security-role-ref>

            <description>
            This security role should be assigned to the
            administrators who are responsible for
            modifying employees.
            </description>

            <role-name>administrators</role-name>

        </security-role-ref>

    </session>

    . . .

</enterprise-beans>
. . .

```

Source 9.8 Declaring a Bean's required security roles.

Step 3: Map Abstract Roles to Actual Roles

Once you've written your bean, you can ship it for resale, build it into an application, or make it part of your company's internal library of beans. The consumer of your bean might be combining beans from all sorts of sources, and each source may have declared security roles a bit differently. For example, we used the string *administrators* in our bean above, but another bean

```
. . .
<enterprise-beans>

  <session>

    <ejb-name>EmployeeManagement</ejb-name>
    <home>examples.EmployeeManagementHome</home>
    . . .

    <security-role-ref>

      <description>
        This security role should be assigned to the
        administrators who are responsible for
        modifying employees.
      </description>

      <role-name>administrators</role-name>
      <!--
        Here we link what we call "administrators" above, to
        a real security-role, called "admins", defined below
      -->
      <role-link>admins</role-link>

    </security-role-ref>

    . . .
  </session>

<assembly-descriptor>

  . . .

  <!--
    This is an example of a real security role.
  -->
  <security-role>
```

Source 9.9 Mapping abstract roles to actual roles.

```
        <description>
        This role is for personnel authorized to perform
        employee administration.
        </description>

        <role-name>admins</role-name>
    </security-role>

    . . .

</assembly-descriptor>

</enterprise-beans>
. . .
```

Source 9.9 Mapping abstract roles to actual roles (*continued*).

provider might use the string *sysadmins* or have completely different security roles. This might be especially true if another developer wrote that bean.

The deployer of your bean is responsible for generating the *real* security roles that the final application will use. Source 9.9 shows this.

Once you've completed your application, you can deploy it in a wide variety of scenarios. For example, if you write a banking application, you could deploy that same application at different branches of that bank, because you haven't hard-coded any specific principals into your application. The deployer of your application is responsible for mapping principals to the roles you've declared. This mapping is called a *security policy descriptor*, a fancy term for the statement, "Every container handles mapping roles to principals differently." The bottom line: Your deployer looks at your security roles and assigns principals to them using proprietary container APIs and tools.

Performing Declarative Authorization

Now that we've seen programmatic authorization, let's move on to declarative authorization. The primary difference between the two models is that with declarative authorization, you *declare* your bean's authorization requirements in your deployment descriptor. The container fulfills these requirements at runtime.

Step 1: Declare Method Permissions

You first need to declare permissions on the bean methods that you want to secure. The container takes these instructions and *generates* security checks in your EJB objects and EJB home objects. Source 9.10 demonstrates this.

```
. . .  
  
<assembly-descriptor>  
  
    . . .  
  
    <!--  
    You can set permissions on the entire bean.  
  
    Example: Allow role "administrators"  
    to call every method on the bean class.  
    -->  
    <method-permission>  
        <role-name>administrators</role-name>  
  
        <method>  
            <ejb-name>EmployeeManagement</ejb-name>  
            <method-name>*</method-name>  
        </method>  
    </method-permission>  
  
    <!--  
    You can set permissions on a method level.  
  
    Example: Allow role "managers" to call method  
            "modifySubordinate()" and "modifySelf()".  
    -->  
    <method-permission>  
        <role-name>managers</role-name>  
  
        <method>  
            <ejb-name>EmployeeManagement</ejb-name>  
            <method-name>modifySubordinate</method-name>  
        </method>  
  
        <method>  
            <ejb-name>EmployeeManagement</ejb-name>  
            <method-name>modifySelf</method-name>  
        </method>  
    </method-permission>  
  
    <!--  
    If you have multiple methods with the same name  
    but that take different parameters, you can even set  
    permissions that distinguish between the two.  
  
    Example: allow role "employees" to call method
```

Source 9.10 Declaring a bean's security policies.

```

        "modifySelf(String)" but not "modifySelf(Int)"
-->
<method-permission>
    <role-name>employees</role-name>

    <method>
        <ejb-name>EmployeeManagement</ejb-name>
        <method-name>modifySelf</method-name>
        <method-params>String</method-params>
    </method>
</method-permission>

<!--
This is the list of methods that we don't want
ANYONE to call. Useful if you receive a bean
from someone with methods that you don't need.
-->
<exclude-list>
    <description>
        We don't have a 401k plan, so we don't
        support this method.
    </description>
    <method>
        <ejb-name>EmployeeManagement</ejb-name>
        <method-name>modify401kPlan</method-name>
        <method-params>String</method-params>
    </method>
</exclude-list>

    . . .

</assembly-descriptor>
. . .

```

Source 9.10 Declaring a bean's security policies (*continued*).

Once defined, the EJB container automatically performs these security checks on your bean's methods at runtime and throws a *java.lang.SecurityException* back to the client code if the client identity is not authenticated or authorized.

Step 2: Declare Security Roles

Declaring security roles is a process similar to programmatic security. We need to define our security roles, and (optionally) describe each so the deployer can understand them. See Source 9.11.

```
<assembly-descriptor>

    . . .

    <security-role>
        <description>
            System administrators
        </description>
        <role-name>administrators</role-name>
    </security-role>

    <security-role>
        <description>
            Employees that manage a group
        </description>
        <role-name>managers</role-name>
    </security-role>

    <security-role>
        <description>
            Employees that don't manage anyone
        </description>
        <role-name>employees</role-name>
    </security-role>

    . . .

</assembly-descriptor>
```

Source 9.11 Declaring security roles for the deployer.

The deployer reads in Source 9.11 and, using the container's tools, maps these roles to principals, as shown in Table 9.1.

Declarative or Programmatic?

As with persistence and transactions, security is a middleware service that you should strive to externalize from your beans. By using declarative security, you decouple your beans' business purpose from specific security policies, enabling others to modify security rules without modifying bean code. No security role strings are hard-coded in your bean logic, keeping your code simple.

In the ideal world, we'd code all our beans with declarative security. But unfortunately, the EJB specification does not provide adequate facilities for

this; specifically, there is no portable way to declaratively perform *instance-level authorization*. This is best illustrated with an example.

Let's say you have an enterprise bean that models a bank account. The caller of the enterprise bean is a bank account manager who wants to withdraw or deposit into that bank account. But this bank account manager is responsible only for bank accounts with balances below \$1,000, and we don't want him modifying bank accounts with larger balances. Declarative authorization has no way to declare in your deployment descriptor that bank account managers can modify only certain bean instances. You can specify security roles only on the enterprise bean class, and those security rules apply for all instances of that class. Thus, you would need to create separate methods for each security role, as we did in Source 9.11. This gets hairy and makes your bean's interface dependent on security roles. For these situations, you should resort to programmatic security.

Security Propagation

Behind the scenes, all security checks are made possible due to *security contexts*. Security contexts encapsulate the current caller's security state. You never see security contexts in your application code, because the container uses them behind the scenes. When you call a method in EJB, the container can propagate your security information by implicitly passing your security context within the stubs and skeletons.

For example, let's say a client is authenticated and has associated security credentials. That client calls bean A, which calls bean B. Should the client's security credentials be sent to bean B, or should bean B receive a different principal? By controlling security context propagation, you can specify the exact semantics of credentials streaming from method to method in a distributed system.

You can control how security information is propagated in your deployment descriptor. The following code takes the client's credentials and propagates them to all other beans you call:

```
...
<enterprise-beans>
  ...
  <session>
    <ejb-name>EmployeeManagement</ejb-name>
    <home>examples.EmployeeManagementHome</home>
    ...
    <security-identity>
      <use-caller-identity/>
    </security-identity>
  ...
</enterprise-beans>
```

```
    </session>
    ...
</enterprise-beans>
```

In comparison, the following code ignores the client's credentials and propagates the role *admins* to all other beans you call:

```
...
<enterprise-beans>
  ...
  <session>
    <ejb-name>EmployeeManagement</ejb-name>
    <home>examples.EmployeeManagementHome</home>
    ...
    <security-identity>
      <run-as>
        <role-name>admins</role-name>
      </run-as>
    </security-identity>
    ...
  </session>

  <assembly-descriptor>
    . . .
    <security-role>
      <description>
        This role is for personnel authorized
        to perform employee administration.
      </description>

      <role-name>admins</role-name>
    </security-role>
    . . .
  </assembly-descriptor>
</enterprise-beans>
```

Your EJB container is responsible for intercepting all method calls and ensuring that your bean is running in the propagation settings you specify. It does this by generating code that executes at the point of interception (inside the EJB objects and EJB home objects).



Since message-driven beans receive JMS messages rather than RMI-IIOP calls, they do not receive any credentials when they are called. It is therefore illegal for message-driven beans to perform any programmatic or declarative security. As far as propagation, it is also illegal for message-driven beans to propagate the nonexistent client's credentials. A message-driven bean can, however, run as a specified identity when calling other beans.

Security Context Propagation Portability

Unfortunately, the EJB specification does not specify how containers should propagate security contexts behind the scenes. What this means to you is that any two EJB containers are likely to be incompatible in how they deal with security. If you call a method from container A into container B, container B will not understand how to receive the security context sent by container A. As the RMI-IIOP interoperability is enhanced, we may see better security context propagation, but that is not likely to emerge until well into the future.

The good news: Most organizations are having enough challenges with a single EJB server! By the time most organizations need security context propagation between different vendors' EJB servers, we should have it.

Understanding Handles

Our final topic in this chapter is the subject of handles. Many EJB applications require that clients are able to disconnect from beans and reconnect later to resume using that bean. For example, if you have a shopping cart that you'd like to save for a later time, and a stateful session bean manifests that shopping cart, you'd want your shopping cart state maintained when you reconnect later.

EJB provides for this need with *EJB object handles*. An EJB object handle is a long-lived proxy for an EJB object. If for some reason you disconnect from the EJB container/server, you can use the EJB object handle to reconnect to your EJB object, so that you don't lose your conversational state with that bean. An EJB object handle is an essentially persistent reference to an EJB object. The following code demonstrates using EJB object handles:

```
// First, get the EJB object handle from the EJB object.
javax.ejb.Handle myHandle = myEJBObject.getHandle();

// Next, serialize myHandle, and then save it in
// permanent storage.
ObjectOutputStream stream = ...;
stream.writeObject(myHandle);

// time passes...

// When we want to use the EJB object again,
// deserialize the EJB object handle
ObjectInputStream stream = . . . ;
Handle myHandle = (Handle) stream.readObject();

// Convert the EJB object handle into an EJB object
```

```
MyRemoteInterface myEJBObject = (MyRemoteInterface)
    javax.rmi.PortableRemoteObject.narrow(
        myHandle.getEJBObject(), MyRemoteInterface.class);

// Resume calling methods again
myEJBObject.callMethod();
```

The EJB specification does not require that handles have the ability to be saved in one environment and then restored in a different environment. This means handles are not guaranteed to be portable across EJB containers, nor across machines.

Home Handles

A variant on EJB object handles are the EJB home handles. These are simply persistent references to home objects, rather than persistent references to EJB objects. The following code shows how to use home handles.

```
// First, get the EJB home handle from the home object.
javax.ejb.HomeHandle homeHandle = myHomeObject.getHomeHandle();

// Next, serialize the home handle, and then save it in
// permanent storage.
ObjectOutputStream stream = ...;
stream.writeObject(homeHandle);

// time passes...

// When we want to use the home object again,
// deserialize the home handle
ObjectInputStream stream = ...;
javax.ejb.HomeHandle homeHandle =
    (HomeHandle) stream.readObject();

// Convert the home object handle into a home object
MyHomeInterface myHomeObject = (MyHomeInterface)
    javax.rmi.PortableRemoteObject.narrow(
        homeHandle.getHomeObject(), MyHomeInterface.class);

// Resume using the home object
myHomeObject.create();
```

Home handles may be useful because you can acquire a reference to a home object, persist it, and then use it again later without knowledge of the home object's JNDI location. But in our opinion, home handles are not going to benefit most applications a tremendous amount. We have never seen any organization make use of them (email us and be the first!).

Summary

In this chapter, we learned a great deal about how to make our beans more robust. We learned how to call beans from other beans, how to use resource factories, how to access environment properties, how to use EJB security, and how to use handles. Most nontrivial EJB deployment will make use of some of these concepts.

This completes Part 2. You've now covered the fundamentals, and should have a strong foundation for learning about advanced concepts. Let's now move on to Part 3, which begins with transactions.

PART THREE

**Advanced Enterprise
JavaBeans Concepts**

If you've read to this point, you should be quite familiar with the basics of Enterprise JavaBeans development. In Part Three, we raise the bar by moving on to more advanced concepts. These include the following:

Transactions. Chapter 10 shows you how to harness transactions to make your EJB deployments reliable. We'll discuss transactions at a conceptual level and how to apply them to EJB. We'll also learn about the Java Transaction API (JTA).

BMP and CMP relationships. Chapter 11 covers how to build *relationships* between entity beans, both BMP and CMP. This is an essential EJB 2.0 topic for anyone performing persistent operations with entity beans.

Persistence Best Practices. In Chapter 12, you'll learn about some of the critical tradeoffs when building a persistence layer—how to choose between session beans and entity beans, how to choose between BMP and CMP—and survey a collection of persistence best practices that we've assembled from our knowledge and experience.

EJB Design Strategies. Chapter 13 is one of the most important chapters in this book. You'll learn about best practices when designing an EJB system. These are not low-level design patterns (those are covered in the companion book, *EJB Design Patterns* by Floyd Marinescu). We'll discuss tradeoffs you can make when designing an EJB systems such as how to choose between local interfaces and remote interfaces, how to choose between stateful and stateless systems, and much more.

Clustering. Chapter 14 shows you how EJBs are clustered in large-scale systems. You'll learn how clustering works behind the scenes, and a few strategies for how containers might achieve clustering. This is a critical topic for anyone building a system that involves several machines working together.

EJB project management. Chapter 15 shows you how to get your project off on the right foot. This includes how to choose whether EJB is right for you, how to build a first-pass of your system, and how to divide your development team.

How to choose an EJB server. In Chapter 16, we'll describe our methodology for how an organization can compare and contrast different vendors' offerings. We'll also list our criteria for what we would want in an EJB server.

Building a real-world EJB-J2EE system. Chapter 17 shows how each of the EJB components can work together to solve a business problem, as well as how EJB and J2EE can be integrated, as through Java Servlets and JavaServer Pages (JSPs).

These are extremely interesting middleware topics; indeed, many books could be written on each subject alone. To understand these concepts, we highly recommend you read Part One and Part Two first. If, however, you're already well-versed in EJB, please join us to explore these advanced issues.

Transactions

Many middleware services are needed for secure, scalable, and reliable server-side development. This includes resource pooling services, security services, remotability services, persistence services, and more.

A key service required for robust server-side development is *transactions*. Transactions, when used properly, can make your mission-critical operations run predictably in an enterprise environment. Transactions are an advanced programming paradigm that allows you to write robust code. Transactions are also very useful constructs to use when performing persistent operations like updates to a database.

In the past, transactions have been difficult to use because developers needed to code directly to a transaction API. With EJB, you can gain the benefits of transactions without performing any transaction programming.

In this chapter, we'll see some of the problems that transactions solve. We'll also see how transactions work and show how they're used in EJB. Because transactions are at the very core of EJB and are somewhat difficult to understand, we'll provide extensive background on the subject. To explain transactions properly, we'll occasionally get a bit theoretical. If the theory presented in this chapter piques your interest, many tomes written on transactions are available for further reading. See the book's accompanying Web site, www.wiley.com/compbooks/roman, for links to more information.

Motivation for Transactions

We begin our discussion with a few motivational problems that transactions address.

Atomic Operations

Imagine that you would like to perform multiple discrete operations yet have them execute as one contiguous, large, *atomic* operation. Take the classic bank account example. When you transfer money from one bank account to another, you want to withdraw funds from one account and deposit those funds into the other account. Ideally, both operations will succeed. But if an error occurs, you would like *both* operations to always fail; otherwise, you'll have incorrect funds in one of the accounts. You never want one operation to succeed and the other to fail, because both operations are part of a single atomic transaction.

One simplistic way to handle this is to perform exception handling. You could use exceptions to write a banking module to transfer funds from one account to another, as in the following pseudo-code:

```
try {
    // Withdraw funds from account 1
}
catch (Exception e) {
    // If an error occurred, do not proceed.
    return;
}
try {
    // Otherwise, deposit funds into account 2
}
catch (Exception e) {
    // If an error occurred, do not proceed,
    // and redeposit the funds back into account 1.
    return;
}
```

This code tries to withdraw funds from account 1. If a problem occurs, the application exits and no permanent operations occur. Otherwise, we try to deposit the funds into account 2. If a problem occurs here, we redeposit the money back into account 1 and exit the application.

There are many problems with this approach:

- The code is bulky and unwieldy.

- We need to consider every possible problem that might occur at every step and code error-handling routines to consider how to roll back our changes.
- Error-handling gets out of control if we perform more complex processes than a simple withdrawal and deposit. It is easy to imagine, for example, a 10-step process that updates several financial records. We'd need to code error-handling routines for each step. In the case of a problem, we need to code facilities to undo each operation. This gets tricky and error-prone to write.
- Testing this code is yet another challenge. You would have to simulate logical problems as well as failures at many different levels.

Ideally, we would like a way to perform *both* operations in a single, large, atomic operation, with a guarantee that both operations either always succeed, or both always fail.

Network or Machine Failure

Let's extend our classic bank account example and assume our bank account logic is distributed across a multitier deployment. This may be necessary for security, scalability, and modularization reasons. In a multitier deployment, any client code that wants to use our bank account application must do so across the network via a remote method invocation. We show this in Figure 10.1.

Distributing our application across the network introduces failure and reliability concerns. For example, what happens if the network crashes during a

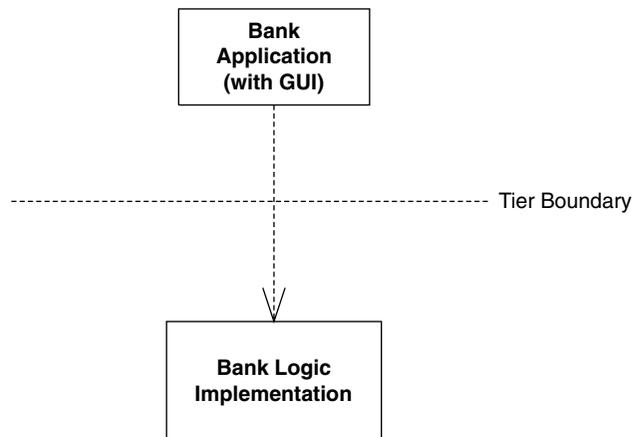


Figure 10.1 A distributed banking application.

banking operation? Typically, an exception (such as a Java RMI *RemoteException*) is generated and thrown back to the client code—but this exception is quite ambiguous. The network may have failed *before* money was withdrawn from an account. It's also possible that the network failed *after* we withdrew the money. There's no way to distinguish between these two cases—all the client code sees is a network failure exception. Thus, we can never know for sure how much money is in the bank account.

The network may not be the only source of problems. In dealing with bank account data, we're dealing with persistent information residing in a database. It's entirely feasible that the database itself could crash. The machine that the database is deployed on could also crash. If a crash occurs during a database write, the database could be in an inconsistent, corrupted state.

None of these situations is acceptable for a mission-critical enterprise application. Mainframe systems and other highly available systems offer preventive measures to avoid system crashes. But in reality, nothing is perfect. Machines, processes, or networks will always fail. There needs to be a recovery process to handle these crashes. Simple exception handling such as Java RMI's *RemoteException* is not sufficient for enterprise-class deployments.

Multiple Users Sharing Data

In any enterprise-level distributed system, you will see the familiar pattern of multiple clients connecting to multiple application servers, with those application servers maintaining some persistent data in a database. Let's assume these application servers all share the same database, as in Figure 10.2. Because each server is tied to the same database image, servers could potentially be modifying the *same* set of data records within that database.

For example, you might have written a tool to maintain your company's catalog of products in a database. Your catalog may contain product information that spans more than one database record. Information about a single product could span several database records or even tables.

Several people in your organization may need to use your tool simultaneously. But if two users modify the same product data simultaneously, their operations may become interleaved. Therefore, your database may contain product data that's been partially supplied by one tool and partially supplied by another tool. This is essentially corrupted data, and it is not acceptable in any serious deployment. The wrong data in a bank account could result in loss of millions of dollars to a bank or the bank's customers.

Thus, there needs to be a mechanism to deal with multiple users concurrently modifying data. We must guarantee data integrity even when many users concurrently update the data.

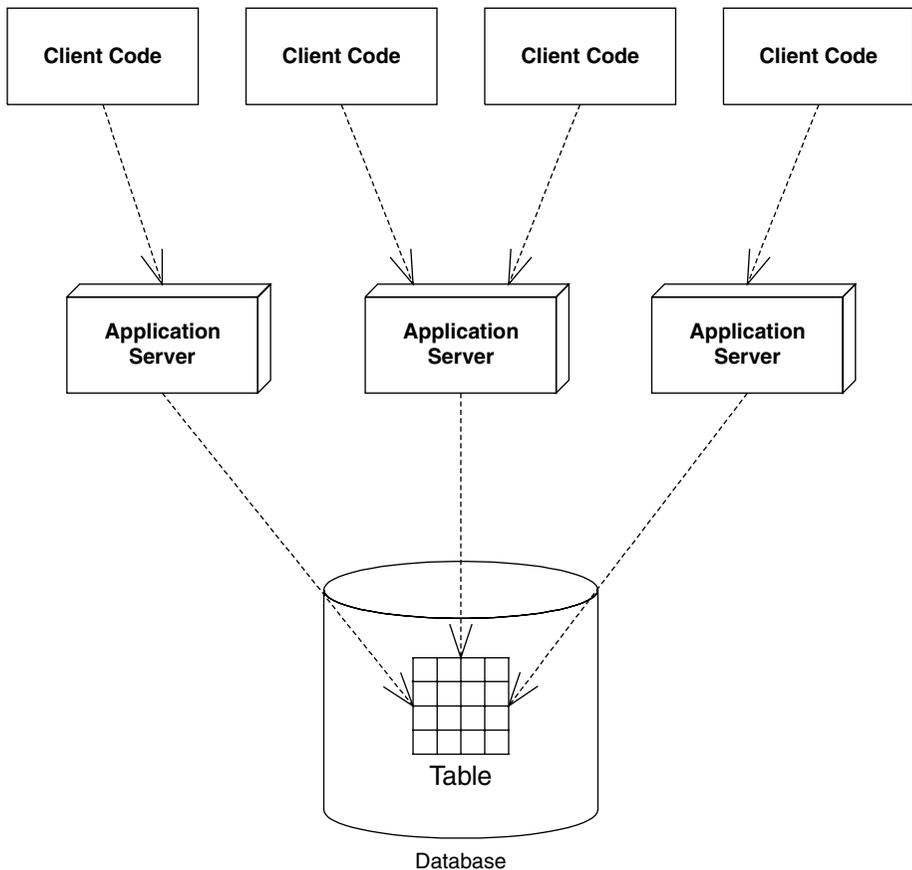


Figure 10.2 Application servers tied to a single database.

Benefits of Transactions

The problems raised in the previous sections can lead to catastrophic errors. You can avoid these problems by properly using *transactions*.

A transaction is a series of operations that appear to execute as one large, atomic operation. Transactions guarantee an all-or-nothing value proposition: Either all of your operations will succeed, or none of them will. Transactions account for network or machine failure in a graceful, reliable way. Transactions allow multiple users to share the same data and guarantee that any set of data they update will be completely and wholly written, with no interleaving of updates from other clients.

By using transactions properly, you can enforce that multiuser interactions with databases (or other storages) occur independently. For example, two clients reading and writing from the same database will be mutually exclusive if transactions are properly used. The database system automatically performs the necessary concurrency control (that is, locking) on the database to keep client threads from affecting each other.

Transactions offer far more than simply letting simultaneous users use the same persistent stores. By having your operations run within a transaction, you are effectively performing an advanced form of concurrency control and exception handling.

The ACID Properties

When you properly use transactions, your operations will always execute with a suite of four guarantees. These four guarantees are well known as the *ACID*

Transaction Vocabulary

Before we get into the specifics of transactions, let's establish a vocabulary. There are several types of participants in a transaction: *transactional objects*, *transaction managers*, *resources*, and *resource managers*. Let's take a look at each of these parties in more detail.

A *transactional object* (or *transactional component*) is an application component, such as a banking component, that is involved in a transaction. This could be an enterprise bean, a Microsoft.NET-managed component, a CORBA component, and so on. These components perform operations that need to execute in a robust fashion, like database interactions.

A *transaction manager* is responsible for managing the transactional operations of the transactional components. It manages the entire overhead of a transaction, running behind the scenes to coordinate things (similar to how a conductor coordinates a symphony).

A *resource* is a persistent storage from which you read or write. A resource could be a database, a message queue, or other storage.

A *resource manager* manages a resource. An example of a resource manager is a driver for a relational database, object database, message queue, or other store. Resource managers are responsible for managing all state that is permanent. The most popular interface for resource managers is the *X/Open XA* resource manager interface. Most database drivers support this interface. Because *X/Open XA* is the de facto standard for resource managers, a deployment with heterogeneous resource managers from different vendors can interoperate.

properties of transactions. The word ACID stands for *atomicity, consistency, isolation, and durability*. Here's the breakdown of each property.

Atomicity guarantees that many operations are bundled together and appear as one contiguous *unit of work*. In our banking example, when you transfer money from one bank account to another, you want to add funds to one account and remove funds from the other account, and you want both operations to occur or neither operation to occur. Atomicity guarantees that operations performed within a transaction undergo an *all-or-nothing paradigm*—either all the database updates are performed, or nothing happens if an error occurs at any time. Many different parties can participate in a transaction, such as an enterprise bean, a CORBA object, a servlet, and a database driver. These transaction participants can force the transaction to result in *nothing* happening for any reason. This is similar to a voting scheme: Each transaction participant votes on whether the transaction should be successful, and if any vote no, the transaction fails. If a transaction fails, all the partial database updates are automatically undone. In this way, you can think of transactions as a robust way of performing error handling.

Consistency guarantees that a transaction leaves the system's state to be *consistent* after a transaction completes. What is a consistent system state? A bank system state could be consistent if the rule *bank account balances must always be positive* is always followed. This is an example of an invariant set of rules that define a consistent system state. During the course of a transaction, these rules may be violated, resulting in a temporarily inconsistent state. For example, your enterprise bean component may temporarily make your account balance negative during a withdrawal. When the transaction completes, the state is consistent once again; that is, your bean never leaves your account at a negative balance. And even though your state can be made inconsistent temporarily, this is not a problem. Remember that transactions execute *atomically* as one, contiguous unit of work (from the atomicity property above). Thus, to a third party, it appears that the system's state is always consistent. Atomicity helps enforce that the system *always* appears to be consistent.

Isolation protects concurrently executing transactions from seeing each other's incomplete results. Isolation allows multiple transactions to read or write to a database without knowing about each other because each transaction is *isolated* from the others. This is useful for multiple clients modifying a database at once. It appears to each client that he or she is the only client modifying the database at that time. The transaction system achieves isolation by using low-level *synchronization protocols* on the underlying database data. This synchronization isolates the work of one transaction from that of another. During a transaction, locks on data are automatically

assigned as necessary. If one transaction holds a lock on data, the lock prevents other concurrent transactions from interacting with that data until the lock is released. For example, if you write bank account data to a database, the transaction may obtain locks on the bank account record or table. The locks guarantee that, while the transaction is occurring, no other concurrent updates can interfere. This allows many users to modify the same set of database records simultaneously without concern for interleaving of database operations.

Durability guarantees that updates to managed resources, such as database records, survive failures. Some examples of failures are machines crashing, networks crashing, hard disks crashing, or power failures. Recoverable resources keep a transactional *log* for exactly this purpose. If the resource crashes, the permanent data can be reconstructed by reapplying the steps in the log.

Transactional Models

Now that you've seen the transaction value proposition, let's dive a bit deeper and explore how transactions work. We begin by taking a look at *transactional models*, which are the different ways you can perform transactions.

There are many different models for performing transactions. Each model adds its own complexity and features to your transactions. The two most popular models are *flat transactions* and *nested transactions*.



To use a particular transaction model, your underlying transaction service must support it. And unfortunately, not all of the vendors who crafted the EJB specification currently implement nested transactions in their products. Hence, Enterprise JavaBeans mandates flat transactions but does not support nested transactions. Note that this may change in the future based on industry demands.

Flat Transactions

A *flat transaction* is the simplest transactional model to understand. A flat transaction is a series of operations that are performed atomically as a single *unit of work*. After a flat transaction begins, your application can perform any number of operations. Some may be persistent operations, and some may not. When you decide to end the transaction, there is always a binary result: either success or failure. A successful transaction is *committed*, while a failed transaction is *aborted*. When a transaction is committed, all of the persistent operations become permanent changes; that is, all of the updates to resources, such as databases, are made durable into permanent storage only if the trans-

action ends with a *commit*. If the transaction is aborted, none of the resource updates are made durable, and thus all changes are *rolled back*. When a transaction aborts, all persistent operations that your application may have performed are automatically undone by the underlying system. Your application can also be notified in case of an abort, so that your application can undo in-memory changes that occurred during the transaction.

This is the *all-or-nothing* proposition we described above. The flat transaction process is outlined in Figure 10.3.

A transaction might abort for many reasons. Many components can be involved in a transaction, and any one component could suffer a problem that would cause an abort. These problems include the following:

Invalid parameters passed to one of the components. For instance, a banking component may be called with a null argument, when it was expecting a bank account ID string.

An invariant system state was violated. For example, if a bank account has a negative balance, your banking component can force the transaction to abort, undoing all associated bank account operations.

Hardware or software failure. If the database that your component is using crashes, the transaction is rolled back and all permanent changes are undone. Similarly, if there is a software failure (such as a distributed system where a JVM crashes) the transaction is rolled back.

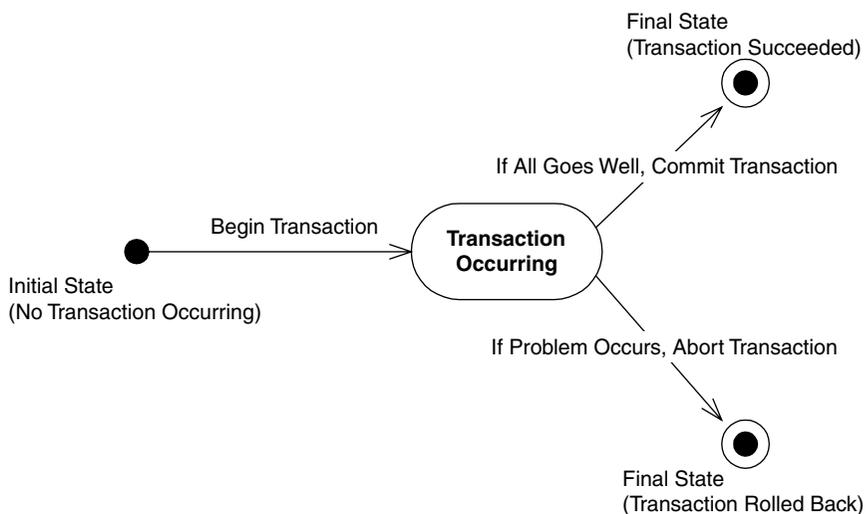


Figure 10.3 The flat transaction.

Any of these problems can cause a transaction to abort. But when an abort occurs, how is the transactional state rolled back? That is the topic of the next section.

How Transactional State Is Rolled Back

Let's assume you're performing a flat transaction that includes operations on physical, permanent resources, such as databases. After the transaction begins, one of your business components requests a connection to a database. This database connection is automatically enlisted in the transaction in which your component is involved. Next, your component performs some persistent operations, such as database updates. But when this happens, your database's resource manager does not permanently apply the updates to the database—your persistent operations are not yet durable and permanent. The resource manager waits until a *commit* statement has been issued. A commit is issued only when all your business components have finished performing all of the operations under that transaction—when the transaction is complete. If the resource is told to commit, it persists the data permanently. If the transaction aborts, the data is not persisted at all.

The take-away point from this discussion is that your business components typically do not perform any rollback of permanent state; if there's an *abort*, the resource (such as a database) does not make your database updates permanent. Your components don't have any *undo* logic for permanent data inside of them; rather, the underlying system does it for you behind the scenes. Your components control the transaction and tell the transaction to abort, but the persistent state rollback is performed for you automatically. Thus, when your business components perform operations under a transaction, each component should perform all persistent operations assuming that the transaction will complete properly.

Now that you've seen flat transactions, let's take a quick look at nested transactions.

Nested Transactions

We begin our nested transactions discussion with a motivational example. Let's say you need to write an application that can plan trips for a travel agency. You need to code your application to plan trips around the world, and your application must purchase the necessary travel tickets for the trip. Consider that your application performs the following operations:

1. Your application purchases a train ticket from Boston, USA, to New York, USA.
2. Your application purchases a plane ticket from New York, USA, to London, England.
3. Your application purchases a balloon ride ticket from London, England, to Paris, France.
4. Your application finds out that there are no outgoing flights from France.

This is the famous *trip-planning problem*. If this sequence of bookings were performed under a flat transaction, your application would have only one option: to roll back the transaction. Thus, because there are no outgoing flights from France, your application has lost all of its bookings! But it may be possible to use another means of transportation out of France, allowing you to salvage the train ticket, plane ticket, and balloon ride. Thus, a flat transaction is insufficient. The all-or-nothing proposition is shooting us in the foot, and we need a more robust transactional model.

A nested transaction solves this problem. A *nested transaction* allows you to embed atomic units of work within other units of work. The unit of work that is nested within another unit of work can roll back without forcing the entire transaction to roll back. Therefore the larger unit can attempt to retry the embedded unit of work. If the embedded unit can be made to succeed, the larger unit can succeed. If the embedded unit of work cannot be made to work, it will ultimately force the entire unit to fail.

You can think of a nested transaction as a *tree* of transactions, all spawning off one *root- or top-level transaction*. The root transaction is the *main* transaction: In our trip-planning example, the root transaction is the overall process of booking tickets around the world. Every other transaction in the tree is called a *sub-transaction*. The subtransactions can be flat or nested transactions. Figure 10.4 illustrates this concept.

What's special about nested transactions is that subtransactions can independently roll back without affecting higher transactions in the tree. That's a very powerful idea, and it solves our trip-planning problem: If each individual booking is a nested transaction, we can roll back any one booking without canceling all our other reservations. But in the end, if the nested transaction cannot be committed, the entire transaction will fail.

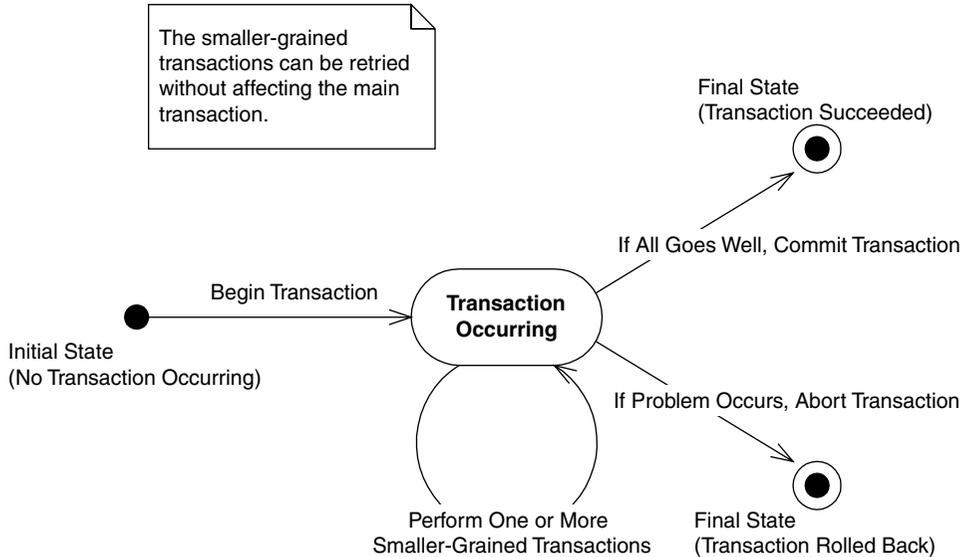


Figure 10.4 The nested transaction.

Other Transactional Models

This concludes our discussion of transactional models. There are other models as well, such as *chained transactions* and *sagas*, but we will not address these subjects here because the EJB specification does not support them. And because the EJB specification does not currently mandate support for nested transactions, for the rest of this chapter we'll assume that our transactions are flat.

Enlisting in Transactions with Enterprise JavaBeans

Let's apply what we've learned so far about transactions to the EJB world.

Enterprise beans can be transactional in nature. This means they can fully leverage the ACID properties to perform reliable, robust server-side operations. Thus, enterprise beans are ideal modules for performing mission-critical tasks.

Underlying Transaction System Abstraction

In EJB, your code never gets directly involved with the low-level transaction system. Your enterprise beans never interact with a transaction manager or a resource manager. You write your application logic at a much higher level, without regard for the specific underlying transaction system. The low-level transaction system is totally abstracted out by the EJB container, which runs behind the scenes. Your bean components are responsible for simply voting on whether a transaction should commit or abort. If things run smoothly, you should commit; otherwise, abort.

Declarative, Programmatic, and Client-Initiated Transactions

Throughout this chapter, we've said that once a transaction begins, it ends with either commit or abort. The key piece of information we're lacking is *who* begins a transaction, *who* issues either a commit or abort, and *when* each of these steps occurs. This is called *demarcating transactional boundaries*. There are three ways to demarcate transactions: *programmatically*, *declaratively*, or *client-initiated*.

Programmatic Transactions

Most existing systems demarcate transactional boundaries *programmatically*. When using programmatic transactions, you are responsible for programming transaction logic into your application code. That is, *you* are responsible for issuing a *begin* statement and either a *commit* or an *abort* statement.

For example, an EJB banking application might have an enterprise bean that acts as a bank teller. A teller bean would expose a method to transfer funds from one bank account to another. With programmatic transactions, the teller bean is responsible for issuing a *begin* statement to start the transaction, performing the transfer of funds, and issuing either a *commit* or *abort* statement. This is the traditional way to perform transactions, and it is shown in Figure 10.5.

Declarative Transactions

Declarative transactions allow for components to *automatically* be enlisted in transactions. That is, your enterprise beans never explicitly issue a *begin*, *commit*, or *abort* statement. The EJB container performs it for you.

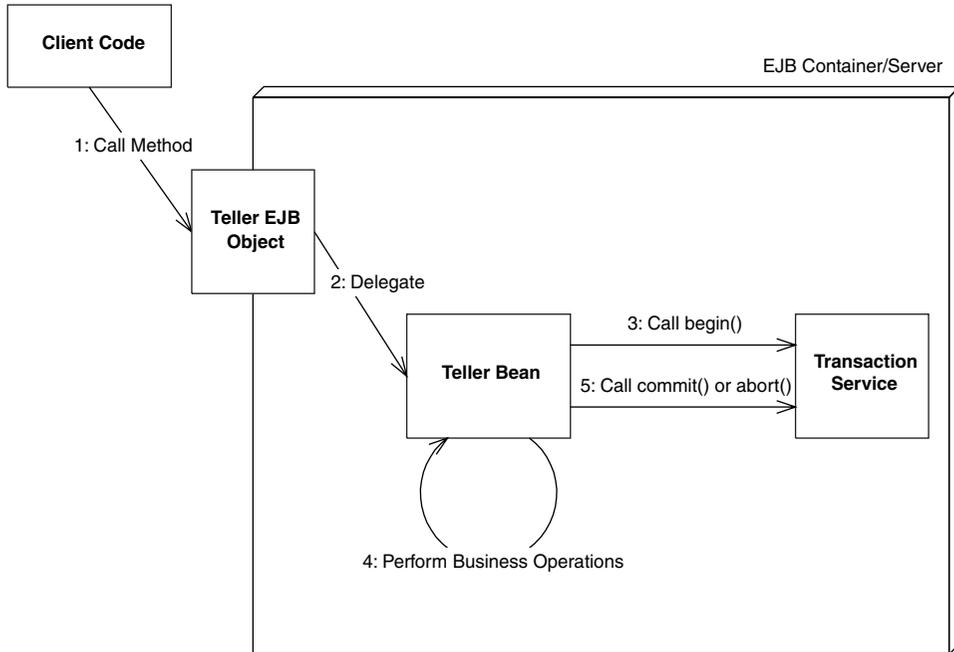


Figure 10.5 Beans with programmatic transactions.

Let's take our bank teller example again, and assume some client code has called our teller bean to transfer funds from one account to another. With declarative transactions, the EJB container *intercepts* the request and starts up a transaction automatically on behalf of your bean. That is, the container issues the *begin* statement to the underlying transaction system to start the transaction. The container then delegates the invocation to your enterprise bean, which performs operations in the scope of that transaction. Your bean can do anything it wants to, such as perform logic, write to a database, send an asynchronous message, or call other enterprise beans. If a problem occurs, the bean can signal to the container that the transaction must abort. When the bean is done, it returns control back to the container. The container then issues either a *commit* or *abort* statement to the underlying transaction system, depending on whether a problem occurred. This is a very simple model, and it is shown in Figure 10.6.

EJB declarative transactions add huge value to your deployments because your beans may not need to interact with any transaction API. In essence, your bean code and your client are not even really aware of transactions happening around them.

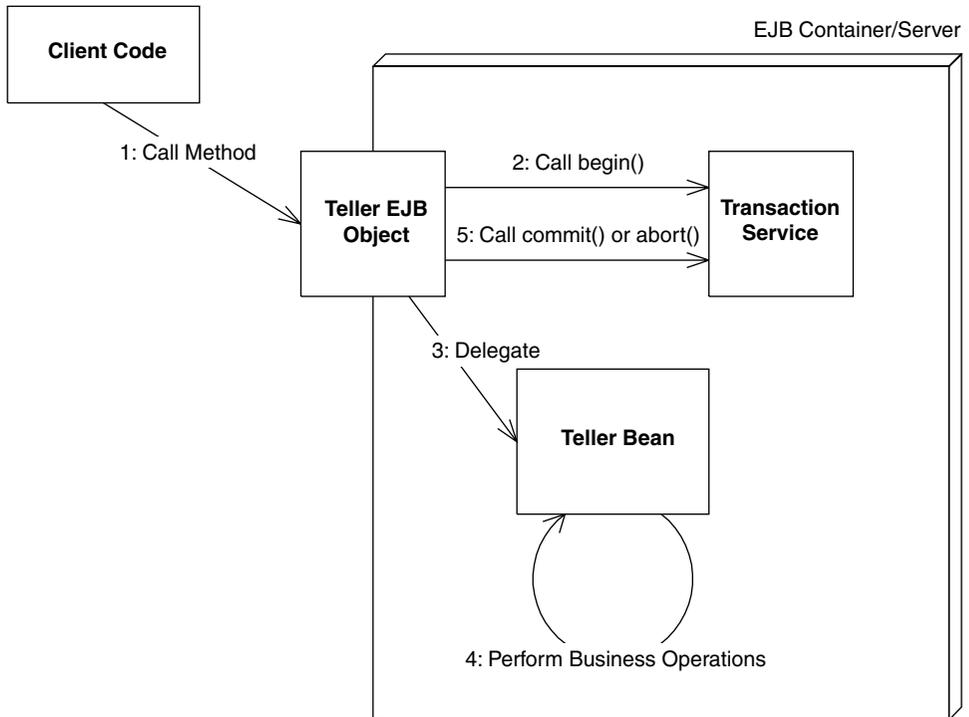


Figure 10.6 Beans with declarative transactions.

So how do you instruct the container about whether your bean is using declarative or programmatic transactions? EJB allows you to specify how your enterprise bean is enrolled in a transaction through the deployment descriptor, as follows:

```
<!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>
  <enterprise-beans>
    <session>
      <ejb-name>Hello</ejb-name>
      <home>examples.HelloHome</home>
      <remote>examples.Hello</remote>
      <ejb-class>examples.HelloBean</ejb-class>
      <session-type>Stateless</session-type>
      <transaction-type>Container</transaction-type>
    </session>
  </enterprise-beans>
</ejb-jar>
```

The above deployment descriptor chooses container-managed (declarative) transactions. If instead of *Container* we chose *Bean*, then we would use bean-managed (programmatic) transactions.

Client-Initiated Transactions

The final way to perform transactions is to write code to start and end the transaction from the client code outside of your bean. For example, if you have a servlet, JSP tag library, application, applet, CORBA client, or other enterprise bean as a client of your beans, you can begin and end the transaction in that client. This is shown in Figure 10.7.

Note that the enterprise bean the client calls would still need to be written to use either programmatic or declarative transactions.

Choosing a Transaction Style

One question that students often ask in our EJB training classes is, “Should I use declarative, programmatic, or client-controlled transactions?”

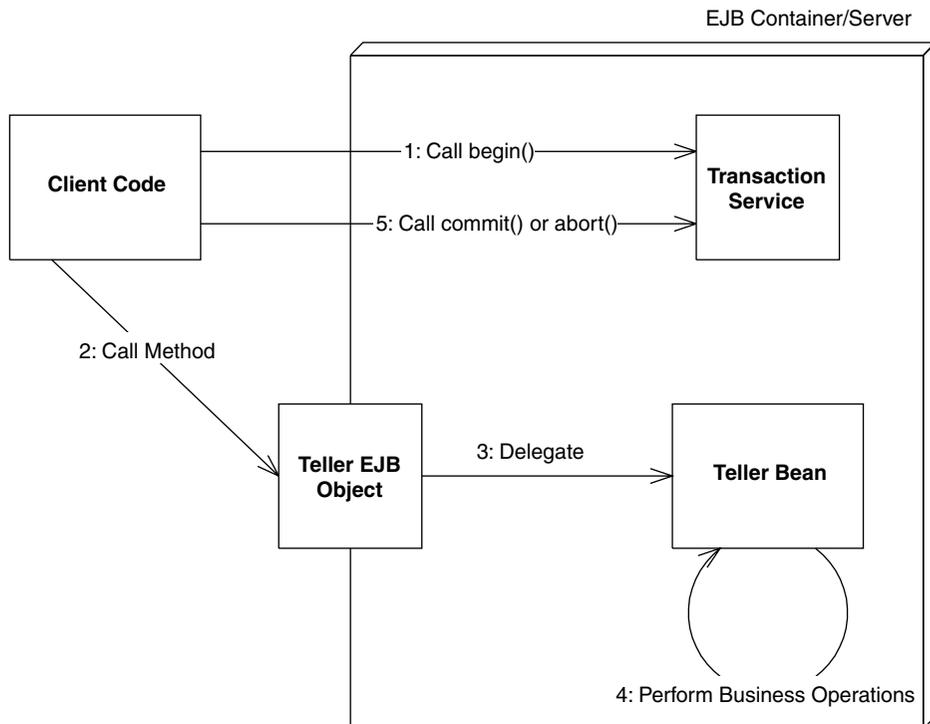


Figure 10.7 Beans with client-initiated transactions.

Transactions and Entity Beans

Perhaps one of the most misunderstood concepts in EJB is how transactions relate to entity beans. Let's explore this concept.

When you call an entity bean in a transaction, the first action that happens is the entity bean needs to load database data through the *ejbLoad()* method, which acquires locks in the database and ensures the entity bean cache is consistent. Then one or more business methods are called. When the transaction is committed, the entity bean's *ejbStore()* method is called, which writes all updates to the database and releases the locks. A transaction should thus span both *ejbLoad()*, business methods, and the final *ejbStore()*, so that if any one of those operations fail, they all fail.

If we were to use bean-managed transactions, we would write code to perform *begin()* and *commit()* methods inside our bean (perhaps around the JDBC code). Perhaps we would start the transaction in *ejbLoad()*, and then commit the transaction in *ejbStore()*. The problem, though, is that you do not call your own *ejbLoad()* or *ejbStore()* methods—the container does. The bean cannot enforce that these methods happen in this order, if at all. Therefore if you started a transaction in *ejbLoad()*, the transaction may never complete.

Because of this, bean-managed transactions are illegal for entity beans. Entity beans *must* use declarative transactions. Session beans or message-driven beans can use bean-managed transactions because a session bean can load database data, perform operations on that data, and then store that data, all in a single method call, and thus is in direct control over the transaction.

A corollary of this discussion is that entity beans do not load and store their data on every method call; rather, they load and store their data on every transaction. If your entity beans are not performing well, it could be because a transaction is happening on each method call, and thus a database read/write is happening on every get/set method. The solution is to make sure your transactions begin earlier and end later, perhaps encompassing many entity bean method calls. By properly controlling the duration of your transactions with transaction attributes (as we will see later in this chapter), you can control when database reads and writes happen with entity beans. For more on this design strategy, see Chapter 13.

The benefit of programmatic transactions is that your bean has full control over transactional boundaries. For instance, you can use programmatic transactions to run a series of minitransactions within a bean method. In comparison, with declarative or client-initiated transactions, your entire bean method must either run under a transaction or not run under a transaction.

The benefit of declarative transactions is that they are simpler. You don't need to write transactional logic into your bean class, which saves coding time and allows you to tune transactions without having access to source code. Also, by having transactions automatically start up and end, you keep client code from misusing your beans. If you're a bean vendor, this will reduce a great number of headaches down the line.

To understand the benefit of client-controlled transactions, consider the following scenario in which we *don't* use client-controlled transactions. Imagine that a nontransactional remote client calls an enterprise bean that performs its own transactions (either programmatically or declaratively). The bean succeeds in the transaction, but the network or application server crashes before the result is returned to a remote client. The remote client would receive a Java RMI *RemoteException* indicating a network error, but would not know whether the transaction that took place in the enterprise bean was a success or a failure. The remote client would then have to write code to check the state of the server if a *RemoteException* was ever thrown. This code can get very messy and is error-prone because it may never be able to contact the server.

With client-controlled transactions, you do not need to worry about this scenario, because the transaction is defined in the client code. If anything goes wrong, the client will know about it. The downside to client-controlled transactions is that if the client is located far from the server, the transactions are likely to roll back due to conflicts. Because of this, use client-transactions sparingly—especially if the client is far away.

Container-Managed Transactions

Let's now assume that we are using container-managed transactions and understand how to implement them. Although we're not writing any code that starts and stops transactions, we still need to provide instructions to the container for how we'd like our transactions to operate. For example, how can we choose whether a bean always runs in a transaction, or whether a bean never runs in a transaction?

A *transaction attribute* is a setting that you give to a bean to control how your bean is enlisted in container-managed transactions. You can specify a different transaction attribute on each bean in your system, regardless of how many beans are working together.

The transactional attribute is a required part of each bean's deployment descriptor. The container knows how transactions should be handled with a bean by reading that bean's transaction attribute from its deployment descriptor. Note that you can specify transaction attributes for entire beans or for indi-

Transactions and Message-Driven Beans

When using message-driven beans, your choice of transaction style has a big impact on your bean.

If you use container-managed transactions, your message-driven bean will read a message off the destination in the same transaction as it performs its business logic. If something goes wrong, the transaction will roll back and the message acknowledgement will occur.

If you use bean-managed transactions, the transaction begins and ends *after* your message-driven bean receives the message. You can then use deployment descriptor *acknowledgement modes* to instruct the container about when to acknowledge messages (see Chapter 8).

If you don't support transactions at all, the container will acknowledge the message at some later time, perhaps when your bean's method completes. The timing is not guaranteed, however.

So which style do you use? If you don't use container-managed transactions, you can't cause the messages to remain on the original destination if something goes wrong, because your bean has no way to indicate a problem.

In general, we recommend using container-managed transactions with message-driven beans. If you want to perform many smaller transactions, consider breaking up your message-driven bean into several other beans, with each bean having a granularity of a single transaction.

Note that there is a huge caveat with container-managed transactions. Let's say you have got an EJB component (any type of component) that sends *and then* receives a message all within one big container-managed transaction. It is possible for the send to never get its message on the queue, because the transaction doesn't commit until after the receive ends. Thus, you'll be waiting for the receive to complete forever. This is called the *infinite block* problem, also known as the *halting problem* in computer science.

Many developers overlook an easy solution to this problem: After sending the request message, you can call *commit()* on the JMS *Session*, which is your JMS transaction helper object. This causes the outgoing message buffer to be flushed.

vidual bean methods. If both are specified, then method-level attributes take precedence. See Source 10.1 transaction.

You must specify transaction attributes on all business methods for your beans. Furthermore, with entity beans you must specify transaction attributes that cover home interface methods, because the home interface creation methods insert database data and thus need to be transactional.

EJB Transaction Attribute Values

Every enterprise bean must have a transaction attribute setting. The following are the possible values for the transaction attribute in the deployment descriptor.

Required

You should use the *Required* mode if you want your bean to *always* run in a transaction. If a transaction is already running, your bean joins in on that transaction. If no transaction is running, the EJB container starts one for you.

For example, say you write a credit card component that performs operations on credit cards, such as charging a credit card or refunding money on a credit card. Let's assume you ship the component with the *Required* transaction attribute. You then sell that component to two customers.

```
<assembly-descriptor>

  <!--
  This demonstrates setting a transaction attribute
  on every method on the bean class.
  -->
  <container-transaction>

    <method>
      <ejb-name>Employee</ejb-name>
      <method-name>*</method-name>
    </method>

    <!--
    Transaction attribute. Can be "NotSupported",
    "Supports", "Required", "RequiresNew",
    "Mandatory", or "Never".
    -->
    <trans-attribute>Required</trans-attribute>

  </container-transaction>

  <!--
  You can also set transaction attributes on individual methods.
  -->
  <container-transaction>

    <method>
```

Source 10.1 Declaring transaction attributes in the deployment descriptor.

```
        <ejb-name>Employee</ejb-name>
        <method-name>setName</method-name>
    </method>

    <trans-attribute>Required</trans-attribute>

</container-transaction>

<!--
You can even set different transaction attributes on
methods with the same name that take different parameters.
-->
<container-transaction>

    <method>
        <ejb-name>Employee</ejb-name>
        <method-name>setName</method-name>
        <method-param>String</method-param>
    </method>

    <trans-attribute>Required</trans-attribute>

</container-transaction>

</assembly-descriptor>
```

Source 10.1 Declaring transaction attributes in the deployment descriptor (continued).

Customer 1 deploys our component in its customer service center, using the component to refund money when an angry customer calls. The customer writes some proprietary code to call your bean as necessary. When the client code calls your bean, the container automatically starts a transaction by calling *begin* and then delegating the call to your bean. When your method completes, the container issues either a *commit* or *abort* statement, depending on whether a problem occurred.

Customer 2 uses our billing component as part of a complete workflow solution. The customer wants to use the credit card component to charge a user's credit card when a user purchases a product from a Web site. The customer then wants to submit an order to manufacture that product, which is handled by a separate component. Thus, the customer has two separate components running but both of them to run under the same transaction. If the credit card cannot be charged, the customer doesn't want the order to be submitted. If the order cannot be submitted, the customer

doesn't want the credit card charged. Therefore the customer produces his or her own workflow bean, which first calls our credit card charging bean and then calls the bean to generate a manufacturing order. The workflow bean is deployed with *Required*, so a transaction automatically starts up. Because your credit card bean is also deployed with *Required*, you *join* that transaction, rather than starting your own transaction. If the order submission component is also deployed with *Required*, it joins the transaction as well. The container commits or aborts the transaction when the workflow bean is done.

Thus, *Required* is a flexible transaction attribute that allows you to start your own transaction or join existing ones, depending on the scenario.

RequiresNew

You should use the *RequiresNew* attribute if you always want a *new* transaction to begin when your bean is called. If a transaction is already underway when your bean is called, that transaction is suspended during the bean invocation. The container then launches a new transaction and delegates the call to the bean. The bean performs its operations and eventually completes. The container then commits or aborts the transaction and finally resumes the old transaction. Of course, if no transaction is running when your bean is called, there is nothing to suspend or resume.

RequiresNew is useful if your bean needs the ACID properties of transactions but wants to run as a single unit of work without allowing other external logic to also run in the transaction.

Supports

When a bean is called with *Supports*, it runs only in a transaction if the client had one running already—it joins that transaction. If the client does not have a transaction, the bean runs with no transaction at all.

Supports is similar in nature to *Required*, with the one exception: *Required* enforces that a new transaction is started if one is not running already. Because *Supports* will sometimes not run within a transaction, you should be careful when using this attribute. Mission-critical operations should be encapsulated with a stricter transaction attribute (like *Required*).

Mandatory

Mandatory mandates that a transaction *must be already running* when your bean method is called. If a transaction isn't running, the `javax.ejb.TransactionRe-`

quiredException exception is thrown back to the caller (or *javax.ejb.TransactionRequiredLocalException* exception if the client is local).

Mandatory is a safe transaction attribute to use. It guarantees that your bean should run in a transaction. There is no way your bean can be called if a transaction isn't already running. However, *Mandatory* relies on a third party to start the transaction before your bean is called. The container will *not* automatically start a transaction; rather, an exception is thrown back to the caller. This is the chief difference between *Mandatory* and *Supports*. *Mandatory* is useful if your component is designed to run within a larger system, such as a workflow system, where your bean is only part of a larger suite of operations, and you want to mandate that the larger operations start a transaction before calling your bean.

NotSupported

If you set your bean to use *NotSupported*, then your bean *cannot* be involved in a transaction at all. For example, assume we have two enterprise beans, A and B. Let's assume bean A begins a transaction and then calls bean B. If bean B is using the *NotSupported* attribute, the transaction that A started is suspended. None of B's operations are transactional, such as reads/writes to databases. When B completes, A's transaction is resumed.

You should use *NotSupported* if you are certain that your bean operations do not need the ACID properties. This should be used only if your beans are performing nonmission-critical operations, where you are not worried about isolating your bean's operations from other concurrent operations. An example here is an enterprise bean that performs rough reporting. If you have an e-commerce Web site, you might write a bean that routinely reports a rough average number of e-commerce purchases per hour by scanning a database. Because this is a low-priority operation and you don't need exact figures, *NotSupported* is an ideal, low-overhead mode to use.

Never

The *Never* transaction attribute means that your bean cannot be involved in a transaction. Furthermore, if the client calls your bean in a transaction, the container throws an exception back to the client (*java.rmi.RemoteException* if remote, *javax.ejb.EJBException* if local).

This transaction attribute is useful when you want to make sure all clients that call your bean do not use transactions. This can help reduce errors in client code, because a client will not be able to erroneously call your bean in a transaction and expect your bean to participate in the ACID properties with other

transaction participants. If you are developing a system that is not transactional in nature and would like to enforce that behavior, consider using the *Never* attribute.

Transaction Attribute Summary

Table 10.1 is a summary of the effects of each transaction attribute. In the chart, T1 and T2 are two different transactions. T1 is a transaction passed with the client request, and T2 is a secondary transaction initiated by the container.

Table 10.1 is important because you can use this information to control the length of your transaction. For example, let's say you want to perform a transfer between two bank accounts. To achieve this, you might have a bank teller session bean that calls into two bank account entity beans. If you deploy all three of these beans with the *Required* transaction attribute, they will all be involved in a single transaction, as shown in Figure 10.8. In this example, assume all three beans are deployed with the *Required* attribute. Notice that

- Transactions always begin and end in the same place. In this case, the Teller EJB object.
- Both Account beans automatically enlist in the Teller's transaction. Thus, we have created a transaction spanning three beans by merely using transaction attributes.

Table 10.1 The Effects of Transaction Attributes

TRANSACTION ATTRIBUTE	CLIENT'S TRANSACTION	BEAN'S TRANSACTION
Required	none	T2
	T1	T1
RequiresNew	none	T2
	T1	T2
Supports	none	none
	T1	T1
Mandatory	none	error
	T1	T1
NotSupported	none	none
	T1	none
Never	none	none
	T1	error

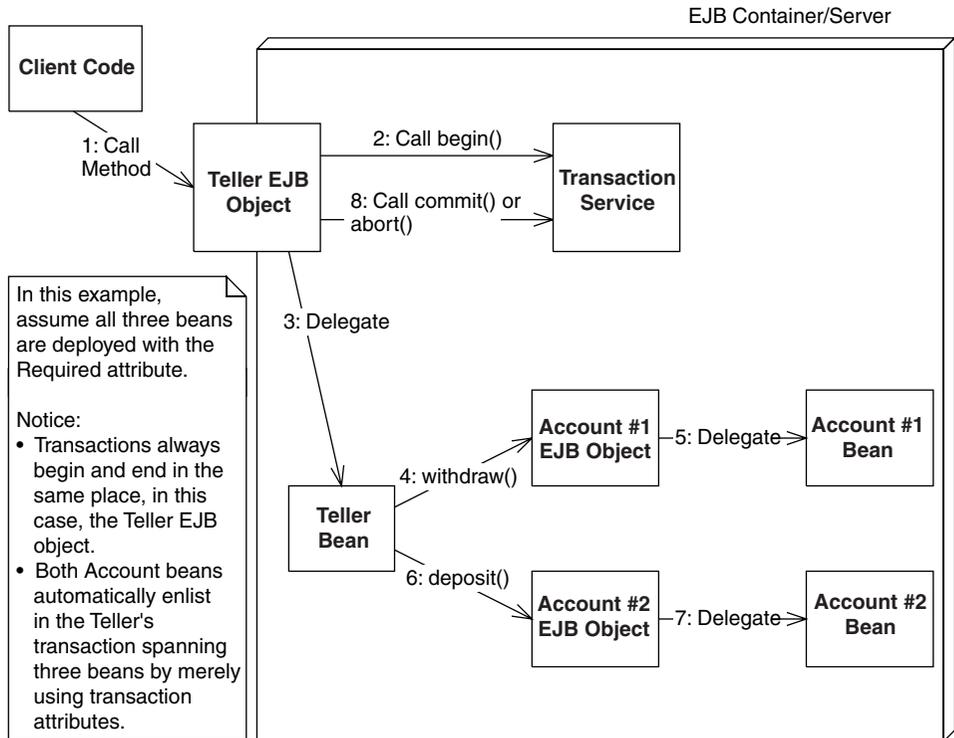


Figure 10.8 Using transaction attributes to control a transaction's length.

Finally, you should note that not all transaction attributes are available for use on all beans. Table 10.2 shows which are permissible.

Here is a brief explanation of why certain transaction attributes are disallowed.

- Entity beans and stateful session beans with *SessionSynchronization* must use transactions. The reason is that both these types of beans are inherently transactional in nature. Entity beans perform database updates, and stateful session beans with *SessionSynchronization* (which we describe later in this chapter) are also transactional. Therefore you normally can't use the following attributes: *Never*, *NotSupported*, *Supports*. Note that the EJB specification *does* allow for containers to *optionally* support these attributes—but only if you're using non-transactional data stores—and with the warning that if you use this, your beans will not be portable, and you may find that you receive inconsistent results.

Table 10.2 Permissible Transaction Attributes for Each Bean Type

TRANSACTION ATTRIBUTE	STATELESS SESSION BEAN	STATEFUL SESSION BEAN IMPLEMENTING		
		SESSION SYNCHRONIZATION	ENTITY BEAN	MESSAGE- DRIVEN BEAN
Required	Yes	Yes	Yes	Yes
RequiresNew	Yes	Yes	Yes	No
Mandatory	Yes	Yes	Yes	No
Supports	Yes	No	No	No
NotSupported	Yes	No	No	Yes
Never	Yes	No	No	No

A client does not call a message-driven bean directly; rather, message-driven beans read messages off a message queue in transactions separate from the client's transaction. There is no client, and therefore transaction attributes that deal with the notion of a client's transaction make no sense for message-driven beans—namely *Never*, *Supports*, *RequiresNew*, and *Mandatory*.

Programmatic Transactions in EJB

Next let's discuss how you can control transactions programmatically in EJB. Programmatic transactions allow for more advanced transaction control than declarative transactions, but they are trickier to use. To control transaction boundaries yourself, you must use the Java Transaction API (JTA). We begin by taking a look at how the JTA was established.

CORBA's Object Transaction Service (OTS)

When we described the ACID properties earlier in this chapter, we mentioned that many parties, such as an enterprise bean and a database driver, can participate in a transaction. This is really an extension to the basic ACID properties, and it's the primary reason that Object Management Group (OMG) developed a standardized *Object Transaction Service* (OTS) as an optional CORBA service. OTS improved on earlier transaction systems that didn't support multiple parties participating in a transaction.

OTS is a suite of well-defined interfaces that specify how transactions can run behind the scenes—interfaces that the transaction manager, resource manager,

and transactional objects use to collaborate. OTS is decomposed into two parts: *CosTransactions* and *CosTSPortability*.

The *CosTransactions* interfaces are the basic interfaces that transactional objects/components, resources, resource managers, and transaction managers use to interoperate. These interfaces ensure that any combination of these parties is possible.

The *CosTSPortability* interface offers a portable way to perform transactions with many participants.

The inner workings of OTS are not relevant to the development of enterprise beans. As an EJB programmer, you should need to think only about writing your application, not about low-level transaction services. This is how EJB achieves rapid application development; you can write a distributed server-side application without understanding complex middleware APIs. EJB shields you from transaction services such as OTS.

The Java Transaction Service (JTS)

Sun realized that you, as an application developer, should not care about most of OTS. Only system-level vendors need to be concerned with the inner workings of OTS. Part of OTS is very applicable to you, however, because it allows you to demarcate transaction boundaries programmatically. Hence, Sun has split up OTS into two sub-APIs: the *Java Transaction Service* (JTS) and the *Java Transaction API* (JTA).

The *Java Transaction Service* (JTS) is a Java mapping of CORBA OTS for system-level vendors. JTS defines the interfaces used by transaction managers and resource managers behind the scenes. It is used to have various vendors' products interoperate. It also defines various objects passed around and used by transaction managers and resource managers. As an application programmer, you should not care about most of OTS, and you should not care about JTS at all. What you should care about is the Java Transaction API (JTA).

The Java Transaction API (JTA)

The *Java Transaction API* (JTA) is a transaction API used by component and application developers. You can use the JTA in your client and bean code to programmatically control transactional boundaries. The JTA package is a standard Java extension, so the package is automatically downloaded if needed.

You can do very useful things with the JTA, such as start a transaction inside your bean, call other beans that also are involved in a transaction, and control whether things commit or abort. Nonbeans can use the JTA as well—the client

code that calls your beans can use the JTA to control transaction boundaries in a workflow scenario, where the client code is calling multiple beans and wants each bean to participate in one transaction.

JTA consists of two sets of interfaces: one for *X/Open XA* resource managers (which we don't need to worry about) and one that we will use to support programmatic transaction control. The interface you use to programmatically control transactions is *javax.transaction.UserTransaction*.

javax.transaction.UserTransaction

The *javax.transaction.UserTransaction* interface allows you to programmatically control transactions. Here is what the *javax.transaction.UserTransaction* interface looks like:

```
public interface javax.transaction.UserTransaction {
    public void begin();
    public void commit();
    public int getStatus();
    public void rollback();
    public void setRollbackOnly();
    public void setTransactionTimeout(int);
}
```

As you can see, six methods are exposed by the *UserTransaction* interface. Three of them—*begin*, *commit*, and *rollback*—are used to begin a new transaction, commit a transaction permanently, and roll back a transaction in case some problem occurred, respectively. The JTA methods are in Table 10.3.

JTA also defines a number of constants that indicate the current status of a transaction. You might see these constants when you call the *UserTransaction.getStatus()* method:

```
public interface javax.transaction.Status {
    public static final int STATUS_ACTIVE;
    public static final int STATUS_NO_TRANSACTION;
    public static final int STATUS_MARKED_ROLLBACK;
    public static final int STATUS_PREPARING;
    public static final int STATUS_PREPARED;
    public static final int STATUS_COMMITTING;
    public static final int STATUS_COMMITTED;
    public static final int STATUS_ROLLING_BACK;
    public static final int STATUS_ROLLEDBACK;
    public static final int STATUS_UNKNOWN;
}
```

Table 10.4 explains the values of those constants.

Table 10.3 The *javax.transaction.UserTransaction* Methods for Transactional Boundary Interaction

METHOD	DESCRIPTION
<code>begin()</code>	Begins a new transaction. This transaction becomes associated with the current thread.
<code>commit()</code>	Runs the two-phase commit protocol on an existing transaction associated with the current thread. Each resource manager will make its updates durable.
<code>getStatus()</code>	Retrieves the status of the transaction associated with this thread.
<code>rollback()</code>	Forces a rollback of the transaction associated with the current thread.
<code>setRollbackOnly()</code>	Calls this to force the current transaction to roll back. This will eventually force the transaction to abort.
<code>setTransactionTimeout(int)</code>	The <i>transaction timeout</i> is the maximum amount of time that a transaction can run before it's aborted. This is useful to avoid deadlock situations, when precious resources are being held by a transaction that is currently running.

Table 10.4 The *javax.transaction.Status* Constants for Transactional Status

CONSTANT	MEANING
<code>STATUS_ACTIVE</code>	A transaction is currently happening and is active.
<code>STATUS_NO_TRANSACTION</code>	No transaction is currently happening.
<code>STATUS_MARKED_ROLLBACK</code>	The current transaction will eventually abort because it's been marked for rollback. This could be because some party called <i>UserTransaction.setRollbackOnly()</i> .
<code>STATUS_PREPARING</code>	The current transaction is preparing to be committed (during Phase One of the two-phase commit protocol).
<code>STATUS_PREPARED</code>	The current transaction has been prepared to be committed (Phase One is complete).
<code>STATUS_COMMITTING</code>	The current transaction is in the process of being committed right now (during Phase Two).
<code>STATUS_COMMITTED</code>	The current transaction has been committed (Phase Two is complete).
<code>STATUS_ROLLING_BACK</code>	The current transaction is in the process of rolling back.
<code>STATUS_ROLLEDBACK</code>	The current transaction has been rolled back.
<code>STATUS_UNKNOWN</code>	The status of the current transaction cannot be determined.

Declarative versus Programmatic Transactions Example

We now show you how to write an enterprise bean in two equivalent ways: using programmatic (or bean-managed) transactions and using declarative (or container-managed) transactions. To illustrate this, we'll use a bank account example. This example has a method called *deposit()* that deposits funds into an account. We'll make this method transactional.

The following code illustrates a deposit method using declarative transactions:

```
/**
 * Deposits amt into account.
 */
public void deposit(double amt) throws AccountException {
    System.out.println("deposit(" + amt + ") called.");

    balance += amt;
}
```

A bean using the preceding method relies on the EJB container to demarcate transactional boundaries. Therefore, the bean's deployment descriptor should use a transaction attribute that provides this (such as *Required*, *Mandatory*, or *RequiresNew*). We showed the code for such a deployment descriptor earlier in this chapter.

The following code illustrates the same method using programmatic transactions:

```
/**
 * Deposits amt into account.
 */
public void deposit(double amt) throws AccountException {

    javax.transaction.UserTransaction userTran = null;

    try {
        System.out.println("deposit(" + amt + ") called.");

        userTran = ctx.getUserTransaction();

        userTran.begin();
        balance += amt;
        userTran.commit();
    }
    catch (Exception e) {
```

Doomed Transactions

Dooming a transaction means to force a transaction to abort. You may need to doom a transaction if something goes wrong, such as a database being unavailable or the client sending you bad parameters.

If you're performing programmatic or client-initiated transactions, you are calling the *begin()* and *commit()* methods. You can easily doom a transaction by calling *rollback()* on the JTA, rather than *commit()*. But how can you doom a transaction if you are participating in a transaction that someone else started? This can occur in one of two cases:

1) Your transaction participant is an EJB component using declarative transactions. The container then starts and ends transactions on your behalf. To instruct the container to abort the transaction, your first instinct might be to throw an exception and expect the container to abort the transaction. But this approach will not work in all cases, because if you are throwing your own custom exception classes, the container has no way of knowing whether the exception is critical enough to indicate a failed transaction and will not abort the transaction. The best way to doom a transaction from a bean with container-managed transactions is to call *setRollbackOnly()* on your *EJB context* object, which we introduced in Chapter 3.

2) Your transaction participant is not an EJB component, such as a Java object. You can doom a transaction by looking up the JTA and calling the JTA's *setRollbackOnly()* method, shown in Table 10.3.

Dooming transactions brings up an interesting side discussion. Imagine you have 10 beans in a chain executing in the same transaction, and bean 2 decides to doom the transaction by calling *setRollbackOnly()*. Why should beans 3 through 10 perform their work if the transaction is doomed to failure anyway? After all, those beans might be performing CPU- or database-intensive operations, and this work will all be wasted when the transaction aborts. The solution is that your beans can *detect* doomed transactions and avoid performing work when a doomed transaction exists. You can detect doomed transactions as follows:

Container-managed transactional beans can detect doomed transactions by calling the *getRollbackOnly()* method on the EJB context object. If this method returns *true*, the transaction is doomed.

Other participants, such as bean-managed transactional beans, can call the JTA's *getStatus()* method, as described in Table 10.3.

You should write code to detect doomed transactions if you expect a good number of transactions to roll back and are performing intensive operations.

```
if (userTran != null) userTran.rollback();
throw new AccountException("Deposit failed because of " +
    e.toString());
}
}
```

Here, we are controlling the transactional boundaries explicitly in code. We first retrieve the JTA from our bean's EJB context object. Then, rather than relying on the EJB container to *begin* and *commit* transactions, we perform these steps ourselves. A bean using the preceding method should be deployed with the deployment descriptor `<transaction-type>` of *Bean*, because the bean is performing its own transaction boundary demarcation.

Take a look at the size difference between the two sets of source code. Bean-managed transactions clutter your source code because you need to write to a transaction API. Container-managed transactions allow you to elegantly write application code and externalize all transaction logic to the container. This is analogous to how we saw entity beans with container-managed persistence as much smaller than those with bean-managed persistence in Chapter 7.



When using programmatic transactions, always try to complete your transactions in the same method that you began them. Doing otherwise results in spaghetti code where it is difficult to track the transactions; the performance decreases because the transaction is held open longer, and the behavior of your system may be odd. See the EJB specification for more details about what the container will do if your transaction is left open.

Transactions from Client Code

The last way you can control transactions is from client code (with the word *client* here meaning anything that calls into your beans, even other enterprise beans). You use the Java Transaction API (JTA) to control transactions from client code.

To control transactions from client code, you must lookup the JTA *UserTransaction* interface with the *Java Naming and Directory Interface* (JNDI). JNDI is a generic lookup facility to lookup resources across a network, and it is fully described in Appendix A. The following code illustrates looking up the JTA *UserTransaction* interface from client code using JNDI:

```
try {
    /*
     * 1: Set environment up. You must set the JNDI Initial
     * Context factory, the Provider URL, and any login
     * names or passwords necessary to access JNDI. See
```

```
*    your application server product's documentation for
*    details on their particular JNDI settings.
*/
java.util.Properties env = ...

/*
 * 2: Get the JNDI initial context
 */
Context ctx = new InitialContext(env);

/*
 * 3: Look up the JTA UserTransaction interface
 *    via JNDI. The container is required to
 *    make the JTA available at the location
 *    java:comp/UserTransaction.
 */
userTran = (javax.transaction.UserTransaction)
    ctx.lookup("java:comp/UserTransaction");

/*
 * 4: Execute the transaction
 */
userTran.begin();

// perform business operations

userTran.commit();
}
catch (Exception e) {
    // deal with any exceptions, including ones
    // indicating an abort.
}
}
```

When you demarcate transactional boundaries in client code, you should be *very* careful. Always strive to keep your transactions as short in duration as possible. Longer-lived transactions result in multiuser performance grinding to a halt. If you need a long transaction (that lasts for minutes, hours, or days) use a distributed locking mechanism, such as the CORBA locking service. Unfortunately, no distributed locking service equivalent currently exists in the Java 2 Platform, Enterprise Edition.

Transactional Isolation

Now that you've seen how to enlist enterprise beans in transactions, let's discuss the *I* in ACID: isolation. Isolation is the guarantee that concurrent users are isolated from one another, even if they are touching the same database data. Isolation is important to understand because it does not come for free. As

we'll see, you can control how isolated your transactions are from one another. Choosing the right level of isolation is critical for the robustness and scalability of your deployment.

The underlying transaction system achieves isolation by performing *concurrency control* behind the scenes. We elaborate on this concept in the following section.

The Need for Concurrency Control

Let's begin our isolation discussion with a motivational example. Imagine there are two instances of the same component executing concurrently, perhaps in two different processes or two different threads. Let's assume that the component wants to update a shared database using a database API such as JDBC or SQL/J. Each of the instances of the component performs the following steps:

1. Read an integer X from a database.
2. Add 10 to X .
3. Write the new value of X to the database.

If each of these three steps executes together in an atomic operation, everything is fine. Neither instance can interfere with the other instance's operations. Remember, though, that the thread-scheduling algorithm being used in the background does not guarantee this. If two instances are executing these three operations, the operations could be interleaved. The following order of operations is possible:

1. Instance A reads integer X from the database. The database now contains $X = 0$.
2. Instance B reads integer X from the database. The database now contains $X = 0$.
3. Instance A adds 10 to its copy of X and persists it to the database. The database now contains $X = 10$.
4. Instance B adds 10 to its copy of X and persists it to the database. The database now contains $X = 10$.

What happened here? Due to the interleaving of database operations, instance B is working with a stale copy of X : The copy before instance A performed a write. Thus, instance A's operations have been lost! This famous problem is known as a *lost update*. It is a very serious situation—instance B has been working with stale data and has overwritten instance A's write. How can transactions avoid this scenario?

The solution to this problem is to use *locking* on the database to prevent the two components from reading data. By locking the data your transaction is using, you guarantee that your transaction and only your transaction has access to that data until you release that lock. This prevents interleaving of sensitive data operations.

In our scenario, if our component acquired an exclusive lock before the transaction began and released that lock after the transaction, then no interleaving would be possible.

1. Request a lock on *X*.
2. Read an integer *X* from a database.
3. Add 10 to *X*.
4. Write the new value of *X* to the database.
5. Release the lock on *X*.

If another component ran concurrently with ours, that component would have to wait until we relinquished our lock, which would give that component our fresh copy of *X*. We explore locking further in the Isolation and Locking sidebar.

Isolation and EJB

As an EJB component developer, you can control how isolated your transactions are from one another. You can enforce strict isolation or allow relaxed isolation. If you have very strict isolation, you can rest assured that each concurrent transaction will be isolated from all other transactions. But sometimes enforcing strict isolation is a hindrance rather than a benefit. Because isolation is achieved by acquiring locks on an underlying data storage, the locks can result in unacceptable performance degradation.

Thus, you need to be smart about how much isolation you really need. *Isolation levels* give you a choice over how much isolation you want and allow you to specify concurrency control at a very high level. If you specify a very strict isolation level, then your transactions will be perfectly isolated from one another, at the expense of performance. If you specify a very loose isolation level, your transactions will not be isolated, but you will achieve higher concurrent transaction performance.

There are four transaction isolation levels:

The *READ UNCOMMITTED* mode does not offer any isolation guarantees but offers the highest performance.

The *READ COMMITTED* mode solves the *dirty read* problem.

Isolation and Locking

During a transaction, a number of *locks* are acquired on the resource being updated. These locks are used to ensure isolation: Multiple clients all updating the same data set cannot interfere with each other. The locks are implicitly retrieved when you interact with resource managers—you do not have to worry about obtaining them yourself.

By intelligently acquiring locks on the resource being used, transactions guarantee a special property: *serializability*. Serializability means that a suite of concurrently executing transactions behaves as if the transactions were executing one after another (nonconcurrently). This is guaranteed no matter how scheduling of the transactions is performed.

The problem with locking is that it physically locks out other concurrent transactions from performing their database updates until you release your locks. This can lead to major performance problems. In addition, a *deadlock* scenario (not specific to databases, by the way) can arise. Deadlock causes the entire system to screech to a dead stop. An example of deadlock occurs when two concurrent transactions are both waiting for each other to release a lock.

To improve performance, transactions distinguish between two main types of locks: *read locks* and *write locks*. Read locks are nonexclusive, in that any number of concurrent transactions can acquire a read lock. In comparison, write locks are exclusive—only one transaction can hold a write lock at any time.

Locking exists in many circles: databases, Version Control Systems, and the Java language itself (through the *synchronized* keyword). The problems experienced in locking are common to all arenas. EJB abstracts concurrency control away from application developers via *isolation levels*.

If you would like more details about locking and transactions, check out *Principles of Database Systems* by Jeffrey D. Ullman (Computer Science Press, 1980). This is a classic, theoretical book on databases that forms the basis for many database systems today.

The **REPEATABLE READ** mode solves the previous problem as well as the *unrepeatable read* problem.

The **SERIALIZABLE** mode solves the previous problems as well as the *phantom* problem.

It's important to understand why *dirty reads*, *unrepeatable reads*, and *phantoms* occur, or you won't be able to use transactions properly in EJB. This section gives you the information you need to make an intelligent isolation level choice when programming with transactions.

The Dirty Read Problem

A dirty read occurs when your application reads data from a database that has not been committed to permanent storage yet. Consider two instances of the same component performing the following:

1. You read integer X from the database. The database now contains $X = 0$.
2. You add 10 to X and save it to the database. The database now contains $X = 10$. You have not issued a *commit* statement yet, however, so your database update has not been made permanent.
3. Another application reads integer X from the database. The value it reads in is $X = 10$.
4. You *abort* your transaction, which restores the database to $X = 0$.
5. The other application adds 10 to X and saves it to the database. The database now contains $X = 20$.

The problem here is the other application read your update before you committed. Because you aborted, the database data has erroneously been set to 20; your database update has been added in despite the abort! This problem of reading uncommitted data is a *dirty read*. (The word *dirty* occurs in many areas of computer science, such as caching algorithms. A dirty cache is a cache that is out of sync with the main source.)

READ UNCOMMITTED

Dirty reads can occur if you use the weakest isolation level, called *READ UNCOMMITTED*. With this isolation level, if your transaction is executing concurrently with another transaction, and the other transaction writes some data to the database *without* committing, your transaction will read that data in. This occurs regardless of the isolation level being used by the other transaction.

READ UNCOMMITTED experiences the other transactional problems as well: unrepeatable reads and phantoms. We'll describe those problems in the pages to come.

When to Use READ UNCOMMITTED

This isolation level is dangerous to use in mission-critical systems with shared data being updated by concurrent transactions. It is inappropriate to use this mode in sensitive calculations, such as in a debit/credit banking transaction. For those scenarios, it's better to go with one of the stricter isolation levels we detail later.

This level is most appropriate if you know beforehand that an instance of your component will be running only when there are no other concurrent transactions. Because there are no other transactions to be isolated from, this isolation level is adequate. But for most applications that use transactions, this isolation level is insufficient.

The advantage of this isolation level is performance. The underlying transaction system doesn't have to acquire any locks on shared data in this mode. This reduces the amount of time that you need to wait before executing, and it also reduces the time concurrent transactions waste waiting for you to finish.

READ COMMITTED

The *READ COMMITTED* isolation level is very similar to *READ UNCOMMITTED*. The chief difference is that your code will read committed data only when running in *READ COMMITTED* mode. When you execute with this isolation level, you will *not* read data that has been written but is uncommitted. This isolation level thus solves the dirty read problem.

Note that this isolation level does not protect against the more advanced transactional problems, such as unrepeatable reads and phantoms.

When to Use READ COMMITTED

This isolation level offers a step up in robustness from the *READ UNCOMMITTED* mode. You aren't going to be reading in data that has just been written but is uncommitted, which means that any data you read is going to be consistent data.

One great use for this mode is for programs that read data from a database to report values of the data. Because reporting tools aren't in general mission-critical, taking a snapshot of committed data in a database makes sense.

When you run in *READ COMMITTED* mode, the underlying concurrency control system needs to acquire additional locking. This makes performance slower than with *READ UNCOMMITTED*. *READ COMMITTED* is the default isolation level for most databases, such as Oracle or Microsoft SQL Server.

The Unrepeatable Read Problem

Our next concurrency control problem is an *Unrepeatable Read*. Unrepeatable reads occur when a component reads some data from a database, but upon rereading the data, the data has been changed. This can arise when another concurrently executing transaction modifies the data being read. For example:

1. You read a data set *X* from the database.
2. Another application overwrites data set *X* with new values.
3. You reread the data set *X* from the database. The values have magically changed.

Again, by using transactional locks to lock out those other transactions from modifying the data, we can guarantee that unrepeatable reads will never occur.

REPEATABLE READ

REPEATABLE READ guarantees yet another property on top of *READ COMMITTED*: Whenever you read committed data from a database, you will be able to reread the same data again at a later time, and the data will have the same values as the first time. Hence, your database reads are *repeatable*. In contrast, if you are using the *READ COMMITTED* mode or a weaker mode, another concurrent transaction may commit data between your reads.

When to Use REPEATABLE READ

Use *REPEATABLE READ* when you need to update one or more data elements in a resource, such as one or more records in a relational database. You want to read each of the rows that you're modifying and then be able to update each row, knowing that none of the rows are being modified by other concurrent transactions. If you choose to reread any of the rows at any time later in the transaction, you'd be guaranteed that the rows still have the same data that they did at the beginning of the transaction.

The Phantom Problem

Finally, we have the phantom problem. A phantom is a *new* set of data that magically appears in a database between two database read operations. For example:

1. Your application queries the database using some criteria and retrieves a data set.
2. Another application inserts new data that would satisfy your query.
3. You perform the query again, and *new* sets of data have magically appeared.

The difference between the unrepeatable read problem and the phantom problem is that unrepeatable reads occur when existing data is changed, whereas

phantoms occur when *new* data that didn't exist before is inserted. For example, if your transaction reads a relational record, and a concurrent transaction commits a new record to the database, a new *phantom record* appears that wasn't there before.

SERIALIZABLE

You can easily avoid phantoms (as well as the other problems described earlier) by utilizing the strictest isolation level: *SERIALIZABLE*. *SERIALIZABLE* guarantees that transactions execute serially with respect to each other, and it enforces the isolation ACID property to its fullest. This means that each transaction truly appears to be independent of the others.

When to Use SERIALIZABLE

Use *SERIALIZABLE* for mission-critical systems that absolutely must have perfect transactional isolation. You are guaranteed that no data will be read that has been uncommitted. You'll be able to reread the same data again and again. And mysterious committed data will not show up in your database while you're operating due to concurrent transactions.

Use this isolation level with care because serializability does have its cost. If all of your operations execute in *SERIALIZABLE* mode, you will quickly see how fast your database performance grinds to a halt. (A personal note: Because transactional errors can be very difficult to detect, due to scheduling of processes, variable throughput, and other issues, we subscribe to the view that it's better to be safe than sorry.)

Transaction Isolation Summary

The various isolation levels and their effects are summarized in Table 10.5.

Table 10.5 The Isolation Levels

ISOLATION LEVEL	DIRTY READS?	UNREPEATABLE READS?	PHANTOM READS?
<i>READ UNCOMMITTED</i>	Yes	Yes	Yes
<i>READ COMMITTED</i>	No	Yes	Yes
<i>REPEATABLE READ</i>	No	No	Yes
<i>SERIALIZABLE</i>	No	No	No

Isolation and EJB

Now that you understand isolation in theory, let's see how to set up isolation in an EJB environment.

If your bean is managing transactions, you specify isolation levels with your resource manager API (such as JDBC). For example, you could call `java.sql.Connection.SetTransactionIsolation(. . .)`.

If your container is managing transactions, there is no way to specify isolation levels in the deployment descriptor. You need to either use resource manager APIs (such as JDBC), or rely on your container's tools or database's tools to specify isolation.

If you're using different resource managers within a single transaction, each resource manager can have a different isolation level, yet all run together under a single transaction. Note that any particular resource manager running under a transaction usually requires a single isolation level for the duration of that transaction. This new model has some drawbacks as well, as described in the following sidebar.

Isolation Portability Issues

Unfortunately, there is no way to specify isolation for container-managed transactional beans in a portable way—you are reliant on container and database tools. This means if you have written an application, you cannot ship that application with built-in isolation. The deployer now needs to know about transaction isolation when he uses the container's tools, and the deployer might not know a whole lot about your application's transactional behavior. This approach is also somewhat error-prone, because the bean provider and application assembler need to informally communicate isolation requirements to the deployer, rather than specifying it declaratively in the deployment descriptor.

When we queried Sun on this matter, Mark Hapner, coauthor of the EJB specification, provided this response: "Isolation was removed because the vendor community found that implementing isolation at the component level was too difficult. Some felt that isolation at the transaction level was the proper solution; however, no consensus was reached on a specific replacement semantics.

"This is a difficult problem that unfortunately has no clear solution at this time . . . The best strategy is to develop EJBs that are as tolerant of isolation differences as possible. This is the typical technique used by many optimistic concurrency libraries that have been layered over JDBC and ODBC."

Pessimistic and Optimistic Concurrency Control

The two basic object concurrency control strategies that your EJBs may follow, pessimistic and optimistic, are summarized in Table 10.6. Pessimistic concurrency control is the algorithm we've been assuming throughout this chapter—you acquire a lock the data for the duration of the transaction, ensuring that nobody messes with your data.

With optimistic concurrency control, your EJB component does not hold the lock for the duration of the transaction. Instead, you *hope* everything will be OK. Then if the database detects a collision, the transaction rolls back. The basic assumption behind optimistic concurrency is that because it is unlikely that separate users will access the same object simultaneously, it is better to handle the occasional collision than to limit the size of your system.

Distributed Transactions

Now that we've concluded our discussion of isolation levels, we'll shift gears and talk about *distributed transactions*, which are transactions over a multitier deployment with several transaction participants.

Table 10.6 Comparing Pessimistic and Optimistic Concurrency Control Strategies.

STRATEGY	ADVANTAGES	DISADVANTAGES
Pessimistic—Your EJB locks the source data for the entire time it needs the data, not allowing anything else (at least anything greater than read/view access) to potentially update the data until it completes its transaction.	<ul style="list-style-type: none"> ■ Brute force approach ■ Provides reliable access to data ■ Suitable for small-scale systems ■ Suitable for systems where concurrent access is rare 	<ul style="list-style-type: none"> ■ Does not scale well because it blocks simultaneous access to common resources
Optimistic—Your EJB implements a strategy to detect whether a change has occurred to the source data between the time it was read and the time it now needs to be updated. Locks are placed on the data only for the small periods of time the EJB interacts with the database.	<ul style="list-style-type: none"> ■ Suitable for large system ■ Suitable for systems requiring significant concurrent access 	<ul style="list-style-type: none"> ■ Requires complex code to be written to support collision detection and handling

The most basic flat transaction occurs with a single application server tied to a single database. Depending on the functionality of your application server's transaction service, you may be able to perform *distributed flat transactions* as well. Distributed flat transactions obey the same rules as simple flat transactions: If one component on one machine aborts the transaction, the entire transaction is aborted. But with distributed flat transactions, you can have many different types of resources coordinating in a single transaction across the network. Here are some possible use-cases for which you may need distributed flat transactions.

- You have multiple application servers coordinating in the same transaction.
- You have updates to different databases in the same transaction.
- You are trying to perform a database update and send or receive a message from a message queue in the same transaction.
- You are connecting to a legacy system as well as one or more other types of storage (such as databases, message queues, or other legacy systems) in the same transaction.

Each of these scenarios requires multiple processes or machines to collaborate, potentially across a network, to solve a business problem. Distributed flat transactions allow multiple transaction participants, *written by different vendors*, to collaborate under one transactional hood.

Durability and the Two-Phase Commit Protocol

One important ACID property is durability. Durability guarantees that all resource updates that are committed are made permanent. Durability is easy to implement if you have one storage into which you are persisting. But what if multiple resource managers are involved? If one of your resources undergoes a catastrophic failure, such as a database crash, you need to have a recovery mechanism. How do transactions accomplish this?

One way would be to log all database operations before they actually happen, allowing you to recover from a crash by consulting the log and reapplying the updates. This is exactly how transactions guarantee durability. To accomplish this, transactions complete in two *phases*.

Phase One begins by sending a *before commit* message to all resources involved in the transaction. At this time, the resources involved in a transaction have a final chance to abort the transaction. If any resource involved decides to abort, the entire transaction is cancelled and no resource updates are performed. Otherwise, the transaction proceeds on course and

cannot be stopped, unless a catastrophic failure occurs. To prevent catastrophic failures, all resource updates are written to a transactional log or journal. This journal is persistent, so it survives crashes and can be consulted after a crash to reapply all resource updates.

Phase Two occurs only if Phase One completed without an abort. At this time, all of the resource managers, which can all be located and controlled separately, perform the actual data updates.

The separation of transaction completion into two phases is called the *two-phase commit protocol* or *2PC*. The two-phase commit protocol is useful because it allows for many transaction managers and resource managers to participate in a transaction across a deployment. If any participant votes that the transaction should abort, all participants must roll back.

In the distributed two-phase commit, there is one master transaction manager called the *distributed transaction coordinator*. The transaction coordinator runs the show and coordinates operations among the other transaction managers across the network. The following steps occur in a distributed two-phase commit transaction:

1. The transaction coordinator sends a *prepare to commit* message to each transaction manager involved.
2. Each transaction manager may propagate this message to the resource managers that are tied to that transaction manager.
3. Each transaction manager reports back to the transaction coordinator. If everyone agrees to commit, the commit operation that's about to happen is logged in case of a crash.
4. Finally, the transaction coordinator tells each transaction manager to commit. Each transaction manager in turn calls each resource manager, which makes all resource updates permanent and durable. If anything goes wrong, the log entry can be used to reapply this last step.

This process is shown in Figure 10.9.

The Transactional Communications Protocol and Transaction Contexts

A distributed two-phase commit transaction complicates matters, because the transaction managers must all agree on a standard mechanism of communicating. Remember that each of the participants in a distributed transaction may have been written by a different vendor, such as a deployment with heterogeneous application servers. The communication mechanism used is called

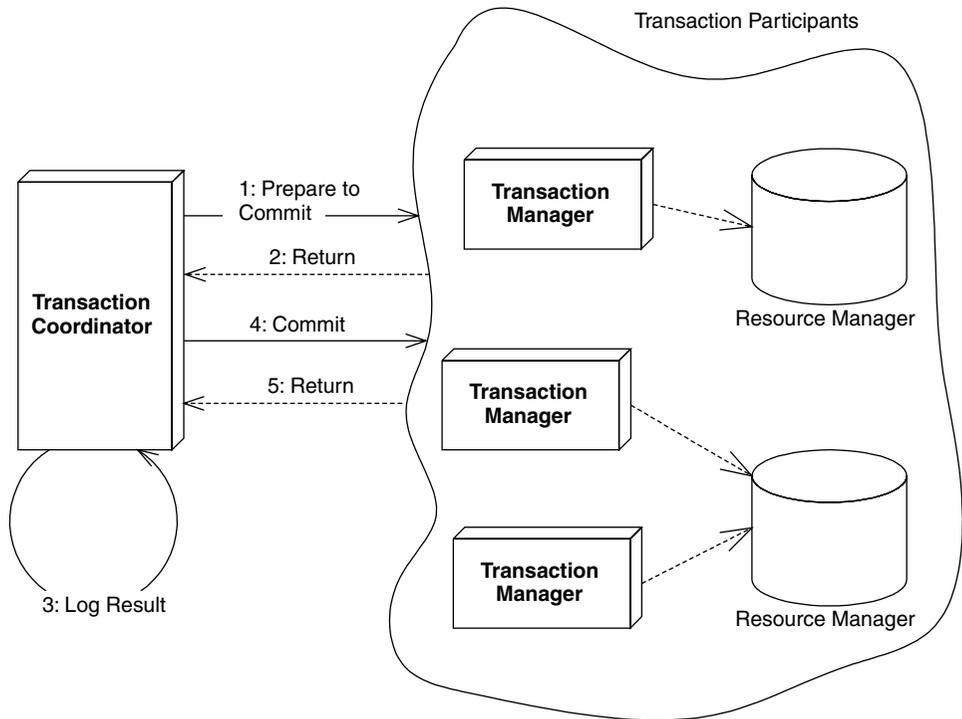


Figure 10.9 A distributed flat transaction using a two-phase commit protocol.

the *transactional communications protocol*. An example of such a protocol is the *Internet Inter-ORB Protocol (IIOP)*, which we describe in Appendix B.

The most important piece of information sent over the transactional communications protocol is the *transaction context*. A transaction context is an object that holds information about the system's current transactional state. It is passed around among parties involved in transactions. By querying the transaction context, you can gain insight into whether you're in a transaction, what stage of a transaction you are at, and other useful data. For any component to be involved in a transaction, the current thread in which the component is executing must have a transaction context associated with it.



The EJB specification suggests, but does not require, that application server vendors support on-the-wire transaction context interoperability. If an application server does support interoperable transactions, EJB requires that it leverage the transaction context propagation facilities built in to CORBA's Object Transaction Service (OTS) and the Internet Inter-ORB Protocol (IIOP). Application servers that use these technologies should be interoperable and run in a distributed 2PC transaction.

Since the EJB specification does not require this level of interoperability, application servers from different vendors cannot be guaranteed to work together and participate in a distributed two-phase commit transaction, because they may not be able to communicate in a standard way.

For most users, this is acceptable because distributed 2PC has poor performance. And more to the point, most organizations struggle enough as it is with a single application server vendor.

It's important to understand which communications protocol your application server uses. If you want to perform a distributed two-phase commit transaction, the transaction participants must agree on a standard protocol.

Designing Transactional Conversations in EJB

In this chapter we've seen that a transactional abort entails an automatic roll-back of database updates that were performed during the transaction. But database updates are only half of the picture. Your application code needs to consider the impacts of a failed transaction as well.

When a transaction aborts, your application code has several choices. You can abort your business process and throw an exception back to the client, or you can attempt to retry the transaction several times. But unfortunately, your application cannot sit in a loop retrying transactions forever, as that would yield horrible performance for concurrent threads of execution. If the transaction cannot eventually be made to succeed, you should consider aborting your business process.

For a stateless session bean, aborting a business process is a simple task—simply throw an exception back to the client. But for a stateful session bean, things are a bit trickier. Stateful session beans represent business processes that span multiple method calls and hence have in-memory *conversational state*. Tossing away that conversation and throwing an exception to the client could entail a significant amount of lost work.

Fortunately, a well-designed stateful session bean can salvage its conversations in the case of failed transactions. The key is to design your beans to be aware of changes to conversational state and to be smart enough to undo any of those changes if a transactional abort occurs.

Because this process is highly application-specific, your application server cannot automate this task for you. Your application server *can* aid you in determining when a transaction failed, enabling you to take application-specific steps. If your session bean needs to be alerted to transaction status (like failed

transactions), your enterprise bean class can implement an optional interface called *javax.ejb.SessionSynchronization*, shown in the following code:

```
public interface javax.ejb.SessionSynchronization
{
    public void afterBegin();
    public void beforeCompletion();
    public void afterCompletion(boolean);
}
```

You should implement this interface in your enterprise bean class and define your own implementations of each of these methods. The container will call your methods automatically at the appropriate times during transactions, alerting you to important transactional events. This adds to the existing arsenal of alerts that your session beans receive already—life-cycle alerts via *ejbCreate()* and *ejbRemove()*, passivation alerts via *ejbActivate()* and *ejbPassivate()*, and now transactional alerts via *afterBegin()*, *beforeCompletion()*, and *afterCompletion()*.

Here's what each of the *SessionSynchronization* methods do:

afterBegin() is called by the container directly after a transaction begins.

beforeCompletion() is called by the container right before a transaction completes.

afterCompletion() is called by the container directly after a transaction completes.

The key method that is most important for rolling back conversations is *afterCompletion()*. The container calls your *afterCompletion()* method when a transaction completes either in a commit or an abort. You can figure out whether a commit or an abort happened by the Boolean parameter that gets passed to you in *afterCompletion()*: True indicates a successful commit, false indicates an abort. If an abort happened, you should roll back your conversational state to preserve your session bean's conversation.

Here's an example of *afterCompletion()* in action:

```
public class CountBean implements SessionBean, SessionSynchronization {

    public int val;
    public int oldVal;

    public void ejbCreate(int val) {
        this.val=val;
        this.oldVal=val;
    }
}
```

```
public void afterBegin() { oldVal = val;}
public void beforeCompletion() {}
public void afterCompletion(boolean b) { if (b == false) val = oldVal; }

public int count() { return ++val; }
public void ejbRemove() {}
public void ejbActivate() {}
public void ejbPassivate() {}
public void setSessionContext(SessionContext ctx) {}
}
```

This is a new version of our `count` bean from Chapter 4. The conversational state is `val`, an integer that increases incrementally whenever `count()` is called. We also keep a backup copy of `val`, called `oldVal`, which we revert back to in case of transactional rollback. Here is what's going on:

1. When our bean is first initialized in `ejbCreate()`, or when a transaction first begins in `afterBegin()`, `val` and `oldVal` are set to the same value.
2. One or more `count()` business methods are called, incrementing `val`.
3. If the transaction fails, the `afterCompletion()` method is called when the transaction completes. If the transaction failed (that is, if a `false` value was passed into `afterCompletion()`), we roll back our conversational state by reverting back to `oldVal`.

Note that for this to work, we must make `count()` transactional in the deployment descriptor using transaction attributes that we described earlier in this chapter.

`SessionSynchronization` is also useful when your stateful session bean caches database data in memory during a transaction. You can use `SessionSynchronization` to track when to cache and when not to cache data as follows.

When the container calls `afterBegin()`, the transaction has just started. You should read in any database data you want to cache in your stateful session bean.

When the container calls `beforeCompletion()`, the transaction has ended. Write out any database data you've cached.



You can implement `SessionSynchronization` only if you're using a stateful session bean with declarative (container-managed) transactions. If your bean is using programmatic (bean-managed) transactions, you are already in control of the transaction because you issue the `begin()`, `commit()`, and `abort()` statements. Stateless session beans do not hold conversations and hence do not need these callbacks.

Summary

Whew! That's a lot of data to digest. You may want to reread this chapter later to make sure you've grasped all the concepts.

In this chapter, we learned about transactions and how they can make a server-side deployment robust. We saw the virtues of transactions, which are called the ACID properties. We looked at different transactional models, including flat and nested transactions.

We then applied this transactional knowledge to EJB. We saw how declarative, programmatic, and client-initiated transactions are useful in EJB and learned how to code with each model. We looked at transaction isolation levels and understood the problems that each level solves. Finally, we learned about distributed transactions and the two-phase commit protocol, and ended with a look at writing transactional conversations.

Reading this chapter will prove well worth the effort, because now you have a wealth of knowledge about the importance and usefulness of transactions in EJB. You should definitely return to this chapter frequently when you're creating transactional beans.

BMP and CMP Relationships

In previous chapters, we looked at how to build entity beans using BMP and CMP. In this chapter, we'll heat things up and learn about *relationships* between data. Examples of relationships include an order having one or more line items, a student registering for a course, and a person having an address. These relationships need to be defined and maintained for advanced data models.

In this chapter, we'll learn about the following relationship topics:

- Cardinality
- Directionality
- Aggregation vs. composition and cascading deletes
- Recursive, circular, and lazily-loaded relationships
- Referential integrity
- Accessing relationships from client code via collections
- How to implement each of the above topics using *both* CMP and BMP

If these concepts are new to you, don't worry—you'll be an expert on them shortly.



To help you understand the concepts and to keep things brief, we'll use a bit of pseudo-code in this chapter. If you would like a complete example of code that you can copy and paste into your deployment illustrating relationships, download the book's source code from the accompanying Web site.

The CMP and BMP Difference

Relationships in EJB are implemented quite differently for CMP and for BMP. BMP entity beans manage relationships explicitly in the bean itself. You need to write a good deal of scaffolding code to maintain the relationship. At the high level, your BMP entity bean looks like this:

```
public class OrderBean implements EntityBean {
    // private fields
    // get/set methods
    // code to manage relationships in ejbXXX methods
}
```

With CMP, you *declare* how you would like your relationships to work in your deployment descriptor. The container then *generates* all the relationship code you need when the container subclasses your entity bean. At the high level, your deployment descriptor looks like this:

```
<ejb-jar>

<enterprise-beans>
    ... define enterprise beans ...
</enterprise-beans>

<relationships>
    ... define EJB relationships ...
</relationships>

</ejb-jar>
```

Let's explore what goes into the comments above by tackling each relationship topic in detail.

Cardinality

Our first relationship topic is *cardinality*. Cardinality specifies how many instances of data can participate in a relationship. There are three flavors of cardinality:

One-to-one (1:1) relationships, such as the relationship between an employee and his home address. Each employee has exactly one home address, and each home address has exactly one employee.

One-to-many (1:N) relationships, such as the relationship between a manager and his employees. Each manager can have many employees working for him, but each employee can have only one manager.

Implementing Relationships in Session Beans

Session beans can perform persistence that involves relationships, just like CMP and BMP entity beans can. If you are familiar with traditional procedural programming, Microsoft programming, or programming involving servlets or JSPs talking straight to a database via JDBC, the session bean approach is quite analogous.

You can use a *stateful session bean* just like an entity bean; the only difference is that with a stateful session bean, you expose methods to a client for loading and storing data, and the client controls when the bean is loaded and stored by calling those methods. In this case, all of the best practices for relationship persistence that apply to BMP entity beans apply to stateful session beans that use JDBC.

You can also use a *stateless session bean* to perform persistence that involves relationships. Stateless session beans do not hold state and therefore do not have an identity, so you can't treat a stateless session bean like an entity bean. You need to use the stateless session bean as a *service* to read and write rows to and from the database, marshaling the state back to the client on each method call. In essence, the stateless session bean serves as a stateless persistence engine, and the relationship code needs to be custom coded.

In general, if you have complex relationships, we do not recommend the session bean approach, due to all the manual coding. The entity bean value proposition really shines through when it comes to relationships.

Many-to-many (M:N) relationships such as the relationship between an employee and an e-mail list. Each employee can be subscribed to many email lists, and each email list can have many employees subscribed.



Just to get you thinking:

Why don't we talk about many-to-one relationships?

Figure 11.1 depicts the three flavors of cardinality visually. Let's look at how to code each type of relationship, for both BMP and CMP.

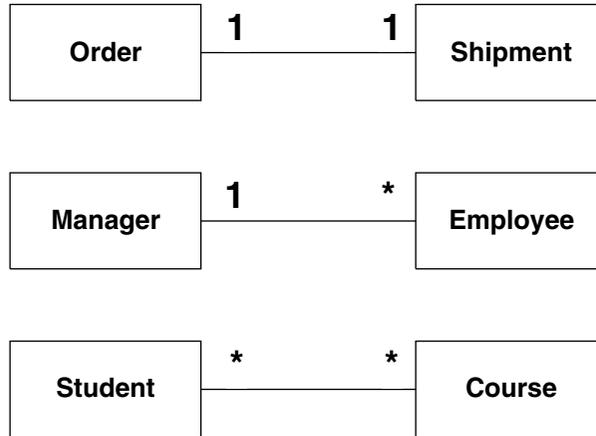


Figure 11.1 The three flavors of cardinality.

1:1 Relationships

In a one-to-one relationship, each constituent can have at most one relationship with the other constituent. Examples of one-to-one relationships include:

- Person:Address
- Car:Windshield
- Order:Shipment

1:1 relationships are typically set up by a *foreign key* relationship in the database. Figure 11.2 shows a possible database setup.

In Figure 11.2, the order has a relationship with a shipment. The order table has a foreign key, which is the shipment table's primary key. This foreign key is the link between the two tables. Note that this isn't the only way to setup a one-to-one relationship. You could also have the shipment point to the order.

Implementing 1:1 Relationships Using BMP

The following code shows how to implement a 1:1 relationship using BMP:

```

public class OrderBean implements EntityBean {
    private String orderPK;
    private String orderName;
    private Shipment shipment; // EJB local object stub

    public Shipment getShipment() { return shipment; }
  }

```

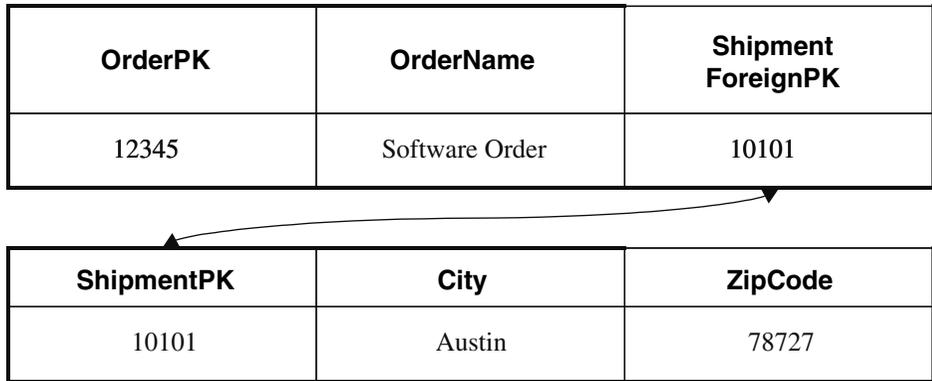


Figure 11.2 A possible one-to-one cardinality database schema.

```

public void setShipment(Shipment s) { this.shipment = s;}

...

public void ejbLoad() {
    // 1: SQL SELECT Order. This also retrieves the
    //      shipment foreign key.
    //
    // 2: JNDI lookup of ShipmentHome
    //
    // 3: Call ShipmentHome.findByPrimaryKey(), passing
    //      in the shipment foreign key
}
public void ejbStore() {
    // 1: Call shipment.getPrimaryKey() to retrieve
    //      the Shipment foreign key
    //
    // 2: SQL UPDATE Order. This also stores the
    //      Shipment foreign key.
}
}

```

As with all BMP entity beans, we must define our SQL statements in our bean. See Chapter 5 for more on this. The special relationship management code is in **bold**.

The relationship management code is only necessary at the instant we transform our bean to-and-from relational data. It is necessary because we can't just persist a stub, like we can with our other fields (such as a String). If we did persist a stub, and (by some miracle) that worked, it would look like a bit-blob in the foreign key column. That bit-blob foreign key would not match up to the primary key in the shipment table.

Here is an explanation for what's happening in our bean:

Our *ejbLoad()* method loads the database data of the order, and part of that data is a foreign key to the shipment. We need to transform that foreign key into a stub to a shipment bean. Therefore we need to perform a JNDI lookup of the shipment home and then call a finder method, passing in the foreign key. This gives us a stub, and we can then call business methods on the shipment.

Our *ejbStore()* method stores the database data for the order, and part of that data is a foreign key to the shipment. We need to transform the shipment stub into a foreign key. Therefore we need to call *getPrimaryKey()* on the shipment stub. This gives us our foreign key, and we can then perform the SQL.

Implementing 1:1 Relationships Using CMP

The following code shows how to implement that same 1:1 relationship using CMP:

```
public abstract class OrderBean implements EntityBean {
    // no fields

    public abstract Shipment getShipment();
    public abstract void setShipment(Shipment s);

    ...

    public void ejbLoad() {} // Empty
    public void ejbStore() {} // Empty
}
```

As with all CMP entity beans, we define our get/set methods as abstract methods and have no fields. The container implements these methods (and defines the fields) when the container subclasses our bean (see Chapter 6).

What's exciting is that our *ejbLoad()* and *ejbStore()* methods are free of any scaffolding code because the container generates all that scaffolding code for us. We do need to specify the relationship in the deployment descriptor, and we do so as follows:

```
<ejb-jar>

<enterprise-beans>
...
</enterprise-beans>

<relationships>
```

```
<!-- This declares a relationship -->
<ejb-relation>

  <!-- The nickname we're giving this relationship -->
  <ejb-relation-name>Order-Shipment</ejb-relation-name>

  <!--
  This declares the 1st half of the relationship
  (the Order side)
  -->
  <ejb-relationship-role>

    <!-- The nickname we're giving this half of the relationship -->
    <ejb-relationship-role-name>
      order-spawns-shipment
    </ejb-relationship-role-name>

    <!-- The Cardinality of this half of the relationship -->
    <multiplicity>One</multiplicity>

    <!--
    The name of the bean corresponding to this
    half of the relationship
    -->
    <relationship-role-source>
      <ejb-name>Order</ejb-name>
    </relationship-role-source>

    <!--
    Recall that a CMP entity bean has an abstract get/set
    method for the relationship. We need to tell the
    container which get/set method corresponds to this
    relationship, so that the container can generate the
    appropriate scaffolding code when subclassing our bean.
    That is the purpose of this element, which is called the
    container-managed relationship (CMR) field. The value
    of the CMR field should be the name of your get/set
    method, but without the get/set, and with a slight
    change in capitalization. getShipment() becomes shipment.
    -->
    <cmr-field><cmr-field-name>shipment</cmr-field-name></cmr-field>
  </ejb-relationship-role>

  <!--
  This declares the 2nd half of the relationship
  (the Shipment side)
  -->
  <ejb-relationship-role>
    <ejb-relationship-role-name>
```

```

        shipment-fulfills-order
    </ejb-relationship-role-name>
    <multiplicity>One</multiplicity>
    <relationship-role-source>
        <ejb-name>Shipment</ejb-name>
    </relationship-role-source>
    </ejb-relationship-role>
</ejb-relation>
</relationships>

</ejb-jar>

```

The deployment descriptor should be self-explanatory. Once we write the proprietary descriptor that maps CMP fields to columns, we will have supplied enough information to the container that its tools should be able to generate any necessary relationship code, such as the code we saw in the BMP example.

1:N Relationships

A one-to-many relationship is one of the more common relationships you'll see in your object model. This is because one-to-one relationships will often be combined into a single data object, rather than having a relationship between two separate data objects. Examples of one-to-many relationships include:

- Order:LineItems
- Customer:Orders
- Company:Employees

1:N relationships are also typically set up by a foreign key relationship in the database. Figure 11.3 shows a possible database setup.

In Figure 11.3, the company has a relationship with many employees. The company has a vector of line-item foreign keys, stored as a bit-blob in the database. We need a vector because we have a relationship with many employees, not just one employee.

The approach shown in Figure 11.3 is not ideal, because it's very nasty to deal with bit-blobs in the database. Queries and reporting become challenging, as databases were not meant to handle relationships in this way. Figure 11.4 shows an alternative.

In Figure 11.4, each employee has a foreign key, which is the company table's primary key. Thus, the employees are pointing back to their company. This may seem backwards if we want to get from the company to the employees. It works, however, because the database doesn't care—it is a flat structure without a sense of direction. You can still construct queries that get from the company to employees.

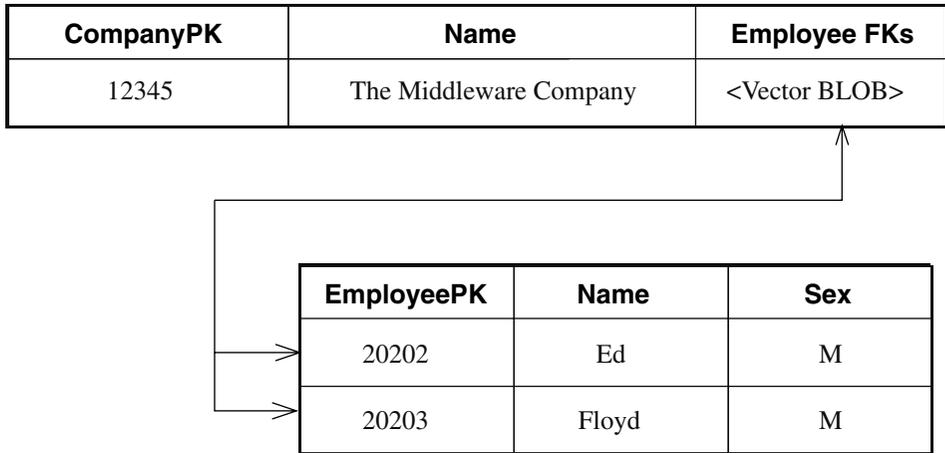


Figure 11.3 A possible one-to-many cardinality database schema.

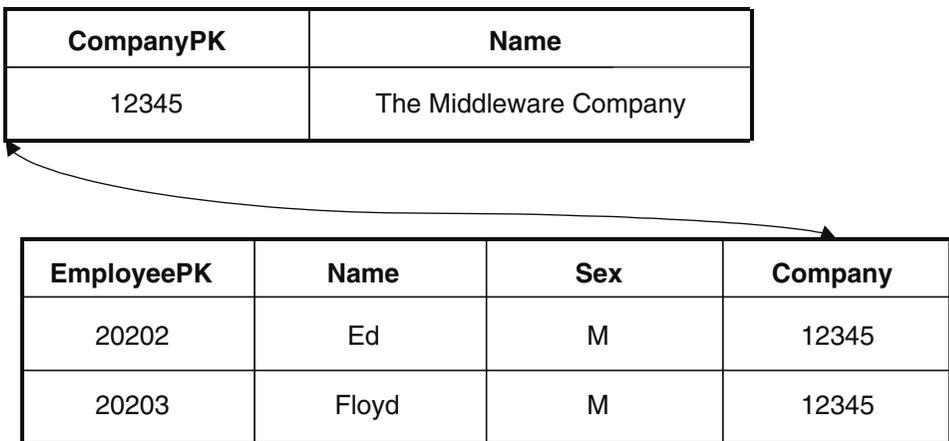


Figure 11.4 Another one-to-many cardinality database schema.

Implementing 1:N Relationships Using BMP

The following code shows how to implement a 1:N relationship using BMP:

```
public class CompanyBean implements EntityBean {
    private String companyPK;
    private String companyName;
```

```
private Vector employees;      // EJB object stubs

public Collection getEmployees() { return employees; }
public void setEmployees(Collection e) {
    this.employees = (Vector) e;
}

...

public void ejbLoad() {
    // 1: SQL SELECT Company
    // 2: JNDI lookup of EmployeeHome
    // 3: Call EmployeeHome.findByCompany(companyPK)
}
public void ejbStore() {
    // 2: SQL UPDATE Company
}
}
```

The code is explained as follows:

- A 1:N relationship has a Vector of stubs, rather than a single stub. Our `get/set` method gets and sets this Vector (a Vector is a Collection).
- Our `ejbLoad()` method is responsible for loading the company state, as well as loading the relationships to employees. How can we achieve this? Remember that the employee table contains the foreign key relationships to the company, not the reverse. Therefore it is natural for the employee bean to access that relationship data, not the company bean. Thus, we do not deal with foreign keys in our bean—we let the employee bean deal with them. We do so by calling a finder method on the employee local home object. That finder method locates each employee that is a member of this company and returns a collection of stubs to us. Note that this causes a second SQL statement to be executed.
- Our `ejbStore()` method is extremely straightforward. Since our `ejbLoad()` method is not responsible for dealing with foreign keys, neither is our `ejbStore()` method. It doesn't even know about the relationship. The employee (not the company) has an `ejbStore()` that persists foreign keys to the company.



If you're good at SQL, you might have noticed that in our example, if we really wanted to do so, we could load both the company and the foreign keys for our employee in one SQL statement. But this would not help us, because we would still need to transform those foreign keys into stubs. We'd need to call `EmployeeHome.findByPrimaryKey()` for each found key, which would generate even more SQL.

Implementing 1:N Relationships Using CMP

The following code shows how to implement a 1:N relationship using CMP:

```
public abstract class CompanyBean implements EntityBean {
    // no fields

    public abstract Collection getEmployees();
    public abstract void setEmployees(Collection employees);

    ...

    public void ejbLoad() {} // Empty
    public void ejbStore() {} // Empty
}
```

Our *ejbLoad()* and *ejbStore()* methods are free of any scaffolding code. The relationships are specified in the deployment descriptor as follows:

```
<ejb-jar>
<enterprise-beans>
...
</enterprise-beans>

<relationships>
<ejb-relation>
<ejb-relation-name>Company-Employees</ejb-relation-name>
<ejb-relationship-role>
<ejb-relationship-role-name>
    Company-Employs-Employees
</ejb-relationship-role-name>
<multiplicity>One</multiplicity>
<relationship-role-source>
<ejb-name>Company</ejb-name>
</relationship-role-source>
<!--
    When you have a relationship with more than one object, you
    can use either a java.util.Collection or a java.util.Set.
    We need to identify which one we're using. How do you choose
    between a Collection and a Set? A Collection can contain
    duplicates, whereas a Set cannot. This needs to match up to
    your bean's get/set methods.
-->
<cmr-field>
<cmr-field-name>employees</cmr-field-name>
<cmr-field-type>java.util.Collection</cmr-field-type>
</cmr-field>
</ejb-relationship-role>
```

```

<ejb-relationship-role>
  <ejb-relationship-role-name>
    Employees-WorkAt-Company
  </ejb-relationship-role-name>
  <multiplicity>Many</multiplicity>
  <relationship-role-source>
    <ejb-name>Employee</ejb-name>
  </relationship-role-source>
</ejb-relationship-role>
</ejb-relation>
</relationships>
</ejb-jar>

```

As you can see, this is much simpler than BMP. If you understood the deployment descriptor for a 1:1 relationship described earlier in this chapter, then you should be able to grasp this one fairly easily.



Relationships with CMP can be much higher performing than their BMP equivalents. To load a 1:N relationship with BMP, we need to perform two SQL statements: We need to *ejbLoad()* the “1” side of the relationship and then *find()* the “N” side of the relationship.

This is an inherent downside to BMP—you are limited to performing SQL operations at the granularity of an entity bean. With CMP, if your container is good, you can optimize and tell the container to perform one gigantic SQL statement to load yourself and your relationships.

M:N Relationships

A many-to-many relationship is not as common as a one-to-many relationship but is still important. Examples of one-to-many relationships include:

- Student:Course
- Investor:MutualFund
- Stock:Portfolio

M:N relationships are typically set up by an *association* table in the database. An association table contains foreign keys to two other tables. Figure 11.5 shows a possible database setup.

What’s interesting about Figure 11.5 is that we’ve created a third table, called an Enrollment table, which models the relationship between a student and a course. The alternative to an association table is for each half of the relationship to have a vector of foreign keys to the other half, persisted as bit-blobs, which is nasty to deal with.

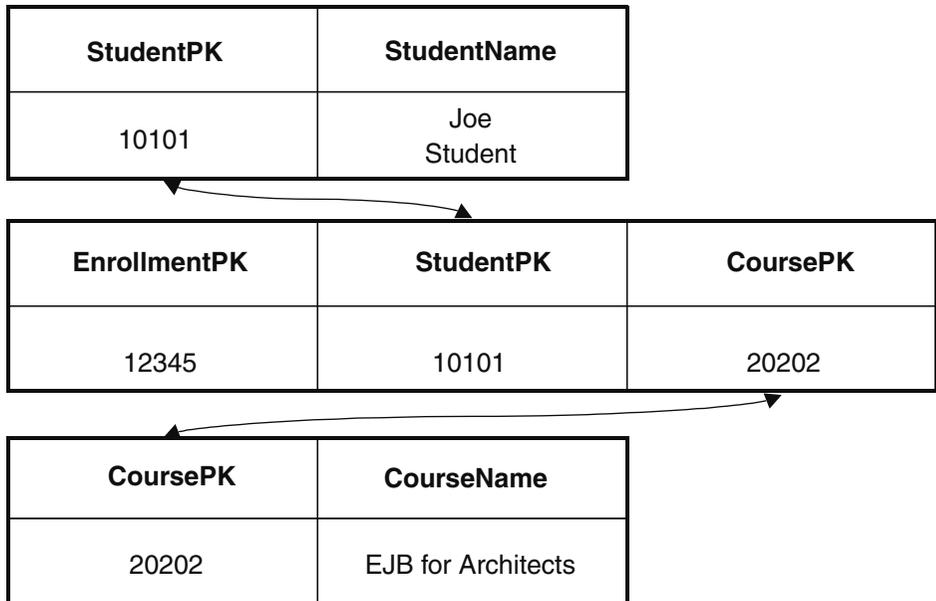


Figure 11.5 A possible many-to-many cardinality database schema.

Two Choices when Implementing M:N Relationships

When you model an M:N relationship using entity beans, you have two choices.

Fake the M:N relationship by introducing a third entity bean. Our enrollment table could theoretically include other information as well, such as the date when the enrollment was made. It then makes sense to model the M:N relationship itself as an entity bean—an enrollment bean. The enrollment bean would map to the association table. This demonstrates a great modeling principle: When you have a many-to-many relationship, consider making the relationship itself into a first-class citizen. When you do this, you are introducing an intermediary. That intermediary has two 1:N relationships. Thus, we have effectively reduced the M:N relationship problem into two 1:N relationship problems!

Model the M:N relationship as a true M:N relationship. If all you're doing is storing relationship information, you might not need to introduce a third entity bean. In this case, you have only two entity beans, each representing half the relationship. Each entity beans would contain a Collection of the other

entity bean. Each entity bean would be persisted to its own table, and each entity bean's Collection would be persisted to the relationships table. With BMP, you are in control of the JDBC, so you can map an entity bean to two tables very easily. With CMP, you're dependent on your container's persister.

We prefer the fake approach, because it keeps your entity beans pure and clean. The fewer relationships that you code into your entity beans, the more reusable your entity beans are in a variety of circumstances, and the less bloat your entity beans incur. This approach also has the advantage that your entity bean and database are similar to one another, making mapping more straightforward.

What's cool, however, is that your EJB components can map to the database however you'd like. That is, both approaches can map to association tables. This is because the database is completely unaware of how it's being represented in the middle tier. You can even switch back and forth between the fake and real approach if you'd like. And even if you're not using an association table but some other approach, you can still map your beans however you'd like, assuming you're good with JDBC (in the BMP case) or assuming your container vendor has a good persistence engine (in the CMP case).

Let's see how to model both fake and real M:N relationships with both BMP and CMP.

Implementing Fake M:N Relationships Using BMP

The following code shows how to implement an M:N relationship as two 1:N relationships using BMP:

```
public class StudentBean implements EntityBean {
    private String studentPK;
    private String studentName;
    ...
    public void ejbLoad() { // SQL SELECT Student }
    public void ejbStore() { // SQL UPDATE Student }
}

public class CourseBean implements EntityBean {
    private String coursePK;
    private String courseName;
    ...
    public void ejbLoad() { // SQL SELECT Course }
    public void ejbStore() { // SQL UPDATE Course }
}

public class EnrollmentBean implements EntityBean {
```

```

private String enrollmentPK;
private Student student;    // EJB local object stub
private Course course;     // EJB local object stub

public Course getCourse() { return course; }
public void setCourse(Course c) { this.course = c; }

public Student getStudent() { return student; }
public void setStudent(Student s) { this.student = s; }

. . .

public void ejbLoad() {
    // 1: SQL SELECT Enrollment. This loads both the
    //    Enrollment plus the foreign keys to Student
    //    and Course.
    //
    // 2: JNDI lookup of StudentHome, CourseHome
    //
    // 3: Call findByPrimaryKey() on both the Student
    //    and Course homes, passing the foreign keys
}
public void ejbStore() {
    // 1: Call getPrimaryKey() on Student, Course. This
    //    gives us our foreign keys.
    //
    // 2: SQL UPDATE Enrollment
}
}

```

As usual, the relationship code is in **bold**. A brand-new entity bean, enrollment, models the relationship between student and course. The enrollment bean keeps a stub for a course and a stub for a student, and has get/set methods for clients to access those stubs. At the point in which object/relational mapping occurs, we transform those stubs to and from their foreign key database representation.

Implementing Fake M:N Relationships Using CMP

The following code shows how to implement a fake M:N relationship using CMP:

```

public abstract class StudentBean implements EntityBean {
    // no fields
    ...
    public void ejbLoad() {} // Empty
    public void ejbStore() {} // Empty
}

```

```

public abstract class CourseBean implements EntityBean {
    // no fields
    ...
    public void ejbLoad() {} // Empty
    public void ejbStore() {} // Empty
}

public abstract class EnrollmentBean implements EntityBean {
    // no fields

    public abstract Course getCourse();
    public abstract void setCourse(Course c);

    public abstract Student getStudent();
    public abstract void setStudent(Student s);

    ...

    public void ejbLoad() {} // Empty
    public void ejbStore() {} // Empty
}

```

Our *ejbLoad()* and *ejbStore()* methods are free of any scaffolding code. The relationships are specified in the deployment descriptor as follows:

```

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

<relationships>
<ejb-relation>
<ejb-relation-name>Enrollment-Student</ejb-relation-name>
<ejb-relationship-role>
<ejb-relationship-role-name>
    Enrollments-AreRegisteredBy-Student
</ejb-relationship-role-name>
<multiplicity>Many</multiplicity>
<relationship-role-source>
<ejb-name>Enrollment</ejb-name>
</relationship-role-source>
<cmr-field><cmr-field-name>student</cmr-field-name></cmr-field>
</ejb-relationship-role>

<ejb-relationship-role>
<ejb-relationship-role-name>
    Student-Has-Enrollments
</ejb-relationship-role-name>
<multiplicity>One</multiplicity>
<relationship-role-source>

```

```

    <ejb-name>Student</ejb-name>
  </relationship-role-source>
</ejb-relationship-role>
</ejb-relation>

<ejb-relation>
  <ejb-relation-name>Enrollment-Course</ejb-relation-name>
  <ejb-relationship-role>
    <ejb-relationship-role-name>
      Enrollments-AreRegistrationsFor-Course
    </ejb-relationship-role-name>
    <multiplicity>Many</multiplicity>
    <relationship-role-source>
      <ejb-name>Enrollment</ejb-name>
    </relationship-role-source>
    <cmr-field><cmr-field-name>course</cmr-field-name></cmr-field>
  </ejb-relationship-role>

  <ejb-relationship-role>
    <ejb-relationship-role-name>
      Course-Has-Enrollments
    </ejb-relationship-role-name>
    <multiplicity>One</multiplicity>
    <relationship-role-source>
      <ejb-name>Course</ejb-name>
    </relationship-role-source>
  </ejb-relationship-role>
</ejb-relation>
</relationships>
</ejb-jar>

```

As you can see from the preceding deployment descriptor, we model our fake M:N relationship as two N:1 relationships (one for each bean in the relationship). An N:1 relationship is conceptually the same as a 1:N relationship, and we learned how to model a 1:N relationship with CMP earlier.

Implementing True M:N Relationships Using BMP

The following code shows how to implement a true M:N relationship using BMP:

```

public class StudentBean implements EntityBean {
    private String studentPK;
    private String name;
    private Vector courses;      // EJB object stubs

    public Collection getCourses() { return courses; }
    public void setCourses(Collection c) { this.courses = c;}

```

```

...

public void ejbLoad() {
    // 1: SQL SELECT Student
    // 2: JNDI lookup of CourseHome
    // 3: Call CourseHome.findByStudent(studentPK)
}
public void ejbStore() {
    // SQL UPDATE Student
}

public class Course implements EntityBean {
    private String coursePK;
    private String name;
    private Vector students;    // EJB object stubs

    public Collection getStudents() { return students; }
    public void setStudents(Collection s) { this.students = s;}

    ...

    public void ejbLoad() {
        // 1: SQL SELECT Course
        // 2: JNDI lookup of StudentHome
        // 3: Call StudentHome.findByCourse(coursePK)
    }
    public void ejbStore() {
        // SQL UPDATE Course
    }
}

```

The relationship code is in **bold**. As you can see, all we've done to model a true M:N relationship is to code a 1:N relationship for each bean in the relationship. This is similar code to the code presented when we learned about 1:N relationships.

Implementing True M:N Relationships Using CMP

The following code shows how to implement a true M:N relationship using CMP:

```

public abstract class StudentBean implements EntityBean {
    // no fields

    public abstract Collection getCourses();
    public abstract void setCourses(Collection courses);

    ...

    public void ejbLoad() {}    // Empty
}

```

```

    public void.ejbStore() {} // Empty
}

public abstract class CourseBean implements EntityBean {
    // no fields

    public abstract Collection getStudents();
    public abstract void setStudents(Collection students);

    ...

    public void.ejbLoad() {} // Empty
    public void.ejbStore() {} // Empty
}

```

Our *ejbLoad()* and *ejbStore()* methods are free of any scaffolding code. The relationships are specified in the deployment descriptor as follows:

```

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

  <relationships>
    <ejb-relation>
      <ejb-relation-name>Student-Course</ejb-relation-name>
      <ejb-relationship-role>
        <ejb-relationship-role-name>
          Students-EnrollIn-Courses
        </ejb-relationship-role-name>
        <multiplicity>Many</multiplicity>
        <relationship-role-source>
          <ejb-name>Student</ejb-name>
        </relationship-role-source>
        <cmr-field>
          <cmr-field-name>courses</cmr-field-name>
          <cmr-field-type>java.util.Collection</cmr-field-type>
        </cmr-field>
      </ejb-relationship-role>

      <ejb-relationship-role>
        <ejb-relationship-role-name>
          Courses-HaveEnrolled-Students
        </ejb-relationship-role-name>
        <multiplicity>Many</multiplicity>
        <relationship-role-source>
          <ejb-name>Course</ejb-name>
        </relationship-role-source>
        <cmr-field>
          <cmr-field-name>students</cmr-field-name>
          <cmr-field-type>java.util.Collection</cmr-field-type>
        </cmr-field>
      </ejb-relationship-role>
    </ejb-relation>
  </relationships>
</ejb-jar>

```

```
</cmr-field>
</ejb-relationship-role>
</ejb-relation>
</relationships>
</ejb-jar>
```

As you can see, modeling a true M:N relationship using CMP is extremely straightforward. We just use the word *Many* on each half of the relationship, and state that each half of the relationship has a collection of the other half.

If you've made it this far, congratulations—this concludes our cardinality discussion! Let's move on to directionality.

Directionality

The *directionality* of a relationship specifies the direction in which you can navigate a relationship. There are two flavors of directionality.

Bidirectional. You can get from entity A to entity B, and can also get from entity B to entity A.

Unidirectional. You can get from entity A to entity B, but *cannot* get from entity B to entity A.

Directionality applies to all cardinalities (1:1, 1:N, and M:N). Directionality and cardinality are orthogonal and complementary concepts. You can mix and match them however you would like.

Let's use our 1:1 relationship example of an order and a shipment to help us figure out directionality.

Implementing Directionality with BMP

The following code is a bidirectional relationship, with the key information in **bold**:

```
public class OrderBean implements EntityBean {
    private String orderPK;
    private String orderName;

    // EJB local object stub, must be stored/loaded
    private Shipment shipment;

    public Shipment getShipment() { return shipment; }
    public void setShipment(Shipment s) { this.shipment = s; }
```

```

    ...
}

public class ShipmentBean implements EntityBean {
    private String shipmentPK;
    private String shipmentName;

    // EJB local object stub, must be stored/loaded
    private Order order;

    public Order getOrder() { return order; }
    public void setOrder(Order o) { this.order = o; }

    ...
}

```

As you can see, in a bidirectional relationship, each bean in the relationship has a field pointing to the other bean, along with a get/set method.

In comparison, with a unidirectional relationship, we don't allow the user to get from the second bean to the first bean.

```

public class OrderBean implements EntityBean {
    private String orderPK;
    private String orderName;

    // EJB local object stub, must be stored/loaded
    private Shipment shipment;

    public Shipment getShipment() { return shipment; }
    public void setShipment(Shipment s) { this.shipment = s; }

    ...
}

public class ShipmentBean implements EntityBean {
    private String shipmentPK;
    private String shipmentName;

    // No Order stub, no Order get/set method

    ...
}

```

Implementing Directionality with CMP

The following is a bidirectional CMP relationship:

```

public abstract class OrderBean implements EntityBean {
    // no fields

```

```

public abstract Shipment getShipment();
public abstract void setShipment(Shipment s);

...

public void ejbLoad() {} // Empty
public void ejbStore() {} // Empty
}

public abstract class ShipmentBean implements EntityBean {
// no fields

public abstract Order getOrder();
public abstract void setOrder(Order o);

...

public void ejbLoad() {} // Empty
public void ejbStore() {} // Empty
}

```

As you can see, in a bidirectional CMP relationship, each bean in the relationship has a pair of abstract get/set methods pointing to the other bean. We need to tell the container that these get/set methods are special relationship methods so that the container can generate relationship code. Here is the deployment descriptor that achieves this.

```

<ejb-jar>

<enterprise-beans>
...
</enterprise-beans>

<relationships>
<ejb-relation>
<ejb-relation-name>Order-Shipment</ejb-relation-name>
<ejb-relationship-role>
<ejb-relationship-role-name>
order-spawns-shipment
</ejb-relationship-role-name>
<multiplicity>One</multiplicity>
<relationship-role-source>
<ejb-name>Order</ejb-name>
</relationship-role-source>
<cmr-field><cmr-field-name>shipment</cmr-field-name></cmr-field>
</ejb-relationship-role>

<ejb-relationship-role>
<ejb-relationship-role-name>

```

```
shipment-fulfills-order
</ejb-relationship-role-name>
<multiplicity>One</multiplicity>
<relationship-role-source>
  <ejb-name>Shipment</ejb-name>
</relationship-role-source>
<cmr-field><cmr-field-name>order</cmr-field-name></cmr-field>
</ejb-relationship-role>
</ejb-relation>
</relationships>

</ejb-jar>
```

In the deployment descriptor, we set up two container-managed relationship (CMR) fields: one for each bean's abstract get/set method pair that points to the other bean.

To make this into a unidirectional relationship, we would simply get rid of an abstract get/set method pair, along with its corresponding CMR field.

Directionality May Not Map to Database Schemas

Note that directionality in entity beans may *not* correspond to the inherent directionality of the database schema. An entity bean can provide for directionality even though the database does not do so easily, and vice versa. For example, Figure 11.6 is a normalized database schema for a Person:Address relationship. Figure 11.7 is a denormalized schema.

Relationships and Local Interfaces

A common theme throughout this book has been to always use local interfaces when possible. This is especially true for entity beans, and has a big impact on relationships.

Specifically, if you decide for some bizarre reason to use remote interfaces with entity beans, then

- **You must not expose *get/set* methods for relationship fields to remote clients. Doing so creates problems because, for example, the client does not have access to the container-implemented collection interface.**
- **Your entity bean can only have unidirectional relationships with other entity beans. The lack of a local interface prevents other entity beans from having a relationship with you.**

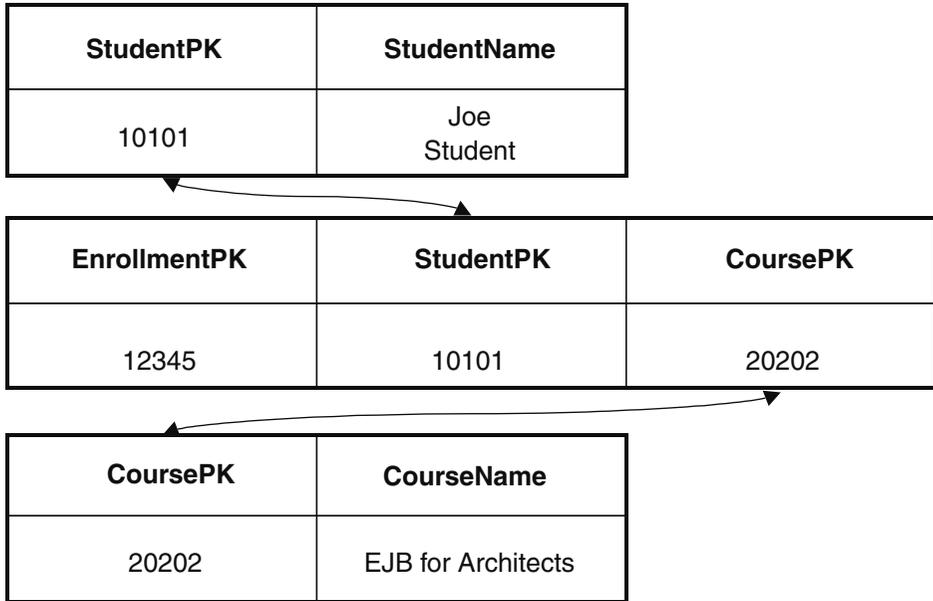


Figure 11.6 A normalized schema.

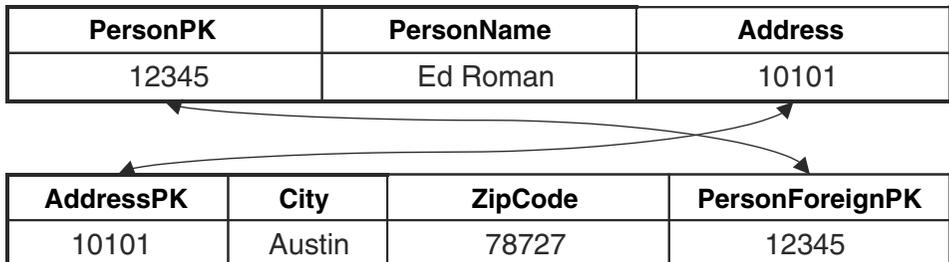


Figure 11.7 A denormalized schema.

Both of these schemas give us enough information to derive relationship information. You can, if you choose to do so, map entity beans to both these schemas and use bidirectional relationships. The difference is that the denormalized schema allows for more efficient SQL. That is the classic computer science space-time tradeoff. If you denormalize the database, you waste space and increase maintenance problems, but you gain speed.

Bidirectional or Unidirectional?

How do you choose between bidirectional and unidirectional relationships? Here are some questions to ask:

- Should each bean know about the other bean? Would that hamper reuse?
- From the client's perspective, does it make intuitive sense to navigate the relationship from the direction in question?
- Is the database mapping of stubs to foreign keys straightforward, or does adding directionality result in mapping to multiple tables, resulting in inadequate performance?

Lazy Loading

All the relationship code we've seen so far makes a big assumption: Whenever an entity bean is loaded, all of the other entity beans that it has a relationship with are also loaded. This is called *aggressive loading*. We saw this, for example, with our Order:Shipment relationship at the beginning of this chapter. The order bean looked up the shipment bean in the order bean's *ejbLoad()* method.

Aggressive loading is nice because you can load all database data in a single transaction. However, it does have its downside. Aggressive loading could lead to loading a very large entity bean graph, and you may not need that entire graph.

Lazy loading means to load only related beans when you need to access those beans. For example, with the Order:Shipment relationship using BMP that we presented at the beginning of this chapter, we would rewrite the code to lazy-load as follows:

```
public class OrderBean implements EntityBean {
    private String orderPK;
    private String orderName;

    private String shipmentFK; // Foreign key to shipment
    private Shipment shipment; // EJB local object stub

    public void ejbLoad() {
        // 1: SQL SELECT Order, loading the shipment foreign key
        // 2: Set shipmentFK field to the loaded key
    }

    public Shipment getShipment() {
        // 1: JNDI lookup of ShipmentHome
        // 2: Call ShipmentHome.findByPrimaryKey(shipmentFK)
        return shipment;
    }
}
```

```
}  
  
...  
  
}
```

In the preceding code, we are looking up the shipment *just in time* when the client calls `getShipment()`, rather than in `ejbLoad()`. `ejbLoad()` merely locates the appropriate foreign key, which `getShipment()` uses.

With CMP, lazy-loading happens automatically behind the scenes. You are, however, reliant on container-specific flags to enable lazy-loading. Most major containers support this, so check your container documentation.

Aggregation vs. Composition and Cascading Deletes

When you have a relationship between two entity beans, you need to think about whether that relationship is an *aggregation* or a *composition* relationship.

An **aggregation relationship** is a *uses* relationship. For example, students use courses. If you delete a student, you don't delete the courses the student is registered in, because other students are using that course. Similarly, if you delete a course, you don't murder a student!

Design Tip: Aggressively Load in One Direction Only for Many-to-Many Relationships

With many-to-many relationships, you need to be careful with how aggressively you load your entity bean graph. For example assume that Larry lives at addresses A, B, and C; Curly at C and D; Moe at C and E; and E is a commune with 37 people living in it. Larry, Curly, Moe, and everyone in the commune is a customer of ours. If we cascade the load across the relationship in both directions when we read in Larry, we would retrieve at least five address objects and 40 customer objects, not to mention any other addresses that the commune people also live in and any customers and their addresses that those retrievals would then cascade to. The same problem arises if we also cascade the deletion in both directions. We need to cascade the retrieval and deletion in one direction, or be incredibly smart about how we cascade in both directions. Unless your entity bean graph is small, we recommend you use lazy-loading for at least one direction of the relationship.

A **composition relationship** is an *is-assembled-of* relationship. For example, orders are assembled of line items. Deleting an order deletes all line items. Line items shouldn't be around if their parent order is gone.

Once you've figured out whether your relationship is an aggregation or composition, you need to write your entity beans so they model the semantics you desire. This all boils down to a concept called a *cascading delete*. An aggregation relationship does not cause a cascading delete, whereas a composition relationship does.

With BMP, you implement a cascading delete manually in your `ejbRemove()` method. For example, an order bean's `ejbRemove()` method would not only perform a SQL DELETE of the order, but would also call the shipment bean's `ejbRemove()` method:

```
public class OrderBean implements EntityBean {
    private String orderPK;
    private String orderName;
    private Shipment shipment; // EJB local object stub

    public Shipment getShipment() { return shipment; }
    public void setShipment(Shipment s) { this.shipment = s;}

    ...

    public void ejbRemove() {
        // 1: SQL DELETE Order
        // 2: shipment.remove();
    }
}
```

With CMP, the container generates cascading delete code for you. If you have a composition relationship, you just need to set up a `<cascade-delete/>` tag in the deployment descriptor, as follows:

```
<ejb-jar>

<enterprise-beans>
...
</enterprise-beans>

<relationships>
<ejb-relation>
<ejb-relation-name>Order-Shipment</ejb-relation-name>
<ejb-relationship-role>
<ejb-relationship-role-name>
    order-spawns-shipment
</ejb-relationship-role-name>
<multiplicity>One</multiplicity>
<relationship-role-source>
```

```

    <ejb-name>Order</ejb-name>
  </relationship-role-source>
  <cmr-field><cmr-field-name>shipment</cmr-field-name></cmr-field>
</ejb-relationship-role>

<ejb-relationship-role>
  <ejb-relationship-role-name>
    shipment-fulfills-order
  </ejb-relationship-role-name>
  <multiplicity>One</multiplicity>
  <cascade-delete/>
  <relationship-role-source>
    <ejb-name>Shipment</ejb-name>
  </relationship-role-source>
  <cmr-field><cmr-field-name>order</cmr-field-name></cmr-field>
</ejb-relationship-role>
</ejb-relation>
</relationships>

</ejb-jar>

```

If you have an aggregation relationship, you just leave the `<cascade-delete/>` tag out.

Relationships and EJB-QL

When setting up CMP relationships, you can also set up special queries using the EJB Query Language (EJB-QL), which we briefly described in Chapter 6 and fully explain in Appendix C. The following is relevant to our discussion and is excerpted from Appendix C.

The big difference between EJB-QL and SQL is that EJB-QL allows you to traverse relationships between entity beans using a dot notation. For example:

```

SELECT o.customer
FROM Order o

```

In this EJB-QL, we are returning all customers that have placed orders. We are navigating from the order entity bean to the customer entity bean easily using a dot notation. This is quite seamless.

What's exciting about this notation is that bean providers don't need to know about tables or columns; they merely need to understand the relationships between the entity beans that they've authored. The container will handle the traversal of relationships for us because we declare our entity beans in the

same deployment descriptor and Ejb-jar file, empowering the container to manage all of our beans and thus understand their relationships.

In fact, you can traverse more than one relationship. That relationship can involve container-managed relationship fields and container-managed persistent fields. For example:

```
SELECT o.customer.address.homePhoneNumber  
FROM Order o
```

The restriction on this type of recursive relationship traversal is that you are limited by the *navigability* of the relationships that you define in the deployment descriptor. For example, let's say that in the deployment descriptor, you declare that orders have a one-to-many relationship with line items, but you do not define the reverse many-to-one relationship that line items have with orders. When performing EJB-QL, you can get from orders to line items, but not from line items to orders.

Recursive Relationships

A recursive relationship is one in which an entity bean instance has a relationship with another instance of the same entity bean class, such as what's shown in Figure 11.8.

Figure 11.8 shows an Employee:Manager relationship. All that this means is that our employee entity bean has a relationship with another employee entity bean.

As you would expect, recursive relationships are implemented exactly as non-recursive relationships. All the principles we learned earlier apply, and nothing is new. We just happen to have a relationship with an instance of an entity bean that uses the same class.

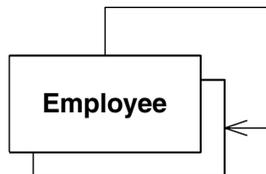


Figure 11.8 A recursive relationship.

Circular Relationships

A circular relationship is similar to a recursive relationship except that instead of involving a single entity bean, it involves several. Figure 11.9 depicts a circular relationship.

The following relationships exist:

- Employees work in a division.
- A division owns one or more workstations.
- An employee has a workstation.

The problem with circular relationships is that if your beans automatically find each other, you will get into an endless circle of finding. The same problem exists for cascading deletes.

So how do you implement circular relationships between EJBs appropriately? You have several implementation strategies.

1. Some containers allow you optimize performance and load an entity bean at the same time that it's found. This is where the circularity issue stems from, because the *ejbLoad()* method performs the cascading find. Not loading an entity bean when it's found means no cascading find operation occurs.
2. Break the circular relationship by removing one of the relationships altogether. This is a harsh approach to resolving the problem.
3. Break the circular relationships within your model by making one or more relationships unidirectional, effectively breaking the circle in both direc-

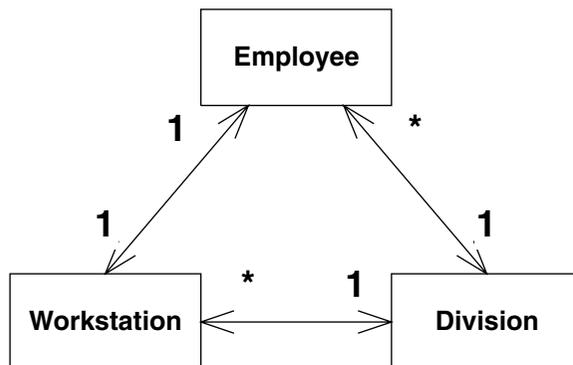


Figure 11.9 A circular relationship.

tions. This isn't always an option because your requirements may not permit it.

4. Use lazy-loading rather than aggressive loading, and do not use cascading deletes.
5. Choose a persister that detects circular relationships. Many persistence administration tools automatically detect and warn you of circular relationships when you define them. This allows you to prevent the problem before it occurs.

Referential Integrity

Referential integrity is the assurance that a reference from one entity to another entity is valid. For example:

- Let's say a company, department, and position each have relationships with an employee. If the employee is removed, all references to it must also be removed, or your system must not allow the removal.
- Let's say an order has a 1:N relationship with a line item. Someone adding a second order to an order line item is trying to change a 1:N relationship to an M:N relationship. We must therefore break the line item's relationship with the original order so that we maintain our intended 1:N semantics.

Referential integrity issues arise in both the database (keeping foreign keys correct) and in the application server (keeping stubs correct). So how do you ensure referential integrity within your EJB applications?. You have three fundamental options:

1. Enforce referential integrity within your database with triggers. For example, you could write a trigger that fires off when an employee is deleted. This trigger would delete the relationships the employee had with other database tables to preserve referential integrity.
2. Enforce referential integrity within your database with stored procedures. Your EJB component would call these stored procedures to perform database operations, and the stored procedures would be responsible for preserving referential integrity.
3. Enforce referential integrity within EJB components.

Implementing referential integrity in your database has the advantage that other nonEJB applications can take advantage of it, your database being the lowest common denominator within your organization. Relational databases

implement triggers for exactly this purpose, and most data modeling tools support the generation of trigger code to simplify this effort for you. The drawback of implementing logic in your database is that it increases the processing burden on your database server(s), running the risk that your database becomes a bottleneck to your application. You can also take a hybrid approach to implementing referential integrity—your EJBs handle the referential integrity for some entities and your database for others.

Of these options, we believe that the EJB approach is the cleanest and easiest to maintain over the long term, because your EJB layer encapsulates all relationships. Here is how you do it with EJB:

With BMP you need to take care of referential integrity on your own. You do so by manually breaking old relationships. If someone tries to assign a second order to your line item, your line item bean should call the order bean and tell the order bean to remove you from its list of line items.

With CMP the container will *automatically* handle referential integrity for you. You never have to worry about these issues. This is one neat feature the container provides that makes CMP a compelling value proposition.

Note that it's not quite this simple. To complicate matters, you might have a farm of EJB application servers, and your component might exist simultaneously on several machines. Furthermore, if you have other applications accessing your database then it is possible that they too have representations of your data in their memory as well. The good news is that *transactions* (see Chapter 10) solve this problem. For example, when you delete an employee and also delete the relationships it has with other entities within a transaction, either all or none of the deletions occur, preserving referential integrity.



Writing code to enforce referential integrity in your EJB components instead of your database works only when all of your applications are written this way (and hopefully they share a common code base). However, this is rarely the case. In many organizations, some or often most applications are written assuming that the database(s) will handle referential integrity. This is clearly an inappropriate layering of these applications because business logic is now implemented on several disparate architectural tiers, making the applications less robust. However, it is a reality that many EJB developers must accept—some of their business logic will be implemented in the database, perhaps through triggers or through Java objects implemented within the database.

Relationships, Referential Integrity, and Client Code

Throughout this chapter, we've seen lots of fascinating relationships. Many of those relationships involved collections. For example, here is our Company: Employee 1:N CMP relationship again.

```
public abstract class CompanyBean implements EntityBean {
    // no fields

    public abstract Collection getEmployees();
    public abstract void setEmployees(Collection employees);

    ...

    public void ejbLoad() {} // Empty
    public void ejbStore() {} // Empty
}
```

This code has methods to get/set entire Collections of employees. But what's interesting is that there is no API for clients to perform operations on *individual* employees.

This is where the Collection comes into play. By using the Collection from client code, you can modify the contents of a 1:N relationship. For example:

```
// Lookup local home objects
Context ctx = new InitialContext(...);
CompanyHome companyHome = (CompanyHome) ctx.lookup("CompanyHome");
EmployeeHome employeeHome = (EmployeeHome) ctx.lookup("EmployeeHome");

// Make a new employee
Employee employeeA = employeeHome.create("Ed Roman");

// Find a company
Company company =
    companyHome.findByPrimaryKey("The Middleware Company");
Collection employees = company.getEmployees();

// Add the employee to the company.
// This demonstrates the add() Collection API method
employees.add(employeeA);

// Look at each employee in the company.
// This demonstrates using iterators to loop through collections
```

```
Iterator i = employees.iterator();
while (i.hasNext()) {
    Employee emp = (Employee) i.next();
    System.out.println(emp.getName());
}

// Remove the employee from the company.
// This demonstrates the remove() Collection API
employees.remove(employeeA);
```

Since we're using local interfaces, the collection that the client modifies is the same as the collection inside the bean. This is because the get/set methods pass the collection by reference rather than by value. Thus, when the client modifies the contents of the collection, he is actually changing the bean's relationships. If remote interfaces were used, the relationships would not be accessible through the remote interface to begin with, due to the remote interface restrictions discussed earlier in this chapter.

The container is responsible for providing an implementation of the Collection interface. This needs to be a *smart* collection that understands how to preserve referential integrity behind the scenes.



Be careful when using iterators and relationships. If you want to modify a relationship while an iterator is at work, use only the `java.util.Iterator.remove()` method. Adding or removing elements from a collection while the iterator is in progress will throw off the iterator.

Table 11.1 lists the effects that client operations have on referential integrity. Note that for the 1:N and M:N rows on the table, we are performing operations on collections and using the collections API for operations such as `add()` and `remove()`.



Try to only look at the first three columns of Table 11.1, and see if you can guess what the fourth column should be.

Table 11.1 Client Operations and Referential Integrity

SITUATION	ORIGINAL RELATIONSHIPS	OPERATION	NEW RELATIONSHIPS
1:1 Relationship Order:Shipment	orderA-shipmentA, orderB-shipmentB	orderA.setShipment (orderB.getShipment());	OrderA-ShipmentB, OrderB-NULL
1:N Relationship Company:Employee	companyA-employeeA, companyA-employeeB, companyB-employeeC, companyB-employeeD	CompanyA.setEmployees (CompanyB. getEmployees());	NULL-employeeA, NULL-employeeB, companyA-employeeC, companyA-employeeD, companyB-Empty Collection
1:N Relationship Company:Employee	companyA-employeeA, companyA-employeeB, companyB-employeeC, companyB-employeeD	EmployeeA.setCompany (employeeC.get Company());	companyB-employeeA, companyA-employeeB, companyB-employeeC, companyB-employeeD
1:N Relationship Company:Employee	companyA-employeeA, companyA-employeeB, companyB-employeeC, companyB-employeeD	CompanyB.get Employees(). add(employeeA);	companyB-employeeA, companyA-employeeB, companyB-employeeC, companyB-employeeD
1:N Relationship Company:Employee	companyA-employeeA, companyA-employeeB, companyB-employeeC, companyB-employeeD	CompanyA.get Employees().remove (employeeA);	NULL-employeeA companyA-employeeB, companyB-employeeC, companyB-employeeD
M:N Relationship Student:Course	studentA-courseA, studentA-courseB, studentB-courseB, studentB-courseC	studentA.setCourses (studentB.getCourses());	studentA-courseB, studentA-courseC, studentB-courseB, studentB-courseC
M:N Relationship Student:Course	studentA-courseA, studentA-courseB, studentB-courseB, studentB-courseC	studentA.getCourses(). add(courseC);	studentA-courseA, studentA-courseB, studentA-courseC, studentB-courseB, studentB-courseC
M:N Relationship Student:Course	studentA-courseA, studentA-courseB, studentB-courseB, studentB-courseC	studentA.getCourses(). remove(courseB);	studentA-courseA, studentB-courseB, studentB-courseC

Summary

Still with us? Fantastic! Pat yourself on the back, because you've achieved a great deal in this chapter. You learned about cardinality, directionality, referential integrity, cascading deletes, recursive relationships, circular relationships, lazily loaded relationships, and how to control relationships from client code. You also saw how to implement each of the above topics using both CMP and BMP.

You should be prepared now to go ahead and implement relationships in your own deployments. For concrete examples without pseudo-code that you can use as a basis for your own deployments, see the book's accompanying source code (www.wiley.com/compbooks/roman).

Persistence Best Practices

Most modern business applications require that you persist data—create, retrieve, update, and delete. Persisting data from EJB components can be as simple as defining a few simple class-to-table mappings using an EJB container’s persistence administration tool, or as difficult as writing sophisticated Java source code.

In this chapter we explore the issues surrounding EJB persistence and explore the various approaches to persistence that you may employ within your EJB applications. We’ll cover the following topics:

- When to use entity beans, and when not to use them
- How to choose between container managed persistence (CMP) and bean managed persistence (BMP)
- A collection of persistence best practices, such as versioning EJB components, and dealing with a legacy data design

This chapter is written with the assumption that you will use one or more relational databases to store your business objects. We are considering only relational databases because that’s what most organizations use.

When to Use Entity Beans

A fundamental issue that you need to address is how to persist the information encapsulated by your EJBs. The approaches are

- Session beans plus JDBC. A session bean persists data manually, typically via JDBC.
- Persisting through entity beans, either BMP or CMP.

Let's first figure out when entity beans (either CMP or BMP) are a good idea compared to session beans plus JDBC. This discussion will highlight the advantages and disadvantages of using entity beans and will help you make that decision in your projects. Then we'll be able to compare and contrast different approaches for persisting using session beans. We'll also compare BMP and CMP.

Control

There are significant control differences between performing persistence via session beans and entity beans. Session beans are more of a service-based architecture because you call methods explicitly to load and store data. Thus, you are in command of when to use JDBC explicitly. This is very similar to the Microsoft approach to business components. In comparison, with entity beans, the container automatically loads and stores data on your behalf.

This loss of control that entity beans give you can be somewhat disturbing. If you're not careful to tune your container properly using its flags, and to start and end transactions at the right times, operations that require a single SQL statement can take several statements. Proper education of your developers will help solve these problems.

Parameter Passing Analogy

Another way to compare session and entity persistence differences is through an analogy to parameter passing conventions.

When you do a query via a session bean, the session bean typically returns a result set to the client. This is similar to pass-by-value because the data is being returned from the session bean to the client. When you do a query via an entity bean home, you get stubs to server-side objects, rather than a result set. This is analogous to pass-by-reference. The first thing that should come to mind is how does the performance compare between these two models?

In a typical deployment, the session beans are colocated with the entity beans in the same process, and they communicate with each other via local interfaces. The fact that entity beans are a pass-by-reference model does not affect performance because there is no need to traverse the network.

If, on the other hand, you're building a GUI client (such as Java servlets, JSPs, applets, or applications communicating remotely to EJB components), you'll typically need to get database data into the GUI client. In this case, a pass-by-reference model will hurt you because the GUI client needs to traverse the network to get data from the entity beans. You can work around this by wrapping your entity beans with colocated session beans. The session beans copy the entity bean data into serializable Java objects, sending them to the GUI client. The GUI client then does not need to traverse the network to access the data.

The point to take away is that there are few performance implications when comparing session beans and entity beans from the pass-by-value versus pass-by-reference paradigm. This should probably not factor into your decision criteria.

Procedural Versus Object-Oriented

Most EJB deployments work with data that is either procedural (tabular) or object-oriented in nature. Session beans that return result sets are naturally suited for tabular, business data. Entity beans, on the other hand, are Java objects. Thus, they benefit from encapsulation and relationships, and represent data in an object-oriented fashion, such as data that requires encapsulation or relationships with other data.

Caching

Middle-tier data caching is extremely important because it empowers you to reduce database traffic, and your database will most likely be your bottleneck.

Session beans do not represent database data and therefore cannot be cached at all. The rows they read in are cache-consistent for the duration of a single transaction only. Entity beans can be cached across multiple transactions if the application server has exclusive access to that part of the database. You set this up using container-specific flags.

If data is shared, entity bean caching benefits are more prominent because that data is likely to be viewed many times. An example of shared data is a product catalog, such as the hottest 100 books on Amazon.com.

If your data is exclusive (not shared), caching offers almost no benefits. An example of exclusive data is a personal account, such as your personal account

settings on Amazon.com. In the exclusive data case, the extra SQL statements that sometimes occur through entity beans may offset the caching benefits, making entity beans a lower-performing solution. However, in most deployments, most data is shared and read-only, and hence caching is an important performance boost that entity beans provide.

Enforcement of Schema Independence

Schema independence is an extremely important feature of your persistence layer. As a nightmare motivational story, a project for one of our clients took three developers four months to change two columns in a database because of spaghetti SQL all over the place. Encapsulating that data with a layer would have avoided those headaches.

Entity beans force developers to go through an entity bean layer, yielding a single entry point to the database. Developers are isolated from the schema, allowing ease of schema evolution and data encapsulation.

In comparison, session beans can also isolate you from the schema, if you take great care when crafting your session bean layer. It takes good internal best practices to enforce that this remains constant throughout your organization.

In the end, if your developers are on top of things, either session beans or entity beans will give you schema independence.

Ease of Use

For most people, session beans are much easier conceptually than entity beans. Session beans represent a more procedural style of programming, where result-sets are returned. People understand how to use a service-based architecture because they are in explicit control of everything.

Entity beans, on the other hand, are a new concept to many people. Thus, it is much easier to screw up performance when using entity beans. We urge you to consider the quality of your developers when making the call about whether to use entity beans.

Migration

Most EJB deployments are based on existing databases. These databases are likely to be tabular in nature, and many have SQL code that has been tuned over the years to be high-performing. It is a known commodity that works well.

Session bean persistence is somewhat procedural in nature and is a natural evolution of those legacy procedural systems. For some deployments, application developers can simply copy SQL code from existing solutions into the new EJB system. This eliminates a risk factor in a new EJB architecture.

In comparison, entity beans represent data in an object-oriented manner and may require some SQL rewrites for an intuitive object-oriented interface. This is a good long-term solution, but may slow you down and introduce short-term risk.

Rapid Application Development

When building a new EJB system, entity beans provide a rapid application development advantage over session beans. Most popular UML editors (Rational Rose, Together/J) allow you to autogenerate entity beans from UML diagrams. Furthermore, tools (such as IDEs or a popular command-line tool called *EJBGen*) allow you to generate complete entity bean Ejb-jar files from a few bits of information about the data you're modeling.

Note, however, that using EJB at all can sometimes increase the time it takes to build a system. This is because of all the overhead you need to go through to write all the files that comprise a bean. If you're not using tools, or at least copying and pasting template code, then you may find yourself bogged down.

Choosing between CMP and BMP

Now that we've compared session beans and entity beans, let's assume we're using entity beans. In Chapter 5, you discovered two approaches for persisting entity beans: With BMP, you are responsible for coding all database logic, while with CMP, the container handles the persistence for you.

The choice between CMP and BMP is not necessarily clear cut. Both bean managed and container managed beans have virtues and drawbacks.

Code Reduction and Rapid Application Development

The promise of CMP is quite compelling. If you tell the EJB container a couple of things about your bean, container managed persistence can perform all data access logic for you. This reduces the size of your bean tremendously—no more JDBC code in your beans—which reduces overall development time. It also makes code easier to understand and maintain. CMP beans are also

fantastic for prototyping. If you need to get something working right away, go with CMP, knowing that you can take a BMP approach later if required.

Know that in reality you still may need to write persistent code with container managed beans. This could be going through a series of wizards to specify how your entity beans map to an underlying store. You also need to specify the logic behind your finder methods. The difference is that with CMP, your data access logic is now specified declaratively, whereas with BMP, you're writing the logic in Java. To CMP's credit, the amount of programming you're doing is much less.

Performance

CMP entity beans, if tuned properly, are much higher performing than BMP entity beans.

For example, with BMP, it takes two SQL statements to load an entity bean: the first to call a finder method (loading only the primary key) and the second during *ejbLoad()* to load the actual bean data. A collection of n bean-managed persistent entity beans requires $n+1$ database calls to load that data (one finder to find a collection of primary keys, and then n loads).

With CMP, the container can reduce the $n+1$ database calls problem to a single call, by performing one giant SELECT statement. You typically set this up using container-specific flags (which do not affect bean portability). Check your container's documentation to see if this feature is supported.



There is a hack work-around to increase BMP performance. It's called the *fat key pattern* and is explained on the companion Web site, www.wiley.com/compbooks/roman.

Bugs

CMP systems tend to be harder to debug than BMP systems. The reason is that with BMP, you are in total control of the JDBC code; if something goes wrong, you can debug that code.

With CMP, you are generating code based on deployment descriptor values. While it may be true that user error is reduced at the database level, serious ramifications occur if there *is* some kind of bug. Because the container is performing your persistence for you, it is tough to figure out what database operations the container is really doing. You may need to trace through container-generated code if it's available, decompile the container, or possibly wait on technical support lines, delaying a project.

Furthermore, since we're all human, we make mistakes writing CMP deployment descriptors, such as having values that are incorrect or that do not match up perfectly to our bean files. Often the container's JDBC code generator is too dumb to point out your error and simply generates bad code. Other times, your container's generator tool might even crash, making it even harder to figure out what the problem is. (This is *really* annoying!)

See Chapter 13 for strategies for debugging misbehaving EJB applications.

Control

BMP gives you ultimate control over JDBC, and thus you have unlimited flexibility for how you map your objects to the database. For CMP, many containers support complex mappings, but some containers don't. For example, if your container-managed persistent entity bean class has a vector of Java objects as a container-managed field, you may need to convert that vector into a bit-blob or other form that the container can handle when mapping to storage.

Application Server and Database Independence

One nice thing about container managed persistence is that you aren't hard-coding a particular database storage API into your beans, such as JDBC. Because you aren't issuing explicit relational database calls in your persistence layer, you can easily move into a different database, such as Oracle instead of SQL Server. Theoretically, you might even port your beans to use object databases without changing code.

Database independence is important for those who are providing beans to others. Often those beans must be able to work with whatever target database the customer has. Given that enterprise beans represent intellectual property, they most likely will not ship with their source code. This means that if an entity bean uses BMP, the customer cannot easily tweak the data access logic. For these vendors, CMP is the only alternative to shipping multiple versions of the same bean code.

Unfortunately, there is no standard way to specify the actual O/R mapping with CMP. Each container has its own tools, wizards, and/or mapping files that specify which fields correspond to which database columns. But what if you want to install your bean in a different container? You'll need to respecify your persistent operations using the new container's tools. If you have a complex object model, this could become a hefty task. Furthermore, since not all

application servers support your complex persistence needs, your beans may not be portable to other containers.

Because of this, sometimes using BMP and allowing your SQL to be hand-tuned via EJB environment properties (see Chapter 9) is the way to go if you want to be application server- and database-neutral.

Relationships

The EJB 2.0 CMP model offers many useful relationship features to bean providers. These include referential integrity, cardinality, relationship management, and cascading deletes. The container can take care of all these issues for you.

With BMP, you must write the scaffolding code to manage and persist the relationships between entity beans. This can get very hairy. You'll notice that a big chunk of your BMP code is dedicated to managing these relationships, which decreases time-to-market and makes your beans more difficult to understand.

Learning Curve and Cost

Most developers already understand how to perform relational database access from Java, and thus BMP does not require much of a learning curve. In comparison, some advanced EJB servers ship with complex O/R mappers for CMP. These mappers provide useful functionality, but do require training and ramp-up time. They also might cost some money, depending on your vendor's policy.



Most people would love to go with CMP, but are afraid to risk its not being flexible enough. If you're in this category, you have a path to try out CMP before you buy. You can make all your entity beans use CMP, and then *subclass* those beans as necessary if you want to use BMP. This works because CMP entity beans are abstract classes.

Choosing the Right Granularity for Entity Beans

If you do decide to go with entity beans (instead of session beans plus JDBC, or session beans plus Java classes), then you need to decide on the *granularity* of your entity beans. The granularity refers to how big (or small) your entity beans are.

In the past, entity beans were restricted to only represent large chunks of data, involving complex joins across multiple tables. Now with the advent of local interfaces and the new CMP model, the container can make many more optimizations. This means the idea of small-grained entity beans is much more viable—if you tune your entity beans properly (see Chapter 13 for tips here).

Another choice you have is to make some of your entity beans be Java classes. These Java classes would hang off other entity beans. For example, you could have an order entity bean that has a vector of line-item Java classes. If you're using CMP, the EJB specification refers to these Java classes as *dependent value classes*. Don't be confused by the terminology—this is just Sun's way of giving a fancy name to Java classes that hang off a CMP entity bean.

For example, here is a dependent value class:

```
package examples;

public class LineItem implements java.io.Serializable {

    private String product;
    private int quantity;

    public void setProduct(String product) { this.product = product; }
    public String getProduct() { return product; }
    public void setQuantity(int quantity) { this.quantity = quantity; }
    public int getQuantity() { return quantity; }
}
```

If you're going to use them, then you should know that there are a few rules for dependent value classes:

- Dependent value classes are defined as CMP fields, and they work just like CMP fields. For example, rather than having a *java.util.String* CMP field, you might have a custom class like *examples.LineItem* instead. Everything we learned about how to use CMP fields in Chapter 7 applies to these custom Java classes too.
- Dependent value classes may not be container-managed relationship (CMR) fields, which we learned about in Chapter 11. Relationships only exist between entity beans, not Java classes.
- Dependent value classes cannot contain references to other entity beans. For example, this sequence of references would be illegal: order (entity bean) points to line item (dependent value class) points to address (entity bean).
- Dependent value classes must be serializable. Getting and setting them are performed by-value rather than by-reference. This hampers performance, but does allow you to access it via the remote interface.

The real value of dependent value classes over entity beans is that they are quick to develop. The downside is that you lose many of the entity bean benefits described earlier in this chapter.

Persistence Tips and Tricks

In this section, we'll present a number of best practices when performing object-to-relational mapping.

Beware the Object-Relational Impedance Mismatch

The object-oriented paradigm, which EJB follows, is based on proven software engineering principles for building applications out of objects that have both data and behavior. The relational paradigm, however, is based on proven mathematical principles for efficiently storing data. Difficulties arise when you attempt to use object and relational technologies together, such as EJBs and relational databases, because of the *impedance mismatch* between the two paradigms. The impedance mismatch becomes apparent when you look at the preferred approach to access: With the object paradigm you traverse objects via their relationships, whereas with the relational paradigm you join the data rows of tables. This fundamental difference results in a nonideal combination of object and relational technologies. Of course, when have you ever used two different things together without a few hitches? To be successful using EJB and relational databases together is to understand both paradigms and their differences, and then make intelligent tradeoffs based on that knowledge.

Hard-Coded Versus Soft-Coded SQL

Most developers hard-code SQL into their BMP entity beans. We showed an example of this in Chapter 6. The problem with this approach is that when your data schema changes, you need to update your source code, retest it, compile it, and redeploy it.

Another possibility is to take a soft-coded approach to SQL, where the mapping of your EJB object schema to your database schema is maintained outside your EJBs. You can keep a list of database mappings in a file or a database, or internally as a data collection, or you can use EJB environment properties accessed via JNDI (see Chapter 9 for more on this). The advantage of the soft-coded approach is that you need to update only the meta data representing your mappings, not the EJB code itself, along the same lines that CMP works for entity beans.

To implement soft-coded SQL within your session beans, you could either build a mapping facility yourself or adopt one of several Java persistence layers/frameworks. The high-level design of a persistence layer, as well as links to several vendors of commercial and open source products, is provided at www.ambyssoft.com/persistenceLayer.html.

When to Use Stored Procedures

Stored procedures are operations that run within a database. A stored procedure typically runs some SQL code, massages the data, and then hands back a response in the form of zero or more records, or a response code, or as a database error message. In the past, stored procedures were written in a proprietary language, such as Oracle's PL/SQL, although Java is quickly becoming the language of choice for database programming. You can invoke stored procedures from a J2EE deployment via JDBC.

The following code invokes a stored procedure (thrown exceptions omitted):

```
// Define the code to invoke a stored function
CallableStatement orderCounter = connection.prepareCall(
    "{call ? = COUNT_CUSTOMER_ORDERS[?]}" );

// Invoke the stored function
orderCounter.registerOutParameter(1, java.sql.Types.FLOAT);
orderCounter.setInt(2, customer.getCustomerID() );
orderCounter.execute();

// Get the return value
numberOfOrders = orderCounter.getFloat(2);

// End the transaction and close the connection
connection.commit();
orderCounter.close();
```

So now that you've seen how to call stored procedures, when should we use them in an EJB environment? Here are some good use cases.

1. **Performance.** Often you're performing data intensive operations with small result sets, and stored procedures then become very appetizing. For example, a good candidate for a stored procedure would be to produce counts listing the number of critical orders (criticality defined by a business rule involving a list of preferred customers, preferred products, and order total) that have been outstanding for more than 30, 60, or 90 days. This operation is data intensive; it would need to take a pass at every order record that has been outstanding for more than 30 days and run it through the defined business rule to determine if it is critical or not. This

involves an amount of data that you wouldn't want to bring across the network to an EJB application server, convert to objects, and then process accordingly. The stored procedure could do all the work on the database server and simply send back the three resulting numbers. Stored procedures are also precompiled, resulting in performance wins.

2. **Shared business rules.** We encourage organizations to first and foremost strive towards centralizing on an EJB layer for all their applications. However, due to political reasons, the reality is that this may not be feasible for all organizations. When your application must share a relational database with other non-EJB applications, such as a legacy system or Microsoft-based system, the database becomes an option for implementing your business rules. This is especially true when legacy applications are unable to access better approaches to implementing business rules, such as an EJB application server or a business rules server. As a result, your relational database becomes the most viable option to implement shared business rules because it is the lowest common denominator that your suite of applications can interact with.
3. **Data security access control logic.** If you have external systems touching your database without going through your EJB layer, you can secure your data through stored procedures. For example, you may want to give another department access to view salary data, but not update it.
4. **Legacy database encapsulation.** You often find that you need to write stored procedures to present a clean view of a legacy database to your EJBs. Most legacy designs are completely inappropriate for access by object-oriented code, or non-object code for that matter, yet cannot easily be reworked due to the large number of legacy applications coupled to them. You can create stored procedures to read and write records that look like the objects that you want. Dealing with legacy databases is discussed later in this chapter.
5. **Centralized SQL.** The SQL is kept in the stored procedures and is written by database experts who excel at writing optimized SQL and do not need to know Java.
6. **Easier migration for fast-changing schemas.** If your database schema changes, then compiling a stored procedure will result in a compile-time error. This makes it easy to find out the ripple effect on schema changes, which is very useful if your schema is being enhanced at a high velocity. SQL code from Java can only be debugged at runtime or by combing through your code.

Note that there are also many reasons to avoid the use of stored procedures:

1. The server can quickly become a bottleneck using this approach. You really need to be careful when moving functionality onto your server: A stored procedure can bring the server to its knees if it is invoked often enough.
2. Stored procedures that are written in a proprietary language can be problematic if you want to be able to port your application to another database vendor in the future. It is quite common to find that you need to port your database to scale it to meet new transaction volumes—don't underestimate the importance of portability. These proprietary languages also increase your learning time before you're productive.
3. You dramatically increase the coupling within your database because stored procedures directly access tables, coupling the tables to the stored procedures. This increased coupling reduces the flexibility of your database administrators. When they want to refactor the database schema, they need to rewrite stored procedures.
4. You increase the maintenance burden for your application because those who maintain your system need to deal with application logic in two places: your EJBs and stored procedures. Your system will become messy over time and difficult to deal with.

The following statement sums up our thoughts on stored procedures: Use them only when necessary.

Normalizing and Denormalizing

When building your data model, you'll often be confronted with a space versus time tradeoff. For example, if you have an *order* that uses a *customer*, you can keep the two separate and unique in the database, or you can copy the customer data into the order table. By duplicating the customer information, you may make queries for orders faster, since you don't have to JOIN across several tables. *Data normalization* is the process of eliminating data redundancy in a database, while *denormalization* is the process of increasing redundancy for performance.

The advantage of having a highly normalized data schema is that information is stored in one place and one place only, reducing the possibility of inconsistent data. Furthermore, highly normalized data schemas in general are closer conceptually to object-oriented schemas, such as those you would create for

your EJB design because the object-oriented goals of promoting high cohesion and loose coupling between classes results in similar solutions (at least from a data point of view). This generally makes it easier to map your EJBs to your data schema.

The disadvantage of normalized data schemas is that when put into production, they often suffer from performance problems. An important part of data modeling is to denormalize portions of your data schema to improve database access times.

For example, often by analyzing the relationships between data, you will see many opportunities for denormalization. One-to-one relationships, such as those between *customer* and *address*, are often prime candidates for denormalization. Their data may be stored in a single *customer* table to improve performance (the address data would be stored as one or more columns within the *customer* table). This is particularly true of leaf tables, tables that are related to only one other table, a trait that the *address* table also exhibited.

Note that if your initial, normalized data design meets the performance needs of your EJBs, it is fine as is. Denormalization should be resorted to only when performance testing shows that you have a problem with your beans and subsequent profiling reveals that you need to improve database access time. Enterprise-ready databases such as Oracle, Sybase, and DB2 include data access monitoring tools that allow you to do exactly this. But if it ain't broke, don't fix it.

Table 12.1 summarizes the three most common normalization rules describing how to put data entities into a series of increasing levels of normalization. Strategies for achieving normalization are classic database challenges that are beyond the scope of this book. *An Introduction to Database Systems, 7th Edition* by C.J. Date (Addison-Wesley, 2000) goes into greater detail.

Table 12.1 Data Normalization Rules

LEVEL	RULE
First normal form	A data entity is in 1NF when it contains no repeating groups of data.
Second normal form	A data entity is in 2NF when it is in 1NF and when all of its non-key attributes are fully dependent on its primary key.
Third normal form	A data entity is in 3NF when it is in 2NF and when all of its attributes are directly dependent on the primary key.



When you are trying to track down the source of an EJB performance problem, you'll often discover that database access is the source of the problem. This is why it is important for your data design to be driven by your EJB design, and for you to be prepared to move away from a pure/normalized database design to one that is denormalized to reflect the actual performance needs of your EJBs.

Use Your EJB Object Model to Drive Your Data Model

For EJB components to map well to a relational database, your EJB schema and relational database schema must reflect one another. This evokes the question should your EJB object model drive your data model or the other way around? Whenever you are given the choice, your EJB object model should drive the development of your data model. Data models take into account only half of the picture (data), whereas object-oriented EJB models take into account the entire picture (data and behavior). By using your EJB models to drive the development of your data models, you ensure that your database schema actually supports the needs of your EJB components.

Note that for this to work, you need to have the freedom to define your data schema, which you will not have if you have to work with a legacy data schema. You also may find that you're not allowed to define the data model; rather, another group at your organization handles that. This approach often proves to be a disaster, resulting in poor performance and significant rework later in the project. In reality, data modeling is an iterative approach. You will likely need to make several iterations of your object model based on feedback from your data modeling efforts, and vice versa.

Follow a Good Data Design Process

Your life as an EJB programmer accessing a relational database will be much more sane if you apply a process to object/relational mapping. We recommend the following steps:

1. Develop a data schema based on your object schema. Strip away the operations from each class, declare the classes to be tables, and remove any tables that have no attributes. Associations between classes, including inheritance, simple associations, aggregation, and composition are translated into relationships between tables. It is important to understand that this provides you with a starting point, not a final solution.

2. Apply data naming conventions. Your organization may have naming conventions for the names of tables and columns; if so, apply them as appropriate. For example, the customer table may be called `TCustomer` and the first name column of that table `FIRST_NAME_C`.
3. Identify keys for each data entity. Each table should have a primary key, one or more columns that uniquely identify an individual row in the table. Foreign keys need to be introduced to implement relationships between tables, and many-to-many relationships between tables need to be resolved via the introduction of an associative table.
4. Normalize and/or denormalize your data schema as required. You normalize your data schema to improve the robustness of your design, although you may find that you need to denormalize occasionally.
5. Refactor your object schema and your data schema as required. Performance problems require that your team tune the container, change the EJB object model, or change the relational database schema to improve the data access times of your EJBs.

Use Surrogate Keys

A common challenge in EJB deployments is to generate unique primary keys. You can generate two basic types of keys

A **natural key** is one or more existing data attributes that are unique to the business concept. For example, a customer table might have two candidate natural keys, *CustomerNumber* and *SocialSecurityNumber*.

A **surrogate key** is a key that has no business meaning, such as an *AddressID* column of an address table. Addresses don't have an easy natural key because you would need to use all of the columns of the address table to form a key for it. Introducing a surrogate key is therefore a much better option in this case.

The foremost advantage of natural keys is that they already exist; you don't need to introduce a new, unnatural value to your data schema. However, the primary disadvantage of natural keys is that because they have business meaning, they may need to change if your business requirements change. For example, if your users decide to make *CustomerNumber* alphanumeric instead of numeric, in addition to updating the schema for the customer table (which is unavoidable), you would have to change every single table where *CustomerNumber* is used as a foreign key. If the customer table instead used a surrogate key, the change would have been localized to just the customer table

itself (*CustomerNumber* in this case would just be a non-key column of the table). Naturally, if you needed to make a similar change to your surrogate key strategy, perhaps adding a couple of extra digits to your key values because you've run out of values, you would have the exact same problem. This points out the need to set a workable surrogate key strategy.

For a key to remain a surrogate, you must never display its value, never allow anyone to edit it, and never allow anyone to use it for anything other than identification. As soon as you display or edit a value you give it business meaning, which effectively makes it a natural key. For example, a *CustomerNumber* could have been originally intended to serve as a surrogate key, but if one day a customer number is printed on an invoice, the customer number has effectively evolved into a natural key. Ideally nobody should know that the persistent object identifier even exists, except perhaps the person(s) debugging your data schema during initial development of your application.



It's important that your primary keys are unique. There are dozens of ways to generate unique keys, such as using a database's built-in counter, an entity bean, an RMI-IIOP object, the current System time, and so forth. Each approach has its advantages and disadvantages. This discussion is fully presented in Floyd Marinescu's book, *EJB Design Patterns*.

Understand the Impacts of Database Updates

It is important to recognize that changes to database data affect the state of the EJB components that represent that data in your application server. A database should not be updated, either by an EJB or a non-EJB application, if the impact of those changes is not fully understood. You can prevent that from happening by setting an internal policy that all database access should go through a common persistence layer (of either session beans, entity beans, or both) and championing that policy to all developers who access that database.

Versioning EJB Components

Sometimes you might need to track *versions* of an EJB component, which means to access old information that no longer exists in your bean. For example, if a customer suddenly got married, her last name might change. You might want to access her maiden name when trying to get information about her that may be stored in a different system. As another example, the historical titles that an employee has held at your organization might be important data for you to determine the next title in her career path.

To develop a versionable EJB component, you have several strategies at your disposal:

1. As your object changes, record those changes in an audit log. You can store entire objects in the log, or you can just store the deltas (changes) to your objects. You might write to this log by using an XML structure or serialized string. To restore an object from the log, either read the object of the appropriate version in, or (if you're using deltas) perform a manual merge.
2. Add versioning columns to your tables. Tables representing versionable objects require several columns to be added, as described in Table 12.2. Whenever an object is updated, a new record is inserted into the appropriate table(s) and made the current version. The previously current version is closed, the *CurrentVersion* column is set to false, and the *EffectiveEnd* column is set to the current datetime. Note that both of those columns are optional: You can determine which row represents the current version of the object by taking the one with the most recent value for *EffectiveStart*, and a previous version can be restored for a specific point in time by taking the row with the effective start date just previous to the requested point in time. This approach is called the *immutable object* design pattern.
3. Add historical tables. With this strategy you have one set of operational tables for your system that you use as you normally would, and a corresponding set of historical tables that have the same schemas with the addition of the *EffectiveEnd* column described in Table 12.2. When an

Table 12.2 Potential Table Columns to Support Versioning

COLUMN	TYPE	PURPOSE
CurrentVersion (Optional)	Boolean	Indicates whether the row represents the current version of the object, simplifying retrieval for most business transactions.
EffectiveStart	Datetime	Indicates the beginning of the period when the values contained in the row were valid. Must be set to the current datetime when the row is first inserted.
EffectiveEnd (Optional)	Datetime	Indicates the end of the period when the values contained in the row were valid. The value is set to the current datetime when the replacement version of an object is first inserted.

object is updated or deleted, the operational tables are changed in the normal way. In addition, the values that had been initially retrieved into memory are written to the corresponding historical table(s), with the `EffectiveEnd` value set to the current datetime.

A few quick observations about making your EJB components versionable:

- The addition of versioning columns is not an option if you are mapping to a legacy database schema.
- For any of these approaches to work, all systems/objects accessing your database must follow them consistently.
- Versioning is performance intensive, requiring additional writes to support updates and deletions as well as more complex retrievals.
- There is no explicit support for versioning with CMP entity beans. If you're using CMP, check your EJB container's documentation to see if it supports versioning.
- If your EJB object model and database schemas vary wildly, the audit log approach is likely your best bet.
- These approaches focus on the versioning of data only, not behavior. To version behavior, you need to support different versions of the same classes and/or apply the *strategy* or *command* design patterns.
- Avoid versioning if you can because it is complex, error-prone, and negatively affects performance.

Living with a Legacy Database Design

For the sake of simplicity we have assumed throughout this chapter that you are in a position to define your data schema. If this is actually your situation, consider yourself among the lucky few. The vast majority of EJB developers are often forced to tolerate an existing legacy design, one that is often difficult, if not impossible, to change because of corresponding changes that would be required to the legacy applications that currently access it. The problem presented by your legacy database is often too difficult to fix immediately; you therefore have to learn to work around it.



This section is *not* about general integration with non-RDBMS legacy systems, such as an SAP R/3 system or a CICS/COBOL system. For strategies on integration with legacy systems, see Chapter 13.

How do you learn to live with a legacy data design? The first step is to understand the scope of the challenge. Start by identifying and understanding the impact of typical data-related problems that you will encounter with legacy data. Table 12.3 lists the most common data problems and summarizes their potential impact on your application. You will likely experience several of these problems in any given database, and any given table or even column within the database will exhibit these problems.

Table 12.3 is lengthy and intended for reference purposes only—you don't need to read or understand the entire table right now. When you encounter a legacy database and want to migrate that into an EJB environment, return to this table.



Both data and database design problems have a common impact on your EJB components: They make it harder to take advantage of CMP because your EJB container needs the ability to overcome the problems appropriately. For those living with a hairy legacy design, we recommend BMP or session beans plus JDBC.

The good news is that your project team isn't the only one facing these sorts of challenges—almost every organization has these problems. As a result, a large market exists for tools to help deal with legacy databases. A sampling is listed in Table 12.4. The basic features are extraction of legacy data, transformation of the legacy data to cleanse it, and the loading of that data into a new data schema that is more robust. Products that support all of these features are referred to as ETL (extract, transform, load) tools.

Patterns for Things That Change with Time

Martin Fowler has developed a pattern language for the development of versionable objects. Posted online at www.martinfowler.com, the language consists of the following patterns:

- 1. Audit log—A simple log of changes, intended to be easily written and nonintrusive.**
- 2. Effectivity—Add a time period to an object to show when it is effective.**
- 3. Snapshot—A view of an object at a point in time.**
- 4. Temporal object—An object that changes over time.**
- 5. Temporal property—A property that changes over time.**
- 6. Time point—Represents a point in time to some granularity.**

Table 12.3 Typical Legacy Data Problems

PROBLEM	EXAMPLE	POTENTIAL IMPACT
A single column is used for several purposes.	Additional information for an inventory item is stored in the Notes column. Additional information will be one or more of a lengthy description of the item, storage requirements, or safety requirements when handling the item.	<ul style="list-style-type: none"> ■ One or more attributes of your EJB components may need to be mapped to this field, requiring a complex parsing algorithm to determine the proper usage of the column. ■ Your EJB component(s) may be forced to implement a similar attribute instead of implementing several attributes as your design originally described.
The purpose of a column is determined by the value of one or more other columns.	If the value of <i>DateType</i> is 17, <i>PersonDate</i> represents the date of birth of the person. If the value is 84, <i>PersonDate</i> is the person's date of graduation from high school. If the value is between 35 and 48, it is the date the person entered high school.	<ul style="list-style-type: none"> ■ A potentially complex mapping is required to work with the value stored in the column.
Incorrect data values	The <i>AgeInYears</i> column for a person row is -3 or the <i>AgeInYears</i> column contains 7, although the <i>BirthDate</i> is August 14, 1967 and the current date is October 10, 2001.	<ul style="list-style-type: none"> ■ Your EJB components need to implement validation code to ensure that their base data values are correct. ■ Strategies to replace incorrect values may need to be defined and implemented. ■ An error-handling strategy needs to be developed to deal with bad data. This may include logging of the error, attempting to fix the error, or dropping the data from processing until the problem is corrected.
Inconsistent/incorrect data formatting	The name of a person is stored in one table in the format <i>FirstName Surname</i> , yet in another table, <i>Surname, FirstName</i> .	<ul style="list-style-type: none"> ■ Parsing code will be required to both retrieve and store the data as appropriate.
Missing data	The date of birth of a person has not been recorded in some records.	<ul style="list-style-type: none"> ■ See strategies for dealing with incorrect data values.

continues

Table 12.3 Typical Legacy Data Problems (*continued*)

PROBLEM	EXAMPLE	POTENTIAL IMPACT
Missing columns	You need a person's middle name, but a column for it does not exist.	<ul style="list-style-type: none"> ■ You may need to add the column to the existing legacy schema. ■ You might need to do without the data. ■ Identify a default value until the data is available. ■ An alternate source for the data may need to be found.
Additional columns	The Social Security number for a person is stored in the database and you don't need it.	<ul style="list-style-type: none"> ■ For columns that are required for other applications, you may be required to implement them in your EJB components to ensure that the other applications can use the data your application generates. ■ You may need to write the appropriate default value to the database when inserting a new record. ■ For database updates, you may need to read the original value, and then write it out again.
Multiple sources for the same data	Customer information is stored in three separate legacy databases.	<ul style="list-style-type: none"> ■ Identify a single source for your information and use only that. ■ Be prepared to access multiple sources for the same information. ■ Identify rules for choosing a preferred source when you discover the same information is stored in several places.
Important entities, attributes, and relationships hidden and floating in text fields	A Notes text field contains the information Clark and Lois Kent, Daily Planet Publications.	<ul style="list-style-type: none"> ■ Develop code to parse the information from the fields. ■ Do without the information.

PROBLEM

Data values that stray from their field descriptions and business rules

EXAMPLE

The maiden name column is being used to store a person's fabric preference for clothing.

POTENTIAL IMPACT

- You need to update the documentation to reflect the actual usage.
- Bean providers that took the documentation at face value may need to update their code.
- Data analysis should be performed to determine the exact usage in case different applications are using the field for different purposes.

Various key strategies for the same type of entity

One table stores customer information using SSN as the key, another uses the ClientID as the key, and another uses a surrogate key.

- You need to be prepared to access similar data via several strategies, implying the need for similar finder operations in some classes.
- Some attributes of an entity bean may be immutable—their value cannot be changed—because they represent part of a key in your relational database. Note that these attributes would not be part of the primary key class for your entity bean.

Unrealized relationships between data records

A customer has a primary residence and a summer home. Both of his homes are recorded in your database, but there is no relationship stored in the database regarding this fact.

- Data may be inadvertently replicated. Eventually a new address record is inadvertently created (and the relationship now defined) for the summer home even though one already exists.
- Additional code may need to be developed to detect potential problems. Procedures for handling the problems will also be required.

One attribute is stored in several fields.

The person class requires a single name field, but is stored in the columns *FirstName* and *Surname* in your database.

- Potentially complex parsing code may be required to retrieve, and then save the data.

continues

Table 12.3 Typical Legacy Data Problems (continued)

PROBLEM	EXAMPLE	POTENTIAL IMPACT
Inconsistent use of special characters	A date uses hyphens to separate the year, month, and day, whereas a numerical value stored as a string uses hyphens to indicate negative numbers.	<ul style="list-style-type: none"> ■ Complexity of parsing code increases. ■ Additional documentation is required to indicate character usage.
Different data types for similar columns	A customer ID is stored as a number in one table and a string in another.	<ul style="list-style-type: none"> ■ You may need to decide how you want the data to be handled by your EJBs and then transform it to/from your data source(s) as appropriate. ■ If foreign key fields have a different type than original data they represent, then table joins, and hence any SQL embedded in your EJBs, become more difficult.
Different levels of detail	A bean requires the total sales for the month, but your database stores individual totals for each order; or a bean requires the weight of individual components of an item, such as the doors and engine of a car, but your database records only the aggregate weight.	<ul style="list-style-type: none"> ■ Potentially complex mapping code may be required to resolve the various levels of detail.
Different modes of operation	Some data is a read-only snapshot of information, but other data is read-write.	<ul style="list-style-type: none"> ■ The design of your EJBs must reflect the nature of the data they are mapped to. EJBs based on read-only data, therefore, cannot update or delete it.
Varying timeliness of data	The customer data is current, address data is one day out of date, and the data pertaining to countries and states is accurate to the end of the previous quarter because you purchase that information from an external source.	<ul style="list-style-type: none"> ■ Your EJB code must reflect, and potentially report to their clients, the timeliness of the information that they are based on.

PROBLEM

Varying default values

Your EJB uses a default of green for a given value, yet another application has been using yellow, resulting in a preponderance (in the opinion of your users) of yellow values stored in the database.

POTENTIAL IMPACT

- You may need to negotiate a new default value with your users.
- You may not be allowed to store your default value (green is an illegal value in the database).

Various representations

The day of the week is stored as T, Tues, 2, and Tuesday in four separate columns.

- Translation code back and forth between a common value that your EJB(s) use will need to be developed.

Database encapsulation scheme exists .

Access to the database is provided only through stored procedures; for example, to create a new customer you must invoke a specified stored procedure. Access to views on the database is permitted; direct table access is denied. The database must be accessed via an API implemented by a C or COBOL wrapper that in turn accesses the database directly. The database must be accessed via predefined data classes/objects.

- To enable CMP the encapsulation scheme must be made to look like a data source that your persistence container recognizes. Otherwise you will be forced to take a Session Beans plus JDBC or BMP approach to persistence.
- The encapsulation scheme will likely increase the response time of database access.
- The individual components of the encapsulation scheme may not be able to be included as a step in a transaction.

Naming conventions

Your database(s) may follow different naming conventions from one another and likely do not follow common Java naming conventions.

- The bean deployer(s) will need to understand all relevant naming conventions.
- Political pressure may be put on your team to follow inappropriate corporate data naming conventions for use with your EJBs.

Inadequate documentation

The documentation for your database is sparse, nonexistent, or out of date.

- A significant legacy data analysis effort will be required to determine the proper usage of each table, column, and stored procedure within your database.

continues

Table 12.3 Typical Legacy Data Problems (*continued*)

PROBLEM	EXAMPLE	POTENTIAL IMPACT
Original design goals are at odds with current project needs.	The legacy database was built for internal use by data entry clerks to capture customer orders in batch mode, whereas you are building a 24x7 order entry application to be deployed over the Internet.	<ul style="list-style-type: none"> ■ Good luck. You'll need it.
Inconsistent key strategy	Your database uses natural keys for some tables, surrogate keys in others, and different strategies for surrogate keys when they are used.	<ul style="list-style-type: none"> ■ Bean providers must understand, and then appropriately code support for the various key strategies for their EJBs. ■ Key generation code increases in complexity to support the various strategies. ■ Additional source code to validate that natural keys are, in fact, unique will be required. ■ Relationship management code increases in complexity because you can't code, and then reuse a single approach.

Table 12.4 Sample Legacy Data Integration Tools

TOOL	URL
Informatica PowerCenter	www.informatica.com
ETI*Extract	www.evtech.com
Information Logistics Network	www.d2k.com
Ascential Software's DataSage	www.ascentialsoftware.com
INTEGRITY Data Re-Engineering Environment	www.vality.com
Trillium Control Center	www.trilliumsoft.com

Once you've identified the challenges in your legacy integration efforts, the second step is to determine how you will address the problems that you have found with your legacy data and the legacy database design. Table 12.5 compares and contrasts several strategies at your disposal.

Although descriptions of how to implement these three strategies is clearly beyond the scope of this book, we can provide some advice:

1. Do not underestimate the effort required to address this problem. If it can be done at all, data migration/improvement efforts often prove to be a project, or a series of projects, that span several years. This is easily on the order of magnitude of your organization's Year 2000 (Y2K) efforts.
2. Think small. A series of small changes, or refactorings, is often preferable to a single big-bang approach in which you need to re-release all of your organization's applications at once. Martin Fowler's book *Refactoring: Improving the Design of Existing Code* (Addison Wesley, 1999) describes the principles and practices of refactoring. It should provide some insight into how to make incremental changes to your legacy data design (many of his refactorings are geared to changing object-oriented designs, but the fundamentals still apply).
3. Did we mention not to underestimate the effort required?

Handling Large Result Sets

A serious problem when writing finder methods of entity beans (or any relational database code for that matter) is handling larger-than-expected result sets. If you locate too much data, you are causing undue performance issues, because you may not need the entire result set. To handle this situation, you have several strategies at your disposal:

Table 12.5 Strategies for Mitigating Legacy Data Problems

STRATEGY	ADVANTAGES	DISADVANTAGES
Create your own private database for new attributes.	<ul style="list-style-type: none"> ■ You have complete control over your database. ■ You may be able to avoid conforming to legacy procedures within your organization, speeding up development. 	<ul style="list-style-type: none"> ■ Replication of common data is likely. ■ Unable to easily take advantage of the existing corporate legacy data. ■ May still be required to integrate with the legacy corporate database(s) via triggers, programmed batch jobs, or ETL tools. ■ Your team must have database expertise. ■ Your project risks significant political problems because you may be perceived as not being team players.
Refactor your data schema.	<ul style="list-style-type: none"> ■ You have a clean database design to work with. ■ Your database schema can be redesigned to reflect the needs of modern, object-oriented and component-based technologies, such as EJB. 	<ul style="list-style-type: none"> ■ This is very difficult to achieve. ■ Legacy applications will need to be updated to reflect the new data schema. ■ You will need to identify and fix all of your data-related problems, requiring significant effort. ■ You need to develop, and then follow, procedures to ensure that your database design remains clean; otherwise you will end up in the same position several years from now.
Encapsulate database access with stored procedures, views, data classes/objects, or an API.	<ul style="list-style-type: none"> ■ A clean access approach to encapsulation can be presented to application developers. 	<ul style="list-style-type: none"> ■ Legacy applications should be rewritten to use the new access approach to ensure integrity within the database. ■ Implementing your encapsulation strategy may require significant effort. ■ Your encapsulation approach may become an architectural bottleneck. ■ Depending on the range of technologies within your organization, you may not be able to find one strategy that works for all applications.

Table 12.5 Strategies for Mitigating Legacy Data Problems (*continued*)**STRATEGY**

Design your EJBs to work with the existing design as is.

- Your EJBs work with the legacy database(s).

DISADVANTAGES

- Significant redesign and coding is likely to be required for this to work.
- The actual problem, a poor database design, is not addressed and will continue to affect future projects.
- This may not be feasible, depending on the extent of the mismatch between the legacy database design and the requirements for your application.
- Performance is likely to be significantly impacted because of the resulting overhead of mapping your EJBs to the database and the transformations required to support those mappings.
- Common approaches to persistence, such as CMP and the use of a persistence layer/framework, is likely not an option if the mismatch is too great.

- Add code to estimate the size of the result set, a feature most relational databases support, and throw an exception if it's too big. This works for Session Beans plus JDBC.
- Learn to live with the large result set. If it doesn't happen very often, it might not be worth your while to write code to deal with this. This works for Session Beans plus JDBC, BMP, and CMP.
- Write tighter SELECT statements by introducing additional parameters to narrow the results. This works for Session Beans plus JDBC, BMP, and CMP.
- Limit the results of the finder via the SQL bound to the method (a feature of most databases).
- Use a scrollable result set. JDBC 2.0 introduced the ability to access the results of SELECT clauses as database cursors, enabling bean providers to write code that brings portions of the result set across the network at a time. This works for Session Beans plus JDBC.
- Use session beans to control how the result set is handled. For example, you can use a stateful session bean that caches a collection of primary keys. When the client requests data, return only a block (say 20 pieces) of data at a time based on the primary keys.
- Let your persistence container handle it. Some persistence containers, such as WebGain's *TOPLink*, implement strategies for dealing with large result sets. This works for entity beans.

Summary

In this chapter, we touched on a variety of best practices and strategies when performing persistence in an EJB environment. We learned when (and when not to) use entity beans, and how to choose between BMP and CMP, and we surveyed a large collection of persistence best practices.

In the next chapter, we'll take a look at an advanced EJB topic—design strategies.

EJB Best Practices and Performance Optimizations

In this chapter, we will discuss *EJB best practices*—tried-and-true approaches to designing, building, and working with EJB. By being aware of these best practices, you will avoid common pitfalls that others have experienced when building EJB systems. We'll also discuss performance issues when building EJB systems.

Let's begin now with our collection of development strategies.



This chapter does *not* cover low-level EJB design patterns. We started to put those together but realized that those patterns deserved a book of their own. That's what gave birth to Floyd Marinescu's book, *EJB Design Patterns*, published by John Wiley & Sons and a companion to this book.

When to Use Stateful versus Stateless

Lately there's been a lot of fuss over statelessness. The limitations of statelessness are often exaggerated, as are its benefits. Many statelessness proponents blindly declare that statelessness leads to increased scalability, while stateful backers argue about having to rearchitect entire systems to accommodate statelessness. What's the real story?

Designed right, statelessness has two virtues:

- With stateless beans, the EJB container is able to easily pool and reuse beans, allowing a few beans to service many clients. While the same

paradigm applies to stateful beans, if the server is out of memory or has reached its bean instance limit, then the bean state may be passivated and activated between method calls, possibly resulting in I/O bottlenecks. So one practical virtue of statelessness is the ability to easily pool and reuse components at little or no overhead.

- Because a stateful session bean caches a client conversation in memory, a bean failure may entail losing your conversation. This can have severe repercussions if you don't write your beans with this in mind or if you don't use an EJB product that provides stateful recovery. In a stateless model, the request could be transparently rerouted to a different component because any component can service the client's needs.

The largest drawback to statelessness is that you need to push client-specific data into the stateless bean for each method invocation. Most stateless session beans need to receive some information that is specific to a certain client, such as a bank account number for a banking bean. This information must be resupplied to stateless beans each time a client request arrives because the bean cannot hold any state on behalf of a particular client.

One way to supply the bean with client-specific data is to pass the data as parameters into the bean's methods. This can lead to performance degradation, however, especially if the data being passed is large. This also clogs the network, reducing available bandwidth for other processes.

Another way to get client-specific data to a stateless bean is for the bean to store data persistently on behalf of a client. The client then does not need to pass the entire state in a method invocation but simply needs to supply an identifier to retrieve the data from persistent storage. The trade-off here is, again, performance—storing conversations persistently could lead to storage I/O bottlenecks, rather than network I/O bottlenecks.

Yet another way to work around the limitations of statelessness is for a bean to store client-specific data in a directory structure using JNDI. The client could later pass the bean an identifier for locating the data in the directory structure. This is quite similar to storing data in a database. The big difference is that a JNDI implementation could be an in-memory implementation (an effect similar to a shared property manager, familiar to COM+ readers). If client data is stored in memory, there is no database hit.

When choosing between stateful and stateless, you should ask if the business process spans multiple invocations, requiring a conversation. If so, the stateful model fits nicely because client-specific conversations can be part of the bean state. On the other hand, if your business process lasts for a single method call, the stateless paradigm better suits your needs.

Note that if you are going to use state, and if you're building a Web-based system, you may be able to achieve what you need with a servlet's *HttpSession* object, which is the Web server equivalent to a stateful session bean and is easier to work with because it does not require custom coding. We have found that a stateful session bean should be used over an *HttpSession* when:

- You need a stateful object that's transactionally aware. Your session bean can achieve this by implementing *SessionSynchronization*, described in Chapter 10.
- You have both Web-based and non-Web-based clients accessing your EJB layer, and both need state.
- You are using a stateful session bean to temporarily store temporary state for a business process that occurs *within* a single HTTP request and involves multiple beans. To understand this point, consider that you are going through a big chain of beans, and a bean deep in the chain needs to access state. You could marshal the state in the parameter list of each bean method (ugly and could be a performance problem if you're using remote interfaces). The better solution is to use a stateful session bean and just pass the object reference through the stack of bean calls.

In summary, most sophisticated deployments are likely to have a complex and interesting combination of the stateless and stateful paradigm. Use the paradigm that's most appropriate for your business problem. The one exception is if there is an obvious bottleneck, such as keeping megabytes of state in memory. But if you are on the fence about stateful versus stateless, you may find that statefulness may not be your primary issue—until you test your code, you're just shooting in the dark. If it turns out that your statefulness is your bottleneck, you could refactor your code if necessary.

When to Use Messaging versus RMI-IIOP

Another hot topic when designing an EJB object model is choosing when (and when not) to use messaging, rather than RMI-IIOP.

The following advantages of messaging provide reasons why you might want to use it:

Database performance. If you are going to perform relational database work, such as persisting an order to a database, it may be advantageous to use messaging. Sending a message to a secondary message queue to be processed later relieves stress on your primary database during peak hours. In the wee hours of the morning, when site traffic is low, you can

What If My Stateful Bean Dies?

Bean failure is an important factor to consider. Because a stateful session bean caches a client conversation in memory, a bean failure may entail losing your conversation. This was not a problem with statelessness—there was no conversation to be lost. Unless you are using an EJB product that routinely checkpoints (that is, persists) your conversations, your conversations will be lost if an application server fails.

Losing a conversation has devastating impacts. If you have large conversations that span time, you've lost important work. And the more stateful session beans that you use in tandem, the larger the existing network of interconnected objects that each rely on the other's stability. This means that if your code is not prepared for a failure, you may have a grim situation. Not an exciting prospect for mission-critical computing, is it?

When designing your stateful beans, use the following guidelines:

- Make sure your problem lends itself to a stateful conversation.
- Keep your conversations short.
- If the performance is feasible, consider checkpointing stateful conversations yourself, to minimize the impacts of bean failure.
- Write smart client code that anticipates a bean failure and reestablishes the conversational state with a fresh stateful session bean.

process messages off the message queue and insert the orders into the database. Note that this only works if the user doesn't need an immediate answer to whether his operation was a success; it would not work, for example, when checking the validity of a credit card.

Quick responses. A client may not want to block and wait for a response that it knows does not exist. For methods that return *void*, the only possible return values are nothing or an exception. If a client never expects to receive an exception, why should it block for a response? Messaging allows clients to process other items when they would otherwise be blocking for the method to return.

Smooth load balancing. In Chapter 8, we discussed how message-driven beans distribute load more smoothly than session or entity beans. With session and entity beans, a load-balancing algorithm makes an educated *guess* about which server is the least burdened. With messaging, the server that is the least burdened and asks for a message, gets the message. This also aids upgrading your system, since you merely need to detect when your

queue size reaches a threshold. This indicates that the number of consumers is not great enough, which means you need to add new machines.

Request prioritization. Asynchronous servers can queue, prioritize, and process messages in a different order than they arrive into the system. Some messaging systems allow message queues to be prioritized to order messages based upon business rules. For example, in a military battle tank, if all requests for the system sent to a centralized dispatch queue are made asynchronously, disaster could result if a fire control message was queued up behind 100 communication messages that had to be processed first. In a military system, it would be advantageous to process any fire control and safety messages before communication messages. A prioritized queue would allow for the reordering of messages on the queue to account for the urgency of fire control in a battle tank.

Rapidly assembling disparate systems. Many legacy systems are based on message-oriented middleware and can easily interact with your J2EE system through messaging. Messaging provides a rapid development environment for systems that have distributed nodes that perform business processing and must communicate with one another.

Loosely coupled systems. Messaging enables loose coupling between applications. Applications do not need to know about each other at compile time. This empowers you to have *dynamic discovery* of applications, which may be useful in a rapidly changing, service-oriented business environment.

Geographically disperse systems. Messaging is very useful when you have applications communicating over the Internet or a wide-area network. The network is slow and unreliable, and RMI-IIOP is not intended for such broad-range communications. Messaging along with guaranteed message delivery adds an element of safety to your transactions. Another choice you have is Web services technologies (XML/HTTP).

Parallel processing. Messaging is a way to perform pseudo-threading in an EJB deployment. You can launch a series of messages and continue processing, which is the distributed equivalent of launching threads.

Reliability. Messaging can be used even if a server is down. System-level problems (such as a database crashing) typically do not affect the success of the operation, because when you're using *guaranteed message delivery* the message remains on the queue until the system-level problem is resolved. Even if the message queue fails, message producers can spool messages and send them when the queue comes back up (called *store and forward*). By combining guaranteed message delivery with store-and-forward, the system will not lose any requests unless there is a complete system failure at all tiers (extremely unlikely).

Many-to-many communications. If you have several parties communicating together, messaging is appropriate since it enables many producers and many consumers to collaborate, whereas RMI-IIOP is a single-source, single-sink request model.

The following are scenarios for which you *might not* want to use messaging:

When you're not sure if the operation will succeed. RMI-IIOP systems can throw exceptions, whereas message-driven beans cannot.

When you need a return result. RMI-IIOP systems can return a result immediately because the request is executed immediately. Not so for messaging. You can return results eventually with messaging, but it's clunky—you need to send a separate return message and have the original client listen for it.

When you need an operation to be part of a larger transaction. When you put a message onto a destination, the receiving message-driven bean does not act upon that message until a future transaction. This is inappropriate when you need the operation to be part of a single, atomic transaction that involves other operations. For example, if you're performing a bank account transfer, it would be a bad idea to deposit money into one bank account using RMI-IIOP and then withdraw money using messaging, because the withdrawal might fail.

When you need to propagate the client's security identity to the server. Since messaging does not propagate the client's security identity to the receiving message-driven bean, you cannot easily secure your business operations.

When you are concerned about request performance. Messaging is inherently slower than RMI-IIOP because there's a middleman (the JMS destination) sitting between the sender and the receiver.

When you want a strongly-typed, OO system. You send messages using a messaging API such as JMS. This is a flat API and is not object-oriented. If you want to perform different operations, the server needs to crack open the message or filter it somehow. In comparison, RMI-IIOP allows you to call different business methods depending on the business operation you want to perform. This is much more intuitive. It's also easier to perform compile-time semantic checking.

When you want a tighter, more straightforward system. Synchronous development tends to be more straightforward than messaging. You have great freedom when sending data types, and the amount of code you need to write is minimal compared to messaging. Debugging is also much more straightforward. When using services that are completely synchronous,

each client thread of control has a single execution path that can be traced from the client to the server and vice versa. The effort to trace any bugs in the system is thus minimal.

How to Guarantee a Response Time with Capacity Planning

Many types of business problems are trivial, such as basic Web sites or non-mission critical applications. But then there are those that *must not fail* and must *guarantee* a certain response time. For example, a trading application needs to guarantee a response time because stock market conditions might change if the trade is delayed. For those serious deployments, capacity planning is essential for your deployment.

The specific amount of hardware that you'll need for your deployment varies greatly depending on the profile of your application, your anticipated user load, and the EJB server you choose. The major EJB server vendors each have strategies for capacity planning that they can share with you.

One strategy, however, works with all EJB server vendors. The idea is to *throttle*, or *limit*, the amount of work any given EJB server instance can process at any one time. Why would you ever want to limit the amount of work a machine can handle? A machine can only guarantee a response time for the clients it serves and be reliable if it isn't using up every last bit of hardware resources it has at its disposal. For example, if your EJB server runs out of memory, it either starts swapping your beans out to disk because of passivation/activation, or it uses virtual memory and uses the hard disk as swap space. Either way, the response time and reliability of your box is jeopardized. You want to prevent this from happening at all costs by limiting the amount of traffic your server can handle at once.

You can throttle (limit) how much traffic your machine can handle using a variety of means. One is by limiting the *thread pool* of your EJB server. By setting an upper bound on the number of threads that can execute concurrently, you effectively limit the number of users that can be processed at any given time. Another possibility is to limit the *bean instance pool*. This lets you control how many EJB components can be instantiated at once, which is great for allowing more requests to execute with lightweight beans and fewer requests to execute with heavyweight beans.

Once you've throttled your machine and tested it to make sure it's throttled correctly, you need to devise a strategy to add more machines to the deployment in case your cluster-wide capacity limit is reached. An easy way to do

this is to have a standby machine that is unused under normal circumstances. When you detect that the limit is breached (such as by observing message queue growth, indicating that your servers cannot take consume off the queue fast enough), the standby machine kicks in and takes over the excess load. A system administrator can then be paged to purchase a new standby machine.

This algorithm guarantees a response time because each individual server cannot exceed its limit, and there's always an extra box waiting if traffic increases.

How to Achieve Singletons with EJB

A *singleton* is a very useful design pattern in software engineering. In a nutshell, a singleton is a single instantiation of a class with one global point of access. You would normally create a singleton in Java by using the *static* keyword when defining a class. However, one restriction of EJB is that you cannot use static fields in your beans. This precludes the use of the singleton design pattern. But we'd still like to use singletons—how?

The answer is JNDI. You can use JNDI to store arbitrary objects to simulate the singleton pattern. If all your objects know of a single, well-known place in a JNDI tree where a particular object is stored, they can effectively treat the object as a single instance. You can perform this by binding an RMI-IIOP stub to a JNDI tree. Any client code that accessed the JNDI tree would get a copy of that remote stub, and each copy would point back to the same RMI-IIOP server object.

The downside to this pattern is you are leaving the EJB sandbox and downgrading to vanilla RMI-IIOP, and thus you lose all the services provided by EJB.

Wrap Entity Beans with Session Beans

Consider the following scenarios:

- A bank teller component performs the business process of banking operations, but the data used by the teller is the bank account data.
- An order-entry component performs the business process of submitting new orders for products, such as submitting an order for a new computer to be delivered to a customer. But the data generated by the order-entry component is the order itself, which contains a number of order line-items describing each part ordered.
- A stock portfolio manager component performs the business process of updating a stock portfolio, such as buying and selling shares of stock. But

the data manipulated by the portfolio manager is the portfolio itself, which might contain other data such as account and stock information.

In each of these scenarios, business process components are manipulating data in some underlying data storage, such as a relational database. The business process components map very well to session beans, and the data components map very well to entity beans. The session beans use entity beans to represent their data, similar to how a bank teller uses a bank account. Thus, a great EJB design strategy is to wrap entity beans with session beans.

Another benefit of this approach is performance. Accessing an entity bean directly over the network is expensive, due to:

- The stub
- The skeleton
- Marshaling/demarshaling
- The network call
- The EJB object interceptor

You can minimize these expensive calls to entity beans by wrapping them with session beans. The session beans perform bulk create, read, update, delete (CRUD) operations on behalf of remote clients. The session bean also serves as a transactional façade, enforcing that transactions occur on the server, rather than involving a remote client. This makes entity beans into an *implementation detail* of session beans. The entity beans are never seen to the external client; rather, entity beans just *happen to be* the way that the session bean performs persistence.

A final benefit of this approach is that your entity beans typically achieve a high level of reuse. For instance, consider an order entry system, where you have an order submission session bean that performs operations on an order entity bean. In the next generation of your application, you may want an order fulfillment session bean, an order reporting session bean, and so on. That same order entity bean can be reused for each of these session beans. This approach allows you to fine-tune and change your session bean business processes over time as user requirements change.

Thus, in practice you can expect the reuse of entity beans to be high. Session beans model a current business process, which can be tweaked and tuned with different algorithms and approaches. Entity beans, on the other hand, define your core business. Data such as purchase orders, customers, and bank accounts do not change very much over time.

There are also a few of caveats about this approach:

- You can also wrap entity beans with other entity beans, if you have a complex object model with relationships.

- The value of session beans as a network performance optimization goes away if you do not have remote clients. This could occur, for example, if you deploy an entire J2EE application into a single process, with servlets and JSPs calling EJB components in-process. However, the session façade could still be used for proper design considerations and to isolate your deployment from any particular multitier configuration. If you're lazy, an alternative is to use an entity bean's home business methods, which are instance-independent business methods that act effectively as stateless session bean methods, except they are located on the entity bean.
- Note that what we've presented here are merely guidelines, not hard-and-fast rules. Indeed, a session bean can contain data-related logic as well, such as a session bean performing a bulk database read via JDBC. The key is that session beans never embody permanent data, but merely provide access to data.

Performance-Tuning Entity Beans

There is a great deal of fear, uncertainty, and doubt (FUD) in the industry about entity beans. Many organizations are using them improperly, creating performance issues. Here are some tips and tricks to make your entity beans high performing:

- Entity beans should not be called directly from remote clients, but rather from session or entity beans located in the same process. Because of this, you should always call entity beans through their local interfaces, not their remote interfaces.
- Use your container's caching options as much as possible. If your beans are read-only, instruct your container to keep them cached permanently. If they are read-mostly or read-write, many containers have algorithms to deal with this. Remember: Memory is cheap.
- Be sure your transactions run on the server, are as short as possible, and encapsulate all of the entity bean operations you'd like to participate in that transaction. This is important because the JDBC occurs at the beginning and end of transactions. If you have a transaction occurring for each entity bean get/set operation, you are performing SQL hits on each method call. The best way to perform transactions with entity beans is to wrap all your entity bean calls within a session bean method. Deploy both the session and entity beans with the container-managed transaction attribute of *Required*. This creates a transaction in the session bean that encapsulates all entity beans in the same transaction.

- Consider having your container batch JDBC updates all at once at the end of the transaction. That way, if you perform many JDBC operations in a single transaction, you only need one network call to the database.
- For performance, use container managed persistence, if possible. As convoluted as it may sound, container managed persistence can actually be higher-performing than bean managed persistence, for reasons outlined in Chapter 12. Just make sure that you're using a good persister that gives you great flexibility when performing O/R mapping.
- If you are not going to access your entire entity bean's data on each transaction, lazy-load some of your fields rather than loading it all when the entity bean is first accessed. You can lazy-load your fields programmatically using BMP by fine-tuning your JDBC code, or you can lazy-load your fields declaratively using CMP if your container tools support it.
- If you're using CMP, instruct your container to persist fields in bulk. For example, BEA Weblogic has the notion of field groups. This empowers you to define groups of fields (even across relationships) that persist together, reducing the amount of SQL required.
- If you're using CMP, use your container tools to force the container to have your finder methods automatically load your bean, rather than having finders and loading happen separately, resulting in two SQL statements. The only time you should not do this is if you're not going to read data from your entity bean (for example, setting fields, but not getting fields).
- Send output to a good logging/tracing system, such as a logging message-driven bean. This allows you to understand the methods that are causing bottlenecks, such as repeated loads or stores.
- Use a performance-profiling tool to identify bottlenecks, such as Optimizelt or JProbe. If your program is hanging on the JDBC driver, chances are the database is your bottleneck.

Performance-tuning your entity beans opens up the possibility to create fine-grained entity beans that model a single row in the database, as well as coarse-grained entity beans that model a complex set of data spanning multiple tables.

Choosing between Local Interfaces and Remote Interfaces

Local interfaces, a new feature in EJB 2.0, allow you to access your EJB components without incurring network traffic. They also allow you to pass

nonserializable parameters around, which is handy. So what is the value of a remote interface? Well, there really isn't a value, unless:

- You need to access your system remotely (say from a remote Web tier).
- You are trying to test EJB components individually and need to access them from a standalone client to perform the testing.
- You need to allow your containers more choices for workload distribution and failover in a clustered-server environment.

For optimal performance, we recommend that you build your system using all local interfaces, and then have one or more session bean wrappers with remote interfaces, exposing the system to remote clients.

Note that the problem with local and remote interfaces is that the code is slightly different for each paradigm. Local interfaces have a different interface name and do not use *PortableRemoteObject.narrow()*, and there are no *RemoteExceptions* thrown. Because of this, you need to recode your clients when switching between the two paradigms. This is an unfortunate consequence of having a programmatic approach to performance-tuning.

To limit the amount of recoding you'll need to do, decide whether the clients of your beans are going to be local or remote *before you start coding*. For example, if you're building a Web-based system, decide whether your system will be a complete J2EE application in a single process, or whether your Web tier will be split off from your EJB tier into a separate process. We discuss the trade-offs of these approaches in Chapter 14.

As a final note, if you are connecting to your EJB deployment from a very distant client (such as an applet or application that gets downloaded by remote users), consider exposing your EJB system as an XML-based Web service, rather than a remote interface. This will be slower than a straight RMI/IIOP call, but is more appropriate for WAN clients.

How to Debug EJB Issues

As EJB is evolving quickly, the containers are evolving as well. The containers or their tools often have small oddities. In addition, users may introduce bugs that are difficult to debug. How do you debug with EJB?

Unfortunately, true debugging is a problem with EJB. Because your beans run under the hood of a container, you'd have to load the container itself into a debugger. But for some containers, this is impossible because you don't have access to the container's source code, or the source code has been obfuscated. For these situations, you may need to use the tried-and-true debugging method of logging.

An even more serious debugging problem occurs if exceptions are being thrown from the EJB container, rather than from your beans. This can happen for a number of reasons:

Your EJB container's generated classes are incorrect because your interfaces, classes, or deployment descriptor haven't fully complied with the EJB specification. Your EJB container's tools should ship with compliance checkers to help resolve this. But know that not everything can be checked. Often because of user error, your deployment descriptor will not match your interfaces. This type of problem is extremely difficult to target, especially if your container tools crash!

Your EJB container has a real bug. This is a definite possibility that you must be prepared to encounter. In the future, however, this should not happen very often because EJB containers that comply with J2EE must test their implementations against Sun Microsystems' robust test suite.

A user error occurs within the EJB container. Probably the most frustrating part of an application is doing the database work. Punctuation errors or misspellings are tough to debug when performing JDBC. This is because your JDBC queries are not compiled—they are interpreted at runtime, so you don't get the nifty things like type checking that the Java language gives you. You are basically at the mercy of the JDBC driver. It may or may not give you useful feedback. For example, let's say that you're modeling a product, and you use the word *desc* rather than *description* to describe your products. Unfortunately, the keyword *desc* is an SQL reserved keyword. This means that your JDBC driver would throw an exception when trying to execute any database updates that involved the word *desc*. These exceptions might be cryptic at best, depending on your JDBC driver. And when you try to figure out what JDBC code is acting up, you will run into a roadblock: With container managed persistence, the JDBC code won't be available because your bean does not perform its own data access! What do you do in this situation?

When you're faced with grim situations like this, contacting your EJB vendor is probably not going to be very helpful. If you are operating with a deadline, it may be too late by the time your vendor comes up with a solution. If you could only somehow get access to the JDBC code, you could try the query yourself using the database's tools.

You can try several options here:

- Some EJB containers support IDE debugging environments, allowing you to step through your code in real time to pinpoint problems. This is something you should look for when choosing a container.
- Check your database's logfile to view a snapshot of what is really happening.

- Use a JDBC driver wrapper that logs all SQL statements (one is available from Provision6, Inc).
- Your EJB container tools may have an option to keep generated Java files, rather than to delete them when compiling them into classes. For example, you can do this with BEA's WebLogic with the *keepgenerated* option to its EJB compiler tool. This is analogous to how you can use the *keepgenerated* option to keep generated proxies with Java RMI's *rmic* compiler.
- As a last resort, you may have to decompile the offending classes to see what's going on. A good decompiler is *Jad* by Pavel Kouznetsov (see the book's accompanying Web site for a link). Of course, decompiling may be illegal, depending on your container's license agreement.

Partitioning Your Resources

When programming with EJB, we've found it very handy to separate the kinds of resources your beans use into two categories: *bean-specific resources* and *bean-independent resources*.

Bean-specific resources are resources your bean uses that are tied to a specific data instance in an underlying storage. For example, a socket connection is a bean-specific resource if that socket is used only when particular bank account data is loaded. That is, the socket is used only when your bean instance is bound to a particular EJB object. Such a resource should be acquired when a bean instance is created in *ejbCreate()* or when activated in *ejbActivate()* and released when the instance is removed in *ejbRemove()* or passivated in *ejbPassivate()*.

Bean-independent resources are resources that can be used over and over again, no matter what underlying data your instance represents. For example, a socket connection is a bean-independent resource if your bean can reuse that socket no matter what bank account your bean represents (that is, no matter what EJB object your bean instance is assigned to). Global resources like these should be acquired when your bean is first created, and they can be used across the board as your bean is assigned to different EJB objects. When the container first instantiates your bean, it associates you with a context object (such as the *setEntityContext()* method); then you should acquire your bean-independent resources. Similarly, when you are disassociated with a context object (such as the *unsetEntityContext()* method), you should release bean-independent resources.

Because acquiring and releasing resources may be costly operations, categorizing your resources as outlined is a vital step. Of course, the most stingy way

to handle resources is to acquire them on a *just-in-time* basis and release them directly after use. For example, you could acquire a database connection only when you're about to use it and release it when you're done. Then there would be no resources to acquire/release during activation/passivation. In this case, you'd let the container pool your resources and thus manage the resources for you. The disadvantage is you need to code resource requests/releases over and over again in your bean code.

Assembling Components

The EJB paradigm enables corporations to assemble applications from existing prewritten components that solve most of the business problem already. As good as this sounds, assembling applications from disparate components is not all roses. The problem with assembling heterogeneous components is getting them all to work together. For example, let's say you purchase a bean that computes prices, and you combine it with some home-grown entity beans, such as an order bean and a product bean. Let's assume we also use a billing component from a different vendor. How do you get these components to work together? None were created with the knowledge of the others.

There is no easy answer to this problem. EJB defines standard interfaces for components to be deployable in any container, but EJB cannot specify how domain-specific components interact. For example, EJB cannot specify the de facto bean to represent a product or an order because each corporation models these differently in its existing information systems.

Unfortunately, you're always going to need to write some workflow component that maps to each vendor's proprietary API and object model. The only way you can get around mapping to APIs is if a standards committee decides on an official object model for a problem domain, such as standardizing what a purchase order looks like. Problem domains such as pricing are open and customizable, which makes this a large challenge.

A second problem with having these components work together is data mapping. How does the billing component understand the data computed by the pricing component? Sure, you might be able to call the billing component's API, but it won't magically know how to deal with the data passed to it. The data was formatted by another vendor's component. You're going to need to write an adapter object that bridges the gap between the two formats. If you purchase components from n vendors, you're going to spend all your time writing adapter code. This is quite mindless and boring, although it is often better than writing your own business logic that you'll have to create, maintain, and test.

The final issue that must be overcome to get components to work together is that every participant component must agree on a standard representation, or schema, for exchanged data. This is a trivial problem when a single vendor writes the components, because that vendor can simply invent a data schema and include it with its components. This becomes a monstrous problem, though, when integrating heterogeneous vendors' components.

We therefore recommend that organizations purchase either small, fine-grained horizontal components that will work anywhere (like a logging service), or large, coarse-grained components that all come from a single vendor (like a sales force automation suite of components). We do not recommend trying to mix and match medium-to-large-sized component groups that make different data schema assumptions.

Developing Components to Be Reusable

Our next best-practice addresses the challenge of developing reusable components. This may be important, for example, if you're developing beans to be reused by other departments within your organization.

First, let's do a reality check. Don't believe anyone who tells you that enterprise beans are reusable by definition—because that is *false*. You need to design your beans correctly if you want them to be reusable. You need to consider the different applications, domains, and users of your enterprise beans, and you need to develop your beans with as much flexibility as possible. Developing a truly reusable set of beans will likely require many iterations of feedback from customers using your beans in real-world situations.

Roughly speaking, bean reusability can fall into three different levels:

Reuse as given. The application assembler uses the acquired bean as it is to build an application. The bean functionality cannot be tailored to fit the application. Most projects will have a difficult time reusing these components because of their inflexibility.

Reuse by customization. The application assembler configures the acquired bean by modifying the bean properties to fit the specific needs of the application. Bean customization typically occurs during development time. To allow for a more flexible maintenance environment, some bean providers allow runtime bean customization.

Reuse by extension (subclass). The application assembler creates custom application-specific beans by subclassing the prebuilt acquired beans. The behavior of the resulting bean is tailored for the application. This level of

reusability is generally more powerful but difficult to achieve. Reuse by extension is made available by only a few bean providers.

The more reusability levels that a bean provides, the more useful a bean is. By leveraging prebuilt beans, organizations can potentially lower the development time of building enterprise applications.



Many organizations have tried—and failed—at truly reusing components. Because of this, it is a perfectly valid strategy to not attempt *true* reuse at all. Rather, you can shoot for a *copy-and-paste reuse* strategy, which means to make the source code for components available in a registry to other team members or other teams. They can take your components and change them as necessary to fit their business problem. While this may not be true reuse, it still offers many benefits. Another approach to reuse is to divide up your applications into Web services that call one another. The Middleware Company offers a service to help organizations rearchitect their applications in this manner.

When to Use XML in an EJB System

XML is a popular buzzword these days, and so we should discuss the appropriateness of XML in an EJB deployment.

XML is useful in the following scenarios:

- As an interface to legacy systems. If you have a large number of legacy systems, or even if you have one big hairy legacy system, you'll need a way to view the data that you send and receive from the legacy system. XML can help you. Rather than sending and receiving data in proprietary structures that the legacy system might understand, you can invent an XML façade to the legacy systems. The façade takes XML input from your EJB components and maps that XML to the proprietary structures that the legacy system supports. When the legacy system returns data to your EJB application, the XML façade transforms the legacy data into XML data that your EJB application can understand. This is also useful for business analysts involved with a J2EE application. They can help perform the data mapping by working with XML, a technology that's fairly simple to understand.
- As a document persistence mechanism. If you are persisting large documents (news reports, articles, books, and so on), representing those documents using XML may be appropriate.
- As a Web service interface. As described in Chapter 1, EJB components can also be wrapped as a Web service. XML then becomes the on-the-wire

data format sent between Web services. We have a free whitepaper on how to build a J2EE-based Web service, as well as how to call another Web service from a J2EE system, available on www.TheServerSide.com.

The one important scenario that XML is not useful for is as an on-the-wire format for communication *between* EJB components.

The idea is that rather than application components sending proprietary data to each other, components could interoperate by passing XML documents as parameters. Because the data is formatted in XML, each component could inspect the XML document to determine what data it received.

Although several J2EE-based workflow solutions use this approach, XML is often inappropriate for EJB-EJB communications because of performance. Parsing XML documents takes time, and sending XML documents over the wire takes even longer. For high-performance enterprise applications, using XML at runtime for routine operations is costly. The performance barrier is slowly becoming less important, however, as XML parsers become higher performing and as people begin to use text compression to send XML documents over the wire. However, it is still the bottleneck in many systems.

Another important reason not to use XML is because it's often simply not needed. Assuming that a single organization writes all your EJB components, there is less need for data mapping between disparate systems, since you control the object model.

Legacy Integration with EJB

Most large-scale EJB/J2EE deployments involve one or more existing systems. Devising a strategy for dealing with these legacy systems is critical. You have two basic choices:

Rewrite that existing system using EJBs. This option is the cleanest solution but requires the most effort. It may, however, be infeasible. Legacy systems tend to be complex. Developers who understand the legacy system may be difficult to find, and the time-to-market needs of the organization may not permit a rewrite. Finally, the performance of existing systems that use native code may not be acceptable in the Java world.

Bridge into that existing system. The *if it ain't broke, don't fix it* rule dictates that this is the most straightforward solution. However, you will need to maintain the bridged solution, which uses two different technologies.

If you decide to bridge into existing systems, we recommend wrapping your legacy system with an EJB layer rather than accessing it directly (from a servlet

or JSP), because this abstraction layer will enable you to replace the legacy system in the future, if you so desire. The EJB layer could be session beans, entity beans, message-driven beans, or all three. The choice of which EJB components to use depends on the nature of your existing system.

If your existing system is highly data-oriented, entity beans are a good choice. The entity beans would represent the legacy system data as Java objects. Note, however, that the data must be able to be represented in an object-oriented way for this to work; otherwise you should consider using session beans.

If your existing system is highly process-oriented, session beans are a good choice. The session beans then serve as a façade to your existing business process engines.

If your existing system uses message-oriented middleware as its interface, message-driven beans are a good choice. You can *send* messages to the existing system using a session bean or message-driven bean. Then a message-driven bean can *receive* messages from the existing system.

The next challenge is how to actually achieve the bridge to the existing system. That is, what is happening inside the EJB layer that talks to the existing system? Again, you have several choices.

Proprietary bridges. You can buy an off-the-shelf bridge that connects to a specific legacy system, perhaps an EJB-COM bridge or a container-provided API. The disadvantage of these proprietary bridges is a loss of portability, since there is no guarantee this code will run in other J2EE-compliant servers.

The Java Native Interface (JNI). JNI enables you to bridge into native code, such as C++ code. The advantage of the JNI is that it's faster than the other approaches. The disadvantages are that it can't connect to any system (just native code), the existing system needs to run-in process, and JNI is platform-specific—so if your code needs to run on multiple platforms you're multiplying the testing and maintenance effort.

The Common Object Request Broker Architecture (CORBA). CORBA is an older middleware technology that competes with EJB (since it has its own component architecture) and also underlies EJB (since some J2EE servers are old CORBA products wearing a new hat). The big difference between CORBA and EJB/J2EE is that CORBA is language-neutral, while EJB/J2EE is specialized to Java. While CORBA has this advantage over EJB/J2EE, it has very little industry momentum behind it and is more appropriate as a technology for performing integration with existing systems. You can bridge into code written in almost any language by calling that legacy

system via CORBA APIs from within your EJB layer. This is highly appropriate for existing systems that are already CORBA-based. The disadvantages of CORBA integration is that it requires an out-of-process remote call which slows performance, and it also requires that you learn a whole new technology if you don't know CORBA already.

Java Message Service (JMS). JMS (along with message-driven beans) enables you to bridge to existing systems using message-oriented middleware. You send messages to existing systems rather than invoking them directly through API calls. This is a bit slower, but also is a loosely coupled paradigm that enables you to build complex messaging workflows. JMS is highly appropriate if your existing system already uses messaging.

Web services. Web services (essentially XML/HTTP) is an attractive approach to integrating to existing systems. You'd use XML to represent the data sent to existing systems, and HTTP is your transport, which allows you to navigate firewalls easily. This is a nonintrusive approach because any system that is Internet-enabled can use Web services without need of a whole separate communications infrastructure such as CORBA or JMS. The disadvantage of Web services is that the XML parsing overhead may slow you down. See www.TheServerSide.com for a J2EE Web services whitepaper.

The J2EE Connector Architecture (JCA). The JCA is a specification that enables you to acquire drivers that connect with existing systems and plug them into your J2EE server to connect to a legacy system. You can connect to any existing system for which drivers exist, and if no driver exists (such as a proprietary internal system you've built in-house), you can write your own driver. A marketplace of JCA-compliant resource adapters is evolving, which is analogous to JDBC drivers connecting to relational databases. Examples include CICS, TUXEDO, MVS/VSAM, SAP R/3, PeopleSoft, and more. This makes the JCA one of the most exciting specifications in J2EE. See <http://java.sun.com/j2ee/connector> for more on the JCA.

Each of these approaches has benefits and drawbacks. You need to decide which approach is right for your business problem. For more help on legacy integration, see Chapter 12.

Summary

In this chapter, we reviewed a series of best practices when working on an EJB project. We hope that you refer back to these design strategies when you begin to work with EJB—after all, an ounce of prevention is worth a pound of cure.

Clustering

In this chapter, we'll talk about *clustering* technology, which exists to address many of the challenges faced by large, high-capacity systems. This chapter also explores many issues relating to EJB and large systems. It is designed to provide you with the broadest possible understanding of the issues as well as solutions that exist to address these issues.

Specifically, we'll cover the following topics:

- Approaches and characteristics of large-scale systems with J2EE application servers
- Approaches that can be employed to instrument clustered EJBs
- Issues related to designing clustered EJB systems
- Issues that impact EJB performance in a clustered system

Overview of Large-Scale Systems

The number of systems being developed is rapidly increasing year after year. Some of these systems are small, targeted at a specific, well-defined user group that is understood when development of the system begins. Other systems are large, targeted at a diverse, massive user group that evolves over time. Given the variety of systems that can be designed, what makes a system large scale? And, more importantly, how can EJB technology operate in a large-scale system?

This section discusses some of the principles behind large systems and defines terminology that will be used throughout the chapter. This section also provides some background and history of theories applicable to large-scale systems in the past.

What Is a Large-Scale System?

Unfortunately, there is no complete computer science definition of a large-scale system. Since requirements for systems vary wildly, what may be considered large for one project is insignificant for another project.

For the purposes of this book, we will define an application server large-scale system as one that requires the use of more than one application server that typically operates in a cluster. A cluster is a loosely coupled group of servers that provide unified services to their clients. Clients that use services deployed into a cluster typically are not aware that their requests are being serviced by a cluster and typically do not have any control over deciding which servers in the cluster process their requests. Servers that operate in a cluster may operate on one or more computers, each of which may have one or more processors.

In comparison, a system that employs multiple databases, firewalls, or Web servers but uses only a single application server would *not* be considered a large-scale system by this definition. This narrowness of the definition is key.



Many organizations fail to estimate the load that their systems will require and so design their system with only small-scale characteristics in mind. For example, we have seen organizations take advantage of threading (which is restricted in EJB), not considering the long-term impact of such a decision when building it out to be large scale.

Because of this, we recommend you always assume the worst: that you will need a large-scale system eventually. This way, you always have a path to follow if your user load increases, due to future business forces that are out of your control.

Large-scale systems can be characterized by the following three properties (collectively called RAS):

Reliability gauges whether the system performs at a constant level as the stresses on that system change. Reliability is *not* the same as performance. For example, if the simplest request takes 10 ms to complete with one user, the system is reliable if the same request takes 10 ms with 1,000,000 concurrent users. The measure of reliability can take many different forms: It can be as broad as supporting a certain number of registered users, or as specific as requiring the round trip time for a single method invocation to be within a discrete range. The most reliable system is one in which a request

can be entirely handled in a single process. Every time a request has to make an interprocess hop, as when a proxy forwards a request to a node in a cluster, the reliability of the system is reduced. For every component added to a system, the number of scenarios that can cause a disruption in reliable service increases and thus decreases the reliability of the overall system. Depending upon the type of architecture employed, a cluster may improve or reduce reliability.

Availability measures the percentage of time that your system is available for use by its clients. A measurement of availability is not related to the effectiveness of servicing those requests; rather, it focuses on whether or not the services are accessible. A system may be unavailable for a variety of reasons, such as network blockage, network latency, or total system failure. The principle of availability states that if the probability of a single application server being available is $1/m$, the probability an application server will be unavailable is $1-1/m$. If there are n application servers in a cluster, the probability of the system being unavailable is $(1-1/m)^n$. The value of $(1-1/m)^n$ decreases as n increases, implying that a cluster will always be more available than a single server.

Serviceability measures how manageable your system is. System management occurs at a variety of levels, including runtime monitoring, configuration, maintenance, upgrades, etc. The principle of serviceability states that two application servers are more complex to service than a single application server. This implies that a cluster is inherently more difficult to service than a nonclustered system.

The takeaway point is that *increasing the reliability of a system impacts its availability and serviceability. A similar argument can be made for attempting to improve the availability or serviceability of a system.* It is important to appreciate that there is no such thing as a perfect system. Any system that has a high level of availability will likely not have a high level of reliability and serviceability.

Basic Terminology

When working on large-scale projects, engineers and developers tend to freely use a variety of terms in relation to clusters and large-scale system without fully understanding the meaning of those terms. Here are definitions for terms used in this chapter:

A cluster is a loosely coupled group of servers that provide a unified, simple view of the services that they offer individually. Servers in a cluster may or may not communicate with one another. Generally, the overall goal of employing a cluster is to increase the availability or reliability of the system.

A **node** is a single server in the cluster.

Load balancing distributes the requests among cluster nodes to optimize the performance of the whole system. A load-balancing algorithm can be systematic or random.

A **highly available system** can process requests even if the initial node designated to handle the request is unavailable.

Fail-over redirects a single request to another node in the cluster because the original node could not process the request.

Request-level fail-over occurs when a packet that is directed to a node for servicing cannot be serviced by that node and is subsequently redirected to another node.

Transparent or automatic session fail-over occurs at the invocation level when a current executing invocation fails and is transparently rerouted to another server to complete execution.

Single access point simplicity is the idea that clients generate requests to the cluster rather than individual nodes of the cluster. The requests are transparently directed to a node in the cluster that can handle the request. The client's view of the cluster is a single, simple system, not a group of collaborating servers.

Transactions per second (TPS) measures the number of transactions executed by a system in a second. A single request can cause zero, one, or more transactions to occur. TPS is a measure of how well the system's transaction management performs and is commonly used to compare the performance of different systems or algorithms.

Requests per second (RPS) measures how many requests can be processed and responded to in a second. In a typical system, a number of threads are responsible for processing messages arriving into the system and placing those messages into a *processing queue*. A separate pool of threads is responsible for taking messages off of the processing queue and actually executing the appropriate service. RPS is a measure of how many messages can be serviced off of the *processing queue* in a second.

Arrivals per second (APS) measures how many incoming messages from clients can be taken from a connection and placed onto a processing queue in a second.

Throughput measures how many requests can be fully serviced in a given time period. This measure includes the processing of arrivals and the handling of requests. If the throughput of a system is high, the system is capable of processing many requests in a given amount of time.

Invocations per second (IPS) measures how many component invocations can be made in a second. IPS usually applies to the number of EJB invocations that can be made in a second.

Partitioning Your Clusters

Now that we've gotten the definitions out of the way, let's look at the different choices you have for how to cluster a J2EE system.

Most modern J2EE servers contain both a Web server and application server. This means that in a Web-based system, the following cluster configurations are possible:

A 3-tier architecture runs the Web server components (servlets and JSPs) and application server components (EJBs) in the same process.

A 4-tier architecture separates the Web server components and application server components into separate processes.

The differences between the two architectures are shown visually in Figure 14.1.



Some may not consider the browser to be a separate tier, since no logic runs in the browser. This doesn't matter for our discussion. All that matters is that you understand the difference between the top picture and the bottom picture.

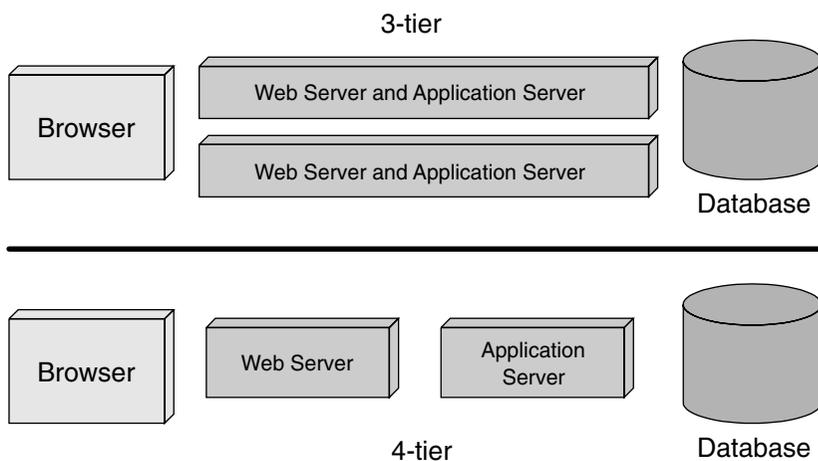


Figure 14.1 3-tier versus 4-tier.

The 3-tier versus 4-tier debate is important because you typically have a fixed budget for your deployment. For example, let's say that we have enough money to buy two machines. We can run a cluster of combined J2EE servers, or we can have a dedicated Web server box and a dedicated application server box. Either way, our budget is fixed, and we can buy two machines.

This debate is scalable to any size project. If you have a larger budget, you can expand and have either n combined J2EE servers, or you can have $n/2$ Web servers and $n/2$ application servers.

The pros and cons of 3-tier versus 4-tier are listed in Table 14.1.

Our favorite is the 3-tier approach. The arguments for 4-tier tend to be weaker in most circumstances. For example:

Static data. The fact that static data can be served faster in 4-tier is not important in many scenarios. Most companies will actually have a separate Web server box just for static data. This can be harnessed in both 3-tier and 4-tier. That static Web page server could also make use of a Storage Area Network (SAN), a network of hard drives to increase throughput.

Security. The fact that you can have a firewall in 4-tier systems (between the Web servers and application servers) doesn't buy you a whole lot. After all, if you expose your EJB components, how many hackers are going to know how to perform a JNDI lookup of your EJB components and call them? Furthermore, you can still secure your 3-tier system by putting a firewall in front of your entire J2EE server cluster.



A common argument in favor of 4-tier is that the presentation and business logic tiers can be scaled separately. Additional nodes can be added to the presentation cluster without adding additional nodes to the back-end cluster and vice versa. This is a flawed argument, because there is no value in being able to scale each tier separately. Remember, we have a fixed budget and can buy only so many machines. We're going to have X machines whether we choose 3-tier or 4-tier. Those machines' hardware resources will be used more efficiently in 3-tier than 4-tier, for reasons listed in Table 14.1.

Instrumenting Clustered EJBs

Although the exact algorithm that your J2EE server uses is proprietary, most application servers have similar approaches to clustering. Let's look at the possible options that application server vendors have for clustering EJBs of all types. We'll then look at the specifics of clustering stateless session, stateful session, entity, and message-driven beans.

Table 14.1 3-Tier Versus 4-Tier

FEATURE	3-TIER	4-TIER	WINNER?
Reliability	High, because there is no interprocess communication. Everything is in a single process.	Low, because there is interprocess communication between Web servers and application servers. There are more network connections and more machines involved with a single request, which means more things can break on a single request.	3-tier
Availability	High, because any machine can fail-over to any other machine.	Higher than no cluster at all, but lower than 3-tier because there are half as many machines that can provide for fail-over in a given tier compared to 3-tier.	3-tier
Serviceability	High, because each box is identical (higher degree of simplicity), and there is no network connection between the Web servers to bother with.	Low, because the Web server cluster must be maintained differently than the application server cluster. There is also a network connection between the tiers.	3-tier
Interfaces used	The Web server components can call high-performing EJB local interfaces.	The Web server components must call the low-performing EJB remote interfaces, which means more marshaling overhead.	3-tier
Network efficiency	No interprocess communication means no sockets to traverse between the Web servers and application servers.	Interprocess communication between Web servers and application servers slows things down significantly.	3-tier
Efficient use of hardware	High, because a J2EE server will be used for whatever purpose its needed for at the current time (Web server tasks or application server tasks).	Low, because you need to predetermine how many machines to devote to Web server tasks, and how many machines to devote to application server tasks. This may not be exactly optimal, and your load characteristics may change over time.	3-tier

continues

Table 14.1 3-Tier Versus 4-Tier (continued)

FEATURE	3-TIER	4-TIER	WINNER?
Security	You can place a firewall between the Web servers and application servers, exposing only what you need to expose to the Internet: the Web servers.	You cannot place a firewall between your Web server and application server, because they are in-process. Therefore, your EJB components are exposed to the world.	4-tier
Site down page	If there is a cluster-wide failure (perhaps due to programmer error) in the application servers or Web servers, the entire site is down.	If there is a cluster-wide failure (perhaps due to programmer error) in the application servers, the Web servers will still be up. You can then display a site down page, which is better than no response at all.	4-tier
Serving quick static data, or simple Web requests that do not involve EJB components	If the application servers are overloaded, static data (such as HTML and images) are served up slowly because the Web servers are competing for hardware resources with the application servers.	If the application servers are overloaded, static data (such as HTML and images) can be served up quickly because the Web servers are dedicated. This is important because most homepages are primarily static content. Most people are willing to wait longer for subsequent requests in your Web site, once they know for sure that the homepage works and the site is up today.	4-tier
Political Ownership	The Web team and EJB team use the same boxes, which could mean ownership conflicts if you're in a political environment.	The Web team and EJB team use different boxes. They don't interfere with each other as much, which is great if you're in a political environment.	4-tier
Load-balancing	You need to set up a separate load-balancing box in front of the <i>J2EE</i> servers. Examples include a software load-balancer running on a cheap Linux box, or a hardware load-balancer like a local director.	You need to set up a separate load-balancing box in front of the <i>Web servers</i> . Examples include a software load-balancer running on a cheap Linux box, or a hardware load-balancer like a local director.	Equal

How EJBs Can Be Clustered

There are many locations that vendors can provide *clustering logic* (such as load-balancing or fail-over logic):

JNDI driver. A vendor could perform load-balancing logic in the JNDI driver (the `InitialContext`) that is used to locate home objects. The JNDI driver could spread traffic across numerous machines.

Container. A vendor could provide clustering logic directly within the container. The containers would communicate with one another behind the scenes using an interserver communication protocol. This protocol could be used to perform clustering operations. For example, if a `ShoppingCart` stateful session bean container has filled up its cache and is constantly activating and passivating EJBs to and from secondary storage, it might be advantageous for the container to send all `create(. . .)` invocations to another container in a different server that hasn't reached its cache limit. When the container's burden has been reduced, it can continue servicing new requests.

Home stub. This object is the first object accessed by remote clients and runs locally on a client's virtual machine. Since stub code is autogenerated by a vendor, the underlying logic in a stub can be vendor-specific. Vendors can instrument method-level load balancing and fail-over schemes directly in a stub. Every `create(. . .)`, `find(. . .)`, and home method invocation can have its request load balanced to a different server in the cluster; it doesn't matter which machine handles each request.

Remote Stub. This object is the client's proxy representing a specific enterprise bean instance. It can perform the same types of load balancing and fail-over as a home stub, but vendors have to be careful about when they choose to do so. Remote stubs must load balance and fail-over requests to instances that can properly handle the request *without disrupting the system*.



The most common scenario is for stubs to be generated at development time through a utility, such as a vendor-specific EJB compiler. This isn't the only option, however. Some application servers can use interception technology such as the JDK 1.3 Proxy class to automatically generate remote home and remote stub logic dynamically at runtime. The JBoss application server is an example of a server that has an EJB container using this approach.

Whether or not an application server uses interception technology or creates custom classes for the stubs and skeletons does not alter the places where cluster-based logic can be inserted. In the following discussions, we continue to reference home stubs, remote stubs, or containers irrespective of how or when these pieces are generated.

The different options that are available to developers and vendors provide a vast array of configurations with which clusterable EJB may be instrumented. By now, you must be thinking, “How do I know what to use, when and where?” The answer lies within the capabilities of any single application server. The rest of this chapter discusses the various issues that application-server vendors face when attempting to provide a clusterable infrastructure for stateless session, stateful session, entity, and message-driven EJBs.



Load-balancing and fail-over logic doesn't exist for local interfaces. Remember that local interfaces do not spread traffic across the network. Parameters must be marshaled by-reference rather than by-value (serialization). If there is no network between the local client and the bean, it does not make sense for a vendor to implement clustering logic for local interfaces. Thus, our discussion applies only to remote clients.

The Concept of Idempotence

An *idempotent* (pronounced i-dim-po-tent, not i-dimp-uh-tent) method is one that can be called repeatedly with the *same* arguments and achieve the *same* results.

An idempotent method in a distributed system doesn't impact the state of the system. It can be called repeatedly without worry of altering the system so that it becomes unusable or provides errant results. Generally any methods that alter a persistent store are not idempotent since two invocations of the same method will alter the persistent store twice. For example, if a sequencer is stored in a database and *m1()* increments the sequencer, two calls to *m1()* will leave the sequencer at a different value than if *m1()* was invoked a single time. An idempotent method leaves the value in the persistent store the same no matter how many invocations of *m1()* occur.

Remote clients that witness a failure situation of a server-side service are faced with a perplexing problem: Exactly how far did the request make it before the system failed? A failed request could have occurred at one of three points:

- After the request has been initiated, but *before* the method invocation on the server has begun to execute. Fail-over of the request to another server should always occur in this scenario.
- After the method invocation on the server has begun to execute, but before the method has completed. Fail-over of the request to another server should only occur if the method is idempotent.
- After the method invocation on the server has completed, but before the response has been successfully transmitted to the remote client. Fail-over

of the request to another server should only occur if the method is idempotent.

Why is this important? A remote stub that witnesses a server failure *never* knows which of the three points of execution the request was in when the failure occurred. Even though failures of requests that haven't even begun method execution should always fail-over, a client can never determine when a failed request is in this category.

Thus, remote stubs can only automatically fail-over requests that were sent to methods flagged as idempotent. In comparison, fail-over of nonidempotent methods must occur programmatically by the client that originated the request. If your EJB server vendor is a major player, it will likely give you the ability to mark an EJB component's method as idempotent or nonidempotent using proprietary container descriptors.



You might think that all methods that are marked to require a new transaction are idempotent. After all, if failure happens, the transaction will roll back, and all transactional state changes (such as transactional JDBC operations) will be undone. So why can't the stub fail-over to another bean to retry the operation?

The answer is that container-managed transactions have an inherent flaw, which we first discussed in Chapter 10. What if the transaction commits on the server, and the network crashes on the return trip to the stub? The stub would then not know whether the server's transaction succeeded or not and would not be able to fail-over.

Stateless Session Bean Clustering

Now, let's take a look at how we can cluster each type of EJB component. We begin with stateless session beans.

Load Balancing

All Java object instances of a stateless session bean class are considered identical. There is no way to tell them apart, since they do not hold state. Therefore all method invocations on the remote home stub and remote stub can be load balanced. Some EJB servers also give you flexibility here, and allow you to *pin* stubs so that they direct requests to a single server only. Some vendors even allow you to configure subsets of methods on a single stub to be pinned or load balanced. This flexibility in load balancing stateless session bean instances is what drives the perception that stateless session EJBs are the most scalable types of synchronous EJB components.

Fail-Over

For stateless session beans, automatic fail-over on *remote home stubs* can always occur. Recall that stateless session bean home stubs have only one method, which is an empty `create()` method. This corresponds to the bean's `ejbCreate()` method. But your container does not call `ejbCreate()` when the client calls `home.create()`—the container can call `ejbCreate()` whenever it feels like kicking beans into the pool, as we saw in Chapter 4. Therefore, your `ejbCreate()` methods should not modify your system's state and should be coded as idempotent.

Automatic fail-over on *remote stubs* can only occur if the called method is idempotent. If your method is nonidempotent, or if your vendor does not support automatic fail-over, you might be able to *manually* fail-over by writing code to retry the method. You need to be careful, however, and factor business rules and other logic into the decision as to whether a fail-over request should be made.

For example, the following pseudo-code manually fails-over any method invocation that is not automatically done so by the remote home or remote stub.

```
InitialContext ctx = null;
SomeHomeStub home = null;
SomeRemoteStub remote = null;

try {
    ctx = ...;
    home = ctx.lookup(...);

    // Loop until create() completes successfully
    boolean createSuccessful = false;
    while (createSuccessful == false) {

        try {

            remote = home.create();

        } catch (CreateException ce) {
            // Handle create exception here.
            // If fail over should occur, call continue;
            continue;

        } catch (RemoteException re) {
            // Handle system exception here.
            // If fail over should occur, call continue;

        } catch (Exception e) {
            // Home stub failure condition detected.
            // If fail over should occur, call continue;
```

```
        continue;

    }

    // If processing gets here, then no failure condition detected.
    createSuccessful = true;

}

boolean answerIsFound = false;
while (answerIsFound == false) {

    try {

        remote.method(...);

    } catch (ApplicationException ae) {
        // Handle application exception here.
        // If fail over should occur, call continue.

    } catch (RemoteException re) {
        // Handle server-side exception here.
        // If fail over should occur, call continue.

    } catch (Exception e) {
        // Failure condition detected.
        // If fail over should occur, call continue.
        continue;

    }

    // If processing gets here, then no failure condition detected.
    answerIsFound = true;

} // while
} catch (Exception e) {}
```

If we wanted it to do so, our EJB component could also assist with this fail-over decision by checking the system state before continuing.

Stateful Session Bean Clustering

Stateful session beans are clustered a bit differently than their stateless cousins. The major EJB server vendors support *replication* of state. It works like this. When a stateful session bean is created, the state must be copied to another machine. The backup copy isn't used unless the primary fails. The bean is routinely synchronized with its backup to ensure that both locations are current. If the container ever has a system failure and loses the primary bean instance, the remote stub of the bean fails-over invocations to another machine. That

other machine can use the backup state and continue processing. A new backup is then nominated, and the state begins to replicate to that new backup. This all occurs magically behind the scenes once you configure your EJB server to replicate state, using your EJB server's proprietary descriptors or administrative console.



Stateful replication should be used with caution. It will limit your performance. Instead, you may want to consider placing critical, transactional, and persistent data in a database via session beans + JDBC or entity beans. Stateful session beans should be used for session-oriented (conversational) data that would not adversely impact the system if the data were lost.

Replication of stateful data typically occurs at one of two points:

At the end of every method. This is not ideal since unnecessary replication of nonaltered data can frequently occur.

After the commit of a transaction. For reasons touched upon in Chapter 10, this is ideal. Transactions give you an all-or-nothing fail-over paradigm. By replicating on transactional boundaries, your stateful session bean state is consistent in time with other changes to your system state (such as performing JDBC work).

Most EJB servers perform stateful fail-over in one of two ways:

In-memory replication. The state could be replicated in-memory across the cluster. In-memory replication is fast. The downside is that most EJB servers limit the replication to only two machines, since memory then becomes a scarce resource.

Persistent storage to a shared hard drive or database. This approach is slower than in-memory replication, but every server in the cluster has access to the persistent state of the replicated bean.

Load Balancing

With stateful session beans, *remote home stubs* can freely load balance *create(...)* requests to different servers in the cluster. These factory methods do not apply to an individual instance in a container but can be serviced by any container in the cluster.

However, *remote stubs* cannot load balance as easily. Your client requests can be sent only to the server that has your state. Note that if your stateful session bean is *replicated* across multiple servers, a remote stub could conceivably load balance different requests to different servers. This wouldn't be ideal, however, since most vendors have a designated *primary* object that requests are

sent to first. The effort involved with load balancing requests in this scenario outweighs any benefits.

Fail-Over

You might think that fail-over can always occur with stateful session beans if the state is replicated across a cluster. After all, if something goes wrong, we can always fail-over to the replica.

However, this is *not* the case. If your bean is in the *middle of a method call*, we still need to worry about idempotency. Your bean might be modifying state elsewhere, such as calling a database using JDBC or a legacy system using the J2EE Connector Architecture. Your stub can fail-over to a backup only if the method is idempotent. The only time your EJB server can disregard idempotency is if your container crashed when nobody was calling it, either between method calls or between transactions, depending on how often you replicate.

For stateful session beans, automatic fail-over on a *remote stub* or *remote home stub* can occur only if your methods are idempotent. Most methods are not idempotent, such as a *create(. . .)* method, which performs JDBC work, or a *set()* method. However, a stateful session beans *can* have some idempotent methods! Any method that does not alter the state of the system or alters the value of the state stored in the stateful session EJB is an idempotent method. For example, if a stateful session EJB has a series of *get()* accessor methods to retrieve the values of state stored in the server, these *get()* accessor methods would be idempotent.

If your method is not idempotent, or if your container does not support replication, you can manually fail-over, similar to our approach to stateless session beans.

Entity Bean Clustering

Now that we've seen session beans, let's see how entity beans are clustered.

Load Balancing

If you're coding your EJB deployments correctly, you will wrap entity beans with a session bean façade. Therefore, most access to entity EJBs should occur over local interfaces by in-process session beans, rather than remote clients. Thus, the need for load balancing goes away. Note that most containers do support load-balancing for entity beans through stubs, similarly to how it works for session beans. But you'll probably never take advantage of it.

Fail-Over

Since you should always access entity beans using local interfaces, fail-over makes little sense. Consider this: If you called an entity bean using a local interface and that failed-over to another machine, you'd suddenly be using its remote interface, changing the client API and entailing pass-by-value rather than pass-by-reference.



If you are accessing entity beans remotely, then as with all other bean types, you can automatically fail-over entity beans only if the methods are idempotent. This usually means *get()*, *finder()*, and possibly some *ejbHome()* business methods.

Entity beans don't have the same replication needs as stateful session beans. This is because entity beans are routinely synchronized with a database via its *store/load* operations. Thus, an entity bean *is* backed up on a regular basis by design. From this perspective, you can think of an entity bean as a stateful session bean that is always replicated by the container on transactional boundaries through *store/load* operations. Those automatic load/store operations are the most important differences between stateful session beans and entity beans.

Since entity beans are backed up on transactional boundaries, transparent fail-over can only occur in-between transactions (and not between methods that are part of a larger transaction). If you have a transaction committing on every method call (for example, through the *Requires New* transaction attribute), fail-over can occur at the method granularity. However, this is not a recommended approach, since your session beans should initiate the transaction and serve as a transactional façade.

Caching

Because entity beans are basically Java objects that represent database data, they are in themselves a middle tier cache for that database. It is a tricky and technically complicated task for an application server to support this cache well. It is also a common misperception that caching always improves the performance of a system. Caching makes a system perform better only when the average overhead associated with updating the cache is less than the overhead that would be needed to access individual instances repeatedly between cache updates. Since the amount of synchronization needed to manage a cache in a cluster is high, a cache generally needs to be accessed three or four times between updates for the benefits of having the cache to outweigh not having it.

Containers provide many different types of caching algorithms. Each of these algorithms has the same principle behind it: to reduce the frequency of *ejbLoad()* and *ejbStore()* methods, which are normally called on transactional boundaries.

You set up these caches using proprietary container tools or descriptors. No Java coding should be required.

Read-Only Caches

A *read-only cache* contains a bunch of read-only entity beans. This is a very useful cache because most enterprise data *is* read-only. This type of caching has enormous benefits.

Since read-only entity beans never change, their *ejbStore()* methods are never called, and they are never called with a transactional context. If your entity bean methods are participating in a read-only cache, they need to have *Never* or *Not Supported* as their transactional attribute.

Read-only caches implement an *invalidation strategy* that determines when the data in the read-only instance is no longer valid and should be reloaded from the persistent store. Common algorithms include:

Timeout. Every *X* seconds, the cache is invalidated and the read-only entity bean is reloaded immediately or upon the next method invocation. You set the time-out interval based on your tolerance for witnessing stale data.

Programmatic. Your vendor provides a home stub or remote stub with *invalidate(...)* or similar methods that allow a client to programmatically invalidate entity beans.

System-wide notification. When someone changes entity beans in a read/write cache, the container invalidates those entity beans that also reside in a read-only cache elsewhere.

It doesn't take long for you to perform operations on a read-only entity bean. The lock on the entity bean only needs to be held just long enough to perform the method call that gets you the data you need. Thus, each server's read-only cache typically keeps a single entity bean instance in memory for each primary key. This saves overhead involved with creating multiple instances and managing the concurrent access.

Distributed Shared Object Caches

A *distributed shared-object cache* is an advanced EJB server feature that few vendors provide today. It is a cluster-wide cache for read/write data. This immediately introduces an obvious problem: *cache consistency*. How does the container stay in sync with the database? What if someone updates the database behind your back? You'll need to refresh your cache.

A distributed shared object cache could theoretically detect collisions at the database level. This might be detected through database triggers, although

this gets very hairy. The idea is that when someone updates the database behind your back, a trigger is fired. The cache is notified by this trigger and updates its contents so that read-only clients can access the latest data. Because each of the servers receives the notification, updating of the data can occur concurrently across the cluster.

A distributed shared object cache also needs stay in sync with other caches in the cluster. It needs to replicate itself to other nodes on regular intervals, similar to the idea of stateful session bean replication. It also needs to implement a distributed *lock manager* that locks objects in memory, similar to how a database locks database rows. Additionally, if a nonreliable messaging infrastructure such as IP multicast is used to send notification messages between servers, a system runs the risk of having two caches trying to lock the same data concurrently—their notification messages might cross in mid-air! An algorithm that allows the pausing of other systems during the period where critical events and notification messages are generated needs to be implemented. As you can see, this convergence of state across multiple nodes is very difficult to implement.



Because of these issues, we do not recommend usage of a distributed shared object cache for most systems. However, if you'd like to give it a shot, we recommend strongly testing your system for a variety of failure conditions before going live.

Read-Mostly Caches

Some application servers provide an exciting *read-mostly* algorithm. This powerful idea allows you to have read-only entity beans that are also updated every now and then, without having the burden of a true distributed shared object cache. The idea is that for any given entity bean class, some instances will be read-only, and some will not be cached at all (read/write).

When you perform a read operation, you use a cached, read-only entity bean for performance.

When you perform a write operation, you use a regular, noncached entity bean. When you modify a regular entity bean and a transaction completes, all of the read-only entity bean caches become invalidated. When the read-only entity beans are next used, they need to reload from the database.

This *read-mostly* pattern has some interesting characteristics:

- Each cache uses a different JNDI name. For example, a read-only cache might have RO appended to the JNDI name, while a read/write cache might have RW appended to the JNDI name. This is somewhat annoying.

- This pattern only requires the use of a read-only cache, which almost all application servers have. You don't need to deal with the complexity of a true distributed shared object cache.

When using a read-mostly algorithm, be sure that your container uses a reliable communications protocol when invalidating the read-only cache. If a message is accidentally lost, you could be working with stale data.

Message-Driven Bean Clustering

Message-driven beans behave differently than session and entity beans and thus have different implications in a cluster. Since message-driven beans do not have home or remote interfaces, they don't have any stubs or skeletons remotely that can perform load balancing and fail-over logic on their behalf.

Message-driven beans are consumers of messages; they behave in a pull scenario grasping for messages to consume, rather than a push scenario in which a remote client sends invocations directly to the consumer. See Chapter 8 for a full discussion of this behavior.

Message-driven bean clustering is really about JMS clustering. A message-driven bean is dependent upon the clusterable features of the JMS server and destinations that it binds itself to. Message-driven beans achieve load balancing by having multiple EJB servers of the same type bound to a single JMS queue for message consumption. If four messages arrive concurrently at the queue and four containers of the same message-driven bean type are bound to the destination, each container is delivered one of the messages for consumption. Each container consumes its message concurrently, achieving a pseudo-load-balancing effect.



We've just discussed how load balancing of messages works for a point-to-point JMS queue. Why don't we worry about load balancing for publish/subscribe messages that are put on a topic?

Fail-over of message-driven beans is integrated into the very nature of the beans themselves. Fail-over occurs any time a message that is being processed is acknowledged as *unsuccessful* to the JMS server. An unacknowledged message is placed back on the destination for reconsumption. The message-driven bean that consumes the message a second (or third, fourth, and so on) time need not be the one that consumed it the first time.

In some advanced JMS server implementations, JMS destination replication allows nonpersistent messages to be replicated across servers in a cluster. Message-driven beans that bind to a replicated destination detect any server

failures and automatically rebind themselves as a consumer to the server hosting the replicated destination.

Other EJB Clustering Issues

This final section discusses some miscellaneous issues about J2EE clustering that can impact the behavior of a system.

First Contact

When a client wants to use an EJB component, whether it is a session, entity, or message-driven bean, the client must always first connect to the JNDI tree:

- Clients that want to use a session or entity bean lookup their home stub.
- Clients that want to send a JMS message to be consumed by a message-driven bean must lookup a JMS *ConnectionFactory* and *Destination* object.

Since all EJB clients use JNDI, naming server clustering ultimately has an impact on the behavior of EJB components in a cluster, too. What kind of clustering enhancements can be made to naming servers, and how does this impact EJBs? There are two types of clustered naming servers:

Centralized. The naming server is hosted on a single server. All EJB servers register their same EJB components on the single naming server, and all clients lookup EJB components on the single naming server. The naming server can even distribute clients to the identical servers in the cluster.

Shared, replicated. Each node in a cluster hosts its own JNDI naming server that contains replicated objects hosted on other servers in the cluster. The naming servers replicate their contents (including home stubs, JDBC *DataSource* objects, JMS *ConnectionFactory* object, JMS *Destination* objects) to the other naming servers in the cluster. Thus, every naming server has a copy of every other naming server's objects in the tree. In a scenario in which the same EJB component is deployed on every server in the cluster, each naming server has a copy of the home stub representing each server. If a server in the cluster crashes, all of the other naming servers that are still active merely have to remove from their naming server the objects hosted on the other machine.

Initial Access Logic

When an application server provides a *centralized naming server*, the logic that clients use to get access to the cluster is simple: They hard-code the DNS name

or IP address of the centralized naming server into all of their *InitialContext* creation calls.

But what about J2EE vendors that support a shared, replicated naming server? Clients can connect to any server in the cluster and make a request for a service hosted anywhere else in the cluster. Architects have a variety of options available to them.

DNS round robin. This allows multiple IP addresses to be bound to a single DNS name. Clients that ask for an *InitialContext* pass in the DNS round-robin name as the URL of the naming server. Every translation of the DNS round-robin name results in the generation of a different IP address that is part of the round-robin list. Using this technique, every client *InitialContext* request is directed to a different server. Networks support this feature or do not—it is *not* dependent upon the capabilities of your application server.

Software proxies. Software proxies maintain open connections to a *list of servers* that are preconfigured in a descriptor file. Software proxies can maintain keep-alive TCP/IP connections with each of the servers to provide better performance instead of attempting to reconnect every request. These software proxies immediately detect any server crash or unresponsiveness because its link is immediately lost. Software proxies can also support a wider range of load balancing algorithms including round robin, random, and weight-based algorithms.

Hardware proxies. Hardware proxies have capabilities similar to software proxies but often can outperform their software counterparts. Hardware proxies can also double as firewalls and gateways.

Summary

In this chapter, we discussed the major challenges and solutions for working with EJB in a clustered system. We discussed what makes a system large and the major characteristics that large systems exhibit. We then compared the 3-tier and 4-tier approaches to clustering. We analyzed the type-specific behavior that can be exhibited by stateless session, stateful session, entity, and message-driven beans in a cluster. And finally, we discussed cluster deployments of EJB, clustered naming servers, and initial access logic to naming servers. So pat yourself on the back! You've just learned a great deal about clustering. Stay with us now and we'll learn all about how to get your EJB project started the right way.

Starting Your EJB Project on the Right Foot

To be successful with an EJB/J2EE project, you must plan and understand a great deal beyond the technologies themselves. You must overcome a wealth of project management challenges, including designing the object model, dividing up your team, and educating your team.

This chapter is a guide for how to get started on your EJB project, with links to external resources that you should consider in your project. They are taken from real-world experiences and intended to help you build enterprise Java systems. As you read the chapter, you may find project management issues that you may not have considered. If this happens, consider weaving these issues into your project plan, or highlight the issues for future reference when you begin an enterprise Java project. While we may not answer every question for each issue, we will point you towards external resources to help you find your own answers.

Get the Business Requirements Down

Before embarking on a J2EE project, try to lock down as many of the business requirements as possible. A suggested process is as follows:

1. Build a complete list of requested features. This is the phase in which you ask questions about user interface requirements, legacy integration requirements, use-case analysis, and so on. If your feature list is incomplete, you should consult with any subject matter experts you can access.

2. Weight each feature based upon corporate and business goals, as well as the time anticipated to implement it. Prioritize the list by feature weight.
3. Gain stakeholder support for the feature list to avoid future bickering.
4. Secure a committed project budget from stakeholders.

You should now have a fairly complete basis for designing an object model.

Decide Whether J2EE is Appropriate

Once you have the business requirements, you need to settle on an architecture. J2EE may be appropriate, but then again, it may not. Spend the time for the appropriate research on the various alternatives up front. Table 15.1 lists external resources to help you make a sound decision.

Decide Whether EJB Is Appropriate

A recent Gartner Group report cited that companies overspent by \$1 billion on EJB in 2000, when they could have gotten by with servlets/JSPs. This motivates our next discussion. Once you've decided whether server-side Java is the way to go, you need to make the all-important decision: Are you actually going to use EJB on this project? Or is EJB overkill?

As an example, consider an e-commerce site that has involved business processes. When you buy something on an e-commerce site, the site needs to:

Table 15.1 Resources to Help You Decide If J2EE Is Appropriate

RESOURCE	DESCRIPTION
TheServerSide.com (www.TheServerSide.com)	Keeps you up-to-date with various news bulletins and articles about the J2EE space.
EJB vs. COM+ Live Debate Transcript featuring Ed Roman and Roger Sessions (www.TheServerSide.com)	Live debate transcript from Austin TX. Ed Roman and Roger Sessions duke it out. A bit outdated, since COM+ has been replaced by .NET.
"J2EE vs. Microsoft.NET" whitepaper by Chad Vawter and Ed Roman (www.TheServerSide.com)	Whitepaper that compares comparing J2EE and Microsoft.NET. J2EE wins.
"J2EE vs. Microsoft.NET" whitepaper by Roger Sessions (www.objectwatch.com)	Whitepaper comparing J2EE and Microsoft.NET. Microsoft.NET wins.

- Validate your credit card.
- Debit your credit card.
- Perhaps run some antifraud checking algorithms.
- Check inventory.
- Send a confirmation e-mail.
- Submit the order.
- Fulfill the order.
- Track the order after it is fulfilled.
- Handle returns.

You can achieve all of this by using vanilla servlets and JSPs. For example, you could have the servlets and JSPs call Java classes. Those Java classes perform your business logic. In fact, we can consider *three* scenarios in which it is possible to use Java classes rather than EJB components:

- In a browser client Web-based system with servlets and JSPs.
- In a Web services system where business partners call your servlets and JSPs using XML/HTTP. (See www.TheServerSide.com for a whitepaper on how to build Web services using J2EE.)
- In a 2-tier client/server system such as a Java applet or Java application connecting to a server, you could use servlets and JSPs on the server. The thick client could communicate with the server via HTTP, which easily navigates through firewalls (compared to IIOP, which does not). Behind the servlets and JSPs could be Java classes instead of EJB components.



A servlet/JSP HTTP layer is important only if the users of your system are going to be behind firewalls, such as anonymous Internet clients, business partners, or other departments within your organization. It is also important if your thick client is located across the Internet, because HTTP is a lightweight protocol that travels across the Internet easily. If there is no firewall issue, or if your users are not located across the Internet (but are on your local area network), you could get rid of your HTTP layer and connect the client to EJB components directly. In this case, the EJB value proposition is strong, because EJB allows the client to call the server using intuitive method names, removes the need to perform XML marshaling, and gives you automatic remotability and load balancing.

So how do you decide which is the right paradigm? Let's start with the reasons that most people think are important for deciding between EJB and Java classes, but which are actually *not important at all*:

EJB server cost. The major J2EE server vendors—IBM, BEA, and Oracle—do not sell their EJB layer separately from their servlet/JSP layer. They sell a

single *J2EE server* that bundles both layers. So if you go with a market leader, you're probably going to buy an EJB server whether you like it or not. The only way to avoid this cost is to purchase an open-source or inexpensive servlet/JSP implementation. Although this is a viable option, it is not recommended for major deployments because the cost of the J2EE server is often a drop in the bucket compared to the total cost of the project. If the server doesn't work out, consider the retraining you'll need to pay for and the code you may need to rewrite if the servers use different versions of the J2EE specifications. Consider the difficulty in hiring skilled professionals if you don't go with a market leader, the cost of relearning how to tune your new server, and the cost of learning to administer that new server. The cost of the application server should not be an issue for most serious deployments. Most major vendors are charging very reasonable fees, *far* less than the \$50,000 per processor that was charged in days gone by. Rather, you should consider whether an EJB layer or a Java class layer is appropriate for *your project*. The professional services fees—we call it *geek time*—tends to dwarf the application server cost.

Resource pooling. Nearly all the major J2EE server vendors allow you to get connection pooling and thread pooling whether you use servlets/JSPs or EJB components.

Clean separation of business logic and presentation logic. EJB is nice because it enforces a separation of presentation logic (servlets and JSPs) from business logic (EJB components). We like this because in the future we can support different types of presentation logic, such as a WAP-enabled phone or an XML data stream client from a business partner. But you can achieve the same results with Java classes. You just need to enforce some coding best practices in your organization for the proper usage of Java classes as a business layer façade.

Now that we've blown away the fear, uncertainty, and doubt (FUD), here are the real reasons not to use EJB:

You can't deal with the limitations of EJB. Examples include threading, static variables, and native code. Most companies can deal with these, because the restrictions exist for good reasons. But for example, if you need to have a multithreaded engine, and you can't deal with the EJB paradigm of load balancing across single-threaded instances, EJB is not a good fit for you. EJB is a square peg—don't try to fit it into a round hole.

You have existing skillsets or investments in a working technology. If your developers are proficient in CORBA, why not stick with it? The Middleware Company once consulted with a client who wrote a CORBA application that assisted with mapping the human genome. It worked well with

CORBA, and our client had no major complaints; and so we recommended they stick with CORBA and avoid the EJB hype.

Your application is a big GUI to a database. If you are just a big GUI to a database—heavy on data logic but no business logic—you could achieve a deployment easily using JSPs with tag libraries connecting to a database via JDBC.

Your application is simple. If you are prototyping, building a simple system, or developing a one-off application that will not evolve over time, EJB may be overkill.

You need an initial system built extremely fast. Using Java classes means you don't need to worry about building all the little files that comprise an Ejb-jar file.

And here are the real reasons to use EJB over Java classes:

In the long term, your system is built faster and more reliably. EJB components benefit from declarative middleware, such as instance pooling, transactions, security, container-managed persistence, container-managed relationships, and data caching. If you used regular Java classes, you'd need to write this middleware yourself over time. Eventually you might find that you have your own middleware framework. *Framework* is a fancy word for building your own home-grown application server. The framework needs to be tested and debugged, and features need to be added. This is a nontrivial task indeed. Can you honestly say your staff is capable of building a better application server than the market leaders who specialize in middleware?

It is easier to hire new employees. If you build your own custom framework using Java classes, new hires need to be trained on this framework. You can no longer look for *EJB* on a resume when hiring developers and expect them to be productive on your system.

You benefit from the best practices the world is building around EJB. You can figure out how to optimize your system by reading articles on the Internet or picking up a book on EJB best practices. This global knowledge base is not at your disposal with a proprietary Java class framework.

You can have different user interfaces. You can reuse the same EJB component layer for a thick client as well as a servlet/JSP client. You cannot achieve this with Java classes because they are not remotely accessible. If you wrapped those Java classes in RMI objects, you'd need to invent your own load balancing, instance pooling, and fail-over.

You can work with industry-standard tools to rapidly develop your system. While in the short run it may seem that Java classes will develop your

system faster than writing all those files that comprise an EJB component, in reality many tools help streamline the EJB development process. Command-line tools generate the files you need, IDEs help you build EJB components, and UML editors help you generate EJB components from UML diagrams. See www.TheServerSide.com for more.

You can separate your Web tier and application server. If you require your business logic to be protected by a firewall, you can deploy the Web server and application server on separate machines and stick a firewall in the middle.

Staff Your Project

When you are building a J2EE project team, you will often find an insufficient number of people in house, if any at all, with experience developing with J2EE technology. Don't despair; many organizations are in the same position. You have several options:

Hire full-time experienced J2EE employees. Full-time experienced employees are the most cost-effective way to staff a development team. However, candidates for full-time employment, particularly those with significant J2EE skills, are often difficult to find. You must have a solid recruiting process.

Educate existing developers on J2EE. For organizations with existing development teams, a much easier alternative to hiring full-time J2EE experts is to educate your existing staff on Java and J2EE through training provided by firms. You can fill in holes in your staff by hiring Java developers or computer science graduates who are both eager to learn J2EE.

Hire short-term consultants. Consultants hired for several days or weeks can help you with specific issues such as choosing a J2EE application server; selecting tools, standards, and guidelines; resolving internal debates; providing an unbiased architectural review of your project; aiding project initiation and planning; and mentoring in a specific technology. Short-term consultants are a bit pricey but provide significant benefit to your project when used effectively for that expertise. Because of their cost, engage these experts for short-term use only. All project teams can benefit from bringing in one or more consultants at the onset of a project.

Hire long-term contractors. Long-term contractors are a happy medium between full-time employees and consultants. They're paid more than employees but often significantly less than consultants. They are often easier to hire because most developers perceive contracting as high paying yet low risk, therefore more people choose this career path and typically have

the experience that you require. (Today's J2EE contractor was yesterday's full-time J2EE employee somewhere else.) Contractors are an effective way to fill out your project team when you don't have enough full-time employees with J2EE experience and don't want to pay consulting rates for a significant portion of your staff. Skills that you should demand of your contractors include expertise in the specific application server that you are using, experience on one or more projects of similar scope, and ideally, experience on one or more projects of a similar nature.

If you decide to go the training or contracting route, the authors of this book may be able to help you. See Table 15.2.

Design Your Complete Object Model

Once your team is assembled and has a good level of J2EE understanding, you are empowered to flesh out your object model. Ideally you should minimize risk by working hand-in-hand with an external J2EE expert who has built such systems in the past.

Whenever you inject a new object into this object model, all layers should be considered. Ignoring the user interface, the business layer, or the data layer could lead to false assumptions that bite you down the line.

See Table 15.3 for suggested resources when building a J2EE object model.

Implement a Single Vertical Slice

Once you have defined an initial architecture, you need to start building to that architecture. We recommend implementing an *initial vertical slice* of the system. A vertical slice is a subset of the use-cases in your system. For example, if

Table 15.2 J2EE-Related Service Vendors

VENDOR	SERVICE FOCUS
The Middleware Company (www.middleware-company.com)	Provides training, consultants, and contractors for Java, EJB, J2EE, and XML projects.
Ronin International (www.ronin-intl.com)	Provides consultants and contractors for object-oriented and component-based architecture and software process development.

Table 15.3 Resources for Building Your J2EE Object Model

RESOURCE	DESCRIPTION
TheServerSide.com (www.TheServerSide.com)	Design Patterns section is invaluable resource for building J2EE systems.
Chapters 13 and 17 of this book	Chapter 13 is a group of EJB design strategies. Chapter 17 is a sample design for a complete EJB/J2EE system.
<i>"EJB Design Patterns"</i> by Floyd Marinescu, published by John Wiley & Sons	Patterns for building EJB object models.
<i>"Core J2EE Patterns"</i> by John Crupi, et al. published by Addison-Wesley	Patterns for building J2EE systems (includes Web tier and EJB tier patterns).
J2EE Blueprints (http://java.sun.com/j2ee)	Best practices guide for J2EE systems.

Reuse of J2EE Components

In our experience, it is a myth that J2EE components achieve high reuse across projects. Components are often copied-and-pasted, but not reused in the true O-O sense.

For large organizations building complex J2EE systems, we recommend investing in a J2EE best practices task force. This task force enforces coding standards across all projects, that all teams speak the same vocabulary for objects in their system, and that correct design patterns are applied in projects. The benefits of this task force include easing communication between projects and enabling developers to easily transition between projects with minimal ramp-up time.

If you are going to go down the path of reuse, we recommend using a reusable component manager, which helps you organize components within your enterprise. ComponentSource and Flashline.com each provide these products.

you're putting together an e-Commerce site, you might have the vertical slice be the search engine or the product catalog. A vertical slice should demonstrate usage of all the J2EE technologies in tandem—you would want to show that a browser can connect to a Web server running servlets, which in turn interacts both with EJBs that access your back-end database and with JSPs to generate HTML to return to the browser. A vertical slice is *not* just EJBs. Developing an initial vertical slice offers several benefits:

Provides experience developing J2EE software. By developing an end-to-end vertical slice, you learn how to work with all of the technologies, tools, and techniques that you are going to apply on your project. You have to start somewhere, and it's better to discover and address any problems as early in your project as possible.

Provides experience deploying J2EE software. The first time you deploy a J2EE application can be confusing. You have several types of nodes to potentially install and configure—Web servers, application servers, database servers, security servers, and so on. You can safely gain this experience by internally deploying your initial vertical slice into your staging area.

Reduces unsoundness risk. By developing an initial vertical slice, you show that the technologies you have chosen all work together, thereby eliminating nasty integration surprises later in your project. Remember the old saying: Everything works well in management presentations, but not necessarily in reality.

Proves to your project stakeholders that your approach works. At the beginning of a project, your stakeholders may support you because they have faith that your team can deliver what you have promised; but their support will be even stronger if you show that you can actually deliver. Furthermore, developing and then deploying (at least internally) an initial vertical slice can be incredibly important to your project politically because your detractors can no longer claim that it isn't going to work.

Answers the question: Will it scale? The vertical slice is a real working piece of your system and should demonstrate how well your design scales under load. You can stress test this slice before building the rest of your system. This reduces risk, especially in situations where you may have questions about whether your object model will work (for example, will EJB entity beans scale?).

Gets the design patterns right early on. Building the vertical slice gives you experience with what works and what doesn't work with J2EE. For example, you'll have an opportunity to compare and contrast different model-view-controller (MVC) paradigms. This leads to discovery of a common technical vision. Once you've locked down that framework, you can apply those best practices to other vertical slices and enable developers with lesser skills to implement them.

Do You Start Fresh or Evolve Your Initial Slice?

Once you have developed your initial vertical slice, you need to make an important decision: Do you throw it away to start fresh on future vertical slices, or do you continue to evolve it into your system? The answer depends on the quality of your work. If it is poor quality, either because you rushed or simply because you were new to the technologies or techniques and made some fundamental mistakes, you should consider starting fresh. There's absolutely nothing wrong with starting fresh—you still gain all the benefits. On the other hand, if the quality of your initial vertical slice is good enough, you can and should consider keeping the code (or at least applicable portions of it) and use it as a base from which to develop your system. This is something that the rational unified process refers to as building the skeleton first.

Choose an Application Server

The choice of an application server is important to your project. Although your J2EE applications may be portable between vendors, the differences make it painful to switch vendors.

Companies that are eager to get started with their EJB development should go with one of the current market leaders. But companies who want to reduce risk before jumping into purchasing a J2EE server should spend the time to research whether the vendor they're thinking about using is right for them. This is applicable for both large and small projects. Our recommended process is as follows:

1. List the features you want in an application server. A consultant can help you build this list.
2. Weight and prioritize the feature list.
3. Eliminate all vendors that don't meet the majority of your criteria.
4. With the two or three vendors left, download and deploy your *initial vertical slice* into those application servers. You can then measure how well these application servers handle your specific business problem, as well as their general usability.

In all cases, we recommend you do not purchase your application server *until* you've deployed your vertical slice into the application server. You may find the application server does not behave as the vendor's marketing propaganda

promised. Download that free evaluation copy and deploy that real, working slice of your system into the server to see for yourself.

The following are suggested resources for choosing an application server:

- Chapter 16 of this book, a guide to choosing an EJB server
- TheServerSide.com application server “Reviews” section (www.TheServerSide.com)
- “Selecting a J2EE Vendor” by Simon Blake (www.TheServerSide.com)
- EJB Portal—Product & Vendors Directory (www.ejbportal.com)
- Enterprise JavaBeans Supporters (Sun Microsystems) (java.sun.com/products/ejb)
- Flashline.com—Application Server Comparison Matrix (www.flashline.com)
- App-Serv Central Contenders List (www.app-serv.com)

Divide Your Team

Dividing your J2EE team is one of your most critical decisions. When assembling a J2EE project team, you have two basic choices:

Horizontal approach. Have folks specialize in different technologies. For example, you’d have a JSP team, a servlets team, an EJB session beans team, and an EJB entity beans team. Members of your team become *specialists* in specific technologies.

Vertical approach. Have folks specialize in vertical business use cases. For example, you’d have a search engine team, a product catalog team, and a payment processing team. Team members become *generalists* and gain experience with all the J2EE technologies involved in that domain, such as servlets, JSPs, and EJBs.

You can also use a hybrid approach, which is a combination of the two. Table 15.4 describes the horizontal, vertical, and hybrid approaches to team organization with their advantages and disadvantages. Table 15.5 lists several recommended resources for building project teams.

So which approach is better? The answer depends on the goals for your project:

- **If your goal is to get your project completed quickly and in a consistent manner**, our experience has shown us that the horizontal or hybrid approach is superior. Design patterns, paradigms, and frameworks are

Table 15.4 Team Organization Strategies

STRATEGY	ADVANTAGES	DISADVANTAGES
<p>Horizontal—Your team is composed of <i>specialists</i> in particular J2EE APIs. Each developer works on several use-cases using that API.</p>	<ul style="list-style-type: none"> ■ The same team uses the same API across all vertical use-cases. This ensures consistency in design patterns, paradigms, and frameworks used. ■ Specialists become proficient with their API, yielding rapid application development. 	<ul style="list-style-type: none"> ■ Specialists do not gain exposure to other APIs, resulting in disconnects between layers. ■ Requires strong planning to achieve parallel development. Need to define rock-solid interfaces between layers. ■ Retention issues arise. Specialists do not have a concept of ownership of a use-case. They only understand only a single part of J2EE, and so their skills grow more slowly.
<p>Vertical—Your team is composed of <i>generalist</i> developers who gain experience with every J2EE technology. They focus on a specific problem domain or use-case.</p>	<ul style="list-style-type: none"> ■ Smooth end-to-end development on an individual use-case basis. ■ Parallel development is easy if use-cases are separated well. Each developer works on his own use-case. ■ Developers have a concept of ownership of a use-case. They gain a wider range of skills. Good for retention. ■ Good for educating developers on different technologies used in your system to give them a broader picture. 	<ul style="list-style-type: none"> ■ Generalists need to know many technologies and are typically highly paid and difficult to find. ■ Generalists typically do not have the specific technical expertise required to quickly solve detailed problems. ■ Subject matter experts must work with several developer groups, increasing their burden. ■ Design patterns, paradigms, and frameworks used may change between use-cases. ■ If use-cases are interdependent, it is difficult to partition the team.
<p>Hybrid—Your team is made up of both generalists and specialists. The generalists have authority over one or more use-cases. They support API specialists who work on many use-cases within a particular API.</p>	<ul style="list-style-type: none"> ■ The same team uses the same API across all vertical use-cases. This ensures consistency in design patterns, paradigms, and frameworks used. ■ Specialists become proficient with their API, yielding rapid application development. ■ Individual use use-cases are implemented consistently . 	<ul style="list-style-type: none"> ■ Requires planning and structure early on in the project. ■ Requires an understanding and buy-in from the team that the generalists have authority within their use-cases. ■ Must still spec out interfaces between layers for specialists to develop in parallel.

Table 15.5 Recommended Resources for Building a Project Team

RESOURCE	DESCRIPTION
<i>Peopleware: Productive Projects and Teams, 2nd Edition</i> , Tom Demarco and Timothy Lister, 1999, Dorset House Publishing	This book is the de facto classic within the information technology industry for how to build and manage a software project team.
<i>Constantine on Peopleware</i> , Larry L. Constantine, 1995, Yourdon Press	This book presents is a collection of writings about the software aspects of software development, including developer productivity, teamwork, group dynamics, and developer personalities. This is a good book to read for anyone trying to understand how to organize and then manage a bunch of software “geeks.”
<i>Organizational Patterns for Teams</i> , Neil B. Harrison, 1996, Pattern Languages of Program Design 2, pages 345–352, Addison-Wesley Publishing Company	The paper describes a collection of patterns for building a software development team, including Unity of Purpose, Diversity of Membership, and Lock ‘Em Up Together.
<i>The Unified Process Inception Phase</i> , Scott W. Ambler & Larry L. Constantine, 2001, CMP Books, www.amblysoft.com	This book describes a collection of activities and best practices for the Rational Unified Process (RUP) Inception phase, including advice for building your project team.

kept consistent across the board. Specialists build a core competency in their API, yielding rapid application development.

- **If your goal is to invest in the education of your developers** to reduce retention issues or to give them a broader long-term skill set, the vertical approach works well. Developers gain experience with every technology in J2EE. The downside is consistency of design patterns across use cases. In a day and age when good software professionals are hard to find, let alone keep, this is an important consideration.

Invest in Tools

A number of tools are worth a look when building your EJB/J2EE deployment. These include testing tools (JUnit), profiling tools (JProbe or OptimizeIt), UML modeling tools (Together/J or Rational Rose), IDEs (JBuilder or WebGain Studio), and more.

Rather than describe each and every tool that's out there, we are providing a *free* up-to-date EJB/J2EE product matrix where users can describe and submit reviews for products and tools in the EJB/J2EE industry. This is located on www.TheServerSide.com.

Invest in a Standard Build Process

An easy choice when picking tools is a tool to help you with a standard build process. If you decide to use a standard build process, you must use some sort of build scripts, written in some scripting language. The build tool does not take place of your build process—it only acts as a tool to support it. What characteristics would you like to have in this scripting language?

Widely understood. It would be nice if your developers (who are more often than not doing the builds) already understood the technology behind the language.

Functional. The scripting language needs to support a wide array of functionality out of the box, especially for Java features.

Extensible. Since no two projects are the same, and projects use all sorts of different technology, it would be useful if you could add functionality to the scripting language to handle your particular needs.

Cross platform. In an enterprise environment, you usually are developing on a Windows machine and doing testing and quality assurance on a non Windows box. You want your build scripts to be as cross-platform as your code.

The Apache group's Ant build tool (<http://jakarta.apache.org>) combines ideas from Java and XML to achieve these goals. Many of our clients are using Ant successfully, and we highly recommend it.

Next Steps

With your initial vertical slice in place, you are in a position to continue your construction efforts by developing additional slices of your system. For each vertical slice, you effectively run through a miniature project lifecycle—fleshing out its requirements, modeling it in sufficient detail, coding it, testing it, and internally deploying it. This approach reduces your project risk because you deliver functionality early in the project; if you can deploy your system internally, you can easily make it available to a subset of your users to gain their feedback. Furthermore, your development team gains significant lifecy-

cle experience early in the project, giving developers a significantly better understanding of how everything fits together.

Summary

In this chapter, we gained greater insight into how to start our EJB projects on the right foot. We learned about a time-tested process that has worked for other organizations to reduce risk and lead to a win-win situation for all parties involved. Armed with this knowledge, you should be able to confidently begin work on your EJB project.

Choosing an EJB Server

Throughout this book, we've explained the concepts behind EJB programming and put the concepts to practice in concrete examples. But perhaps an even more daunting task than learning about EJB is choosing from the legion of container/server product vendors out there—currently more than 30 such products. For the uninformed, this is a harrowing task. What should you be looking for when choosing an EJB product? That is the focus of this chapter.

To best make use of this chapter, first ask which application server features are most important to you in your deployment, including specific features that you need (such as support for a particular database). Once you've listed your requirements, assign a weight to each feature. For example, if transparent fail-over is important in your deployment, you might rank it a 7 of 10. Once you've weighted each feature, you can begin evaluating application server products and create a scorecard for each product.

Once you've reduced your list of servers to two or three, we recommend that you deploy code into those servers and test them out for yourself. You should measure both quantitative data (how many transactions per second can the server support?) as well as qualitative data (how easy is the server to use?). See Chapter 15 for more details on how choosing an EJB server fits into a larger EJB development process.

The remainder of this chapter discusses our criteria for choosing an EJB server.



This chapter *does not* recommend a particular EJB server. Why not? Because by the time this book fell into your hands, the information would already be out of date. Instead, we are hosting a free section on www.wiley.com/compbooks/roman that contains application server reviews.

J2EE 1.3 Brand

Perhaps the most important issue to think about when choosing an EJB container/server product is compatibility. When you make your purchase decision, you need to write code and purchase beans that are compatible with your container/server product. If in the future you decide to switch to a different vendor's product, the transition will surely not be free and it will always involve some migration headaches. While the EJB standard defines the interfaces that should make products compatible, realistically, every vendor's product will differ from the next in some semantic ways, which will impact your deployment. Ideally, you want to make the right choice on your first purchase.

J2EE v1.3 ships with a compatibility test suite, which verifies that a particular vendor's product is indeed compatible with the J2EE 1.3 specifications, including EJB 2.0. You can verify compatibility by looking for a J2EE seal of approval, which Sun Microsystems stamps on J2EE-compliant products.

Pluggable JRE

Some containers are hard-coded to a specific version of the Java Runtime Environment (JRE). Other vendors are more flexible, supporting many different JREs. This may be important to you if you have existing applications that depend on specific JRE versions.

Conversion Tools

Does the EJB server ship with tools to migrate old J2EE code into the latest version? Consider that even though J2EE 1.3 sounds new today, it won't be new tomorrow. You may need to upgrade your application in the future to a new version of J2EE, and your vendor's historical support of migration tools is most indicative of whether it will support such migration in the future.

Complex Mappings

Be sure your EJB server allows you to perform any complex database mappings that you may need, such as mapping to stored procedures and mapping to complex JOINS across a relational database, as well as the ability to write custom SQL if necessary.

Third-Party JDBC Driver Support

Some servers do not allow the substitution of JDBC drivers—or if they do, they may disable features such as connection pooling. Be sure your vendor supports your database connection needs.

Lazy-Loading

Lazy-loading means to load entity bean data on demand. This is important for large object graphs where the user may only need access to a small portion of the graph. Note, however, that your EJB server should allow you to tune lazy-loading on a per-bean basis, so that you can load an entire object graph if you know you need that entire graph.

Deferred Database Writes

A deferred database write means to defer JDBC write operations until transaction commit time. This is important, because if you have a transaction involving many EJB components and thus many database writes, it is counterproductive to perform many network roundtrips to the database. The superior approach is to perform one large write at transaction commit time.

Pluggable Persistence Providers

Some EJB containers provide proprietary APIs for plugging in third-party persistence modules, such as a module that persists your entity beans to an object database rather than a relational database. Other possibilities include persisting to a file, persisting to a relational database using a simple object-relational

mapping, persisting to a relational database using a complex object-relational mapping, or persisting using user-defined persistence routine (which may implement persistence through a legacy application).

If you're planning on plugging in a third-party persister into your container, be sure that you can gain transactions and connection pooling.

In-Memory Data Cache

If you are using entity beans (and many deployments will, now that they can be accessed in a high-performance way through local interfaces), be aware that entity bean performance is not equal between application servers.

Some application servers work in a *pass-through* mode, which means that any entity bean operations are passed through to the database, resulting in a low-level database transaction. Other vendors implement smart caching of entity beans, allowing some operations to occur in memory rather than at the database level. For example, if you're merely reading the same data over and over again from an underlying storage, you should not need to hit the database on every method call. The difference between pass-through and caching application servers is tremendous. Some vendors have reported a 30-fold or more performance increase over the pass-through application server.

There is even a third-party marketplace for such caching providers. For example, *Javlin* is a product that plugs into an EJB server to add caching support.

Integrated Tier Support

Throughout this book, we've concentrated on EJB as a server-side component model. But for many deployments, Web components written with servlets and JSPs need to access the EJB layer. Some EJB servers offer the ability to run servlets and JSPs in the same JVM as your enterprise beans. If you want this style of deployment, look for this feature.

Scalability

Your EJB server should scale linearly with the amount of resources thrown at it. If you add extra machines with equal power (memory, processor power, disk space, and network bandwidth), the number of concurrent users your server-side deployment can support and the number of transactions your system can execute per second, should increase linearly. Be sure to ask your EJB

server vendor for case studies and customer references to back up its scalability story.

Other questions to ask include:

- How long does it take for the EJB server to start up or restart? This is important for development as well as for production deployment. If the restart cycle is long it makes it inconvenient to develop and debug with the EJB server. In production, a slow startup cycle affects the availability of the application to the clients.
- Can the EJB server recover from backend database loss and restart? For example, if the EJB server temporarily loses connectivity to the database, does it need to be restarted to reestablish connectivity to the database, or can it do so automatically?

High Availability

High availability is critical for server-side deployments. You want the highest level of confidence that your EJB server won't come down, and you can look for a number of things to increase your confidence. Your EJB server vendor should have compelling numbers indicating the availability of its product, backed up by existing customers. Realize, too, that your EJB server is only as available as the operating system and hardware that it's deployed on. Be sure to ask your EJB server vendor what operating systems and hardware configurations they support.

Security

A typical EJB deployment leverages predefined security lists that are already available in existing systems. For example, an IT shop may store access control lists of users in a Lotus Notes LDAP server; you may need to use these lists in your EJB deployments. Many EJB products offer assistance with importing and exporting ACLs from existing deployments, so that you won't have to create your own solutions from scratch, saving you time when deploying EJB products. Some systems can even tap into existing security systems—they get the user and authorization information from the existing security service.

Standardized support for the Java Authentication and Authorization Service (JAAS) will enable you to plug in different security providers.

Other questions include:

- Can the server integrate with LDAP in real-time for authentication and authorization?
- Does the security system support SSL?
- Can a firewall be placed between the servlet container and the EJB container? Between a third-party Web server and the servlet container? Can it be an application proxy-type firewall or only a packet filtering firewall?

IDE Integration

An essential component of any development is an easy-to-use *Integrated Development Environment* (IDE), such as Webgain's VisualCafe, Inprise's JBuilder, iPlanet's Forte, or IBM's VisualAge for Java. IDEs can assist in code management, automate programming tasks, and aid in debugging.

Some EJB container/server vendors are IDE vendors as well (IBM, Inprise, and iPlanet). This duality allows them to seamlessly integrate their EJB server product with their IDE. The end result is compelling: The IDE can aid in coding, debugging, and deploying your beans by working together with the application server. Other EJB server vendors who do not have their own IDE are forming strategic alliances with IDE vendors to gain a competitive edge in the marketplace.

Some useful features to look for include:

- Automatic creation of home and remote interfaces from bean
- Automatic identification of business methods
- Creation and editing of deployment descriptors
- Construction of Ejb-jars, Web archives (.wars), and enterprise archives (.ears) from within the IDE
- Direct deployment from the IDE to the J2EE server
- Debugging into the container via a Java remote debug protocol

UML Editor Integration

The diagrams in this book were drawn using the Unified Modeling Language (UML), the de facto standard for communicating development architectures between programmers. A number of visual UML editors are on the market, such as Rational Software's *Rational Rose* and Togethersoft's *Together/J*. Many of these UML editors enable you to visually design EJB components, and then

automatically generate and deploy those components into the EJB server of your choice, yielding rapid application development. Be sure to ask your EJB server vendor about which UML editors support their servers.

Intelligent Load Balancing

A common deployment scenario involves a set of machines, all working together to provide an n -tier solution.

The variety of ways to *load balance* requests across a set of machines include random, round-robin, active load-balancing, weighted load-balancing, and custom algorithms (see Chapter 14 for more on this).

In the long run, if you have many requests hitting your servers, the particular load-balancing algorithm that you choose will likely not matter, as long as your server supports some algorithm. Load-balancing algorithms become particularly important in two cases: if you have a heterogeneous hardware situation and need to skew the load to one machine; or if you only have a few, heavy-duty requests coming into your system. If you're among these cases, be sure that your EJB server supports the load-balancing algorithms you require.

Stateless Transparent Fail-over

When your application server crashes, there should be a transparent rerouting of all requests to a different application server. The natural place to put this process is in intelligent client-side proxies, which intercept all network-related problems and retry methods on alternative application servers or in the object request broker runtime. Transparent fail-over is fairly easy to implement if you restrict the client to invoke only on a stateless server, and assume that all transactional resource updates can be rolled back.

Clustering

A more advanced level of transparent fail-over is stateful transparent fail-over or clustering. With clustering, your application server is replicating conversational state across servers. If an application server crashes, another server can pick up the pieces since it has replicated state. If your application server supports clustering both for Web components (servlets, JSP scripts) as well as clustering for enterprise beans, you can completely eliminate single points of failure from your deployment, ensuring uninterrupted business processes.

An extension of clustering is application partitioning—configuring components to run only particular nodes within a cluster. High-end servers provide tools for managing this complexity.

Java Management Extension (JMX)

JMX is a J2EE API for monitoring a deployment. If your EJB server supports JMX, you can write an application that monitors your EJB server. Your application could set properties in your EJB server as well, such as modifying the current thread pool, redeploying an EJB component, and so on. If you wish to write an application that performs advanced monitoring or control over your EJB deployment, JMX is the way to go.

Administrative Support

A Web-based administrative console allows system administrators to monitor your deployment from afar. Web-based consoles are superior to thick client administrative consoles because you can easily access your system from any machine, and firewalls don't get in the way.

Command-line based administration is also important. It is necessary to allow the automation of deployment and management. After all, your automated testing process will need to quickly deploy beans into a server. It is inappropriate to require a human to click on a Web-based console to achieve this. Common tasks that need to be automated from the command line include:

- Start, stop, and restart the EJB server
- Deploy, redeploy, and undeploy an application

Hot Deployment

Hot deployment means redeploying EJB components into a running EJB server without shutting down the EJB server. This feature may be necessary if you are in a 24x7 environment where even a small amount of downtime during scheduled maintenance is unacceptable.

Instance Pooling

Instance pooling is the pooling and reuse of EJB components. Advanced EJB servers can pool and reuse any type of component, be it a stateful/stateless

session bean, CMP/BMP entity bean, or message-driven bean. Look for flexibility in configuring this pool size, including configurable rules for dynamically increasing and decreasing its size under various load conditions.

Automatic EJB Generation

Some EJB servers ship with wizard-like or command-line tools to automatically generate EJB components for you. For example, you could supply the name of an entity bean, along with the names and types of its persistent fields. From this, the tool should be able to generate your bean class, component interfaces, home interfaces, deployment descriptor, and Ejb-jar file.

Clean Shutdown

What happens when you want to take down an application server for maintenance? Perhaps you want to reboot the machine the application server is installed on, upgrade the application server, or install software on the machine. But if you simply kill the process, any connected clients' work would be lost, potentially resulting in financial errors or other catastrophes.

This leads to another area of value that EJB products can provide: a clean way to shut the application server down without having a gross impact on clients. For example, the EJB application server may simply have a routine that refuses connections from new clients and allows for all existing clients to gracefully disconnect.

Real-Time Deployment

Starting up and shutting down an EJB application server is usually a fairly heavyweight operation. If you're debugging an EJB application, having to restart the EJB application server each time you regenerate your beans is a hassle. Having to shut down an application server to deploy new beans has an even greater impact, because that application server cannot service clients when it is down.

An enhanced value that some EJB products can provide above and beyond the EJB specification is a mechanism for deploying enterprise beans in real time. This means the ability to deploy and redeploy beans without shutting down a running application server.

Distributed Transactions

In Chapter 10, we examined transactions in depth and noted how multiple processes on different physical machines could participate in one large transaction. This is known as a *distributed transaction*, and it is a fairly heavyweight operation. It necessitates the use of the *distributed two-phase commit protocol*, a reliable but cumbersome dance that transaction participants must take part in for a distributed transaction to succeed.

If you require distributed transactions, make sure your EJB server supports them, and also supports a clean recovery in case of transactional failure. For a two-phase commit transaction to work, you also need to have the same transaction service deployed on all participant machines or to have interoperable transaction services. Note that very few deployed EJB systems actually employ distributed transactions today.

Superior Messaging Architecture

If you are using messaging in your system, realize that not all messaging architectures were created equal. Some messaging architectures do not allow you to cluster your JMS destinations, which creates single points of failure. Other messaging architectures cannot support as many messages per second as others. Be sure to get these statistics from your EJB server vendor.

You also might want to look for additional quality of services (if you need them) such as synchronous and asynchronous delivery, publish/subscribe and point-to-point, acknowledgement (ACK) and negative acknowledgement (NACK) guaranteed message delivery, certified delivery, and transactional delivery.

Provided EJB Components

Some EJB servers ship EJB components as optional packages with their servers. This can include personalization components, marketing components, e-commerce components, vertical industry specific components, and many more. Making use of any of these components may help to shorten your development cycle.

If, on the other hand, you plan to use a third-party bean provider's components, you should ask if the components are certified on the EJB servers you are choosing between. If not, you run the risk of incompatibility.

J2EE Connector Architecture (JCA)

The JCA is an extremely important architecture for integrating with existing systems. If you have existing systems, be sure to ask if your container supports the JCA, and if so, which JCA adapters are certified on that application server.

Web Services

Web services technologies (XML/HTTP) enable you to integrate with existing systems seamlessly and are also useful for B2B integration. The major EJB servers will support the following up-and-coming J2EE Web services APIs:

The Java API for XML Registries (JAXR) enables you to access registries, such as *UDDI* servers.

The Java API for XML Binding (JAXB) transforms Java objects to and from XML documents.

The Java API for XML Parsing (JAXP) enables you to parse XML in a parser-independent manner.

The Java API for XML-based RPC (JAX/RPC) enables you to send and receive XML documents using XML-based protocols such as *SOAP* and *XMLP*.

The Java API for XML Messaging (JAXM) is for performing messaging in XML, such as *ebXML*.

The Java API for WSDL (jWSDL) enables you to manipulate *WSDL* documents.

To learn more about these acronyms and how to build Web services, see the free whitepaper by The Middleware Company on how to build J2EE Web services, located on www.TheServerSide.com.

Workflow

Advanced EJB servers, as well as third-party independent software developers, are shipping J2EE-based workflow engines. A workflow engine enables you to model business processes. A business process could span EJB components, existing systems, B2B partners, and more. A workflow engine typically has a flow of control that can change depending upon the state of the current system. These flows are designed visually using a graphical workflow design GUI. This is extremely useful for involving business analysts in the development of business processes.

Currently there is no standard J2EE API for performing workflow, so any workflow solution you find will be proprietary. When shopping for a workflow engine, ask your vendor the following:

- Does the workflow engine integrate with J2EE?
- Is the workflow engine itself written in J2EE, so that you are not injecting foreign technology into your deployment?
- Does the workflow engine support message-oriented middleware, such as IBM MQSeries?
- Does the workflow engine allow for long-lived business processes that may take days, weeks, or even months? One example is a workflow that begins with purchasing a book on a Web site, then leads to fulfilling that book, and finally handles a possible return shipment of the book. That business process could span quite awhile.

Open Source

Some EJB servers are open source-code servers, similar to Linux, in that anyone can contribute to their development. Examples are *jBoss* and *OpenEJB*.

If you choose an open source code EJB server, be sure you choose one for the right reasons—you'd like more fine-grained control over the code base, you are running a nonmission critical deployment, or you'd like to foster the open source code community.

The wrong reason is usually price. Realize that the total cost of ownership of your deployment includes far more than just the cost of the application server. The total cost of ownership includes hardware costs, software costs, costs to employ developers, costs to train developers, and opportunity costs if your system is not ideal. Unless you are strapped for cash, we recommend you take price out of the picture.

Specialized Services

EJB vendors provide numerous other features to differentiate their products. Some of these features do not impact your code at all. For instance, your bean code should always remain the same no matter what load-balancing scheme your application server uses. Other features may require explicit coding on your part, such as ERP integration. When choosing a product, ask yourself how much explicit coding you would need to write to a particular vendor's

proprietary API. The more of these APIs you use, the less portable your EJB code becomes.

Some examples of special features offered in EJB products are:

- Optimized mechanisms for pooling and reusing sockets, multiplexing many clients over a single socket
- Enhanced Java Virtual Machines for added performance
- Advanced systems management integration to professional monitoring tools

As you can see, the emergence of these services becomes one of the chief advantages of EJB as a competitive playing field that encourages vendors to provide unique qualities of service.

Nontechnical Criteria

There is a host of nontechnical criteria to consider as well:

Reputable vendor. Does the vendor have a brand name and a history of distributed transaction processing systems? How large is the firm? How many years has it been in operation?

High-quality technical support available after hours. If a crisis situation ensues in the middle of the night, will your vendor be available to resolve problems?

Verifiable customer success stories. Look for large, well-known (ideally Fortune 500) companies implementing solutions with the vendor's product. Don't hesitate to ask tough questions to get beyond the marketing hype.

Training and consulting services available. The company should have its own internal training and consulting services or should have partnerships with other firms to provide those services to you. Be sure that the vendor's training/consulting department is adequately staffed to provide the care you need, and that the vendor is not overburdened with other projects.

Historical velocity to meet standards. Looking back in time, how close has the vendor's release cycle been to the J2EE specifications' release cycle? The shorter, the better.

Free evaluation copy. Any deserving vendor should let you evaluate its product free of charge for either a limited time period or as a stripped-down product version. Otherwise, rule that vendor out immediately.

Summary

In this chapter, you've surveyed the criteria for making an EJB application server purchase decision. The EJB specifications (as well as the products that implement it) are evolving rapidly. The features offered in the marketplace are likely to change over time. For the latest information about EJB products and news, check out the following resources:

TheServerSide.com. TheServerSide.com has a section where users submit application server reviews. There are also ECPerf benchmark results to compare the performance of different application servers.

Online white papers. Some research firms offer whitepaper reviews they have performed on EJB products. An example of such a site is TechMetrix.com.

The Sun Microsystems Web site. The java.sun.com/products/ejb page has an *EJB Directory* of current EJB products with links to vendor information.

Flashline.com. Flashline.com has an EJB matrix of application servers and their associated costs.

Magazine article reviews. Some Java-based print magazines offer comparisons of EJB products as well. Examples here include *Java Report*, *Java Developer's Journal*, and *JavaPro*.

EJB-J2EE Integration: Building a Complete Application

In this chapter, we will see how to design and build a complete EJB/J2EE system. In particular, you'll learn how to use entity beans, session beans, and message-driven beans *together*, and how to call EJB components from Java servlets and JavaServer Pages (JSPs).

We will first motivate our deployment by describing the business problem. We'll then design the object model. The complete source code is available on the book's accompanying Web site, www.wiley.com/compbooks/roman. The code is fully commented and ready to run.

The Business Problem

Jasmine's Computer Parts, Inc. is a fictitious manufacturing company that makes a wide variety of computer equipment, including motherboards, processors, and memory. Jasmine, the company's owner, has been selling her products using direct mail catalogs, as well as a network of distributors and resellers.

Jasmine wants to lower the cost of doing business by selling her computer parts directly to the end customer through an e-commerce Web-based sales model. Jasmine has given us a high-level description of the functionality of the

e-commerce solution. She'd like the following features in the system we provide for her:

User authentication. Registered users would first log in to the Web site to access the complete catalog. Only registered users should be able to browse and purchase from her online store.

An online catalog. Users should be able to browse her complete product line on the Web and view details of each product.

Shopping cart functionality. While browsing the catalog, a user should be able to choose the products he or she wants. The user should be able to perform standard shopping cart operations, such as viewing the current shopping cart or changing quantities of items already picked out.

Specialized pricing functionality. Users who order items in bulk should get a percentage discount. For example, if I order five memory modules, I get a 10-percent discount on that memory. In addition, registered users who frequent the store often should get additional discounts.

Order generation. Once the user is happy with his or her selections and has committed to ordering the products, a permanent order should be generated. A separate fulfillment application (which we won't write) would use the data in the orders to manufacture the products and ship them. The user would be able to return to the Web site later to view the status of current orders.

Billing functionality. Once the user has placed the order, we should bill it to him or her. If the user does not have enough funds to pay, the order should be cancelled.

E-mail confirmation. After the order has been placed and the credit card debited, a confirmation e-mail should be sent to the user.

This is definitely going to be a full-featured deployment!

A Preview of the Final Web Site

To give Jasmine an idea of what the final product should be like, our sales team has put together a series of screenshots. The screenshots show what the e-commerce system will look like when an end user hits the Web site.

Figure 17.1 shows a user logging into the system initially. Our authentication will be through login names and passwords.

When the user has been recognized, he or she is presented with a Web storefront. The Web storefront is the main page for Jasmine's online store. This is shown in Figure 17.2. From the Web storefront, the user can jump to the

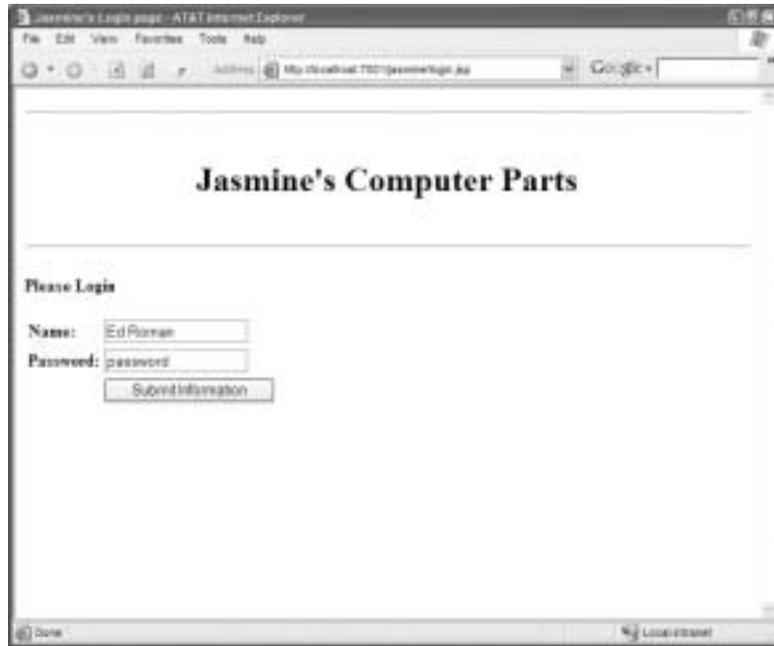


Figure 17.1 A user logging into Jasmine's Computer Parts.

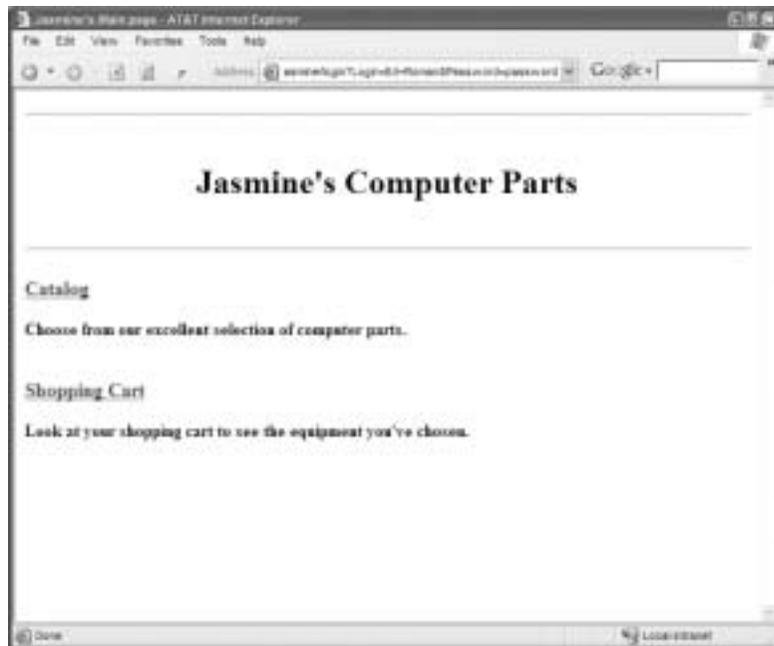


Figure 17.2 The Web storefront for our online store.

catalog of products that Jasmine offers, shown in Figure 17.3. A user who wants to view details about a product can check out the product detail screen, shown in Figure 17.4. The user can also add the product to the current shopping cart—a temporary selection of products that the user has made but has not committed to purchasing yet.

Once the user has made product choices, the user can view a cart for the current selections (and make any last-minute changes), as shown in Figure 17.5. When the user clicks the button to purchase the selection, he or she is billed and a new order is generated. Finally, the user is given the order number for future reference (see Figure 17.6).

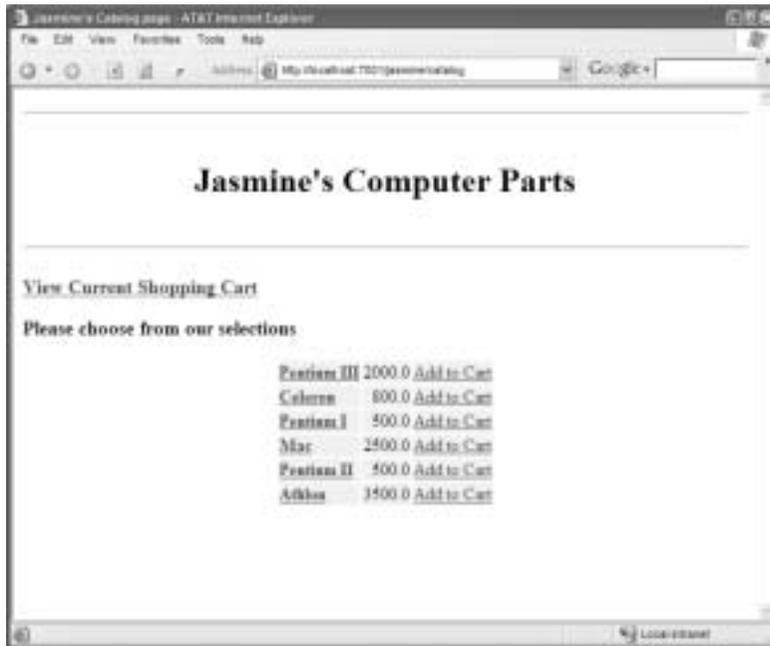


Figure 17.3 Browsing the online catalog.

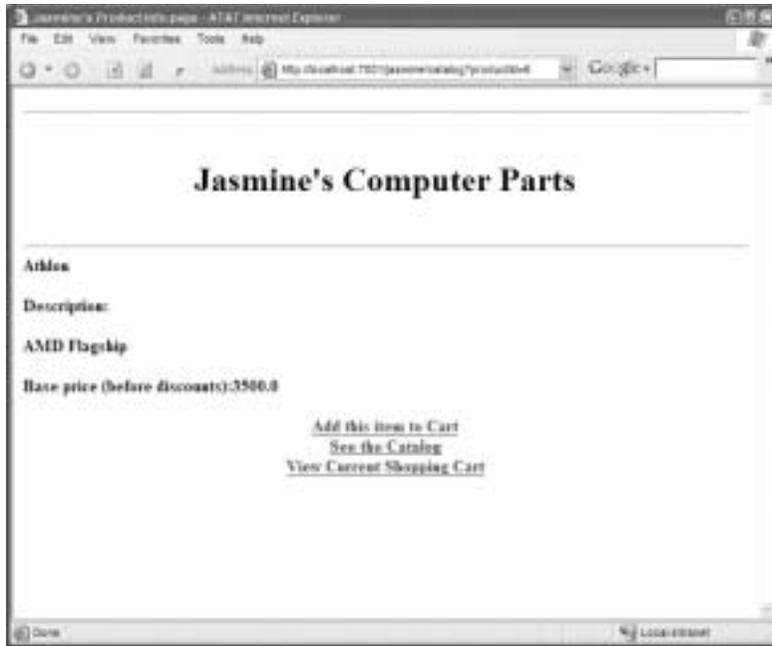


Figure 17.4 Viewing a particular product.



Figure 17.5 Viewing and modifying a cart.



Figure 17.6 Making a purchase.

Scoping the Technical Requirements

While meeting Jasmine's requirements, we'd like to develop an extensible infrastructure that she can add to in the future. That means making the right abstractions to loosen the coupling between our components. Ideally, Jasmine should be able to plug in a different implementation of any part of the system with very few modifications.

Our deployment will be partitioned into three tiers:

The presentation tier involves one or more Web servers, each responsible for interacting with the end user. The presentation tier displays the requested information in HTML to the end user; it also reads in and interprets the user's selections and makes invocations to the business tier's EJB components. The implementation of the presentation tier uses *servlets* and *JavaServer Pages (JSPs)*.

The business logic tier consists of multiple EJB components running under the hood of an EJB container/server. These are reusable components that are independent of any user interface logic. We should be able to, for

example, take our business tier and port it to a different presentation tier (such as a disconnected sales force's laptop) with no modifications. Our business tier is made up of session beans, entity beans, and message-driven beans.

The **data tier** is where our permanent data stores reside. The databases aggregate all persistent information related to the e-commerce site.

Object Model for the Business Logic Tier

Let's begin designing our EJB object model. The major beans are shown in Figure 17.7. As you can see from this diagram, we are following the design strategy of wrapping entity beans with session beans, described in Chapter 13. We'll explain this UML diagram as we go.

We will co-locate our servlets/JSPs in the same process as our EJB components, allowing us to exclusively use efficient local interfaces.

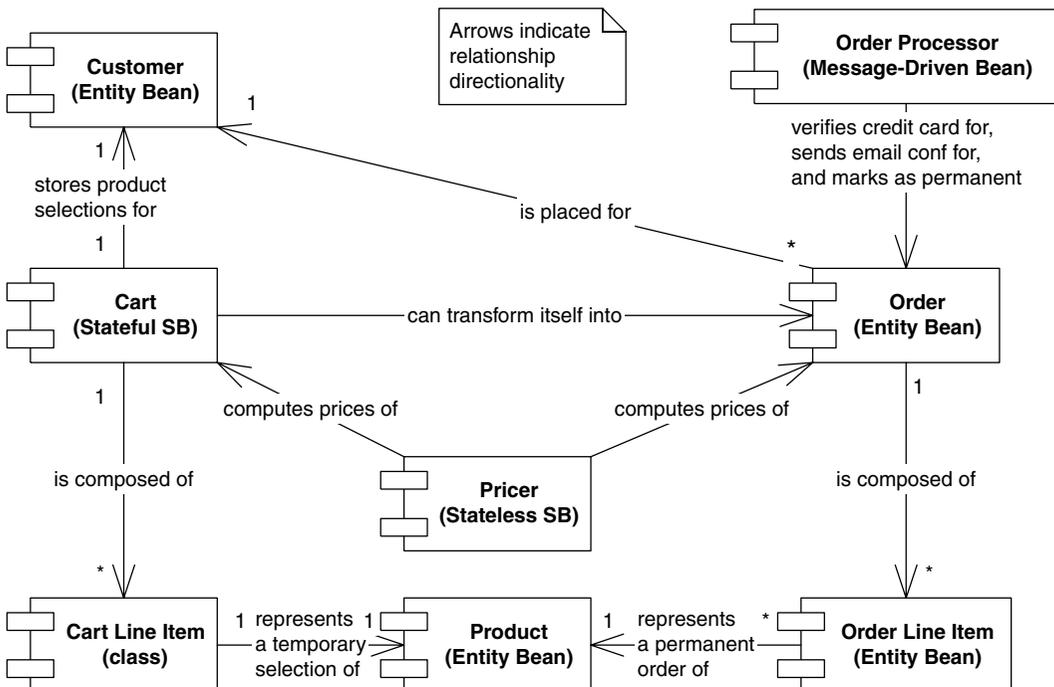


Figure 17.7 The major EJB components in our system.

Products

First, we need to model the products that Jasmine is selling. A product could be a motherboard, a monitor, or any other component. Products should be persistent parts of the deployment that last forever. Our product abstractions should represent the following data:

- The unique product ID
- The product name
- A description of the product
- The base price of the product (indicating the price of the product, with no discounts or taxes applied)

Jasmine should be able to add and delete products from the system using a maintenance tool. Because products are permanent, persistent parts of the system, they are best modeled as entity beans. Our product entity bean should have methods to get and set the above fields.

We also have a catalog session bean that wraps this entity bean, serving as a transactional networked façade.



Rather than entity beans, we could have used Java classes to represent the entities in our system, such as products, customers, and so on. However, many of these nouns (especially products) are prime candidates to be cached by the container. This means that by going with entity beans, our performance would actually improve. Furthermore, by using local interfaces, the access time is comparable to a Java class. And by using automatic entity bean code generators (see www.wiley.com/compbooks/roman), we can write about the same amount of code as a Java class.

Customers

Next, we need to represent information about Jasmine's customers. A customer represents an end user—perhaps an individual or a corporation that purchases goods from our Web site. Our customer abstraction contains the following data:

- Customer ID
- The customer's name (also used as the customer's login name for our Web site)
- The customer's address
- The customer's password (used to verify the customer's identity)

We also have a UserManager session bean that wraps this entity bean, serving as a transactional networked façade.



New customers, products, and so on can be added to the system in many ways. Jasmine could have users log in through a separate Web site and input their name, address information, password, and other profile data. We could also develop a custom maintenance tool (standalone or Web-based) for adding new products. To keep this example simple, we'll manually insert direct database data, but feel free to extend this for your purposes.

Carts

Next, we need to keep track of the selections a customer has made while navigating our catalog by modeling a shopping cart.

Each customer who has logged in should have his or her own temporary and separate cart in which to work. Therefore, our carts need to hold client-specific state in them. They should not be persistent, because the user can always cancel the cart.

This naturally lends itself to the stateful session bean paradigm. Each cart stateful session bean holds conversational state about the user's current cart. It allows us to treat the entire cart as one coarse-grained object. A new cart needs to be generated every time a user logs in. Each cart bean contains the following information:

- The customer, whom we authenticated at the login screen. We need to store customer information so that we know who to bill, what discounts to apply, and where to ship the manufactured products.
- The products and quantities that the customer currently has selected. This data is best represented in its own separate bean, called a *Cart Line Item*, described later.
- The subtotal for the cart, taking into account all the prices of the products the user wants as well as any discounts the user gets.
- The taxes charged. This is added to the subtotal for the final grand total.

In addition to this data, the cart beans will be smart and will know how to generate permanent orders from themselves. We describe orders a bit later.



When making shopping cart-like functionality, you have several choices. You can use session beans (as we are) for temporary shopping carts. You can also use servlets or JSPs session objects, which is appropriate if your shopping cart is primitive in functionality and shouldn't be reused for other graphical user interfaces.

A final choice is to use entity beans and to keep the shopping cart data in the database. The entity bean approach is appropriate for persistent shopping carts, where you'd like the user to retrieve the shopping cart when returning later. This might be useful if it requires complex configuration to get an item into a shopping cart, such as custom configuring a laptop computer. The downside to the entity bean approach is you need to write a shopping cart cleaning program that routinely sweeps abandoned shopping carts from the database.

Cart Line Items

As the user navigates the Web site, he or she will add products to the cart. For convenience of manipulation, we'd like to separate a cart into individual line items, where each line item represents data pertaining to a single product the user has currently selected. A cart has a 1:N relationship with its constituent line items.

Cart line items contain the following data:

- The ID of the line item
- The product that the user wants to buy
- The quantity of that product
- Any discounts the customer gets from the base price of the product

The cart line item is specific to one customer and is not persistent. It is best modeled as either a stateful session bean or a Java class that hangs off the cart. We'll choose to make it a Java class because there is no middleware that we need when a cart calls a line item. Furthermore, we may need to marshal the line item out to a servlet or JSP so that the contents can be displayed to a user. By modeling our line items as classes to begin with, we can achieve this easily.

Pricers

Because Jasmine wants customized pricing, we need the concept of a pricer—a component that takes a cart as input and calculates the price of that cart based on a set of pricing rules. A pricing rule might be, “Customer X gets a 5-percent discount,” or, “If you purchase 10 motherboards you get a 15-percent discount.” These pricing rules could be read in from a database or set via EJB environment properties (see Chapter 9).

Our pricer takes a cart as input and computes the subtotal (before taxes) of that cart. It figures out the subtotal by computing a discount for each cart line item in that bean and subtracting the discounts from the total price.

Our pricer works on any cart and holds no client-specific state. Once the pricer has computed a price on a cart, it is available to perform another computation on a different cart. It is also not a persistent object—it would not make sense to save a pricer because a pricer simply performs logic and holds no state. This means our pricer fits into the EJB world best as a stateless session bean.

Orders

Next, we need to model a permanent order for goods. We'll define an order abstraction for this purpose. An order is a cart that has been converted into a work request. An order represents a real business action that needs to take place, such as the production of goods. Generating an order and billing a customer go hand in hand.

An order contains the following information:

- The ID of this order (which the user can use to check on order status)
- The customer for which this order is generated (used for shipping address information)
- The products and quantities that should be ordered (as with carts, best represented as separate information; contained in *order line items*, described later)
- The subtotal and taxes on the order
- The date the order was placed

Orders are permanent, persistent objects. You want an order's state to be around if your deployment crashes for any reason because an order means money. Therefore, orders are best depicted as entity beans. In comparison, carts are not permanent—they represent temporary interactions with the customer. You don't want to write a cart's data to a database during a customer interaction, but you do want to keep track of the user's information—hence the stateful session bean is best applied for carts.

Order Line Items

For convenience of manipulation, we break up our notion of an order into individual line items, where each line item represents data pertaining to a single product the user has ordered. An order has a 1:N relationship with its constituent line items. Our order line item abstraction contains the following data:

- The ID of this order line item
- The product that this order line item represents (used by manufacturing to reveal which product to make)

- The quantity of the product that should be manufactured
- The discount that the customer received on this product

Because order line items are permanent, persistent objects, they are best represented as entity beans. At first, you might think an order line item is too small and fine-grained to be an entity bean and might better be represented as Java classes for performance. However, with EJB 2.0 local interfaces and by properly tweaking your EJB server, it is now possible to have both fine-grained and large-grained entity beans. Chapter 13 has more detail about how to optimize such entity beans for performance.

Order Processor

The last challenge we face is how to *generate* orders in our system. We'd like for the user to continue browsing the Web site when he has placed the order, rather than waiting to see if his credit card is approved. This is similar to Amazon.com's one-click functionality. We'd also like to email the user afterwards indicating whether the order was successfully placed.

The best paradigm to achieve this is messaging. When the user wants to order the shopping cart, we could send a JMS message containing the shopping cart reference. Then later, the message will be processed off the queue by an *order processor* message-driven bean. This order processor is responsible for querying the shopping cart, checking the user's credit card, checking inventory, e-mailing the user a confirmation, and creating the order (entity bean).

The challenge of sending data through JMS is that we cannot marshal EJB stubs in a JMS message. Thus, we couldn't send a shopping cart stub as a serialized bit-blob in a JMS message. This is a fundamental problem with message-driven beans: It's very challenging to send data into a message-driven bean that comes from another bean.

An alternative is to use *EJB object handles* (described in Chapter 9), which are serializable stubs. However, this might not work either, since the stateful session bean cart might time-out before the JMS message was processed.

Another alternative is to create a custom, serializable representation of the shopping cart, perhaps by using serializable Java objects. The problem here is we'd need to create all these extra Java classes, which is very annoying.

The best solution for us is to submit the order *before* sending a JMS message. We'd then mark the order status as *unverified*. The order processor receives the primary key of the order, retrieves the order entity bean from the database, checks the credit card, send the confirmation e-mail, and then changes the order status to *submitted*.

Our notion of an order can be easily extended to include order status, such as *Manufacturing* or *Shipping* and other order fulfillment information. It would also be interesting to e-mail the order status to the end user at regular intervals using the *JavaMail* API. Since we do not fulfill orders, we leave this as an exercise to the reader.

Object Model for the Presentation Tier

Our next task is to design our presentation tier, which displays the graphical user interface to the end user. For our presentation tier, we will use a few Java servlets and JSPs to interact with a client over HTTP.

What Are Servlets?

A *servlet* is a module that runs within a request/response-oriented server. A *request* is a call from a client, perhaps remotely located. Requests contain data that the client wants to send to the server. A *response* is data that the server wants to return to the client to answer the request. A servlet is a Java object that takes a request as input, parses its data, performs some logic, and issues a response back to the client. This is shown in Figure 17.8.

Servlets work on the same concept as CGI scripts. A CGI script is a program that takes data from standard input, performs some logic, and sends data to standard output. It doesn't matter what language you use to write a CGI script, as long as it follows this convention of reading data from standard input and writing data to standard output. This is shown in Figure 17.9.

The problem with CGI is that your server must restart the CGI script every time a new request is issued. This means every time a client communicates, your server needs to begin a new process. Starting and stopping processes are expensive operations.

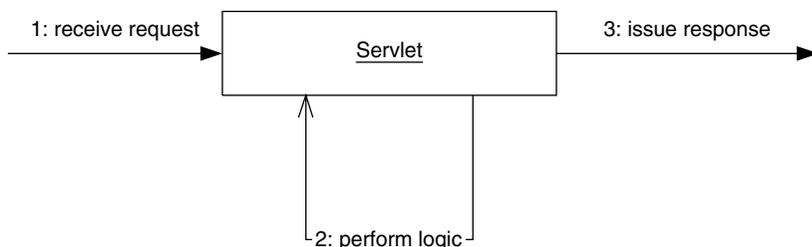


Figure 17.8 The basic servlet paradigm.

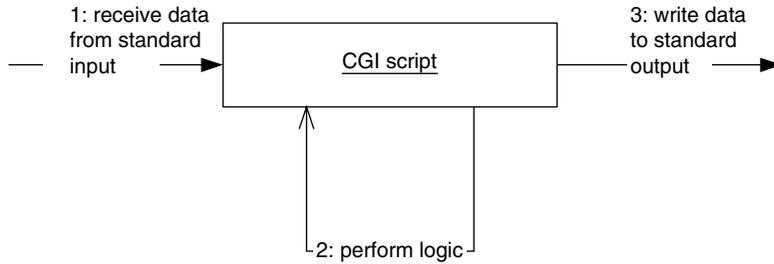


Figure 17.9 CGI scripts.

This is where servlets have an edge over CGI. Servlets are Java objects that function the same way as CGI scripts—taking in data (a request), performing some logic, and writing out data (a response). The difference between CGI and servlets is that CGI scripts must be restarted for every request, where servlets are pooled and reused over and over again to service many requests. This means you don't have to restart a process every time a new request comes over the network. This greatly enhances performance. The container pools servlets in the same way that the container pools enterprise beans.

Note that there are many different types of servlets, because the servlet concept is protocol-neutral. You could theoretically develop an FTP servlet, a POP servlet, and so on. An HTTP servlet, the most common type of servlet, works using the HTTP protocol. This is shown in Figure 17.10.

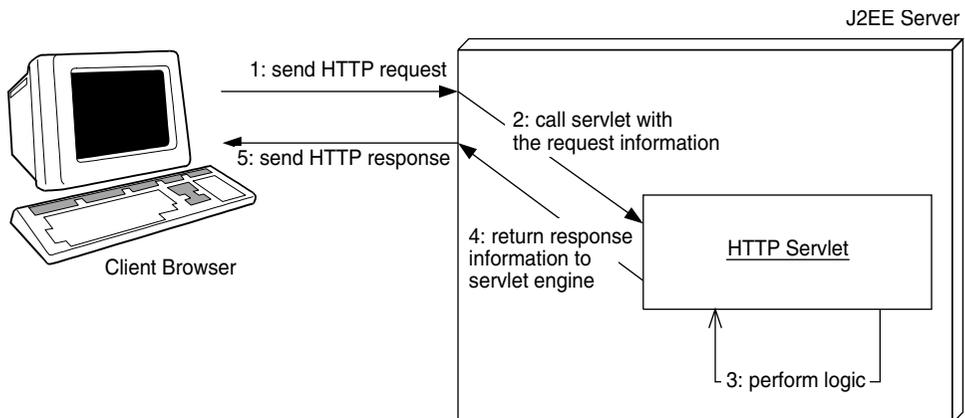


Figure 17.10 HTTP servlets.

An example of an HTTP servlet is shown in Source 17.1.

As you can see, HTTP servlets are very straightforward. They have a simple method called *service()* which responds to HTTP requests. In that method, we write some HTML back to the browser. If properly configured, the J2EE server will pool and reuse this servlet to service many HTTP requests at once.

We can also do trickier things—respond differently to different types of HTTP requests, maintain user sessions, read input parameters from Web forms (using the *HttpServletRequest* object), and call EJB components.

The great thing about servlets is that they are written in Java and therefore can be debugged just like any other Java code. The downside to servlets is that they require Java knowledge. It is therefore inappropriate to use servlets to write large amounts of HTML back to the user, because that HTML is interlaced with Java code, as we saw in Source 17.1. This makes it very challenging for Web designers to get involved with your deployment.

What are JavaServer Pages?

A JavaServer Page (JSP) is a flat file that is translated at runtime into a servlet. JSPs are generally useful for presentation-oriented tasks, such as HTML rendering. You don't need to know Java to write a JSP, which makes JSPs ideal for Web designers. A sample JSP is shown in Source 17.2.

As you can see, this just looks like HTML and is easily maintained by a graphic designer. You can do fancy things as well, such as interlacing Java code with

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

public class HelloWorld extends HttpServlet {

    public void service(
        HttpServletRequest req, HttpServletResponse rsp)
        throws ServletException, IOException
    {
        PrintWriter out = rsp.getWriter();
        out.println("<H1>Hello World</H1>");
    }
}
```

Source 17.1 A sample HTTP servlet.

```
<!doctype html public "-//w3c/dtd HTML 4.0//en">
<html>
<body>
<H1>Hello World</H1>
</body>
</html>
```

Source 17.2 A sample JSP.

the JSP, managing user sessions, and so on. Just about anything you can do in a servlet can be done in a JSP. The difference is that a JSP is a flat file that is translated into a servlet later. The code in source 17.2 would be translated into a servlet with *out.println()* statements for the HTML code.

How Do I Combine Servlets, JSPs, and EJB Components?

You have several choices when architecting your Web-based system. Here are just a few examples.

The JSP files can have embedded Java code that call EJB components. For example, we could interlace the following code into a JSP:

```
<html>
<H1>About to call EJB ... </H1>
<%
javax.naming.Context ctx = new javax.naming.InitialContext();
Object obj = ctx.lookup("MyEJBHome");
...
%>
</html>
```

When this JSP is translated into a servlet, the Java code would be inserted into the generated servlet. This is a bad idea, because the JSP files cannot be easily maintained by a graphic designer due to the large amount of Java code in the JSP.

The JSP files can communicate with EJB components via custom tags. You can design custom JSP tags that know how to interact with EJB components, called *JSP tag libraries*. Tag libraries are appealing because once you've designed them, graphic designers can call EJB components by using familiar tag-style editing rather than writing Java code. The tags then call Java code that understands how to invoke EJB components.

Servlets can call EJB components and then call JSPs. You can write one or more Java servlets that understand how to call EJB components and pass their results to JSP files. This is a model-view-controller paradigm, because the EJB layer is the model, the JSPs are the view, and the servlet(s) are the controller—they understand which EJB components to call and then which JSPs to call. This is shown in Figure 17.11. The advantage of this paradigm is that it pushes most of the Java code into servlets and EJB components. The JSPs have almost no Java code in them at all and can be maintained by graphic designers.

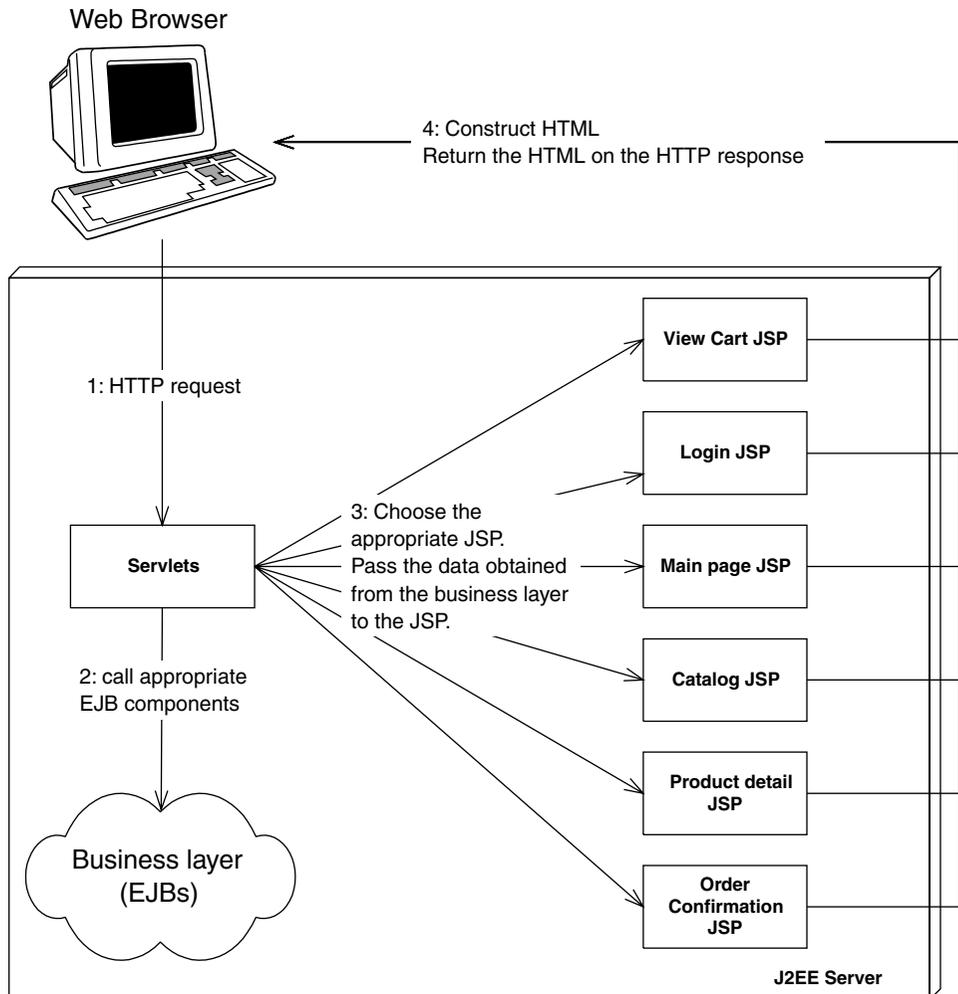


Figure 17.11 The EJB-JSP-Servlet Model-View-Controller Paradigm.

You can go with an off-the-shelf Web framework. Several off-the-shelf Web frameworks aid in building Web-based systems, such as Jakarta Struts.

JSPs in Our E-Commerce Deployment

We will choose a model-view-controller paradigm for our e-commerce deployment. We will have servlets that perform the controller processing, call our EJB components, and select the appropriate JSP based on the results of the EJB layer processing.

To fulfill Jasmine's requirements, we'll define the following servlets and JSPs:

A Login Page. The login page will be the first page the user deals with when going to Jasmine's Web site. It is responsible for reading in the user's name and then retrieving the appropriate customer entity bean that matches that name. It compares the user's submitted password with the permanent password stored with the customer entity bean. If the passwords match, a new cart stateful session bean is created for this customer. The customer information is stored in the cart so the cart contains the user's billing and shipping information. If the passwords don't match, an error is displayed and the user is given another chance to enter a password.

A Web Storefront Page. The user who gets through the login page is redirected to the Web storefront, which is the main page for Jasmine's store. This is the main navigation page for Jasmine's store. It links to the Catalog page and the view cart page.

A Catalog Page. To start adding products to the cart, the user can browse the list of available products by going to the catalog page. The user can also view details of a particular product, in which case we direct the user to the product detail page.

A Product Detail Page. When the user wants information about a particular product in the catalog, the product detail page shows that information. From this screen, the user can add the product to his or her cart.

A View Cart Page. This page allows the user to view and modify the shopping cart. This means deleting items or changing quantities. Every time the user changes something, we recalculate the price of the cart by calling the pricer stateless session bean.

An Order Confirmation Page. Finally, when the user is happy, he or she can convert the cart stateful session bean into an order entity bean. The user is then shown his or her Order Number, which is extracted from the order bean. We then send a JMS message to the OrderProcessor bean, which asynchronously processes the order.

This completes our object model design for our presentation tier. The flow of control for our pages is depicted in Figure 17.12. Note that the JSPs do not directly call each other: servlets receive all requests, call the appropriate EJB components, and route the results to the appropriate JSP for the HTML to be rendered.

Once we've developed the application, we need to package and deploy it. A J2EE application is packaged this way:

An Ejb-jar file (.jar) contains EJB components.

A Web archive file (.war) contains Web components, such as servlets, JSPs, HTML, images, and JSP tag libraries.

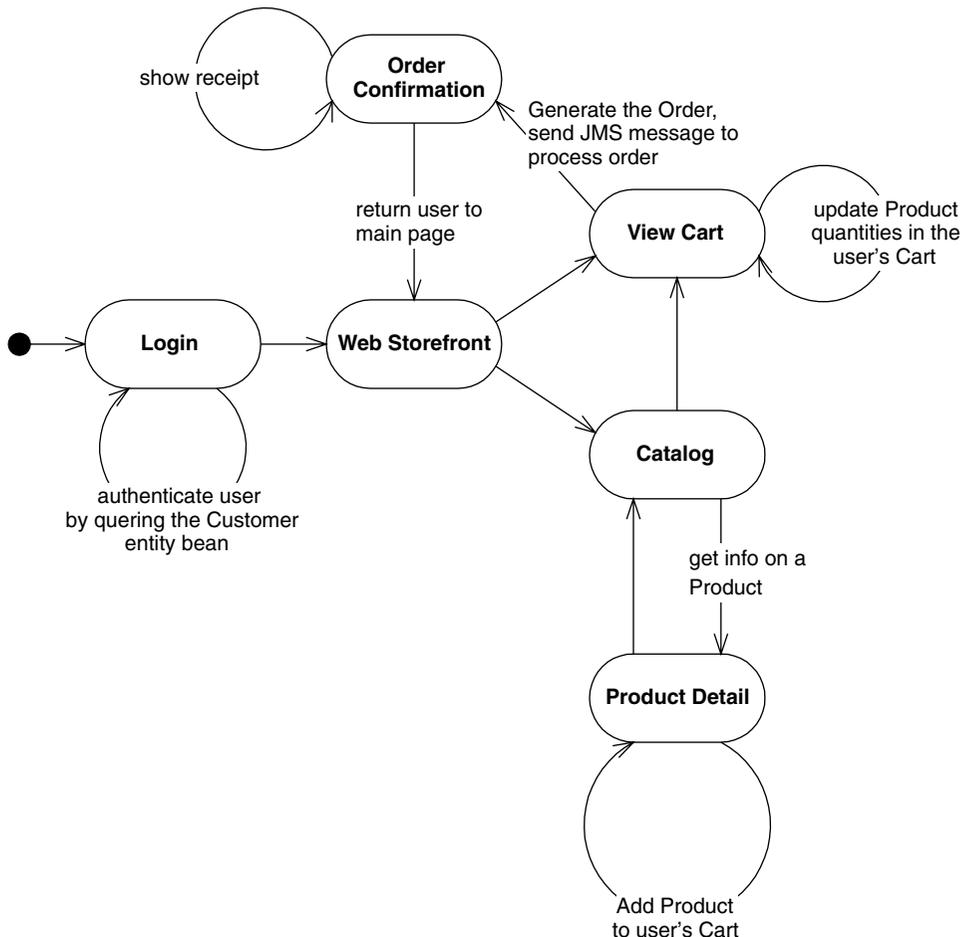


Figure 17.12 State diagram for our e-commerce site.

An **enterprise archive file (.ear)** is a J2EE application that contains a .jar file and a .war file. This is the unit of deployment you care most about, because it represents a J2EE application.

Each of these files follows the .ZIP rules for compression. The idea is that you first create the Ejb-jar file, then the Web archive, and then zip them up together into an enterprise archive. You deploy the enterprise archive into your application server using its tools, or perhaps by copying it into the appropriate folder. For code examples of how to build and deploy these archives, see the book's accompanying source code.

Example Code

Before concluding this chapter, let's look at an example of a servlet calling EJB components and then calling a JSP file, so that you can grasp how this model-view-controller paradigm is achieved.

We'll take the example of logging into our site. Source 17.3 shows the login JSP.

```
<%--
  This JSP displays a login screen.  When the user fills out
  the login screen, it will submit it to the Login Servlet,
  which will verify the user's credentials by calling
  EJB components.

  if verification is unsuccessful, the login servlet will return
  the user to this page to re-enter his credentials.

  if verification is successful, Jasmine's main page will be
  displayed.
--%>

<html>
<head>
  <title>Jasmine's Login page </title>
</head>

<body>

<%-- Include the title, which is "Jasmine's Computer Parts"--%>
<jsp:include page="title.jsp" />
```

Source 17.3 The login JSP.

```
<%-- Indicate the error page to use if an error occurs --%>
<jsp:directive.page errorPage="error.jsp" />

<%-- Display the login form --%>
<h4>Please Login</h4>
<p>
<form action="/jasmine/login" method="get">
  <table>
    <tr>
      <td><b>Name:</b></td>
      <td>
        <input type="text" name="Login" value="Ed Roman" size="19">
      </td>
    </tr>
    <tr>
      <td><b>Password:</b></td>
      <td>
        <input type="text" name="Password" value="password" size="19">
      </td>
    </tr>
    <tr>
      <td></td>
      <td>
        <input type="submit" value="Submit Information">
      </td>
    </tr>
  </table>
</form>

<%
  // get whether the person logged in successfully
  Boolean failed = (Boolean) request.getAttribute("loginFailed");
  if (failed != null) {
    if (failed.booleanValue() == true) {
%>
  <p>
    <strong>Could not log in! Please try again.</strong>
  <p>
<%
  }
}
%>

</body>
</html>
```

Source 17.3 The login JSP (*continued*).

Source 17.4 shows our login servlet.

The login servlet is self-documenting. It cracks open the request, figures out which EJB components to call, and selects the appropriate JSP.

```
package examples;

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

/**
 * This is the very first servlet the client deals with.
 * It's a Login authentication servlet. It asks the user
 * for his name and password, and passes it to the UserManager
 * stateless session bean for verification.
 * If the user authenticates properly, reference to a new
 * Cart is saved in his HttpSession object, and the user can
 * begin to add items to his cart and shop around.
 */
public class LoginServlet extends HttpServlet {

    /**
     * UserManager home object for authenticating user
     */
    private UserManagerHome userManagerHome;

    /**
     * Cart home object for creating a new cart when
     * the user logs in.
     */
    private CartHome cartHome;

    /**
     * The servlet engine calls this method once to
     * initialize a servlet instance.
     *
     * In the body of this method, we acquire all the
     * EJB home objects we'll need later.
     */
    public void init(ServletConfig config) throws ServletException {

        super.init(config);

        try {
```

Source 17.4 The login servlet.

```
/*
 * Get the JNDI initialization parameters.
 * We externalize these settings to the
 * servlet properties to allow end-
 * users to dynamically reconfigure their
 * environment without recompilation.
 */
String initCtxFactory =
    getInitParameter(Context.INITIAL_CONTEXT_FACTORY);

String providerURL =
    getInitParameter(Context.PROVIDER_URL);

/*
 * Add the JNDI init parameters to a
 * properties object.
 */
Properties env = new Properties();
env.put(Context.INITIAL_CONTEXT_FACTORY, initCtxFactory);
env.put(Context.PROVIDER_URL, providerURL);

/*
 * Get the initial JNDI context using the above
 * startup params.
 */
Context ctx = new InitialContext(env);

/*
 * Look up the UserManager and Cart Home Objects
 * we need via JNDI
 */
userManagerHome = (UserManagerHome)
    ctx.lookup("UserManagerHome");

cartHome = (CartHome) ctx.lookup("CartHome");
}
catch (Exception e) {
    log(e);
    throw new ServletException(e.toString());
}
}

/**
 * The servlet engine calls this method when the user's
 * desktop browser sends an HTTP request.
 */
public void service(HttpServletRequest request,
```

Source 17.4 The login servlet (*continued*).

```
        HttpServletResponse response)
        throws ServletException, IOException {

    /*
     * Set up the user's HttpSession
     */
    HttpSession session = request.getSession(true);

    /*
     * Retrieve the login name / password from the
     * URL string.
     */
    String loginName = request.getParameter("Login");
    String password = request.getParameter("Password");
    boolean isLogin=false;

    /*
     * If user has not tried to log in yet, present
     * him with the login screen.
     */
    if ((loginName == null) || (password == null)) {
        writeForm(request, response, false);
    }

    /*
     * Otherwise, the user has been to this screen
     * already, and has entered some information.
     * Verify that information.
     */
    else {
        /*
         * Uses the UserManager Stateless Session bean to
         * authenticate the user credentials.
         */
        try {
            UserManager userManager=userManagerHome.create();
            isLogin= userManager.validateUser(loginName,password);
        }
        catch (Exception e) {
            writeForm(request, response, true);
            e.printStackTrace();
            return;
        }
        /*
         * If the passwords match, make a new Cart Session
         * Bean, and add it to the user's HttpSession
         * object. When the user navigates to other
```

Source 17.4 The login servlet (*continued*).

```
* servlets, the other servlets can access the
* HttpSession to get the user's Cart.
*/
if (isLogin) {
    try {
        Cart cart = cartHome.create(loginName);
        session.setAttribute("cart", cart);

        /*
         * Call the main page
         */
        RequestDispatcher disp =
            this.getServletContext().getRequestDispatcher("/wsf.jsp");
        disp.forward(request, response);

        return;
    }
    catch (Exception e) {
        log(e);
        throw new ServletException(e.toString());
    }
}

/*
 * If there was no match, the user is
 * not authenticated. Present another
 * login screen to him, with an error
 * message indicating that he is not
 * authenticated.
 */
writeForm(request, response, true);
}

/**
 * Writes the Login Screen (private use only)
 *
 * @param showError true means show an error b/c client
 * was not authenticated last time.
 */
private void writeForm(HttpServletRequest request,
                        HttpServletResponse response,
                        boolean showError)
    throws ServletException, IOException {

    /*
     * Set a variable indicating whether or not we failed to
```

Source 17.4 The login servlet (*continued*).

```
    * log-in. The JSP will read this variable.
    */
    request.setAttribute("loginFailed", new Boolean(showError));

    /*
    * Forward the request to the login JSP
    */
    RequestDispatcher disp =
        this.getServletContext().getRequestDispatcher("/login.jsp");
    disp.forward(request, response);
}

private void log(Exception e) {
    e.printStackTrace();
}

public String getServletInfo() {
    return "The Login servlet verifies a user.";
}
}
```

Source 17.4 The login servlet (*continued*).

If you're curious to see how the other use-cases are implemented, see the book's accompanying source code. And as a reminder, this is just one of many ways to implement a Web architecture.

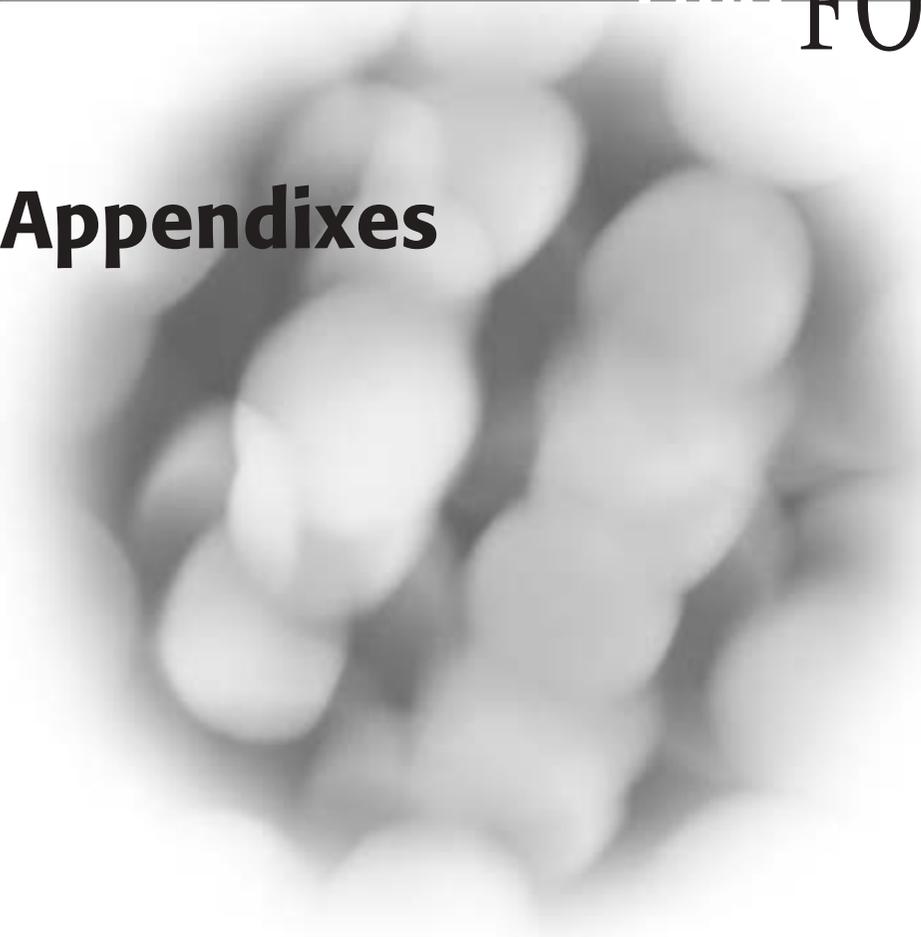


As an alternative, we could have also chosen a single-servlet architecture with only one servlet and many JSPs. This single servlet would call Java classes, and each Java class would represent a Web use-case and understand the EJB components to call. For example, we could have a Java class that understood how to verify login credentials. The advantage of this paradigm is we could reuse these Web use-case classes in several pages, and our servlet layer would be completely decoupled from our EJB components.

Summary

In this chapter, we've painted a picture of how our e-commerce system should behave. Now that we've made the proper abstractions, our components should fall into place easily. By performing this high-level analysis, we can be confident that our final product will be extensible and reusable for some time to come.

Appendixes



The appendixes in this book include topics you might not need to learn right away, but that are nonetheless useful, including:

Java Remote Method Invocation over the Internet Inter-ORB Protocol

(RMI-IIOP). Appendix A kicks things off by exploring Java RMI-IIOP.

Here you'll gain a solid foundation in RMI-IIOP, at both the conceptual and programmatic levels, which is necessary for understanding how EJB works behind the scenes.

The Java Naming and Directory Interface (JNDI). Appendix A continues by investigating the JNDI, a standard Java extension package for performing naming and directory operations. We'll learn the concepts, how to program with JNDI, and how JNDI relates to EJB.

CORBA interoperability. Appendix B discusses how you can integrate CORBA and EJB systems. We'll learn why RMI-IIOP was invented as a bridge between CORBA and Java systems. We'll also see code that calls an EJB component from a CORBA client.

Reference guides. Appendix C is a quick reference guide for those building deployment descriptors; Appendix D explains the EJB Query Language (EJB-QL) for container managed persistent entity beans; and Appendix E is an API and diagram reference guide. We don't recommend you read these appendixes until you're programming with EJB and need a reference guide.

RMI-IIOP and JNDI Tutorial

To help you to truly understand EJB, this appendix explains the technologies that EJB depends upon—Java RMI-IIOP and the Java Naming and Directory Interface (JNDI).

The goal of this appendix is to teach you enough about RMI-IIOP and JNDI for you to be productive in an EJB environment. This tutorial will cover the basics, but is by no means a complete RMI-IIOP and JNDI tutorial, and for good reason—most organizations will not need to use these technologies beyond the extent we describe in this appendix, and your reading time is valuable.

Interested readers who want to learn more about RMI-IIOP and JNDI should consult the following references:

The RMI-IIOP and JNDI Tutorials. These are available for free on the Sun Microsystems Web site, <http://java.sun.com>.

The RMI-IIOP and JNDI Specifications. The specifications define the core of RMI-IIOP and JNDI. They are a bit more technical but should not be tough after reading this appendix. They are also downloadable from <http://java.sun.com>.



Your J2EE server implementation should ship with RMI-IIOP and JNDI implementations. It is generally a bad idea to mix and match implementations, like combining Sun's RMI-IIOP package with BEA's JNDI implementation. For the easiest development path, stick with a single-vendor solution.

Java RMI-IIOP

Java RMI-IIOP (which stands for Java Remote Method Invocation over the Internet Inter-ORB Protocol) is J2EE's de facto mechanism for performing simple, powerful networking. RMI-IIOP allows you to write distributed objects in Java, enabling objects to communicate in memory, across Java Virtual Machines, and across physical devices.



RMI-IIOP is not your only choice for performing remote method invocations in Java. You can also use Java Remote Method Invocation (RMI). RMI was the original way to perform remote method invocations in Java and uses the *java.rmi* package. RMI-IIOP is a special version of RMI that is compliant with CORBA and uses both *java.rmi* and *javax.rmi*.

RMI has some interesting features not available in RMI-IIOP, such as distributed garbage collection, object activation, and downloadable class files. But EJB and J2EE mandate that you use RMI-IIOP, not RMI. Therefore we will not cover RMI.

If you wish to learn more about why RMI-IIOP was invented as an extension of RMI and also survey the CORBA compatibility features of RMI-IIOP, read Appendix B.

Remote Method Invocations

A *remote procedure call (RPC)* is a procedural invocation from a process on one machine to a process on another machine. RPCs enable traditional procedures to reside on multiple machines, yet still remain in communication. They are a simple way to perform cross-process or cross-machine networking.

A *remote method invocation* in Java takes the RPC concept one step further and allows for distributed *object* communications. RMI-IIOP allows you to invoke methods on objects remotely, not merely procedures. You can build your networked code as full objects. This yields the benefits of an object-oriented programming, such as inheritance, encapsulation, and polymorphism.

Remote method invocations are by no means simple. These are just some of the issues that arise:

Marshalling and unmarshalling. RMIs (as well as RPCs) allow you to pass parameters, including Java primitives and Java objects, over the network. But what if the target machine represents data differently than the way you represent data? For example, what happens if one machine uses a different binary standard to represent numbers? The problem becomes even more apparent when you start talking about objects. What happens if you send an object reference over the wire? That pointer is not usable on the other

machine because that machine's memory layout is completely different from yours. *Marshalling* and *unmarshalling* is the process of massaging parameters so that they are usable on the machine being invoked on remotely. It is the packaging and unpackaging of parameters so that they are usable in two heterogeneous environments. As we shall see, this is taken care of for you by Java and RMI-IIOP.

Parameter passing conventions. There are two major ways to pass parameters when calling a method: *pass-by-value* and *pass-by-reference*, as shown in Figure A.1. When you pass-by-value, you pass a copy of your data so that the target method is using a copy, rather than the original data. Any changes to the argument are reflected only in the copy, not the original. Pass-by-reference, on the other hand, does not make a copy. With pass-by-reference, any modifications to parameters made by the remote host affect the original data. The flexibility of both the pass-by-reference and pass-by-value models is advantageous, and RMI-IIOP supports both. We'll see how in the pages to come.

Network or machine instability. With a single JVM application, a crash of the JVM brings the entire application down. But consider a distributed object application, which has many JVMs working together to solve a business problem. In this scenario, a crash of a single JVM should not cause the distributed object system to grind to a halt. To enforce this, remote method invocations need a standardized way of handling a JVM crash, a machine crash, or network instability. When some code performs a remote invocation, the code should be informed of any problems encountered during the operation. RMI-IIOP performs this for you, abstracting out any JVM, machine, or network problems from your code.

As you can see, there's a lot involved in performing RMIs. RMI-IIOP contains measures to handle many of these nasty networking issues for you. This reduces the total time spent dealing with the distribution of your application, allowing you to focus on the core functionality.

J2EE 1.3-compliant servers are required to ship RMI-IIOP implementations to enable you to perform networking. Your RMI-IIOP code is then portable to any hardware or operating system on which these implementations execute. Contrast this with proprietary, platform-dependent RPC libraries, and you can see some real value in RMI-IIOP.

The Remote Interface

We begin our exploration of RMI-IIOP by reviewing one of object-oriented design's great programming practices—the separation of the interface of code from its implementation.

RMI-IIOP Client Address Space

RMI-IIOP Server Address Space

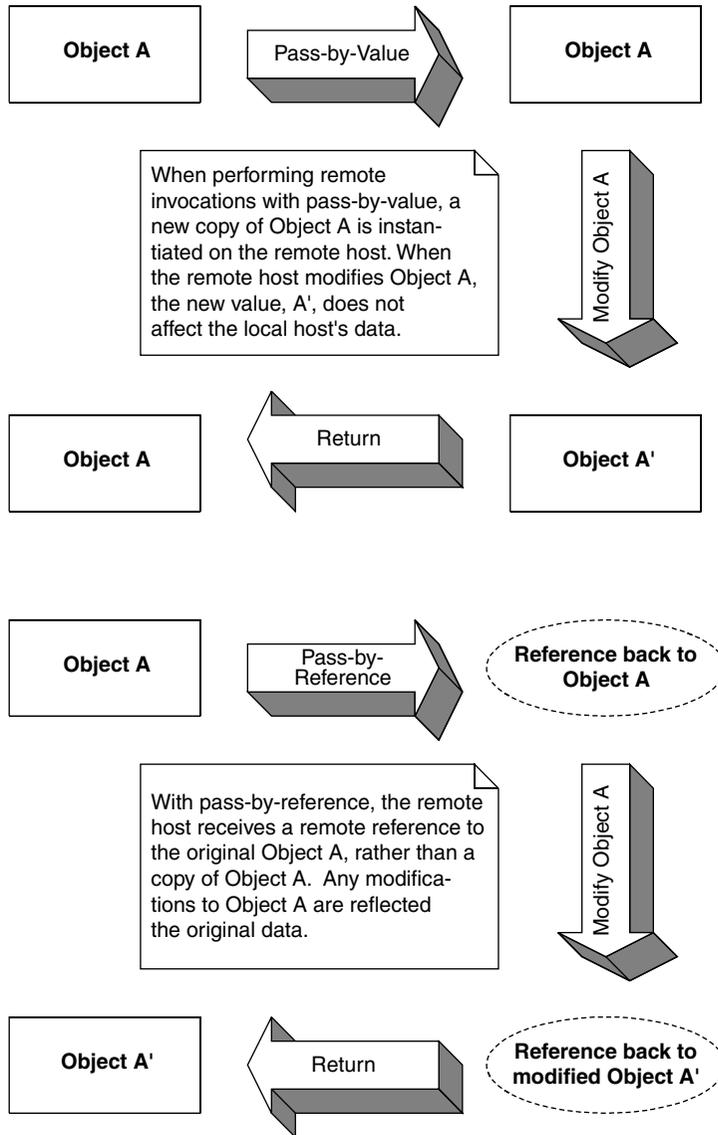


Figure A.1 Pass-by-value versus pass-by-reference.

The **interface** defines the exposed information about an object, such as the names of its methods and what parameters those methods take. It's what the client works with. The interface masks the implementation from the

viewpoint of clients of the object, so clients deal only with the end result: the methods the object exposes.

The implementation is the core programming logic that an object provides. It has some very specific algorithms, logic, and data.

By separating interface from implementation, you can vary an object's proprietary logic without changing any client code. For example, you can plug in a different algorithm that performs the same task more efficiently.

RMI-IIOP makes extensive use of this concept. All networking code you write is applied to interfaces, *not* implementations. In fact, you *must* use this paradigm in RMI-IIOP—you do not have a choice. It is impossible to perform a remote invocation directly on an object implementation. You can operate solely on the interface to that object's class.

Therefore, when using RMI-IIOP, you must build a custom interface, called a *remote interface*. This remote interface should *extend* the interface *java.rmi.Remote*. Your interface should have within it a copy of each method your remote object exposes.

We'll now begin to build a simple example illustrating the basics of RMI-IIOP. In this example, a remote object exposes one method: *generate()*. *generate()* returns a new, unique long number each time it's called. This is useful, for example, when generating primary keys for data, such as entity beans (discussed in Chapter 5).

Source A.1 is a valid remote interface.

Client code that wants to call methods on your remote object must operate on *IPKGenerator*. Notice that each method must also throw a *java.rmi.RemoteException*. A *RemoteException* is thrown when there is a problem with the network, such as a machine crashing or the network dying.

```
import java.rmi.Remote;
import java.rmi.RemoteException;

/**
 * The remote interface for the remote object. Clients use this
 * remote interface to perform any operations on the remote object.
 */
public interface IPKGenerator extends Remote {
    public long generate() throws RemoteException;
}
```

Source A.1 IPKGenerator.java.



With RMI-IIOP, you can never fully separate your application from the network. At some point, you'll need to deal with remote exceptions being thrown due to networking issues. Some may consider this a limitation of RMI-IIOP because the network is not entirely seamless: Remote exceptions force you to differentiate a local method from a remote method. But in some ways, this is an advantage of RMI-IIOP as well. Interlacing your code with remote exceptions forces you to think about the network and encourages distributed object developers to consider issues such as the network failing, the size of parameters going across the network, and more.

The Remote Object Implementation

Remote objects are networked object implementations that can be called by another JVM. They *implement* a remote interface and thus expose methods that can be invoked by remote clients.

The physical locations of remote objects and the clients that invoke them are not important. For example, it is possible for a client running in the same address space as a remote object to invoke a method on that object. It's also possible for a client across the Internet to do the same thing. To the remote object, both invocations appear to be the same.

To make your object a remote object available to be invoked on by remote hosts, your remote class must perform *one* of the following steps:

Extend the class `javax.rmi.PortableRemoteObject`. *PortableRemoteObject* is a base class from which you can derive your remote objects. When your remote object is constructed, it automatically calls the *PortableRemoteObject*'s constructor, which makes the object available to be called remotely.

Don't extend `javax.rmi.PortableRemoteObject`. Perhaps your remote object class needs to inherit implementation from another custom class. In this case, because Java does not allow for multiple implementation inheritance, you cannot extend *PortableRemoteObject*. If you do this, you must manually export your object so that it is available to be invoked on by remote hosts. To export your object, call `javax.rmi.PortableRemoteObject.exportObject()`.

Now let's create the remote object class. This class implements the *IPKGenerator* interface, and it is shown in Source A.2.

In our remote object constructor, the superclass makes our object available to be called remotely. This makes the object available at a random port number. Once the remote object's constructor is complete, this object is available forever for any virtual machine to invoke on; that is, until someone calls `unexportObject()`.

```
import java.rmi.RemoteException;
import javax.rmi.PortableRemoteObject;

/**
 * The remote object which generates primary keys
 */
public class PKGenerator
    extends PortableRemoteObject
    implements IPKGenerator {

    /*
     * Our remote object's constructor
     */
    public PKGenerator() throws Exception, RemoteException {

        /*
         * Since we extend PortableRemoteObject, the super
         * class will export our remote object here.
         */
        super();
    }

    /*
     * Generates a unique primary key
     */
    public synchronized long generate() throws RemoteException {
        return i++;
    }

    private static long i = System.currentTimeMillis();
}
```

Source A.2 PKGenerator.java.



This primary key generator has its own shortcoming as well: to generate a primary key, someone needs to invoke a remote method, which could be a performance bottleneck.

If you need to generate primary keys in production, see the companion to this book, Floyd Marinescu's *EJB Design Patterns*, published by John Wiley & Sons.

Stubs and Skeletons

Now that we've seen the server code, let's look at the architecture behind how the networking is achieved in RMI-IIOP. One of the benefits of RMI-IIOP is an

Issues with Our Primary Key Generation Algorithm

Our primary key generation algorithm is to simply increment a number each time someone calls our server. This generator overcomes two common challenges when writing an RMI implementation:

Threading. RMI-IIOP allows many clients to connect to a server at once. Thus, our remote object implementation may have many threads running inside of it. But when generating primary keys, we never want to generate a duplicate key because our keys are not unique and thus would not be good candidates to use in a database. Therefore, it is important to have the synchronized block around the *generate()* method, so that only one client can generate a primary key at once.

JVM crashes. We must protect against a JVM crash (or hardware failure). Thus, we initialize our generator to the current time (the number of milliseconds that have elapsed since 1970). This is to ensure that our primary key generator increases monotonically (that is, primary keys are always going up in value) in case of a JVM crash. Note that we haven't considered daylight savings time resulting in duplicate keys. If we were to use this code in production, we would need to account for that.

almost illusionary, transparent networking. You can invoke methods on remote objects just as you would invoke a method on any other Java object. In fact, RMI-IIOP completely masks whether the object you're invoking on is local or remote. This is called *local/remote transparency*.

Local/remote transparency is not as easy as it sounds. To mask that you're invoking on an object residing on a remote host, RMI-IIOP needs to somehow simulate a local object that you can invoke on. This local object is called a *stub*. It is responsible for accepting method calls locally and *delegating* those method calls to their actual object implementations, which are possibly located across the network. This effectively makes every remote invocation appear to be a local invocation. You can think of a stub as a placeholder for an object that knows how to look over the network for the real object. Because you invoke on local stubs, all the nasty networking issues are hidden.

Stubs are only half of the picture. We'd like the remote objects themselves—the objects that are being invoked on from remote hosts—to not worry about networking issues as well. Just as a client invokes on a stub that is local to that client, your remote object needs to accept calls from a *skeleton* that is local to that remote object. Skeletons are responsible for receiving calls over the net-

work (perhaps from a stub) and delegating those calls to the remote object implementation. This is shown in Figure A.2.

Your RMI-IIOP implementation (that is, your J2EE server) should provide a means to *generate* the needed stubs and skeletons, thus relieving you of the networking burden. Typically, this is achieved through command-line tools. For example, Sun's J2EE reference implementation ships with a tool called *rmic* (which stands for the RMI compiler) to generate stub and skeleton classes. As you can see from Figure A.2, you should deploy the stub on the client machine and the skeleton on the server machine.

Object Serialization and Parameter Passing

One of the more interesting responsibilities of stubs and skeletons is to handle your parameters. This next section discusses how parameters are passed in Java RMI-IIOP. We also explore the power of *object serialization* and how it assists in parameter passing.

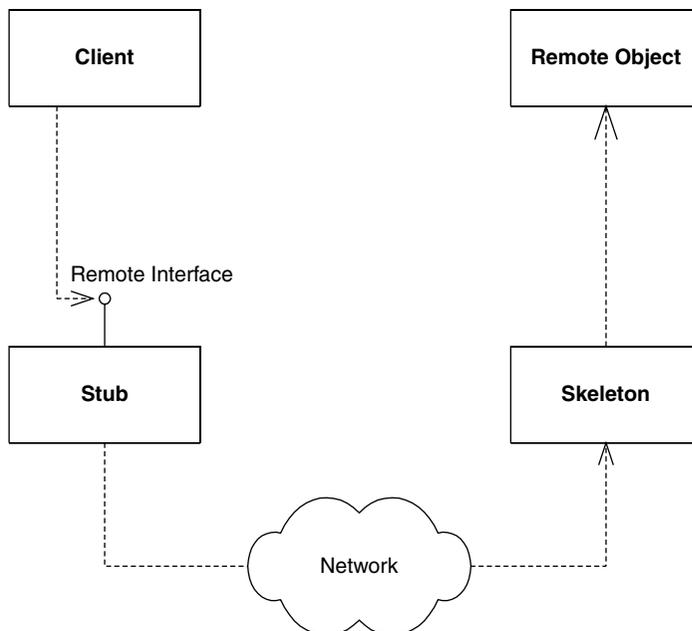


Figure A.2 Stubs and skeletons.

Passing By-Value

When invoking a method using RMI-IIOP, all parameters to the remote method are passed *by-value*. This means that when a client calls a server, all parameters are copied from one machine to the other.

Passing objects by-value is very different from the Java programming language. When you call a method in Java and pass an object as a parameter, that object is passed *by reference*. More specifically, the reference to the object is copied, but the actual object's data is not.

There's a big problem with passing by-value. If you're trying to pass an object over the network and that object contains references to other objects, how are those references resolved on the target machine? A memory address on one machine does not map to the same memory address on another machine. Also, the referenced object may not even exist on the target machine. How do we get around this?

Object Serialization

Java introduces the concept of *object serialization* to handle this problem. *Serialization* is the conversion of a Java object into a bit-blob representation of that object. You can send bit-blobs anywhere. For example, you can use object serialization as an instant file-format for your objects and save them to your hard disk. RMI-IIOP also uses object serialization to send parameters over the network. When you're ready to use the object again, you must deserialize the bit-blob back into a Java object. Then it's magically usable again.

The Java language handles the low-level details of serialization. In most cases, you don't need to worry about any of it. To tell Java that your object is serializable, your object must implement the `java.lang.Serializable` interface. That's all there is to it: Take this one simple step, and let Java handle the rest. `java.lang.Serializable` defines no methods at all—it's simply a *marker interface* that identifies your object as something that can be serialized and deserialized.

You can provide your own custom serialization by implementing the `writeObject()` method on your object, or provide custom deserialization by implementing `readObject()`. This might be useful if you'd like to perform some sort of compression on your data before your object is converted into a bit-blob and decompression after the bit-blob is restored to an object.

Figure A.3 shows the serialization/deserialization process, where `writeObject()` is responsible for saving the state of the class, and `readObject()` is responsible for restoring the state of the class. These two methods will be called automatically when an object instance is being serialized or deserialized. If you choose not to

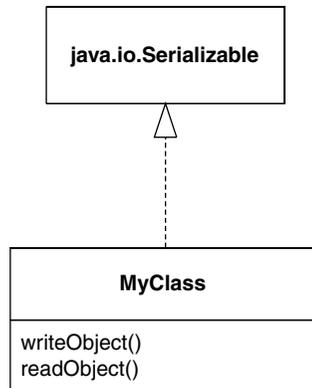


Figure A.3 The Java serialization process.

define these methods, then the default serialization mechanisms will be applied. The default mechanisms are good enough for most situations.

Rules for Serialization

Java serialization has the following rules for member variables held in serialized objects:

- Any basic primitive type (int, char, and so on) is automatically serialized with the object and is available when deserialized.
- Java objects can be included with the serialized bit-blob or not; it's your choice. The way you make your choice is as follows:
 - Objects marked with the *transient* keyword are not serialized with the object and are not available when deserialized.
 - Any object that is not marked with the transient keyword must implement *java.lang.Serializable*. These objects are converted to bit-blob format along with the original object. If your Java objects are neither transient nor implement *java.lang.Serializable*, a *NotSerializable* Exception is thrown when *writeObject()* is called.

Thus, when you serialize an object, you also serialize all nontransient subobjects as well. This means you also serialize all nontransient sub-subobjects (the objects referenced from the subobjects). This is repeated recursively for every object until the entire reference graph of objects is serialized. This recursion is handled automatically by Java serialization, as shown in Figure A.4. You simply need to make sure that each of your member objects implements the

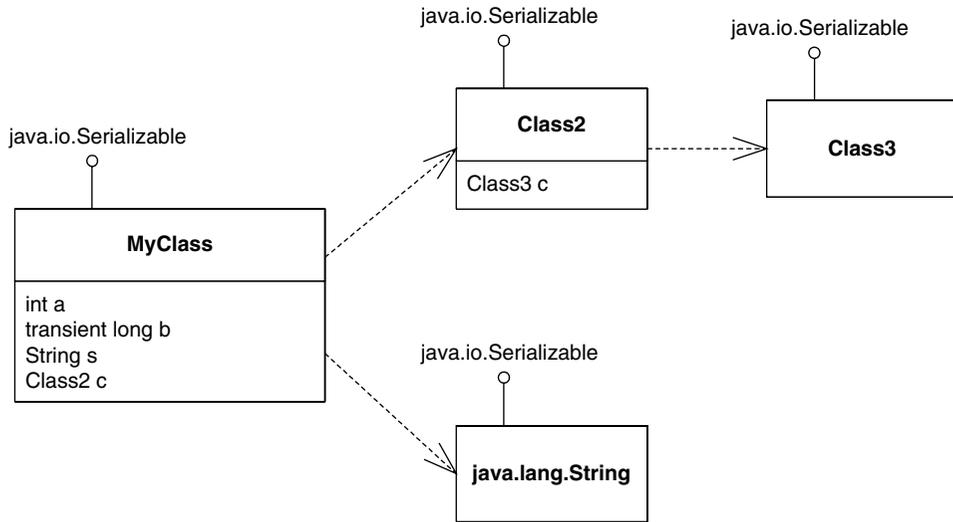


Figure A.4 Object serialization recursion.

java.lang.Serializable interface. When serializing `MyClass`, Object Serialization will recurse through the dependencies shown, packaging up the entire graph of objects as a stream. In this diagram, everything will get serialized except for `transient long b`, since it is marked as `transient`.

What Should You Make Transient?

How do you know which member variables should be marked `transient` and which should not? Here are some good reasons to mark an object as `transient`:

- The object is large. Large objects may not be suitable for serialization because operations you do with the serialized blob may be very intensive. Examples here include saving the blob to disk or transporting the blob across the network.
- The object represents a resource that cannot be reconstructed on the target machine. Some examples of such resources are database connections and sockets.
- The object represents sensitive information that you do not want to pass in a serialized stream.

Note that object serialization is not free—it is a heavyweight operation for large graphs of objects. Make sure you take this into account when designing your distributed object application.

Object Serialization and RMI-IIOP

Java RMI-IIOP relies on object serialization for passing parameters via remote method invocations. Figure A.5 shows what `MyObject`'s object graph could look like. Notice that every field and subfield is a valid type for Java serialization.

Figure A.5 shows how RMI-IIOP handles *pass-by-value*, where an entire graph of objects is serialized into a bit-blob, sent across the network, and then deserialized on the target machine. But passing parameters by-value can lead to inefficiencies. What if your referenced graph of objects is very large? What if you

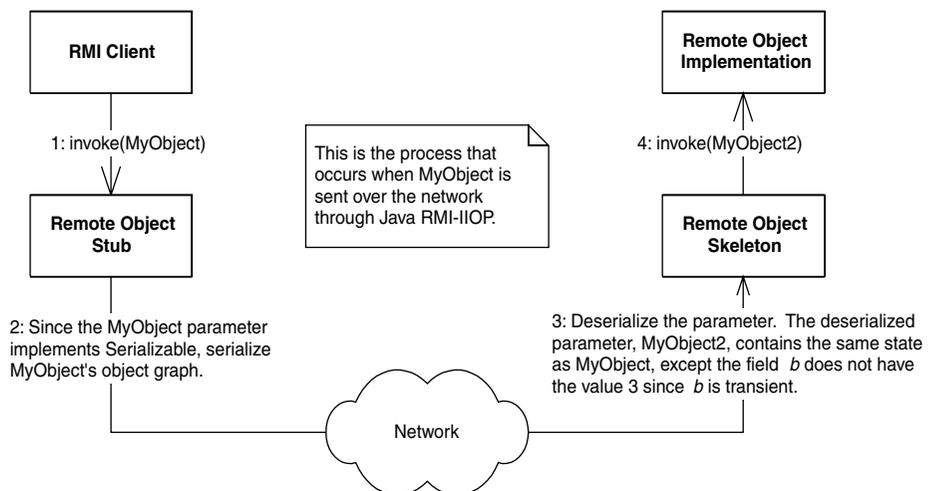
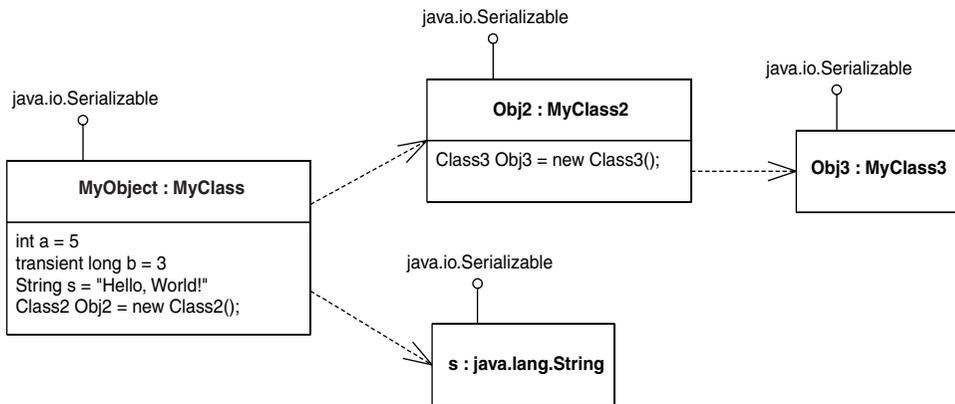


Figure A.5 Java RMI-IIOP and object serialization.

have lots of state to send across the network? The ensuing network lag from performing the invocation may be unacceptable.

There is another way to pass arguments. RMI-IIOP *simulates* a pass-by-reference convention, which means the arguments are not copied over. Rather, the server modifies the client's copy of the parameter.

If you want to pass a parameter by-reference, the parameter must itself be a remote object. The parameter is thus an object that is callable remotely. When the client calls the server, the RMI-IIOP runtime sends a stub to that remote object to the server. The server can perform a callback on that stub, which connects the server to the remote object living on the client machine. Figure A.6 shows the process that occurs when *MyRemoteObject*, an instance of *MyRemoteClass*, is sent over the network through Java RMI-IIOP.

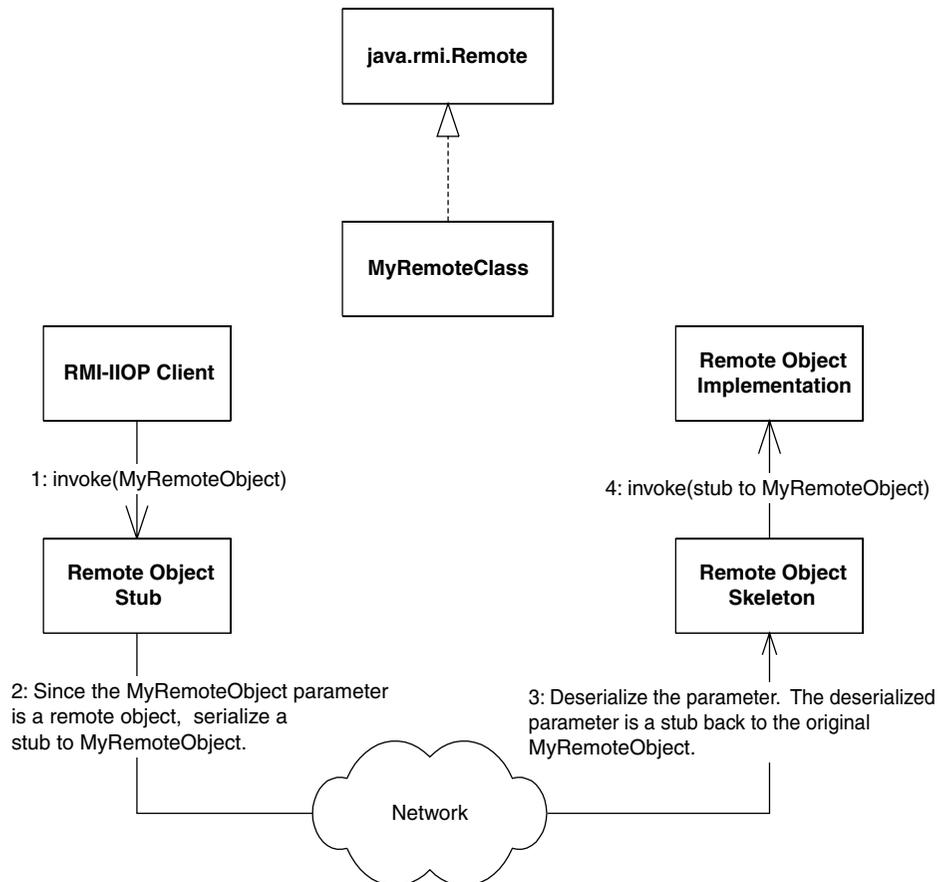


Figure A.6 Pass-by-reference with Java RMI-IIOP.

The best way to understand this paradigm is by analogy. In Java programming language, when you pass an object as a parameter, the object reference is copied. In RMI-IIOP, when you pass an object as a parameter, the stub is copied. Both of these strategies achieve pass-by-reference because they are cloning the thing that points to the object, rather than the object itself.

Because Java RMI-IIOP stubs are also serializable, they are passable over the network as a bit-blob. This is why earlier we said that *all* parameters in Java RMI-IIOP are passed by-value. Thus, Java RMI-IIOP only *simulates* pass-by-reference by passing a serializable stub, rather than serializing the original object. By making your parameters remote objects, you can effectively avoid the network lag in passing large objects.

In summary, we have the following rules for passing objects using Java RMI-IIOP:

- All Java basic primitives are passed by-value when calling methods remotely. This means copies are made of the parameters. Any changes to the data on the remote host are not reflected in the original data.
- If you want to pass an object over the network by-value, it must implement `java.lang.Serializable`. Anything referenced from within that object must follow the rules for Java serialization. Again, any changes to the data on the remote host are not reflected in the original data.
- If you want to pass an object over the network by-reference, it must be a remote object, and it must implement `java.rmi.Remote`. A stub for the remote object is serialized and passed to the remote host. The remote host can then use that stub to invoke callbacks on your remote object. There is only one copy of the object at any time, which means that all hosts are calling the same object.

Now that you understand parameter passing, let's move on. For us to complete our RMI-IIOP sample application, we need some way to publish the server and have the client locate that server. This process, called *bootstrapping*, is achieved via the JNDI. Let's put our RMI-IIOP example on hold while we learn about JNDI. We'll return later to complete the example.

The Java Naming and Directory Interface (JNDI)

The JNDI is a J2EE API that provides a standard interface for locating users, machines, networks, objects, and services. For example, you can use JNDI to locate a printer on your corporate intranet. You can also use it to locate a Java object or to connect with a database. JNDI is used in EJB, RMI-IIOP, JDBC, and more. It is the standard way of looking things up over the network.

Naming and Directory Services

To understand JNDI, we must first understand the concept of naming and directory services.

A *naming service* is analogous to a telephone operator. When you want to call someone over the phone and you don't know that person's phone number, you can call your telephone company's information service operator to *look up* the person you want to talk with. You supply the telephone operator with the *name* of the person. The operator then looks up the phone number of the person you want to speak with and can dial the number for you, connecting you to that person.

A naming service is an entity that performs the following tasks.

- It associates names with objects. We call this *binding* names to objects. This is similar to a telephone company's associating a person's name with a specific residence's telephone number.
- It provides a facility to find an object based on a name. We call this *looking up* or *searching* for an object. This is similar to a telephone operator finding a person's telephone number based on that person's name and connecting the two people.

Naming services are everywhere in computing. When you want to locate a machine on the network, the *Domain Name System (DNS)* is used to translate a machine name to an IP address. If you look up `wiley.com` on the Internet, the name `wiley.com` is translated into the object (which happens to be a String) `199.171.201.14` by the DNS.

Another example of naming occurs in file systems. When you access a file on your hard disk, you supply a name for the file such as `c:\autoexec.bat` or `/etc/fstab`. How is this name translated into an actual file of data? A file system naming service can be consulted to provide this functionality.

In general, a naming service can be used to find any kind of generic object, like a file handle on your hard drive or a printer located across the network. But one type of object is of particular importance: a *directory object* (or *directory entry*). A directory object differs from a generic object because you can store *attributes* with directory objects. These attributes can be used for a wide variety of purposes.

For example, you can use a directory object to represent a user in your company. You can store information about that user, like the user's password, as attributes in the directory object. If you have an application that requires authentication, you can store a user's login name and password in directory

object attributes. When a client connects to your application, the client supplies a login name and password, which you can compare with the login name and password that are stored as a directory object's attributes. If the data matches, the user is authenticated. If the data doesn't match, your application can return an error. You can store other attributes, too, besides a login name and password, including a user's email address, phone number, and postal address.

A *directory service* is a naming service that has been extended and enhanced to provide directory object operations for manipulating attributes. A directory is a system of directory objects that are all connected. Some examples of directory products are Netscape Directory Server and Microsoft's Active Directory. Your company probably uses a directory to store internal company information (locations of computers, current printer status, personnel data, and so on).

What does a directory look like internally? The directory's contents—the set of connected directory objects—usually forms a hierarchical tree-like structure. Why would you want a tree-like structure? A tree's form suggests the way a real-world company is organized. For example, the *root* (or top node) of your directory tree can represent your entire company. One branch off the root could represent people in the company, while another branch could represent network services. Each branch could have subtrees that decrease in granularity more and more, until you are at individual user objects, printer objects, machine objects, and the like. This is illustrated in Figure A.7.

All in all, directories are not very different from databases. A database can store arbitrary data, just like a directory. Databases provide query operations to look up items in a database, just like directories. You can think of a directory as a scaled-down, simplified database. In fact, most directories are implemented by a database behind the scenes.

Problems with Naming and Directories

There are many popular naming and directory products out today. Directory vendors differentiate their product lines by offering different types of services. Unfortunately, this leads to different naming and directory standards. And each directory standard has a different protocol for accessing the directory. For example, directories based on the *Lightweight Directory Access Protocol* (LDAP) are accessed differently than those based on the *Network Information System* (NIS) or Novell's *Network Directory System* (NDS).

This means that if you want to switch directory vendors, you need to rewrite all of your client code that accesses the directory. It also means you need to

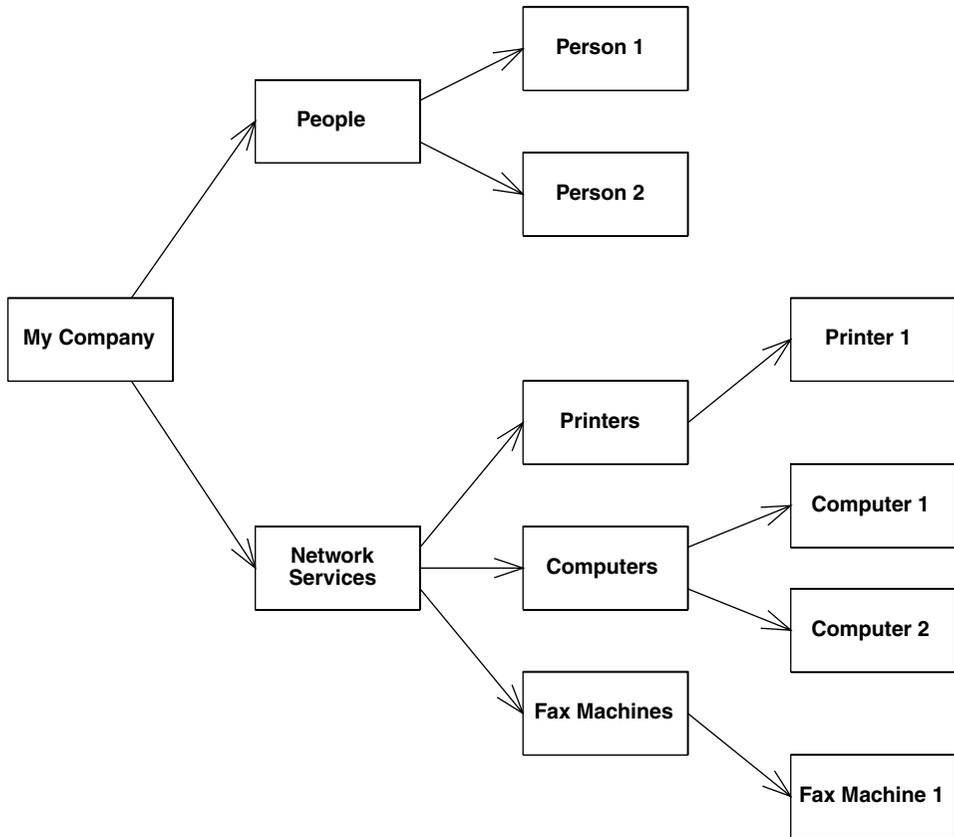


Figure A.7 A hierarchical directory structure.

download a new library, learn a new API, and test new code each time you use a different directory.

Initially, LDAP was meant to resolve this problem by becoming *the* ubiquitous protocol for directories. LDAP is straightforward and is being adopted quickly by the industry—IBM’s Lotus Notes and Microsoft’s Active Directory both are LDAP-based. However, not all directory products are LDAP-based.

Enter JNDI

JNDI is a system for Java-based clients to interact with naming and directory systems. JNDI is a *bridge* over naming and directory services, a beast that provides one common interface to disparate directories. Users who need to access an LDAP directory use the same API as users who want to access an NIS direc-

tory or Novell's directory. All directory operations are done through the JNDI interface, providing a common framework.

Benefits of JNDI

The following surveys the advantages that JNDI has to offer:

- You only need to learn a single API to access all sorts of directory service information, such as security credentials, phone numbers, electronic and postal mail addresses, application preferences, network addresses, machine configurations, and more.
- JNDI insulates the application from protocol and implementation details.
- You can use JNDI to read and write whole Java objects from directories.
- You can link different types of directories, such as an LDAP directory with an NDS directory, and have the combination appear to be one large, federated directory. The federated directory appears to the client to be one contiguous directory.

In J2EE, you can use JNDI for many purposes. These include:

- Using JNDI to acquire a reference to the *Java Transaction API (JTA) User-Transaction* interface
- Using JNDI to connect to resource factories, such as JDBC drivers or Java Message Service (JMS) drivers
- Using JNDI for beans to look up other beans

See Chapters 9 and 10 for examples of achieving these operations.

JNDI Architecture

JNDI is made up of two halves: *the client API* and the *Service Provider Interface (SPI)*. The client API allows Java code to perform directory operations. This API is uniform for all types of directories. You will spend the most time using the client API.

The JNDI SPI is an interface to which naming and directory service vendors can plug in. The SPI is the converse of the API: While the API allows clients to code to a single, unified interface, the SPI allows naming and directory service vendors to fit their particular proprietary protocols into the system, as shown in Figure A.8. This allows for client code to leverage proprietary naming and directory services in Java while maintaining a high level of code portability.

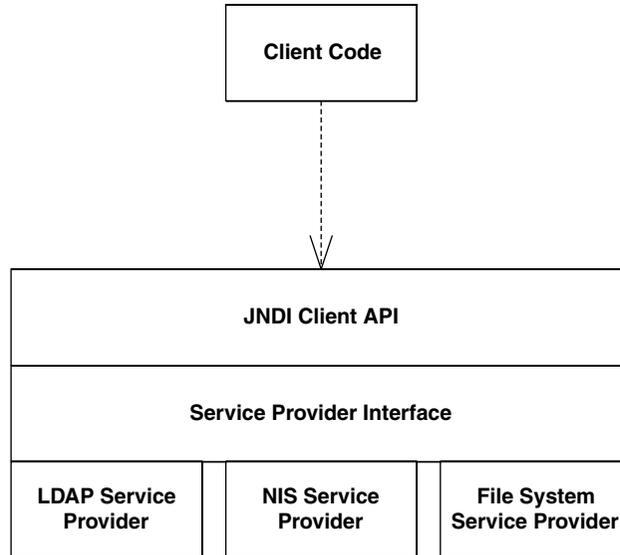


Figure A.8 JNDI architecture.

JNDI's architecture is somewhat like the Java Database Connectivity package (JDBC):

- In JDBC, one uniform client API performs database operations. In JNDI, naming and directory service clients invoke a unified API for performing naming and directory operations.
- In JDBC, relational database vendors provide JDBC drivers to access their particular databases. In JNDI, directory vendors provide *service providers* to access their specific directories. These providers are aware of specific directory protocols, and they plug in to the JNDI SPI.

For example, Sun Microsystems gives away an LDAP service provider for free. The LDAP service provider knows how to map a JNDI client API operation into an LDAP operation. It then executes the LDAP operation on an LDAP directory, using the specific LDAP protocol.

A number of JNDI service providers are available today, including LDAP, NIS, Novell NDS, SLP, CORBA, File System, RMI-IIOP, and many more. The JNDI homepage (<http://java.sun.com/products/jndi>) has a list of service providers.

J2EE servers *bundle* a JNDI implementation with their product. Typically this is a custom implementation provided by the J2EE server vendor. JNDI then just

becomes another service provided by the server, along with RMI-IIOP, JMS, and so on. Many servers ship JNDI implementations that are fault tolerant, providing a high level of availability. These JNDI implementations are intended to integrate with the other J2EE services, such as RMI-IIOP, JDBC, EJB, and JMS.

JNDI Concepts

We begin our JNDI exploration with naming concepts. There are several kinds of names in JNDI:

- An *atomic name* is a simple, basic, indivisible component of a name. For example, in the string */etc/fstab*, *etc* and *fstab* are atomic names.
- A *compound name* is zero or more atomic names put together. In the previous example, the entire string */etc/fstab* is a compound name.

A *binding* is an association of a name with an object. For example, the filename *autoexec.bat* in the Windows file system has a binding to the file data on your hard disk. Your *c:\windows* folder is a name that is bound to a folder on your hard drive. Note that a compound name such as */usr/people/ed/.cshrc* consists of multiple bindings, one to *usr*, one to *people*, one to *ed*, and one to *.cshrc*.

A context is an object that contains zero or more bindings. Each binding has a distinct atomic name. So for example, in the UNIX file system, let's consider a folder named */etc* that contains files named *mtab* and *exports*. In JNDI, the */etc* folder is a context containing bindings with atomic names *mtab* and *exports*. Each of the *mtab* and *exports* atomic names is bound to a file on the hard disk.

To expand this further, consider a folder named */usr* with subfolders */usr/people*, */usr/bin*, and */usr/local*. Here, the */usr* folder is a context that contains the *people*, *bin*, and *local* atomic names. Each of these atomic names is bound to a subfolder. In JNDI terms, these subfolders are called *subcontexts*. Each subcontext is a full-fledged context in its own right, and it can contain more name-object bindings, such as other files or other folders. Figure A.9 depicts the concepts we have learned so far.

Naming Systems, Namespaces, and Composite Names

A *naming system* is a connected set of contexts. For example, a branch of an LDAP tree could be considered a naming system, as could a folder tree in a file system. Unfortunately, naming systems each have a different syntax for accessing contexts. For example, in an LDAP tree, a compound name is

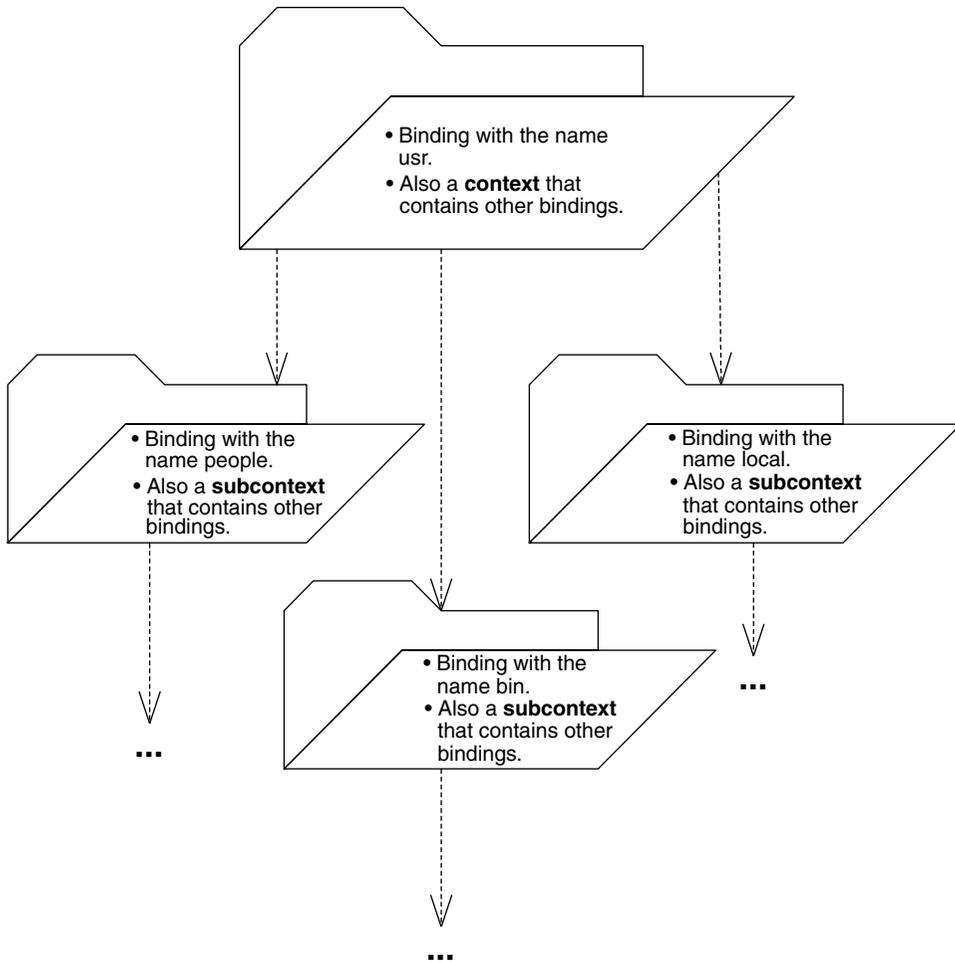


Figure A.9 JNDI naming concepts.

identified by a string such as `cn=Ed Roman, ou=People, o=Middleware-Company.com, c=us`, whereas a file system compound name might look like `c:\java\lib\tools.jar`.

A *namespace* is all the names contained within naming system. Your hard drive's entire collection of filenames and directories or folders is your hard drive file system's namespace. The set of all names in an LDAP directory's tree is an LDAP server's namespace. Naming systems and namespaces are shown in Figure A.10. This branch of a hard disk is an example of a naming system

because it's a connected set of contexts. Within this naming system, the namespace is every name shown.

A *composite name* is a name that spans multiple naming systems. For example, on the Web, the URL `http://java.sun.com/products/ejb/index.html` is composed of the following namespaces:

- `http` comes from the *URL scheme-id* namespace. You can use other scheme-ids, such as `ftp` and `telnet`. This namespace defines the protocol you use to communicate.
- `java.sun.com` uses the DNS to translate machine names into IP addresses.
- `products` and `ejb` and `index.html` are from the file system namespace on the Web server machine.

By linking multiple naming systems like the preceding URL, we can arrive at a unified *composite namespace* (also called a *federated namespace*) containing all the bindings of each naming system.

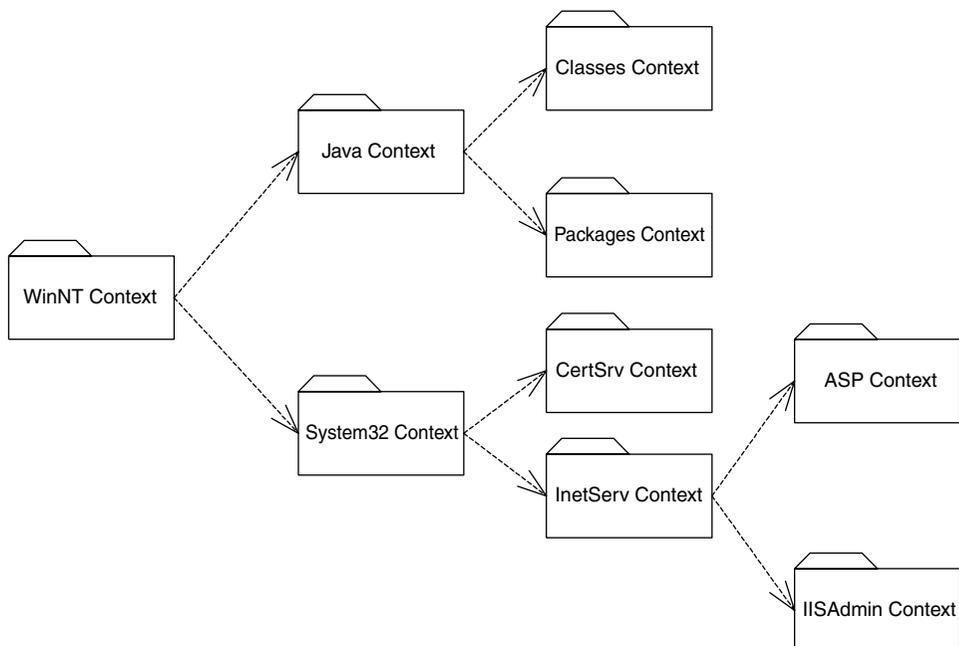


Figure A.10 Naming systems and namespaces.

Initial Context Factories

Students in our training classes commonly ask, If you are to traverse a composite namespace, how do you know which naming system to look into first? For example, which namespace do you first look in when traversing the string `http://www.TheServerSide.com/events/index.jsp`?

The starting point of exploring a namespace is called an *initial context*. An initial context simply is the first context you happen to use. An initial context is a starting point for performing all naming and directory operations.

To acquire an initial context, you use an *initial context factory*. An initial context factory is responsible for churning out initial contexts. An initial context factory basically *is* your JNDI driver. For example, there is an LDAP initial context factory, as well as a file system initial context factory. These initial context factories know the specific semantics of a particular directory structure. They know how to acquire an arbitrary context that you can use as an initial starting context for traversing a directory structure.

Initial context factories are used for bootstrapping, or jump-starting, your naming and directory service operations. You use an initial context factory as a bootstrapping mechanism for identifying an initial naming system.

When you acquire an initial context, you must supply the necessary information for JNDI to acquire that initial context. For example, if you're trying to access a JNDI implementation that runs within a J2EE server, you might supply:

- The IP address of the J2EE server
- The port number that the J2EE server accepts
- The starting location within the JNDI tree
- Any username/password necessary to use the J2EE server

You could use this same paradigm to access an LDAP server—just substitute *LDAP server* for *J2EE server* in the preceding list.

Initial contexts and composite namespaces are illustrated in Figure A.11.



A quick performance tip for you: some J2EE servers take a long time to create an initial context. If this is the case, we strongly recommend caching an initial context for future use. Create it once, and then reuse it when needed later.

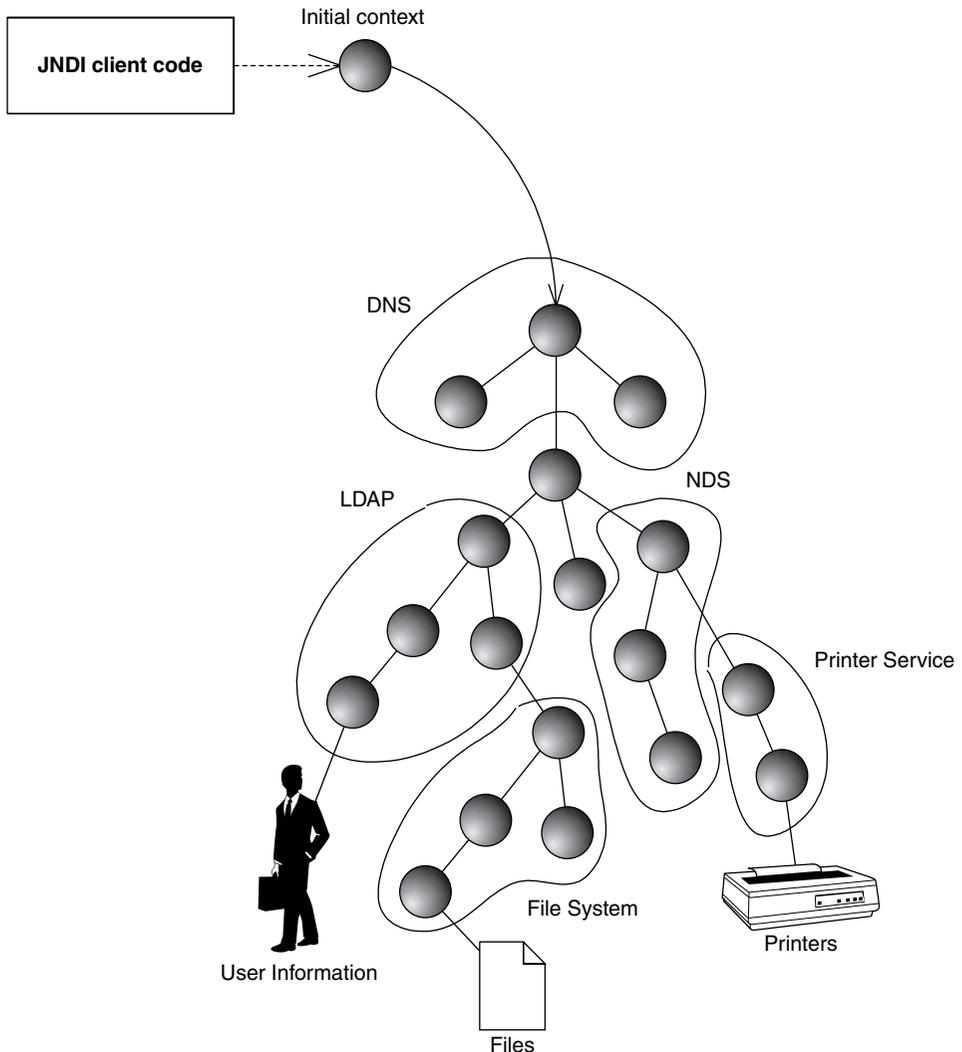


Figure A.11 A composite (federated) namespace with an initial context.

Programming with JNDI

Now that you've seen the concepts behind JNDI, let's put our theory into concrete use. Source A.3 shows a simple JNDI example.

The code simply acquires an initial context, and then the program completes. The specific JNDI driver that we use is based on the system properties passed in at the command-line. For example, to connect to your file system, you

```
package examples;

public class InitCtx {

    public static void main(String args[]) throws Exception {
        // Form an Initial Context
        javax.naming.Context ctx =
            new javax.naming.InitialContext(System.getProperties());
        System.err.println("Success!");
    }
}
```

Source A.3 InitCtx.java.

would use Sun's *file system* JNDI service provider, which is a driver that connects you to your own hard disk to browse the file system. You would then run the program as follows:

```
java
-Djava.naming.factory.initial=
    com.sun.jndi.fscontext.RefFSContextFactory
-Djava.naming.provider.url=
    file:c:\
examples.InitCtx
```

The *java.naming.factory.initial* parameter identifies the class of the JNDI driver. Then we identify the starting point on the file system that we want to begin navigating; specifically, the *c:* folder. This starting point is structured in the form of a Uniform Resource Locator (URL). In JNDI, it is called the *provider URL* because it is a URL that the service provider accepts for bootstrapping.

We can reuse this same code to connect to an LDAP server as follows:

```
java
-Djava.naming.factory.initial=com.sun.jndi.ldap.LdapCtxFactory
-Djava.naming.provider.url="ldap://ldap.funet.fi:389/c=fi"
examples.InitCtx
```

As you can see, this data-driven mechanism of performing JNDI has its advantages. It allows you to avoid recompiling your Java code. It also enables non-Java-savvy users to customize the behavior of your programs without modifying source code, which is important if you ship your products only as *.class* files.

Other JNDI Operations

After acquiring the initial context, you could begin to execute JNDI operations, such as reading or writing data to and from the JNDI tree by using the other API calls available in JNDI. Here is a brief list of available operations that you can call on the context:

list() retrieves a list of contents available at the current context. This typically includes names of objects bound to the JNDI tree, as well as subcontexts. In a file system, this might be a list of file names and folder names. If you're connecting to a proprietary J2EE server's JNDI implementation, you might see a list of bound objects as well as subcontexts to which you can navigate.

lookup() moves from one context to another context, such as going from `c:\` to `c:\windows`. You can also use `lookup()` to look up objects bound to the JNDI tree. The return type of `lookup()` is JNDI driver specific. For example, if you're looking up RMI-IIOP objects, you would receive a *java.rmi.Remote* object; if you're looking up a file in a file system, you would receive a *java.io.File*.

rename() gives a context a new name, such as renaming `c:\temp` to `c:\tmp`.

createSubcontext() creates a subcontext from the current context, such as creating `c:\foo\bar` from the folder `c:\foo`.

destroySubcontext() destroys a subcontext from the current context, such as destroying `c:\foo\bar` from the folder `c:\foo`.

bind() writes something to the JNDI tree at the current context. As with *lookup()*, JNDI drivers accept different parameters to *bind()*.

rebind() is the same operation as `bind`, except it forces a bind even if there is already something in the JNDI tree with the same name.

Integrating RMI-IIOP and JNDI

Now that you've seen both RMI-IIOP and JNDI, let's see how to combine them and complete our RMI-IIOP example. There are essentially two uses of JNDI with RMI-IIOP:

- An RMI-IIOP server first publishes itself to a JNDI tree using the JNDI API.
- A client then uses JNDI to lookup an RMI-IIOP server.

This process is shown in Figure A.12.

Your JNDI tree implementation is typically bundled with the J2EE server runtime. Therefore, when you startup your J2EE server, the JNDI tree runs in-process to the J2EE server and starts up as well. J2EE servers also ship with a JNDI driver that can connect to that JNDI tree implementation, which clients call.

Binding an RMI-IIOP Server to JNDI

The source code for binding the RMI-IIOP server to the JNDI tree is in Source A.4

The Startup class instantiates a remote object, acquires an initial context, binds the remote object to the context, and then waits for a client to call. It assumes that your J2EE server's JNDI implementation is already up and running. Note that you must supply your J2EE server's JNDI driver initialization parameters

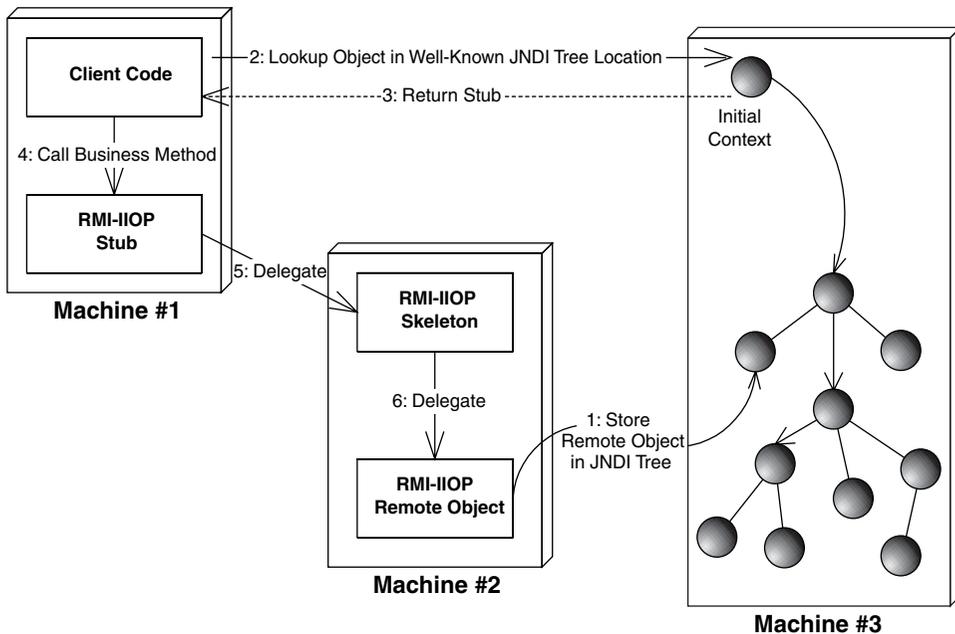


Figure A.12 Bootstrapping with JNDI.

```
import javax.naming.*;

/**
 * A helper class which starts our RMI-IIOP server
 */
public class Startup {

    /**
     * Our main() method starts things up
     */
    public static void main(String args[]) throws Exception {

        /**
         * Start up our PKGenerator remote object. It will
         * automatically export itself.
         */
        PKGenerator generator = new PKGenerator();

        /**
         * Bind our PKGenerator remote object to the JNDI tree
         */
        Context ctx = new InitialContext(System.getProperties());
        ctx.rebind("PKGenerator", generator);
        System.out.println("PKGenerator bound to JNDI tree.");

        synchronized (generator) {
            generator.wait();
        }
    }
}
```

Source A.4 Startup.java.

via the command line, as we showed earlier in this chapter when we ran our JNDI initial context example. Check your server's documentation or see the book's accompanying source code for this.

Looking up an RMI-IIOP Server with JNDI

Our client code that looks up the RMI-IIOP server via JNDI is in Source A.5.

Our client code is self-explanatory, with one exception. After looking up our remote object, we do a very strange operation: *javax.rmi.PortableRemoteObject.narrow()*. This is a static method on an RMI-IIOP class called *Portable*

```
import javax.naming.*;
import java.rmi.*;

public class Client {

    public static void main (String[] args) throws Exception {

        // Lookup the remote object via JNDI
        Context ctx = new InitialContext(System.getProperties());
        Object remoteObject = ctx.lookup("PKGenerator");

        // Cast the remote object, RMI-IIOP style
        IPKGenerator generator = (IPKGenerator)
            javax.rmi.PortableRemoteObject.narrow(
                remoteObject, IPKGenerator.class);

        // Generate a PK by calling the RMI-IIOP stub
        System.err.println(generator.generate());
    }
}
```

Source A.5 Client.java.

RemoteObject. This method casts the generic object that we looked up via JNDI to our RMI-IIOP interface type. This *narrow()* operation is required whenever you lookup an RMI-IIOP object via JNDI. Why do we need it, and why don't we just cast it using a regular Java cast? The short answer is CORBA interoperability. And if you're *really* curious, the long answer is in Appendix B.

As with the server, to run the client, you must supply your J2EE server's JNDI driver initialization parameters via the command line, as we showed earlier in this chapter when we ran our JNDI initial context example. Check your server's documentation or see the book's accompanying source code for this.

Summary

In this appendix, we've learned how Java RMI-IIOP and JNDI are fundamental underlying technologies in an EJB deployment. We looked at RMI-IIOP's architecture, comparing it to traditional RPCs. We examined stubs and skeletons, parameter passing, and object serialization. We concluded our RMI-IIOP discussion by introducing a sample primary key generator RMI-IIOP server.

Next, we delved into JNDI. We looked at the basics of naming and directory concepts, and saw how to acquire an initial context. We then investigated how to bind and lookup an RMI-IIOP object using JNDI.

In the next chapter, we'll delve into RMI-IIOP at a deeper level, by examining how it can be used for CORBA interoperability. This topic is important for anyone who has existing CORBA systems, or existing systems written in other languages that they'd like to bridge into their EJB system.

CORBA Interoperability

EJB would not be complete without a way to integrate with CORBA. CORBA allows EJB applications to communicate with existing CORBA applications, as well as to integrate with existing investments written in non-Java languages such as C++ and COBOL. Indeed, CORBA and EJB are related—many of the concepts in Java 2 Platform, Enterprise Edition came from CORBA.

In this appendix, we'll learn the high-level concepts behind CORBA. We'll then see how J2EE can integrate with CORBA via RMI-IIOP. Finally, we'll look at how to use CORBA clients to access EJB systems.

What Is CORBA?

The *Common Object Request Broker Architecture* (CORBA) is a unifying standard for writing distributed object systems. The standard is completely neutral with respect to platform, language, and vendor. CORBA incorporates a host of technologies and is very broad in scope.

The Object Management Group (OMG), a consortium of companies that began in 1989, invented CORBA. CORBA itself is simply a standard, just like EJB. CORBA-compliant products implement the CORBA specification, just as EJB-compliant servers implement the EJB specification.

In addition to CORBA, the OMG has defined a protocol called *Internet Inter-ORB Protocol* (IIOP, pronounced “eye-op”). IIOP is the standard Internet

protocol for CORBA. You never see IIOP; it is used behind the scenes for distributed object communications.

CORBA as the Basis for EJB

Many of the concepts in EJB came out of CORBA. In a sense, you can think of EJB as CORBA with a new hat. EJB and J2EE bring a Java-centric, component-based approach to traditional middleware programming—an architecture suitable for rapid application development. CORBA, on the other hand, offers a much broader suite of middleware features to work with. This includes a time service, a distributed locking service, a relationship service, and more. The primary advantage of EJB over CORBA is that EJB has more industry momentum; indeed, the very middleware vendors who offered CORBA implementations in the past are now focused on their EJB implementations.

Just like EJB, a group of companies jointly developed CORBA. This prevents CORBA from becoming a standard that's specific to one product or architecture (in the way that Microsoft.NET, for example, is specific to Windows).

Why Should I Care about CORBA?

Why would you want to use CORBA? There are several reasons.

You can use CORBA for legacy integration. If you have an existing investment (such as a legacy banking application) you can leverage that investment today using CORBA. For example, let's say you have a banking application written in C++. CORBA gives you the ability to preserve and reuse it. You can wrap your existing investment as a CORBA object, allowing it to be called from any application. As we'll find out, CORBA is a language-neutral standard and allows code written in several languages to communicate. Thus, CORBA is an ideal platform for code written in different languages to cooperate.

CORBA allows for advanced middleware development. Remember that EJB is not supposed to be an end-all to every problem. But if there is a middleware service that can be generalized, you're likely to find it standardized as a CORBA service. For those who need it, CORBA gives great functionality.

CORBA and EJB have hooks connecting them. Some EJB products will allow your enterprise beans to be called from two different kinds of clients: clients written to use the J2EE suite of APIs and clients written to use CORBA APIs. This means that code written in C++ or Smalltalk can call your enterprise beans.

Drawbacks of CORBA

As usual, the world isn't perfect. Using CORBA has disadvantages as well as advantages:

CORBA is slow-moving. All standards committees are bureaucratic and slow to make decisions. This is because the standards committee itself is not driven by revenues, but rather by individual interests from participating companies. CORBA experiences benefits from not being owned by one company, but its openness is also a drawback. The cycle time for the OMG to adopt a new CORBA feature is on the order of years.

CORBA has a steep learning curve. As CORBA has evolved over the years, it has undergone *feature creep*. More and more features have been added, which makes CORBA a robust standard but also increases the learning curve. Indeed, the specifications that define the whole of CORBA are thousands of pages long and are quite challenging to master. The nice thing about CORBA is that you don't have to learn it all to use it—you can learn optional CORBA *services* as you need them.

Products developed under CORBA may have incompatible features. It's great that CORBA is a unifying standard. Because no single company controls the standard, it levels the playing field for companies competing to build CORBA products. But there remain the problems of multivendor solutions. As with EJB products, if you mix and match CORBA products, you will inevitably run into assumptions that vendors have made but that are specific to their own products. This is the trade-off between a one-vendor solution, such as Microsoft, and an open standard, such as CORBA or EJB. The price of freedom is eternal vigilance.

Understanding How CORBA Works

Before we delve into CORBA/EJB interoperability, we'll cover the core CORBA fundamental concepts. This will lay the groundwork for us to discuss how CORBA and EJB are compatible.

Object Request Brokers

An *Object Request Broker* (ORB) facilitates network communication. ORBs enable disparate applications to communicate without being aware of the underlying communications mechanism. They are responsible for finding objects to service method calls, handling parameter passing, and returning results. Whenever you have multiple objects interacting in a CORBA environment, ORBs facilitate the communications. This is shown in Figure B.1.

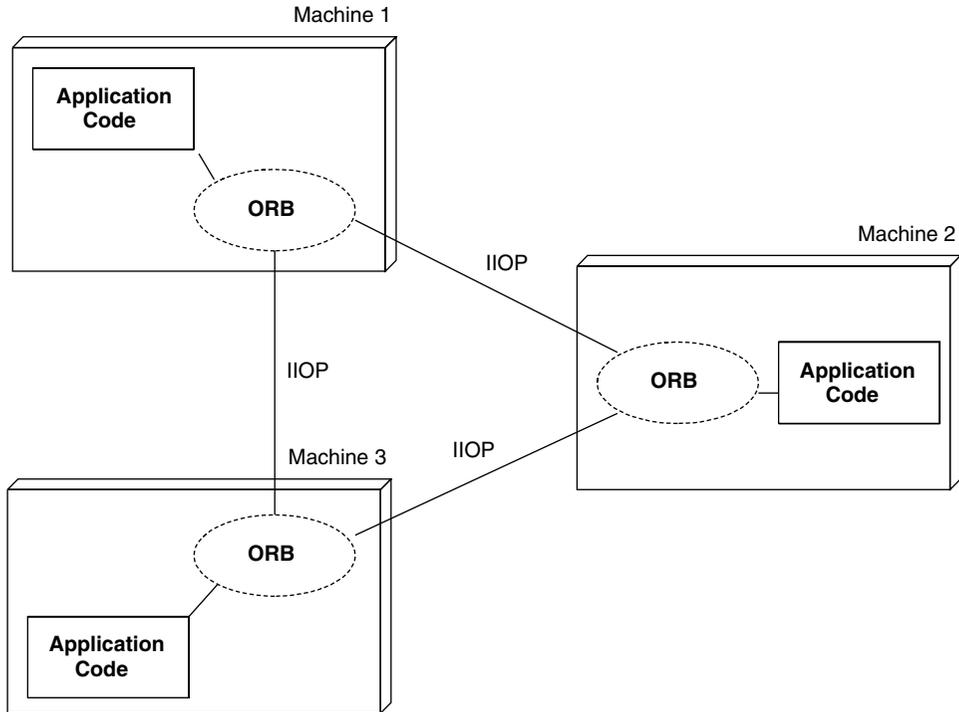


Figure B.1 The ORB facilitates your networking needs.

Numerous CORBA ORBs are on the market. Some examples are Iona's OrbixWeb, Inprise's VisiBroker, and IBM's ComponentBroker. Each vendor offers various qualities of service that differentiate that vendor's product from those of other vendors in the marketplace.



The concept of an ORB is absolutely not specific to CORBA. Both J2EE implementations and Microsoft.NET contain ORB functionality as well because both architectures facilitate network communications and hence, serve as object request brokers. For the rest of this chapter, however, we'll assume we're dealing with CORBA ORBs.

OMG's Interface Definition Language

The cornerstone of CORBA is the OMG's *interface definition language* (OMG IDL). OMG IDL is a language that CORBA uses to define the interfaces between clients and the objects they call. When you write a CORBA object implementation, that object implementation must have corresponding IDL

that defines the interface for that object implementation. By programming with OMG IDL, you force a clear distinction between interface and implementation—you can vary your implementation without changing the interface your clients use. The IDL concept is shown in Figure B.2.

Another great benefit to OMG IDL is that it is a *language-neutral* interface for object implementations. You can write your IDL once and then define your object implementations in any language that CORBA supports, such as C++ or Smalltalk. And because IDL is language-neutral, client code that calls your object implementations can be written in any language that CORBA supports

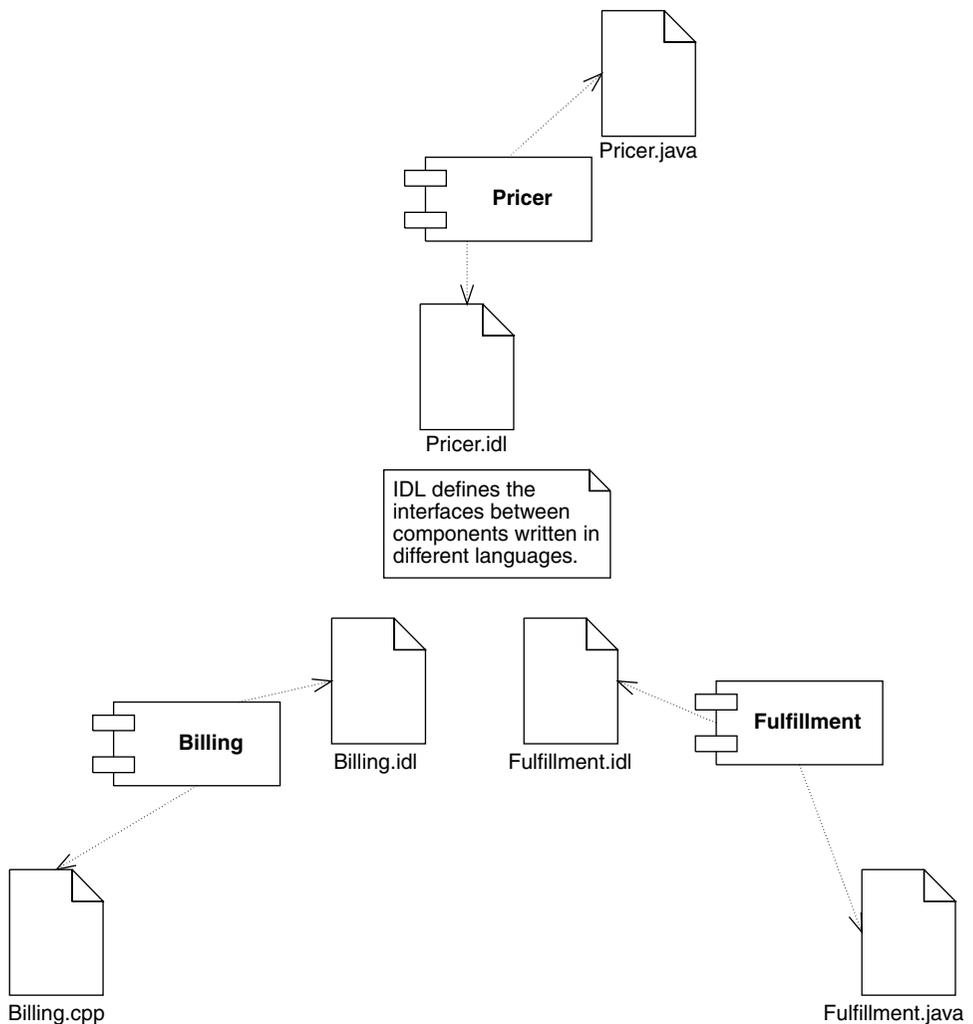


Figure B.2 The Interface Definition Language concept.

as well. Thus, IDL enables you to have a deployment mixing heterogeneous languages.

IDL is also inherently *platform-neutral*, allowing clients and object implementations to be deployed in different platforms. For example, your clients can exist on a Windows box and talk to business objects deployed on a Sun Solaris box.

You should think of IDL as a middleman language—a common ground that in theory is independent of language change. IDL allows you to write a distributed application with the illusion that it's all written in one language.

Here is a sample snippet of IDL:

```
module examples {  
  
    interface HelloWorld {  
        string sayHello(in string myName);  
    }  
  
}
```

As you can see, IDL is very similar to C++ and Java.

There are many different types in IDL, including basic types (such as *short* and *float*) and constructed types (such as *struct* and *enumeration*). You'll find that if you know C++, learning to use OMG IDL is pretty straightforward. If you're a Java programmer, you should not have too much difficulty using IDL to define your object's interfaces either, because Java's syntax is similar to C++.



We only briefly describe IDL in this chapter. Most CORBA books will have a section explaining IDL fully. And if you're serious about CORBA, take a look at the specifications on the OMG Web site (www.omg.org), which detail OMG IDL rigorously.

OMG IDL Maps to Concrete Languages

IDL is only a descriptive language in that it describes the interfaces to your objects. You cannot execute IDL. Neither your CORBA object implementations nor your CORBA clients ever see IDL. You program your clients and object implementations in whatever language you're using, such as Java or C++. But how, then, do you refer to CORBA objects? The answer is the OMG IDL *maps* to specific languages, such as Java or C++. If you go to the OMG Web site (www.omg.org), you'll see that there are specifications detailing how OMG IDL maps to various languages. For instance, there is an IDL-to-Java mapping specification that defines how IDL maps to Java. With the IDL-to-Java mapping, the *string* type in OMG IDL maps to the *java.lang.String* object in Java.

It is important to realize that, although IDL is a language, it is more of an abstraction because you never write client code or object implementations that use IDL files. Rather, you use IDL to define the interfaces to your objects and then *map* that IDL into your particular language using an *IDL compiler*. For example, an IDL-to-Java compiler would take as input an IDL file and generate Java interfaces for your object implementations. Once this is done, you can implement those interfaces in Java. You could then map the IDL to a different language, such as C++, by using an IDL-to-C++ compiler. This would allow you to write client code in C++ that calls your Java object implementations.



For the sake of brevity, we do not cover the IDL-to-Java mapping here. You can download the complete IDL-to-Java mapping specification for free from the OMG Web site.

CORBA Static Invocations

As we've said, the ORB facilitates client/server communications, simplifying client networking needs. But how does a client invoke a method on a remote CORBA object? The answer is via a local method call, which gets translated into a remote method call across the network. This is quite analogous to how networking is accomplished in Java RMI.

The conventional way to perform distributed computing in CORBA is to have the client invoke locally on a pregenerated *stub*. The stub is a proxy for the real object implementation, which exists elsewhere on the network. The stub is responsible for going through the client-side ORB runtime, which channels the request over the network via IIOP.

The receiving server-side ORB runtime receives the IIOP request, then calls a *skeleton* to handle the request. The server-side skeleton is a pregenerated file, just like the stub. The skeleton is responsible for delegating the invocation to the actual server-side CORBA object (also called a *servant*) that will service the request. The skeleton is also responsible for coordinating with an *object adapter*. This object adapter performs many tasks, such as mapping object references to servants, activating servants in case they don't exist already, housekeeping of threads, and more. Modern ORB implementations have object adapters coded to the CORBA *Portable Object Adapter* (POA) specification.

The CORBA invocation process is shown in Figure B.3.

Both the stub and skeleton are pregenerated files. They are usually generated from the IDL file that defines the server-side CORBA object's method signatures. They need to be pregenerated for two reasons.

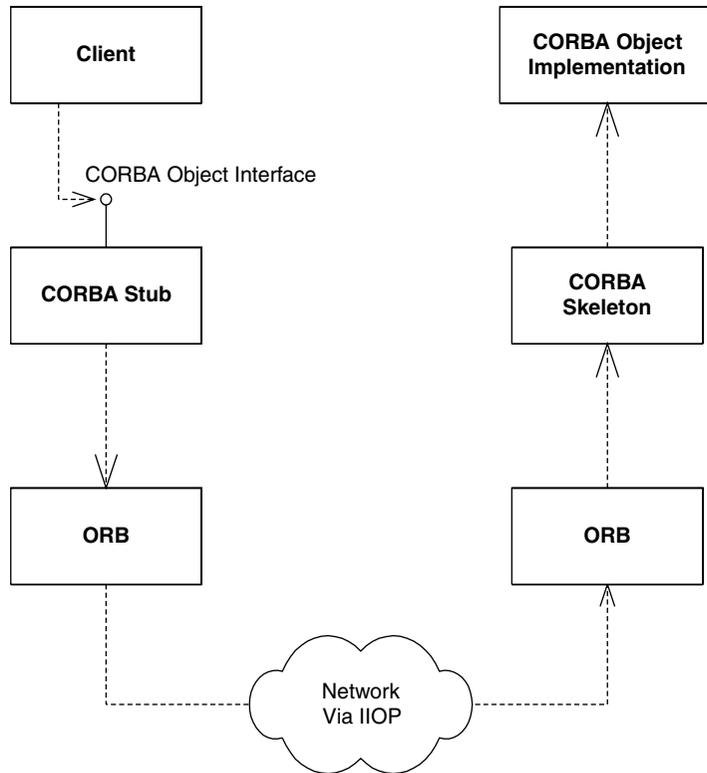


Figure B.3 Calling a method in CORBA.

CORBA objects are inherently cross-language. This means you need to pre-generate the stubs and skeletons in the particular language you're using. You're free to use any language to which IDL maps.

Stubs and skeletons contain specific syntax about your particular CORBA object's method signatures. Thus, you must generate them for each of your CORBA objects because each object will have different method signatures. By generating them, you can simulate an environment where clients can invoke on proxies using the actual method signatures of the real object, located elsewhere.

This invocation mechanism is called a *static invocation*, because you're statically binding your client code to stubs at compile time.

CORBA's Many Services

In addition to enabling objects to communicate over the network, the OMG has published a set of CORBA Object Services (known as CORBA Services [COS]) that give your networked objects additional capabilities. These services are optionally provided by CORBA vendors. Most serious CORBA products give you one or more services to aid development. These include:

- The *CORBA Naming Service* (COS Naming) is a CORBA service that allows you to look up CORBA objects by name, a technology similar to the Java Naming and Directory Interface (JNDI).
- The *CORBA Event Service* allows for asynchronous communications between CORBA objects.
- The *CORBA Object Transaction Service* (OTS) enables CORBA objects to perform transactions.
- The *Concurrency Control Service* allows for multiple clients to concurrently interact with a resource.
- The *CORBA Security Service* adds secure functionality to your CORBA system.

A final specification, called *CORBA Components*, adds component features to CORBA objects, allowing them to function similarly to enterprise beans. This means that CORBA now has a proposal that allows for true components to be developed in the CORBA world. CORBA Components is very similar to Enterprise JavaBeans. This was done intentionally so that CORBA Components and enterprise beans can reside together. One goal of CORBA Components is to integrate with enterprise beans. Thus, it should be possible to do either of the following:

- Make a CORBA Component appear as though it were an enterprise bean.
- Make an enterprise bean appear as though it were a CORBA Component.

For now, there is very little industry momentum for CORBA Components. This will definitely be an interesting standard to keep an eye on as EJB and CORBA evolve.

The Need for RMI-IIOP

Now that you've seen the basics of CORBA theory, let's compare Java RMI to CORBA. We'll first see why people use RMI and CORBA. Next, we'll look at the semantic differences that must be overcome to merge CORBA and RMI.

Finally, we'll look at how the industry merged RMI and CORBA with RMI-IIOP. This standard is the key to EJB-CORBA compatibility.

The Need for RMI-CORBA Interoperability

RMI and CORBA are very similar technologies with slightly different goals. One technology is not better than the other—it all depends on what kind of development you're doing.

CORBA is a robust distributed object standard that allows for language interoperability. RMI, on the other hand, was built for very simple distributed object communications in Java. While RMI does not contain CORBA's cross-language support, it is well suited for pure Java development due to Java-specific features such as distributed garbage collection, object activation, and downloadable class files.

While both RMI and CORBA are intended for distributed object communications, neither technology contains high-end middleware services, such as persistence or transactions. CORBA programmers can gain those middleware services by leveraging CORBA's optional services that we described earlier. RMI programmers can gain those middleware services by leveraging the Java 2 Platform, Enterprise Edition suite.

Unfortunately, although RMI and CORBA are similar in nature, they historically have been incompatible technologies. When you program code with Java RMI, you need to write your code to the RMI API. Similarly, when you program code with CORBA, you need to write your code to the CORBA API. This is terrible for code reuse: If you write code in either RMI or CORBA, you'll need to rewrite major pieces of your code if you want to switch networking technologies.

Ideally, we'd like a world where we could perform the following:

Combine client-side Java RMI with server-side CORBA. We should be able to write an object implementation to the CORBA API and write client code to the Java RMI API that calls that CORBA object. This is shown in Figure B.4.

Combine client-side CORBA with server-side Java RMI. We should be able to write a remote object implementation with the RMI API and have a client written to the CORBA API call that object. This is shown in Figure B.5.

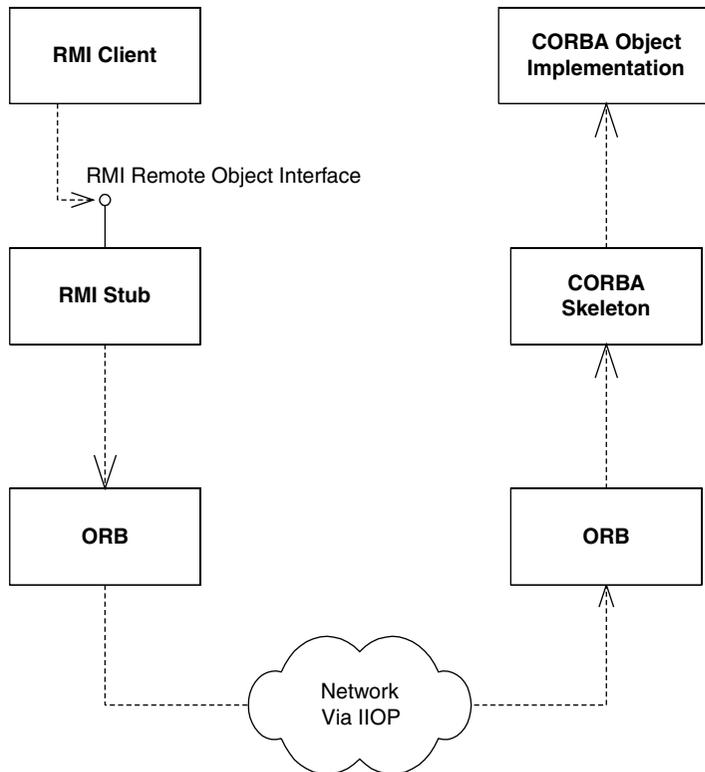


Figure B.4 An RMI client calling a CORBA object implementation.

Combining RMI with CORBA

What strategy should we use to combine the CORBA world with the Java RMI world? To begin to answer this, let's compare how CORBA and RMI work behind the scenes:

- Both CORBA (except in its dynamic communications mechanism) and RMI use pregenerated stubs and skeletons to perform network communications.
- Behind the scenes, CORBA uses IIOP as the protocol to perform client/server communications. This occurs beneath the stub/skeleton layer.
- Behind the scenes, Java RMI uses the Java Remote Method Protocol (JRMP) protocol for performing client/server communications. This occurs beneath the stub/skeleton layer as well.

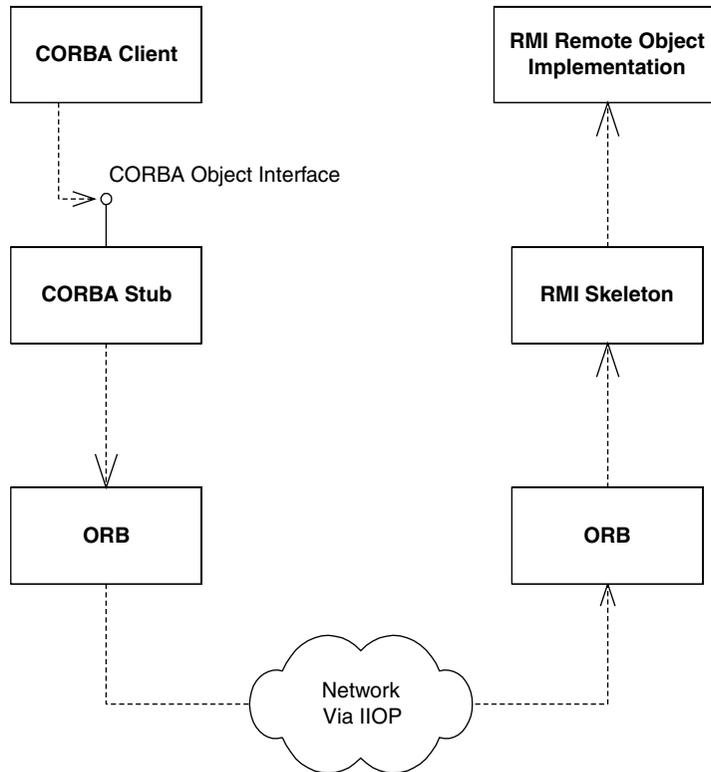


Figure B.5 A CORBA client calling an RMI remote object implementation.

The protocol being used is the key to interoperability of CORBA and RMI. RMI skeletons always expect a request to come in via the JRMP protocol, and CORBA skeletons are always expecting data to come in using the IIOP protocol. But this protocol layer should be totally pluggable. For example, we should be able to switch out RMI's JRMP protocol and switch in the IIOP protocol. If we did this, we could achieve Figures B.4 and B.5.

So why is IIOP our protocol of choice, rather than JRMP? The reason is that IIOP is a much more robust protocol than JRMP. IIOP is supported by numerous vendors in the industry and has been designed with interoperability of heterogeneous distributed objects in mind.

The scheme we've just presented is the basis for the unification of CORBA and RMI, and it is called RMI-IIOP (pronounced RMI over IIOP). RMI-IIOP allows for CORBA clients, RMI clients, CORBA object implementations, and RMI object implementations to be mixed and matched. This accomplishes our goal

Table B.1 Combinations Possible Using RMI-IIOP

CLIENT	SERVER
RMI-IIOP client	RMI-IIOP server
RMI-IIOP client	CORBA server
CORBA client	RMI-IIOP server
CORBA client	CORBA server

of creating a bridge between RMI and CORBA. Table B.1 shows the RMI-IIOP combinations that are possible.

Resolving the Differences between RMI and CORBA

Combining Java RMI with CORBA was not a straightforward task. There were incompatibilities. For example, in RMI you can pass parameters by-value using serialization (see Appendix A). There was no way in CORBA to marshal parameters by-value. Thus, a new CORBA specification called *Objects-by-Value* was developed to address this. The CORBA ORB you use with RMI-IIOP must implement this specification if you wish to marshal objects by value. The curious reader can download the complete *Objects-by-Value* specification free from the OMG Web site, www.omg.org.

Parameter passing conventions were not the only differences between RMI and CORBA. There are other semantic differences as well. Let's take a look at the major concerns:

Distributed garbage collection. RMI gives you an automatic way of cleaning up objects over the network with a distributed garbage collector. CORBA, on the other hand, has no such mechanism. Why? Because not every language that CORBA maps to has the concept of even regular in-process garbage collection.

Narrowing. When you receive an object using Java RMI, you can simply cast it into the desired object using a Java cast. This is possible because RMI automatically downloads the appropriate stub for the object you're dealing with. CORBA, however, does not have a mechanism for automatic stub downloading.

Java RMI programmers don't want to learn OMG IDL. One of the niceties of Java RMI is that it's all Java, which means you don't need to learn a separate interface definition language (such as OMG IDL) to handle your networking needs. But with RMI-IIOP, you can mix CORBA clients with RMI

server object implementations. Those CORBA clients are pure CORBA clients (with pure CORBA stubs), and they need to work with some IDL. That IDL needs to come from somewhere. Should we force Java RMI programmers to churn out an IDL file? If we make Java RMI coders learn OMG IDL, a large benefit of RMI has been lost.

A separate OMG specification called the *Java-to-IDL Mapping* specification resolves the semantic differences between RMI and CORBA. This document details all of the subtleties of combining the RMI API with the IIOP protocol. It addresses issues such as distributed garbage collection and inheritance, as well as the resolution of the differences between RMI and CORBA. In essence, the *Java-to-IDL Mapping* document is the complete specification for RMI-IIOP.

Let's take a look at how *Java-to-IDL* solves some of the semantic differences between RMI and CORBA.

Distributed garbage collection (DGC). RMI-IIOP does not propose to accomplish distributed garbage collection. And rightfully so—DGC is in general a hard problem to solve. Instead, the *Java-to-IDL* specification mandates that RMI coders cannot rely on distributed garbage collection when using RMI-IIOP.

Narrowing. When using RMI-IIOP, you cannot simply cast an object you receive over the network, because the stub class file may not exist on the client machine. RMI-IIOP does not allow downloadable class files because CORBA does not mandate support for this feature. Thus, you must perform a *safe cast* by explicitly calling the `javax.rmi.PortableRemoteObject.narrow()` method. This static method converts an object into the desired remote interface type for you. You pass `narrow()` the object you desire to cast, and the class to which you wish to cast it. The `narrow()` method returns the resultant object or throws an exception if the narrow is not possible, perhaps because the class does not exist.

Java RMI programmers don't want to learn OMG IDL. One great benefit of Java RMI is that you don't need to learn a separate interface definition language to perform remote computing. We'd like to preserve this feature. RMI-IIOP therefore defines a mapping from RMI/IDL types to OMG IDL types. This mapping provides a well-defined way for Java language types used by RMI-IIOP to be automatically mapped into OMG IDL. Once we have this, a vendor can write a tool that automatically performs this mapping. Such a tool is called a *java-to-idl compiler*. It takes in code written in Java and spits out OMG IDL. This IDL can be used by CORBA clients when calling your RMI remote object implementations. The IDL can also be used by CORBA object implementations that your RMI clients call.

Java-to-IDL allows you to build complete distributed applications in Java and then use apps written in other languages to invoke on your distributed application. The Java-to-IDL Mapping simplifies your network programming tremendously. No longer do you have to write IDL and then

The IIOP Debate

The IIOP standardization solution that the J2EE and CORBA communities reached was not the only possible solution. In fact, this solution was heavily debated and highly political. Here are the dirty details.

The problem here is that originally, not all J2EE server vendors based their servers on CORBA. BEA WebLogic, for example, built its server from scratch with a custom RMI implementation, as did many other server vendors. These vendors did not want to standardize on IIOP, because that meant they had to rewrite their server to use CORBA instead. This added up to increased development time and loss of control over what happened at the network level, which could introduce performance or functionality constraints.

To understand a possible alternative solution, you must understand the concept of context propagation. Let's say your code is involved in a transaction or has security credentials associated with it. Most systems (including EJB) manifest transaction and security data in *contexts*. These contexts are invisible to you and are passed along transparently. Typically they are kept in *Thread Local Storage*, which is a pocket of memory associated with each thread. When you perform an invocation, you want this context information to be *propagated* automatically to the receiving system so the object implementation can execute its business logic for you within the same contexts. RMI-IIOP standardizes the way contexts are propagated, which achieves interoperability.

But there is another solution as well. Imagine if you have two EJB servers talking to one another—say, Weblogic calling WebSphere. Weblogic is invoking on a Websphere stub (not Weblogic stub) that has been deployed on the Weblogic machine. That Websphere stub understands how to talk to a Websphere skeleton. Therefore it should not be necessary to standardize on a protocol such as IIOP. Rather, there should be a standard mechanism for a stub to retrieve transaction and security contexts from the EJB server's thread local storage.

In the end, a compromise was reached. J2EE server vendors are allowed to use other protocols besides IIOP. However, they need to *support* IIOP in case interoperability is required with CORBA systems. That is the irony of the phrase RMI-IIOP—it does not necessarily mean usage of the IIOP protocol. Rather, it means standardizing on the *PortableRemoteObject.narrow()* method.

translate that into Java. Java-to-IDL compilers allow you to write your Java app as you normally would, yet they allow for CORBA interoperability by generating IDL for you. This is a great convenience—Java RMI programmers gain the benefits of CORBA/IIOP interoperability, such as cross-language support, at a very low cost.

Steps to Take for RMI and CORBA to Work Together: An Overview

Now that you've seen the theory of combining RMI with CORBA, let's see exactly what steps you need to take for interoperability.

RMI-IIOP Client with a CORBA Object Implementation

Our first scenario depicts an RMI-IIOP client with a CORBA object implementation. To develop such a system, perform the following steps.

1. **Write your RMI-IIOP remote interface.** You write the remote interface in Java. The remote interface is RMI-IIOP's client/server contract for distributed objects.
2. **Generate the needed client-side RMI-IIOP stubs.** The RMI-IIOP client will use the stubs to invoke on the CORBA object implementation. You can generate these stubs using your J2EE server's RMI-IIOP implementation.
3. **Generate the OMG IDL.** When you define your CORBA object implementations, you're going to need IDL. This IDL must match your RMI-IIOP remote interface if you want RMI-IIOP clients to be able to invoke your CORBA object implementations. Rather than laboriously writing it yourself, you can automatically generate it through a Java-to-IDL compiler. The Java-to-IDL compiler takes in your RMI-IIOP remote interface and spits out OMG IDL. Where do you get a Java-to-IDL compiler? It typically ships with your container's tools. For example, some J2EE servers have a flag on their RMI-IIOP compiler (`rmic`) to spit out IDL.
4. **Generate the needed server-side CORBA files.** You're going to need some helper code, such as skeletons for your CORBA object implementations. And remember that this helper code can be in any CORBA-compliant language in which you choose to implement your CORBA object implementations. This is where the IDL you generated in step 3 comes into play. When you define your CORBA object implementations, you can

use any language to which IDL maps. You then use an IDL compiler to take in your IDL and produce network management code in the language in which you're implementing your objects. For example, if you use Java, you'll need an IDL-to-Java compiler. Most major ORB vendors that support Java include an IDL-to-Java tool with their products, and J2EE servers that bundle ORBs should do so as well.

5. **Write the client and the server.** You can now write your RMI-IIOP client and your CORBA object implementations.

CORBA Client with an RMI-IIOP Object Implementation

The second scenario depicts a CORBA client with an RMI-IIOP object implementation. To achieve this, you perform the following steps:

1. **Write your RMI-IIOP remote interface.** You write the remote interface in Java. The remote interface is RMI-IIOP's client/server contract for distributed objects.
2. **Generate the needed server-side RMI-IIOP skeletons.** The skeletons will be used to receive invocations and delegate them to your RMI-IIOP remote object implementations. You can generate these skeletons using your J2EE server's RMI-IIOP implementation.
3. **Generate the OMG IDL.** When you define your CORBA clients, you're going to need IDL. This IDL must match your RMI remote interface if you want CORBA clients to call your RMI-IIOP object implementations. Rather than laboriously writing it yourself, you can automatically generate it through a Java-to-IDL compiler. The Java-to-IDL compiler takes in your RMI-IIOP remote interface and spits out OMG IDL, and ships with your ORB or J2EE server.
4. **Generate the needed client-side CORBA files.** As in the previous section, you need to generate helper code, such as stubs for your CORBA clients. Thus, you need to generate these network plumbing classes from the IDL with an IDL compiler, such as an IDL-to-Java compiler that ships with your ORB or J2EE server.
5. **Write the client and the server.** You can now write your CORBA client and your RMI-IIOP object implementations.



As you can see, mixing and matching RMI-IIOP and CORBA is not trivial. Several tricky steps are necessary, and different J2EE servers may behave differently. Be prepared to experience a few headaches if you want to get RMI-IIOP and CORBA working together.

Bootstrapping with RMI-IIOP and CORBA

Recall from earlier in this chapter that CORBA has its own built-in naming service, called the CORBA Naming Service (or COS Naming). COS Naming is the standard way CORBA clients look up remote CORBA objects. But again, this is simply looking up an arbitrary resource over the network—the resource just happens to be CORBA objects rather than printers or RMI-IIOP objects. Therefore, COS Naming is a perfect fit for JNDI. You can lookup CORBA objects using JNDI by using a special CORBA-compatible JNDI driver. One such driver is the COS Naming service provider, downloadable for free from <http://java.sun.com>. Note that you should check your J2EE server's documentation for the specific driver they recommend.

What's great about this paradigm is that our RMI-IIOP client code can access both RMI-IIOP servers and CORBA servers without changing code, but rather by merely plugging in a different JNDI driver.

The Big Picture: CORBA and EJB Together

CORBA and EJB have an interesting relationship. They compete with one another in some respects (due to CORBA Components), and at the same time, they complement each other. This is because CORBA is often the enabling technology that resides beneath the EJB level. Many EJB server vendors layer their EJB products on top of an existing CORBA infrastructure, and RMI-IIOP allows just this to happen.

Although CORBA/EJB interoperability is still in its infancy, the vision is solid, and we hope several benefits will come from it. The biggest benefit is that CORBA clients written in any language (that OMG IDL maps to) should be able to call your enterprise beans.

Another benefit of CORBA/EJB interoperability is at the transaction and security level. Clients should be able to mix calls to both CORBA objects and enterprise beans under the hood of the same transaction. Similarly, you should be able to construct a distributed transaction that spans heterogeneous EJB servers. And finally, you should eventually be able to propagate security contexts from one EJB server to another, allowing for single sign-on between different EJB server vendors. The specifications are maturing slowly but surely to make this a reality in the future.

What You Don't Get from CORBA-EJB Interoperability

We want to make it clear that there is one benefit that you do *not* get from EJB-CORBA interoperability. CORBA-EJB interoperability is for connecting a CORBA *client* to an enterprise bean written in *Java*. You *cannot* write your enterprise beans in any language but Java. If you want to write your server-side components using another language, see Chapter 13 for legacy integration strategies.

Sample Code

Now let's write some sample code to illustrate how to use CORBA to call an EJB component. We'll use the Hello, World bean developed in Chapter 3. The key thing to notice is that we are taking a bean that we called using RMI-IIOP in Chapter 3, and are reusing the bean without modification and accessing it from a CORBA client.

We'll use the following to access our bean:

- COS Naming to look up the home object
- OTS to demarcate transaction boundaries
- The Java language to write our CORBA client

Source B.1 shows the implementation.

```
import java.util.*;
import org.omg.CosNaming.*;
import org.omg.CosTransactions.*;

public class CORBAclient {

    public static void main(String[] args) throws Exception {

        /*
         * Initialize the ORB.
         */
        Properties p = new Properties();
        p.put("org.omg.CORBA.ORBClass", <..Your ORB class..>);
        org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args, p);
```

Source B.1 Example CORBA EJB client.

```
    /*
     * Get a reference to a naming context
     */
    NamingContext context = NamingContextHelper.narrow
        (orb.resolve_initial_references("NameService"));

    /*
     * Look up the home object using COS Naming
     */
    NameComponent[] names = { new NameComponent("HelloHome", "") };
    HelloHome helloHome = HelloHomeHelper.narrow
        (context.resolve(names));

    /*
     * Get the CORBA OTS Current interface for
     * controlling transactions
     */
    Current currentTX = CurrentHelper.narrow
        (orb.resolve_initial_references("TransactionCurrent"));

    /*
     * Begin the transaction
     */
    currentTX.begin();

    /*
     * Use the home object to create an EJB object
     */
    Hello hello = helloHome.create();

    /*
     * Call a business method
     */
    System.out.println(hello.hello());

    /*
     * Remove the EJB object
     */
    hello.remove();

    /*
     * Commit the transaction
     */
    currentTX.commit(true);
}
}
```

Source B.1 Example CORBA EJB client (*continued*).

As you can see, CORBA clients are a bit more complex than RMI-IIOP clients. We first need to initialize the ORB before beginning any CORBA operations. Next we get a reference to a naming context via COS Naming, which we use to look up home objects. Once we've retrieved the home object, calling methods on enterprise beans is syntactically similar to RMI-IIOP. We also get a reference to the *OTS Current* interface, which is used to demarcate transactional boundaries, analogous to the Java Transaction API (JTA) described in Chapter 10. The *begin()* and *commit()* calls have the same semantic meaning as their JTA equivalents. See the book's accompanying source code for vendor-specific build scripts for this code.

Summary

In this chapter, you've experienced a whirlwind tour of CORBA and IIOP. We've displayed CORBA's advantages and the reasons that CORBA is a useful technology. We then delved into the inner workings of CORBA and explored its architecture. We also glanced at CORBA's services and touched on the IDL-to-Java mapping.

We then compared RMI to CORBA and reasoned why the two worlds need cohesion. We designed the requirements for RMI-IIOP interoperability and dived into several scenarios illustrating RMI and CORBA working in unison. We wrapped up our discussion of RMI-IIOP by illustrating the steps necessary for you to write RMI-IIOP code.

In the last section of this chapter, we caught a glimpse of the future—EJB and CORBA interoperability—and showed some example code.

Deployment Descriptor Reference

This appendix is a reference guide for building XML deployment descriptors. You do not need to read this appendix front-to-back; rather, you should use it as a reference guide when you have questions about deployment descriptors. This is a handy reference when programming deployment descriptors, because you can quickly lookup the structure in question.

How to Read a DTD

This appendix is a consolidated view of the EJB 2.0 XML deployment descriptor document type definition (DTD). A DTD is a schema for an XML document. It constrains how you can form your XML so that a computer program, such as an EJB compiler tool, can interpret the resulting document.

The tables in this appendix use the following syntax:

ELEMENT	DESCRIPTION
element?	A question mark (?) indicates this element is optional.
element*	An asterisk (*) indicates zero or more of these elements may exist.
element+	A plus sign (+) indicates one or more of these elements may exist.
elementA elementB	This means you can have either have elementA or elementB, but not both.
Element	No punctuation means there must be exactly one element.

Note that the ordering of elements in the tables below is important. Your deployment descriptor will not be valid unless you follow the exact ordering in the charts. For example, it would be invalid to define a local home interface before defining a home interface.

Elements are also case sensitive. Be sure you use the correct capitalization.

The Header and Root Element

All XML deployment descriptors should be named `ejb-jar.xml` and be located in the META-INF folder of your Ejb-jar file. The XML file is a flat text file that begins with the following declaration:

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

The *ejb-jar* element is the root element of all deployment descriptors.

ELEMENT	DESCRIPTION	WHERE TO GO FOR MORE INFORMATION
description?	A text description of this Ejb-jar file.	
display-name?	A short name of this Ejb-jar file, to be displayed by tools	
small-icon?	The relative path within the Ejb-jar file that you can find a 16 × 16 icon image (either JPEG or GIF) to be displayed by tools when manipulating this Ejb-jar file.	
large-icon?	Same as small icon, except a 32 × 32 image.	
enterprise-beans	Defines one or more enterprise beans.	See <i>session</i> , <i>entity</i> , or <i>message-driven</i> later in this appendix.
relationships?	Defines CMP relationships.	See <i>relationships</i> later in this appendix.
assembly-descriptor?	Defines application assembly information, such as transactions and security.	See <i>assembly-descriptor</i> later in this appendix.
ejb-client-jar?	Specifies an optional JAR file that contains classes that remote clients use to access beans, such as stubs and interfaces. Only useful if you have remote clients.	Chapter 3

Here is an example of *ejb-jar*:

```
<ejb-jar>
  <description>E-Commerce System</description>
  <display-name>E-Commerce EJB-JAR file</display-name>
  <small-icon>small.gif</small-icon>
  <large-icon>large.gif</large-icon>

  <enterprise-beans>
    ... One or more session, entity, and message-driven . . .
  </enterprise-beans>

  <relationships>
    ... Define relationships ...
  </relationships>

  <assembly-descriptor>
    ... Define application assembly information ...
  </assembly-descriptor>

  <ejb-client-jar>ECommerceClient.jar</ejb-client-jar>
</ejb-jar>
```

Defining Session Beans

The following is an example of how to set up a session bean. Descriptions follow.

```
<ejb-jar>
  <enterprise-beans>
    <session>
      <ejb-name>Count</ejb-name>
      <home>examples.CountHome</home>
      <remote>examples.Count</remote>
      <ejb-class>examples.CountBean</ejb-class>
      <session-type>Stateful</session-type>
      <transaction-type>Container</transaction-type>
    </session>
  </enterprise-beans>
</ejb-jar>
```

<session>

The *session* element defines a session bean. Used in *ejb-jar*.

ELEMENT	DESCRIPTION	WHERE TO GO FOR MORE INFORMATION
description?	A text description of this bean.	
display-name?	A short name of this bean, to be displayed by tools.	
small-icon?	The relative path within the Ejb-jar file that you can find a 16 × 16 icon image (either JPEG or GIF) representing this bean, to be displayed by tools.	
large-icon?	Same as small icon, except a 32 × 32 image.	
ejb-name	The nickname that you want to give this bean. This Ejb-name can be referenced later in the deployment descriptor.	Chapters 3, 4
home?	Home interface class. Remember to include the package too!	Chapters 3, 4
remote?	Remote interface class. Remember to include the package too!	Chapters 3, 4
local-home?	Local home interface class. Remember to include the package too!	Chapters 3, 4
local?	Local interface class. Remember to include the package too!	Chapters 3, 4
ejb-class	Session bean class. Remember to include the package too!	Chapters 3, 4
session-type	For a stateful session bean, set this to <i>Stateful</i> . For a stateless session bean, set this to <i>Stateless</i> .	Chapters 3, 4
transaction-type	For declarative transactions, set this to <i>Container</i> . For bean-managed transactions, set this to <i>Bean</i> .	Chapter 10
env-entry*	Declares environment properties for this bean.	Chapter 9
ejb-ref*	Declares references to other beans.	Chapter 9
ejb-local-ref*	Declares local references to other beans.	Chapter 9
security-role-ref*	Declares security role references.	Chapter 9
security-identity?	Declares how to perform security context propagation.	Chapter 9
resource-ref*	Declares resource factory references (such as JDBC driver used in bean).	Chapter 9
resource-env-ref*	Binds resource factories to JNDI nicknames.	Chapter 9

Note that you must define home/remote or local-home/local in pairs. For example, it would be invalid to define a home interface without a remote interface. You must also define at least one pair, meaning you must use either remote interfaces, local interfaces, or both.

Defining Entity Beans

The following is an example of how to set up an entity bean. Descriptions follow.

```
<ejb-jar>
  <enterprise-beans>
    <entity>
      <ejb-name>Product</ejb-name>
      <local-home>examples.ProductHome</local-home>
      <local>examples.Product</local>
      <ejb-class>examples.ProductBean</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>Product</abstract-schema-name>
      <cmp-field><field-name>productID</field-name></cmp-field>
      <cmp-field><field-name>name</field-name></cmp-field>
      <primkey-field>productID</primkey-field>
      <query>
        <query-method>
          <method-name>findByName</method-name>
          <method-params>
            <method-param>java.lang.String</method-param>
          </method-params>
        </query-method>
      <ejb-ql>
        <![CDATA[SELECT OBJECT(a) FROM Product AS a WHERE name 5 ?1]]>
      </ejb-ql>
      </query>
    </entity>
  </enterprise-beans>
</ejb-jar>
```

<entity>

The *entity* element defines an entity bean. Used in *ejb-jar*.

ELEMENT	DESCRIPTION	WHERE TO GO FOR MORE INFORMATION
description?	A text description of this bean.	
display-name?	A short name of this bean, to be displayed by tools.	
small-icon?	The relative path within the Ejb-jar file that you can find a 16 × 16 icon image (either JPEG or GIF) representing this bean, to be displayed by tools.	
large-icon?	Same as small icon, except a 32 × 32 image.	
ejb-name	The nickname that you want to give this bean. This Ejb-name can be referenced later in the deployment descriptor.	Chapters 6, 7
home?	Home interface class. Remember to include the package too! We don't recommend you use this, because entity beans should always be accessed via their local interfaces.	Chapters 6, 7
remote?	Remote interface class. Remember to include the package too! We don't recommend you use this, because entity beans should always be accessed via their local interfaces.	Chapters 6, 7
local-home?	Local home interface class. Remember to include the package too!	Chapters 6, 7
local?	Local interface class. Remember to include the package too!	Chapters 6, 7
ejb-class	Session bean class. Remember to include the package too! For a CMP entity bean, set this to <i>Container</i> . For a BMP entity bean, set this to <i>Bean</i> .	Chapters 6, 7 Chapters 6, 7
prim-key-class	Primary key class (if you have one). Remember to include the package too!	Chapters 6, 7
reentrant	Set to <i>True</i> or <i>False</i> depending on whether you want to be able to call yourself through another bean.	Chapters 6, 7
cmp-version?	Set to <i>1.x</i> or <i>2.x</i> depending on whether you're using the old EJB 1.1 style of entity bean programming, or the new EJB 2.0 style. The book's accompanying source code has EJB 1.1 and EJB 2.0 style examples.	Chapters 6, 7

continued

ELEMENT	DESCRIPTION	WHERE TO GO FOR MORE INFORMATION
abstract-schema-name?	Declares a nickname for this bean's CMP field definition. Used within EJB-QL queries.	Chapters 7, 11 Appendix D
cmp-field*	Defines a container-managed persistent field for a CMP entity bean.	Chapter 7
primkey-field?	If you're not using a primary key class, declares one of your CMP fields to be a primary key.	Chapter 7
env-entry*	Declares environment properties for this bean.	Chapter 9
ejb-ref*	Declares references to other beans.	Chapter 9
ejb-local-ref*	Declares local references to other beans.	Chapter 9
security-role-ref*	Declares security role references.	Chapter 9
security-identity?	Declares how to perform security context propagation.	Chapter 9
resource-ref*	Declares resource factory references (such as JDBC driver used in bean).	Chapter 9
resource-env-ref*	Binds resource factories to JNDI nicknames.	Chapter 9
query*	Defines a CMP EJB-QL query for a finder or select method.	Chapters 7, 11 Appendix D

<cmp-field>

The *cmp-field* element defines a CMP field within an entity bean definition. Used in *entity*.

ELEMENT	DESCRIPTION	WHERE TO GO FOR MORE INFORMATION
description?	A text description of this CMP field.	
field-name	The name of the get/set method corresponding to this CMP field. Leave off the get/set prefix, and make sure the first letter is lowercase.	Chapter 7

<query>

The *query* element defines an EJB-QL query for a finder or select method. Applies only to CMP entity beans. Used in *entity*.

ELEMENT	DESCRIPTION	WHERE TO GO FOR MORE INFORMATION
description?	A text description of this query	Chapter 7 Appendix D
query-method	The finder or select method that will be associated with this EJB-QL query	Chapter 7 Appendix D See later table.
result-type-mapping?	Maps the return results of the EJB-QL query to either remote interfaces (set to <i>Remote</i>) or local interfaces (set to <i>Local</i>). The default is <i>Local</i> .	Chapter 7 Appendix D
ejb-ql	The actual EJB-QL formatted string to query the storage	Chapter 7 Appendix D

<query-method>

The *query-method* element declares the finder or select method that will be associated with this EJB-QL query. Applies only to CMP entity beans. Used in *query*.

ELEMENT	DESCRIPTION	WHERE TO GO FOR MORE INFORMATION
method-name	The name of this method. Leave off the get/set and make sure the first letter is lowercase.	Chapter 7 Appendix D
method-params	A list of fully qualified Java types that are parameters to this method.	Chapter 7 Appendix D See later table.

<method-params>

The *method-params* element declares a list of fully qualified Java types that are parameters to this method. Applies only to CMP entity beans. Used in *query-method*.

ELEMENT	DESCRIPTION	WHERE TO GO FOR MORE INFORMATION
method-param*	Zero or more fully qualified Java types for method parameters that this query accepts.	Chapter 7 Appendix D

Defining Message-Driven Beans

The following is an example of how to set up a message-driven bean. Descriptions follow. See Chapter 8 for complete documentation.

```
<ejb-jar>
  <enterprise-beans>
    <message-driven>
      <ejb-name>Hello</ejb-name>
      <ejb-class>examples.HelloBean</ejb-class>
      <transaction-type>Container</transaction-type>
      <message-driven-destination>
        <destination-type>javax.jms.Topic</destination-type>
      </message-driven-destination>
    </message-driven>
  </enterprise-beans>
</ejb-jar>
```

<message-driven>

The *message-driven* element defines a message-driven bean. Used in *ejb-jar*.

ELEMENT	DESCRIPTION	WHERE TO GO FOR MORE INFORMATION
description?	A text description of this bean.	
display-name?	A short name of this bean, to be displayed by tools.	

continued

ELEMENT	DESCRIPTION	WHERE TO GO FOR MORE INFORMATION
small-icon?	The relative path within the Ejb-jar file that you can find a 16 × 16 icon image (either JPEG or GIF) representing this bean, to be displayed by tools.	
large-icon?	Same as small icon, except a 32 × 32 image.	
ejb-name	The nickname that you want to give this bean. This Ejb-name can be referenced later in the deployment descriptor.	Chapter 8
ejb-class	Session bean class. Remember to include the package too!	Chapter 8
transaction-type	For declarative transactions, set this to <i>Container</i> . For bean-managed transactions, set this to <i>Bean</i> .	Chapter 10
message-selector?	Filters for messages based on a special JMS selector string.	Chapter 8
acknowledge-mode?	If you perform bean-managed transactions, you must specify how to acknowledge messages when your <code>onMessage()</code> method is called. Set this to either <i>Auto-acknowledge</i> or <i>Dups-ok-acknowledge</i> .	Chapter 8
message-driven-destination?	Specifies the destination that you wish to listen for messages.	Chapter 8
env-entry*	Declares environment properties for this bean.	Chapter 9
ejb-ref*	Declares references to other beans.	Chapter 9
ejb-local-ref*	Declares local references to other beans.	Chapter 9
security-role-ref*	Declares security role references.	Chapter 9
security-identity?	Declares how to perform security context propagation.	Chapter 9
resource-ref*	Declares resource factory references (such as JDBC driver used in bean).	Chapter 9
resource-env-ref*	Binds resource factories to JNDI nicknames.	Chapter 9

<message-driven-destination>

The *message-driven-destination* element specifies the destination that you wish to listen for messages. Used in *message-driven*.

ELEMENT	DESCRIPTION	WHERE TO GO FOR MORE INFORMATION
destination-type	The type of destination, either <i>javax.jms.Queue</i> (for point-to-point) or <i>javax.jms.Topic</i> (for publish/subscribe).	Chapter 8
subscription-durability?	Indicates durability of messages, either <i>Durable</i> for durable messages, or <i>NonDurable</i> for nondurable messages.	Chapter 8

Defining Environment Properties

The following is an example of how to set up environment properties. Descriptions follow. See Chapter 9 for complete documentation.

```
<enterprise-beans>
  <session>
    <ejb-name>Pricer</ejb-name>
    <home>examples.PricerHome</home>
    <remote>examples.Pricer</remote>
    <ejb-class>examples.PricerBean</ejb-class>
    <session-type>Stateless</session-type>
    <transaction-type>Container</transaction-type>
    <env-entry>
      <description>
        The algorithm for this pricing engine.
      </description>
      <env-entry-name>Pricer/algorithm</env-entry-name>
      <env-entry-type>java.lang.String</env-entry-type>
      <env-entry-value>NoTaxes</env-entry-value>
    </env-entry>
  </session>
</enterprise-beans>
```

<env-entry>

The *env-entry* element defines an environment property that the bean can access via JNDI to customize its functionality at runtime. Used in *session*, *entity*, and *message-driven*.

Defining EJB References

ELEMENT	DESCRIPTION	WHERE TO GO FOR MORE INFORMATION
description?	A text description for this environment property.	Chapter 9
env-entry-name	The JNDI name relative to <code>java:comp/env</code> where the bean can lookup this environment property.	Chapter 9
env-entry-type	The fully qualified Java type of this environment property (such as <code>java.lang.String</code> or <code>java.lang.Integer</code>).	Chapter 9
env-entry-value?	The text value of this environment property.	Chapter 9

The following is an example of how to set up references from one bean to another. This is useful because beans can look each other up without needing to initialize JNDI to any particular driver. Descriptions follow. See Chapter 9 for complete documentation.

```
<ejb-jar>
<enterprise-beans>

  <session>
    <ejb-name>Catalog</ejb-name>
    ... define a catalog session bean ...
  </session>

  <session>
    <ejb-name>Pricer</ejb-name>
    <home>examples.PricerHome</home>
    <remote>examples.Pricer</remote>
    <ejb-class>examples.PricerBean</ejb-class>
    <session-type>Stateless</session-type>
    <transaction-type>Container</transaction-type>
```

```

<ejb-ref>
  <description>
    This reference is from the Pricer to the Catalog
  </description>
  <ejb-ref-name>ejb/CatalogHome</ejb-ref-name>
  <ejb-ref-type>Session</ejb-ref-type>
  <home>CatalogHome</home>
  <remote>Catalog</remote>
  <ejb-link>Catalog</ejb-link>
</ejb-ref>
</session>
</enterprise-beans>
</ejb-jar>

```

<ejb-ref>

The *ejb-ref* element defines a remote reference from one bean to another. This should be used sparingly, since local interfaces are the superior way to call from one bean to another. Used in *session*, *entity*, and *message-driven*.

ELEMENT	DESCRIPTION	WHERE TO GO FOR MORE INFORMATION
description?	A text description of this EJB reference.	Chapter 9
ejb-ref-name	The JNDI name relative to <i>java:comp/env</i> that will be used to lookup this EJB reference. Recommended: prefix with <i>ejb/</i> .	Chapter 9
ejb-ref-type	The type of the bean we have a reference to. Could be either <i>Session</i> or <i>Entity</i> . (Message-driven beans are not referred to directly because they have no home and are accessed via JMS.)	Chapter 9
home	The home interface class being referred to. Remember to include the package too!	Chapter 9
remote	The remote interface class being referred to. Remember to include the package too!	Chapter 9
ejb-link?	The Ejb-name of the bean we're referring to. Note: You can also refer to beans in a different jar file, by using a syntax such as <i>../products/product.jar#ProductEJB</i> .	Chapter 9

<ejb-local-ref>

The *ejb-local-ref* element defines a local reference from one bean to another. We recommend you use these types of references, since local interfaces are the superior way to call from one bean to another. Used in *session*, *entity*, and *message-driven*.

ELEMENT	DESCRIPTION	WHERE TO GO FOR MORE INFORMATION
description?	A text description of this EJB reference.	Chapter 9
ejb-ref-name	The JNDI name relative to <i>java:comp/env</i> that will be used to lookup this EJB reference. Recommended: prefix with <i>ejb/</i> .	Chapter 9
ejb-ref-type	The type of the bean we have a reference to. Could either be <i>Session</i> or <i>Entity</i> . (message-driven beans are not referred to directly because they have no home, and are accessed via JMS.)	Chapter 9
local-home	The local home interface class being referred to. Remember to include the package too!	Chapter 9
local	The local interface class being referred to. Remember to include the package too!	Chapter 9
ejb-link?	The Ejb-name of the bean we're referring to. Note: You can also refer to beans in a different jar file, by using a syntax such as <i>../products/product.jar#ProductEJB</i> .	Chapter 9

Defining Security

Security is a bit difficult to explain without a more thorough explanation. See Chapter 9 for complete documentation.

<security-role-ref>

The *security-role-ref* element defines a security role that your bean depends upon. Used in *session*, *entity*, and *message-driven*.

ELEMENT	DESCRIPTION	WHERE TO GO FOR MORE INFORMATION
description?	A text description of the security role.	Chapter 9
role-name	A text string for the security role that this bean references and depends upon.	Chapter 9
role-link?	Maps the above role-name abstract security role to a real security role defined in the assembly descriptor.	Chapter 9

<security-identity>

The *security-identity* element defines whether the caller's security identity is to be used when this bean executes, or whether another security identity should be used. Used in *session*, *entity*, and *message-driven*.

ELEMENT	DESCRIPTION	WHERE TO GO FOR MORE INFORMATION
description?	A text description	Chapter 9
use-caller-identity run-as	If you want to use the caller's identity when executing, set to the empty element <use-caller-identity/>. If you want to run as another security identity, define the <run-as> element.	Chapter 9

<run-as>

The *run-as* element allows your bean to run as a specified identity. Used in *security-identity*.

ELEMENT	DESCRIPTION	WHERE TO GO FOR MORE INFORMATION
description?	A text description	Chapter 9
role-name	The name of the security role you wish to run as.	Chapter 9

Defining Resource Factories

The following is an example of how to set up resource factories, which are drivers to external systems. Descriptions follow. See Chapter 9 for complete documentation.

```
<ejb-bar>
  <enterprise-beans>
    <entity>
      <ejb-name>Account</ejb-name>
      <home>examples.AccountHome</home>
      <remote>examples.Account</remote>
      <local-home>examples.AccountLocalHome</local-home>
      <local>examples.AccountLocal</local>
      <ejb-class>examples.AccountBean</ejb-class>
      <persistence-type>Bean</persistence-type>
      <prim-key-class>examples.AccountPK</prim-key-class>
      <reentrant>False</reentrant>

      <resource-ref>
        <res-ref-name>jdbc/ejbPool</res-ref-name>
        <res-type>javax.sql.DataSource</res-type>
        <res-auth>Container</res-auth>
      </resource-ref>
    </entity>
  </enterprise-beans>
```

<resource-ref>

The *resource-ref* element defines a reference to a resource factory. Used in *session*, *entity*, and *message-driven*.

ELEMENT	DESCRIPTION	WHERE TO GO FOR MORE INFORMATION
description?	A text description.	Chapter 9
res-ref-name	The JNDI name to which you wish to bind the resource factory, referenced off of <i>java:comp/env</i> .	Chapter 9
res-type	The fully qualified Java type of the resource factory, such as <i>javax.jms.ConnectionFactory</i> .	Chapter 9
res-auth	Set to <i>Application</i> if you are providing your own security to access this resource factory. Set to <i>Container</i> if the container is handling security access to this resource factory.	Chapter 9
res-sharing-scope?	Specifies whether connections obtained from this resource factory are shareable. Must be either <i>Shareable</i> or <i>Unshareable</i> .	Chapter 9

<resource-env-ref>

The *resource-env-ref* element defines a reference to an administered object. Used in *session*, *entity*, and *message-driven*.

ELEMENT	DESCRIPTION	WHERE TO GO FOR MORE INFORMATION
description?	A text description.	Chapter 9
resource-env-ref-name	The name of the administered object.	Chapter 9
resource-env-ref-type	The fully qualified type of the administered object.	Chapter 9

Defining Relationships

The following is an example of how to set up relationships. Descriptions follow. See Chapter 11 for complete relationships documentation.

```
<ejb-jar>

  <enterprise-beans>
  ...
</enterprise-beans>

  <relationships>
  <ejb-relation>
  <ejb-relation-name>Order-shipment</ejb-relation-name>
  <ejb-relationship-role>
  <ejb-relationship-role-name>
    order-spawns-shipment
  </ejb-relationship-role-name>
  <multiplicity>One</multiplicity>
  <relationship-role-source>
  <ejb-name>Order</ejb-name>
  </relationship-role-source>
  <cmr-field>
  <cmr-field-name>shipment</cmr-field-name>
  </cmr-field>
  </ejb-relationship-role>

  <ejb-relationship-role>
  <ejb-relationship-role-name>
    shipment-fulfills-order
```

```

    </ejb-relationship-role-name>
    <multiplicity>One</multiplicity>
    <cascade-delete/>
    <relationship-role-source>
      <ejb-name>Shipment</ejb-name>
    </relationship-role-source>
    <cmr-field>
      <cmr-field-name>order</cmr-field-name>
    </cmr-field>
  </ejb-relationship-role>
</ejb-relation>
</relationships>
</ejb-jar>

```

<relationships>

The *relationships* element defines CMP relationships. Used in *ejb-jar*.

ELEMENT	DESCRIPTION	WHERE TO GO FOR MORE INFORMATION
description?	A text description of the relationships.	Chapter 11
ejb-relation+	Defines one or more relationships.	Chapter 11 see next table.

<ejb-relation>

Each *ejb-relation* defines a single CMP relationship. Used in *relationships*.

ELEMENT	DESCRIPTION	WHERE TO GO FOR MORE INFORMATION
description?	A text description of this relationship.	Chapter 11
ejb-relation-name?	A unique nickname for this relationship.	Chapter 11
ejb-relationship-role	The first half of the relationship.	Chapter 11
ejb-relationship-role	The second half of the relationship.	Chapter 11

<ejb-relationship-role>

Each *ejb-relationship-role* defines half of a CMP relationship. Used in *ejb-relation*.

ELEMENT	DESCRIPTION	WHERE TO GO FOR MORE INFORMATION
description?	A text description of this half of the relationship.	Chapter 11
ejb-relationship-role-name?	A unique nickname for this half of the relationship.	Chapter 11
multiplicity	Relationships can be One:One, One:Many, Many:One, or Many:Many. This element declares this half of the relationship to either be <i>One</i> or <i>Many</i> .	Chapter 11
cascade-delete?	By declaring this empty element, when the other half of the relationship is removed, so is this half. <i>Note: The other half of the relationship must have a One multiplicity, because otherwise you could get into an infinite cascading deletion loop.</i>	Chapter 11
relationship-role-source	Identifies which bean is participating in this relationship.	Chapter 11
cmr-field?	Identifies the get/set method that will be located on this bean and will access the other half of the relationship.	Chapter 11

<relationship-role-source>

A *relationship-role-source* identifies which bean is participating in a relationship. Used in *ejb-relationship-role*.

ELEMENT	DESCRIPTION	WHERE TO GO FOR MORE INFORMATION
description?	A text description of this bean participating in the relationship.	Chapter 11
ejb-name	The Ejb-name of the bean participating in this relationship.	Chapter 11

<cmr-field>

A *cmr-field* identifies the get/set method that will be associated with a bean to access the other half of a relationship. Used in *ejb-relationship-role*.

ELEMENT	DESCRIPTION	WHERE TO GO FOR MORE INFORMATION
description?	A text description of this container-managed relationship field.	Chapter 11
cmr-field-name	The name of the get/set method associated with accessing the other half of this relationship. <i>Note: Leave off the get/set prefix, and make sure the first letter is lowercase.</i>	Chapter 11
cmr-field-type	If the other half of the relationship has a multiplicity of <i>Many</i> then you need to choose either: <i>java.util.Collection</i> (can contain duplicates) <i>java.util.Set</i> (cannot contain duplicates) This needs to match up to your bean's get/set methods.	Chapter 11

Defining the Assembly Descriptor

The following is an example of how to set up an assembly descriptor. Descriptions follow.

```
<ejb-jar>
...
<assembly-descriptor>

  <security-role>
    <description>
      Personnel authorized to perform employee administration
    </description>
    <role-name>admins</role-name>
  </security-role>

  <method-permission>
    <role-name>administrators</role-name>

  <method>
    <ejb-name>EmployeeManagement</ejb-name>
```

```

        <method-name>*</method-name>
    </method>
</method-permission>

<container-transaction>
    <method>
        <ejb-name>EmployeeManagement</ejb-name>
        <method-name>*</method-name>
    </method>
    <trans-attribute>Required</trans-attribute>
</container-transaction>

<exclude-list>
    <description>
        We don't have a 401k plan, so we
        don't support this method.
    </description>
    <method>
        <ejb-name>EmployeeManagement</ejb-name>
        <method-name>modify401kPlan</method-name>
        <method-params>String</method-params>
    </method>
</exclude-list>

</assembly-descriptor>
</ejb-jar>

```

<assembly-descriptor>

The *assembly-descriptor* element is the root of the assembly descriptor. Used in *ejb-jar*.

ELEMENT	DESCRIPTION	WHERE TO GO FOR MORE INFORMATION
security-role*	Identifies zero or more security roles that the application uses. This corresponds to the <i>role-link</i> element defined earlier in this appendix.	Chapter 9 See later table.
method-permission*	Sets up permissions for specific methods on your bean.	Chapter 9 See later table.
container-transaction*	Sets up transactions associates with specific methods on your bean.	Chapter 10 See later table.
exclude-list?	A list of methods that should never be callable. Useful if you acquire a bean from a third party and don't want to make use of all its functionality.	Chapter 9

<security-role>

The *security-role* element defines a security role that the application uses. This corresponds to the *role-link* element defined earlier in this appendix. Used in *assembly-descriptor*.

ELEMENT	DESCRIPTION	WHERE TO GO FOR MORE INFORMATION
description?	A text description of this security role.	Chapter 9
role-name	The text string naming this security role.	Chapter 9

<method-permission>

The *method-permission* element sets up permission on a specific method in your bean. Used in *assembly-descriptor*.

ELEMENT	DESCRIPTION	WHERE TO GO FOR MORE INFORMATION
description?	A text description of this method permission.	Chapter 9
role-name+ unchecked	The names of one or more security roles that can call these methods. Or, alternatively, you can specify the <code><unchecked/></code> empty element to disable security checks on these methods.	Chapter 9
method+	A list of one or more methods that these security permissions apply to.	Chapter 9 See later table.

<container-transaction>

The *container-transaction* element associates one or more methods with a container-managed (declarative) transaction. Used in *assembly-descriptor*.

ELEMENT	DESCRIPTION	WHERE TO GO FOR MORE INFORMATION
description?	A text description of this transaction.	Chapter 10
method+	A list of one or more methods that this transaction applies to.	Chapter 10 See later table.
trans-attribute	The style of transaction you'd like, either <i>NotSupported</i> , <i>Supports</i> , <i>Required</i> , <i>RequiresNew</i> , <i>Mandatory</i> , or <i>Never</i> .	Chapter 10

<exclude-list>

The *exclude-list* element is a list of methods that should never be callable. This is useful if you acquire a bean from a third party and don't want to make use of all its functionality. Used in *assembly-descriptor*.

ELEMENT	DESCRIPTION	WHERE TO GO FOR MORE INFORMATION
description?	A text description of why we are excluding these methods.	Chapter 9
method+	A list of one or more methods to exclude clients from being able to call.	Chapter 9 See later table.

<method>

The *method* element specifies a method on a bean. Used in *method-permission*, *container-transaction*, and *exclude-list*.

ELEMENT	DESCRIPTION	WHERE TO GO FOR MORE INFORMATION
description?	A text description.	
ejb-name	The Ejb-name of the bean we're interested in.	
method-intf?	Optionally identifies the interface name that we're specifying the method for, either <i>Home</i> , <i>Remote</i> , <i>LocalHome</i> , or <i>Local</i> . Useful if there is a naming conflict between two interface method signatures.	
method-name	The name of the method, capitalized properly. Can also use an asterisk (*) to specify all methods.	
method-params?	An optional list of parameters. Useful for disambiguating methods with the same signature.	See next table.

<method-params>

The *method-params* element is useful for disambiguating methods with the same signature Used in *method*.

ELEMENT	DESCRIPTION	WHERE TO GO FOR MORE INFORMATION
method-param*	Zero or more fully qualified Java types of parameters.	

The EJB Query Language (EJB-QL)

This appendix will help you fully understand the syntax and semantics of the *EJB Query Language* (EJB-QL), the language used to describe query methods for container managed persistent entity beans in EJB 2.0. To understand this appendix, you should first be familiar with the chapters on entity beans—Chapters 5, 6, and 7. Chapter 11 will also help.



You can begin coding with EJB without fully understanding EJB-QL. We recommend that you read this appendix if you are struggling with understanding the basics of EJB-QL, or if you are doing EJB-QL coding and need a guide.

Overview

EJB-QL is a standard and portable language for expressing container managed persistent entity bean query operations. These entity bean query operations can include finder methods (used by external entity bean clients), as well as select methods (used internally by the entity bean itself). EJB-QL is not necessary for bean managed persistence because the bean provider writes the database access code, which is integrated into the entity bean class itself.

EJB-QL is a new addition to EJB 2.0. Before EJB 2.0, you needed to explain to the container how to implement your query operations in a proprietary way. For example, you might bundle a container-specific flat-file with your bean. This flat-file would not be portable to other containers, which is annoying for bean providers who wish to write components that are container-agnostic.

Throughout this appendix, we will use an e-commerce object model to illustrate EJB-QL, using such entity beans as orders, line items, products, and customers. We designed that object model in Chapter 17.

A Simple Example

Let's kick things off with a simple EJB-QL example. Take the following entity bean remote finder method:

```
public java.util.Collection findAvailableProducts() throws
    FinderException, RemoteException;
```

This finder method means to find all products that are currently in stock. The following EJB-QL in the deployment descriptor instructs the container on how to generate the database access code that corresponds to this finder method:

```
...

<entity>
  <ejb-name>Product</ejb-name>
  <home>examples.ProductHome</home>
  <remote>examples.Product</remote>
  <ejb-class>examples.ProductBean</ejb-class>
  <persistence-type>Container</persistence-type>
  <prim-key-class>examples.ProductPK</prim-key-class>
  <reentrant>False</reentrant>

  <cmp-version>2.x</cmp-version>
  <abstract-schema-name>Product</abstract-schema-name>

  <cmp-field>
    <field-name>inventory</field-name>
  </cmp-field>
  ...more container-managed persistent fields...

  <query>
    <query-method>
      <method-name>findAvailableProducts</method-name>
      <method-params>
        </method-params>
    </query-method>
    <ejb-ql>
      <![CDATA[SELECT OBJECT(p) FROM Product AS p WHERE p.inventory >
0]]>
    </ejb-ql>
  </query>

  ...
</ejb-jar>
```

In the preceding code, we put together a query that resembles SQL or OQL. See Chapter 5 for more on object query language (OQL). We can refer to entity beans inside of the EJB-QL by using that entity bean's *abstract-schema-name* defined earlier in the deployment descriptor. We can also query its container managed fields or container managed relationships, or other entity beans.

In fact, if we're using a relational database, the container will translate this EJB-QL code into SQL code in the form of JDBC statements. The following SQL is an example of what might be generated depending on your container implementation:

```
SELECT DISTINCT p.PKEY
FROM PRODUCT p
WHERE p.INVENTORY > 0
```

This SQL returns primary keys (not rows) back to the container. The container then wraps those primary keys in EJB objects and returns RMI-IIOP stubs to the client who called the finder method. When the client calls business methods on those stubs, the EJB objects intercept the call, and the *ejbLoad()* method is called on the entity beans. The container then loads the actual rows from the database. Note that this process may be optimized depending on your container implementation.

EJB-QL is useful for both home interfaces and local home interfaces. A single EJB-QL definition will inform the container about how to implement the SQL code for both a home object and local home object that have identically named finder methods.



The XML parser that reads the prior deployment descriptor might get confused when confronted with the greater-than sign (>), thinking it might be a closing tag. To remove this ambiguity, we use a CDATA section, which instructs the XML parser to ignore that text for parsing purposes.

The Power of Relationships

The big difference between EJB-QL and SQL is that EJB-QL allows you to traverse relationships between entity beans using a dot-notation. For example:

```
SELECT o.customer
FROM Order o
```

In this EJB-QL, we are returning all customers that have placed orders. We are navigating from the order entity bean to the customer entity bean easily using a dot-notation. This is quite seamless.

What's exciting about this notation is that bean providers don't need to know about tables or columns; they merely need to understand the relationships

between the entity beans that they've authored. The container handles the traversal of relationships for us because we declare our entity beans in the same deployment descriptor and Ejb-jar file, empowering the container to manage all of our beans and thus understand their relationships.

In fact, you can traverse more than one relationship. That relationship can involve container managed relationship fields and container managed persistent fields. For example:

```
SELECT o.customer.address.homePhoneNumber
FROM Order o
```

The restriction on this type of recursive relationship traversal is that you are limited by the *navigability* of the relationships that you define in the deployment descriptor. For example, let's say that in the deployment descriptor, you declare that orders have a one-to-many relationship with line items, but you do not define the reverse many-to-one relationship that line items have with orders. When performing EJB-QL, you can get from orders to line items, but not from line items to orders. Even though the database is directionally neutral, the line items entity bean should have no knowledge of orders, and thus this traversal cannot take place. For more about how to define these types of relationships, see Chapter 11.

EJB-QL Syntax

An EJB-QL query contains three parts:

- A required SELECT clause
- A required FROM clause
- An optional WHERE clause

We now discuss the details of each of these clauses. We'll do the SELECT clause last because that indicates the return results of a query.

The FROM Clause

The FROM clause *constricts* the *domain* of a query. It indicates what part of the data storage you are querying over—that is, what entity beans you are going to look at. In the case of a relational database, the FROM clause typically restricts which tables you are querying over. For example, the following FROM clause means we are looking only at order entity beans:

```
SELECT OBJECT(o)
FROM Order AS o
```

What we're doing here is *declaring a variable* in the FROM clause. We are creating a variable, *o*, which can be used later in the query. In this case, we are reusing that variable in the SELECT clause. You can also reuse that variable in the WHERE clause.

Note that declaring variables will restrict your queries even if you don't use the variables. For example:

```
SELECT OBJECT(o)
FROM Order AS o, Customer AS c
```

The above query finds all orders if and only if there are any customers (which do *not* need to be related to the Order objects). Even though we aren't using the variable *c* anywhere else, we are still excluding orders if there are no customers. By declaring a variable, you are constraining the domain of the query. This is similar to the following SQL statement that returns all orders so long as there are one or more records in the Customer table:

```
SELECT o.*
FROM Order o, Customer c
```

Finally, you should note that the phrase *AS* is optional and is merely syntactic sugar to help make the query look better. This query produces the same result as the previous EJB-QL statement:

```
SELECT OBJECT(o)
FROM Order o, Customer c
```

Declaring Collection Variables

Sometimes you need to declare variables in the FROM clause that represent a collection of values. For example, let's say we want to find all of the line items that are attached to orders. The following query achieves that:

```
SELECT OBJECT(l)
FROM Order AS o, IN(o.lineItems) l
```

The preceding EJB-QL declares two variables:

- The phrase `Order AS o` declares a variable `o`, which represents any order entity bean.
- The phrase `IN(o.lineItems) l` declares a variable `l`, which represents any line item from any order bean's collection of line items.

As you can see, since the evaluation order is left to right, you can use variables on the right that were declared on the left.

Thus, you use the *AS* syntax when declaring a variable representing a single value and the *IN* syntax when declaring a variable representing a collection of

values. Note that both *AS* and *IN* queries can return multiple values from the EJB-QL query—the difference is that the *IN* syntax is necessary when traversing an underlying entity bean relationship that uses a *java.util.Collection*, such as an order that points to a collection of line items.

Variables Represent Only One Value at a Time

Next, consider the following query, which returns all line items that are attached to orders that are attached to customers:

```
SELECT OBJECT(l)
FROM Customer AS c, IN(c.orders) o, IN(o.lineItems) l
```

Notice the phrase *o.lineItems*. While *o* is a collection variable, it represents only one element of that collection at a time. Thus, it is perfectly legal to use the phrase *o.lineItems* because in that phrase, *o* represents an individual order, not a collection of orders.

The WHERE Clause

The EJB-QL WHERE clause restricts the *results* of a query. It is where you choose the values you want from the declared variables in the FROM clause. The general syntax of the WHERE clause is *WHERE <conditional expression>*. For example:

```
SELECT OBJECT(o)
FROM Order o
WHERE o.lineItems IS NOT EMPTY
```

The query finds all orders that have line items.

Handling Input Parameters

When performing a query, you'll often want to query based upon parameters supplied by the client. For example, to implement the following finder method that finds a product based on a description:

```
findProductByDescription(String s)
```

A WHERE clause can be used as follows:

```
SELECT OBJECT(p)
FROM Product p
WHERE p.description = ?1
```

Here, ?1 represents the first parameter passed in. Additional parameters would be numbered as ?2, ?3, and so on. Note that you don't need to use all variables declared in the finder/select method.

Conditional Expressions

Many conditional expressions are built-in to EJB-QL. The complete list is in Table D.1.

Note that you can have more than one conditional expression and use parentheses to denote order of execution. Your container may provide proprietary extensions to these conditional expressions as well, perhaps in a separate deployment descriptor.

EJB-QL also contains the following built-in functions:

CONCAT(String, String) combines two strings into one and returns a String.

SUBSTRING(String, start, length) cuts a String into a smaller String, beginning at *start* and being *length* long.

LOCATE(String, String [, start]) returns an *int* denoting where a String is located within another String. You can use the optional *start* parameter to indicate where to begin locating.

LENGTH(String) gives you a string's length, returned as an *int*.

ABS(number) returns the absolute value of a number, which can be an *int*, *float*, or *double*.

SQRT(double) takes the square root of a number and returns it as a *double*.

Dealing with Collections

Normally if you want to use collections in the WHERE clause, you should declare those collections as variables in the FROM clause. For example, the following is invalid:

```
SELECT OBJECT(1)
FROM Order AS o
WHERE o.lineItems.product.name = 'chip'
```

The above is invalid because we are trying to reference a variable from a collection. The following is the correct way to write this EJB-QL:

```
SELECT OBJECT(1)
FROM Order AS o, IN(o.lineItems) l
WHERE l.product.name = 'chip'
```

Table D.1 EJB-QL Conditional Expressions

CONDITIONAL EXPRESSION	EXAMPLE	NOTES
Mathematical operations: +, -, *, / Comparison operations: =, >, <, >=, <=, <> (not equal) Logical operators: NOT, AND, OR	Find all products that are computer chips and whose profit margin is positive: SELECT OBJECT(p) FROM Product p WHERE (p.description = 'chip') AND (p.basePrice - p.cost > 0)	<ul style="list-style-type: none"> ■ Two entity beans are equal if and only if they share the same primary key value. ■ You cannot compare two different entity bean classes.
Between expressions	Find all products whose price is at least 1000 and at most 2000: SELECT OBJECT(p) FROM Product p WHERE p.basePrice BETWEEN 1000 AND 2000	Can also use NOT BETWEEN to return all data that is not between two values.
In expressions	Find all products whose manufacturer is either Intel or Sun: SELECT OBJECT(p) FROM Product p WHERE p.manufacturer IN ('Intel', 'Sun')	Can also use NOT IN to return all data that is not in a range.
Like expressions	Find all products with ids that begin with 12 and end with 3. For example, 123 or 12993 qualifies, but not 1234: SELECT OBJECT(p) FROM Product p WHERE product.productID LIKE '12*3' Find all products with ids that begin with 123 and are a total of four characters long. For example, 123c qualifies, but not 14 nor 12345: SELECT OBJECT(p) FROM Product p WHERE product.productID LIKE '123_'	<ul style="list-style-type: none"> ■ % stands for any sequence of zero or more characters. ■ _ stands for a single character. ■ You can represent the literal % or _ character by using special escape sequences (see the EJB spec for more). ■ You can also use NOT LIKE to achieve the opposite effect.

continued

Table D.1 EJB-QL Conditional Expressions (continued)

CONDITIONAL EXPRESSION	EXAMPLE	NOTES
Null comparison expressions	Find all products that have NULL descriptions: <pre>SELECT OBJECT(p) FROM Product p WHERE product.description IS NULL</pre>	You can also use NOT NULL to find all data that has non-NULL values.
Empty collection comparison expressions	Find all orders that have no line items: <pre>SELECT OBJECT(o) FROM Order o WHERE o.lineItems IS EMPTY</pre>	<ul style="list-style-type: none"> ■ You can also use IS NOT EMPTY to find valid collections. ■ In this special case, you can declare collections in the WHERE clause rather than declaring them as variables first in the FROM clause.
Collection member expressions	Find all line items that are attached to orders: <pre>SELECT OBJECT(l) FROM Order o, LineItem l WHERE l MEMBER OF o.lineItems</pre>	<ul style="list-style-type: none"> ■ The word OF is optional. ■ In this special case, you can declare collections in the WHERE clause rather than declaring them as variables first in the FROM clause. ■ Can also use NOT MEMBER OF to locate data where elements are not members of collections.

The two special exceptions to this rule are when you use the `EMPTY` or `MEMBER` conditional expressions, shown in Table D.1. In these cases, you can use collections in the `WHERE` clause.

Performing Comparisons

Sometimes you may need to declare more than one variable that represents the same entity bean. When you are performing comparisons, this comes in very handy. For example:

```
SELECT OBJECT(p1)
FROM Product p1, Product p2
WHERE p1.quantityInStock > p2.quantityInStock AND
      p2.name='Pentium 866'
```

The preceding query finds all products that have a greater quantity in stock than a Pentium 866 chip.

The SELECT Clause

The EJB-QL `SELECT` clause specifies the return results of a query. To understand why we need the `SELECT` clause, consider the following query, which returns all orders that contain line items:

```
SELECT OBJECT(o)
FROM Order AS o, IN(o.lineItems) l
```

In this query, we have defined two variables in the `FROM` clause: *o* and *l*. The `SELECT` clause is necessary because it affirms that we want to return *o* (and not *l*) to the client that called the query.

How to Traverse Relationships

The `SELECT` clause can traverse relationships. For example, the following query returns all the products in all the orders that contain line items:

```
SELECT l.product FROM Order AS o, IN(o.lineItems) l
```

As you can see, we can use the convenient dot-notation to traverse relationships in the `SELECT` clause. Behind the scenes, a SQL `JOIN` statement might occur.

If you've been paying careful attention, you may have noticed that in the earlier example we wrapped a variable *o* with the phrase `OBJECT()`, but in this example, we didn't use the phrase `OBJECT()` at all. The EJB-QL rule is that you wrap your return result with the phrase `OBJECT()` only if you are returning a stand-alone variable that does not traverse a relationship using the dot-notation.

How to Deal with Collections

Let's say we want to find all line items on all orders. We are thus asking for a collection of return results. Unfortunately, the following SELECT clause will not work:

```
SELECT o.lineItems
FROM Order AS o
```

The reason the above doesn't work is that SELECT clauses may only return single variables, not collections. To get around this restriction, you need to define a variable in the FROM clause. The following demonstrates this as a legal way to find all line items on all orders:

```
SELECT OBJECT(l)
FROM Order AS o, IN(o.lineItems) l
```

How to Filter for Duplicates

You can control whether SELECT clauses return duplicates. For example, take our previous EJB-QL query that finds all products in all order line items:

```
SELECT l.product FROM Order AS o, IN(o.lineItems) l
```

The above query may return duplicate products because two different people may have ordered the same product. To get a unique list, you must apply the DISTINCT filter, as follows:

```
SELECT DISTINCT l.product FROM Order AS o, IN(o.lineItems) l
```

Another choice is to declare your finder or select method to return a *java.util.Set*, which may not contain duplicates compared to a *java.util.Collection*. If you use a *java.util.Set*, both of the preceding EJB-QL statements would return the same unique results.

How to Control What Gets Returned in Finders

EJB-QL queries return results differently depending on how the client initiates the query. For example, take the following finder queries (thrown exceptions omitted):

```
// declared on the home interface
public java.util.Collection findAllProducts();
```

```
// declared on the local home interface
public java.util.Collection findAllProducts();
```

We want EJB objects to be returned for the first query and EJB local objects to be returned for the second query. The EJB-QL code in the deployment descriptor for both of these query methods could be:

```
<query>
  <query-method>
    <method-name>findAllProducts</method-name>
    <method-params>
      </method-params>
    </query-method>
  <ejb-ql>
    <![CDATA[SELECT OBJECT (p) FROM Product AS p]]>
  </ejb-ql>
</>
```

What's great here is that we wrote our EJB-QL once, yet we can reuse it for both the home interface and local home interface. The container will automatically wrap the return results in an EJBObject or EJBLocalObject (or collections of EJBObjects/EJBLocalObjects). These are the only possible types you can return from a finder query.

How to Control What Gets Returned in Selects

With finder methods, the container knows whether the results of a finder should be EJB objects or EJB local objects, because the container could look at whether the query was defined on the home interface or local home interface, respectively. But what about ejbSelect() methods (see Chapter 7)? Consider the following ejbSelect():

```
public abstract java.util.Collection ejbSelectAllProducts();
```

Here, we define the ejbSelect() method on the entity bean class, which doesn't give the container any information about whether our query should return EJB objects or EJB local objects. How does the container know what objects to wrap around the results?

To get around this, EJB requires that you set up a special stanza in the deployment descriptor to inform the container about whether the results should be local or remote objects:

```
<query>
  <query-method>
    <method-name>ejbSelectAllProducts</method-name>
    <method-params>
      </method-params>
    </query-method>
  <result-type-mapping>Local</result-type-mapping>
  <ejb-ql>
```

```

    <![CDATA[SELECT OBJECT (p) FROM Product AS p]]>
  </ejb-ql>
</>

```

The preceding code will cause the *ejbSelect()* method to return a collection of EJB local objects. If you want the results to be a collection of EJB objects, change the *result-type-mapping* element to have the value *Remote*.

Finally, note that *ejbSelect()* methods can also return container managed fields. For example:

```
public abstract java.lang.String ejbSelectProductName();
```

Finder methods cannot return container managed fields because finder methods can operate remotely and at the granularity of entity beans, not parts of entity beans.

Truth Tables

Let's wrap up our EJB-QL lesson with a look at the truth tables for how the operations AND, OR, and NOT evaluate (see Tables D.2, D.3, and D.4). The

Table D.2 The AND Truth Table

AND	TRUE	FALSE	UNKNOWN
True	True	False	Unknown
False	False	False	False
Unknown	Unknown	False	Unknown

Table D.3 The OR Truth Table

OR	TRUE	FALSE	UNKNOWN
True	True	True	True
False	True	False	Unknown
Unknown	True	Unknown	Unknown

Table D.4 The NOT Truth Table

NOT	
True	False
False	True
Unknown	Unknown

way you read these tables is to combine the column header with the row header using the operator in the upper left-hand corner. That should give you the result in the cell located at the intersection of the column and row. Note also that in the tables, the case of *unknown* means expressions that produce an unknown result, such as the clause:

```
WHERE NULL IN ('Intel', 'Sun')
```



In the final section of this appendix, you can test your knowledge of EJB-QL. Here is a list of queries that we'd like to implement. Try to figure out the EJB-QL without looking at the description, or try to figure out the description by looking at the EJB-QL. (Answers at the end of this appendix.)

1. Find all line items.
2. Find all customers' home addresses.
3. Find all customers' home addresses without duplicates.
4. Find all line items that are attached to orders.
5. Find all orders that contain line items.
6. Find all orders that do not contain line items.
7. Find all products whose descriptions are either chip or motherboard.
8. Find all products that have a zero inventory.
9. Find all products with inventory greater than a parameter passed in.
10. Find all products with inventory between 10 and 20.
11. Find all products whose remaining inventory is greater than the remaining inventory for products manufactured by Intel.
12. Find the names of all customers whose names begin with A.

Final Note

Be forewarned that while EJB-QL is a convenient layer of indirection that isolates you from the database, a danger lurks under the covers. The danger is that your generated SQL code could perform poorly because you are not hand-tuning SQL code but rather, dealing with high-level EJB-QL code.

Standard performance best practices of optimizing SQL still apply with EJB-QL. Check and recheck the optimization of the container-generated SQL by examining your generated helper files or your database log. Here are some possible ways to optimize your queries:

- Optimize your entity beans using specific container flags such as lazy-loading flags (check your container documentation).
- If available, use your container-tools to help generate more optimal SQL from EJB-QL.
- Redesign your EJB-QL.
- Rewrite some of your finder or select methods.
- Redesign/denormalize your database schema.
- Rethink your entity bean design.
- Manually write your SQL.

Summary

In this appendix, we've learned about EJB-QL. EJB-QL is a great advancement in EJB because it allows bean providers to ship code that contains queries that are portable across containers. We learned about the syntax of EJB-QL, including the SELECT, FROM, and WHERE clauses. We then went through several EJB-QL examples. You should now be empowered to try writing your own EJB-QL and begin experimenting with container managed persistent entity beans.



Answers to Quiz on Previous Page

1. `SELECT OBJECT(l) FROM LineItem l`
2. `SELECT c.homeAddress FROM Customer c`
3. `SELECT DISTINCT c.homeAddress FROM Customer c`
4. `SELECT OBJECT(l) FROM Order o, IN(o.lineItems) l`
5. `SELECT OBJECT(o) FROM Order o, IN(o.lineItems) l`
6. `SELECT OBJECT(o) FROM Order o WHERE o.lineItems IS EMPTY`
7. `SELECT OBJECT(p) FROM Product p WHERE p.description IN ('chip', 'motherboard')`
8. `SELECT OBJECT(p) FROM Product p WHERE p.inventory = 0`
9. `SELECT OBJECT(p) FROM Product p WHERE p.inventory > ?1`
10. `SELECT OBJECT(p) FROM Product p WHERE p.inventory BETWEEN 10 AND 20`
11. `SELECT OBJECT(p1) FROM Product p1, Product p2 WHERE p1.inventory > p2.inventory AND p2.manufacturer = 'Intel'`
12. `SELECT c.name FROM Customer c WHERE c.name LIKE 'A%'`

EJB Quick Reference Guide

This appendix is a quick reference for programmers to use during EJB development. In the first section, you'll find Figures E.1 through E.13 illustrating what's really going on in an EJB system. These were taken directly from the EJB specification; I've condensed the diagrams and commented on them to clarify their meaning. You'll also find summaries and explanations of each method in the EJB architecture, as well as a transaction reference.

Session Bean Diagrams

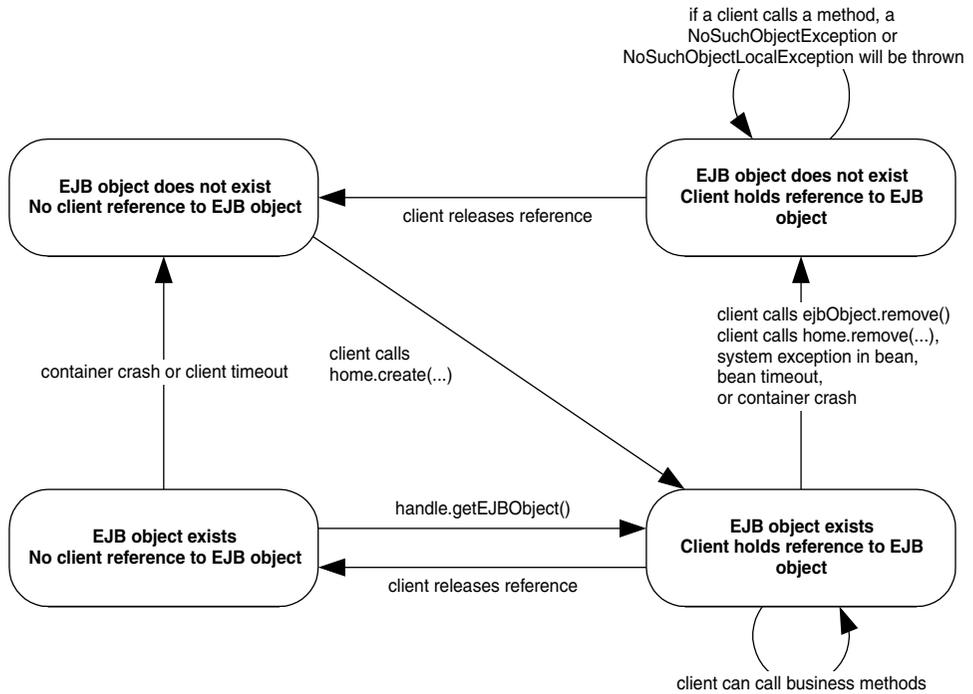


Figure E.1 The client's view of a session bean object life cycle.

Stateless Session Bean Diagrams

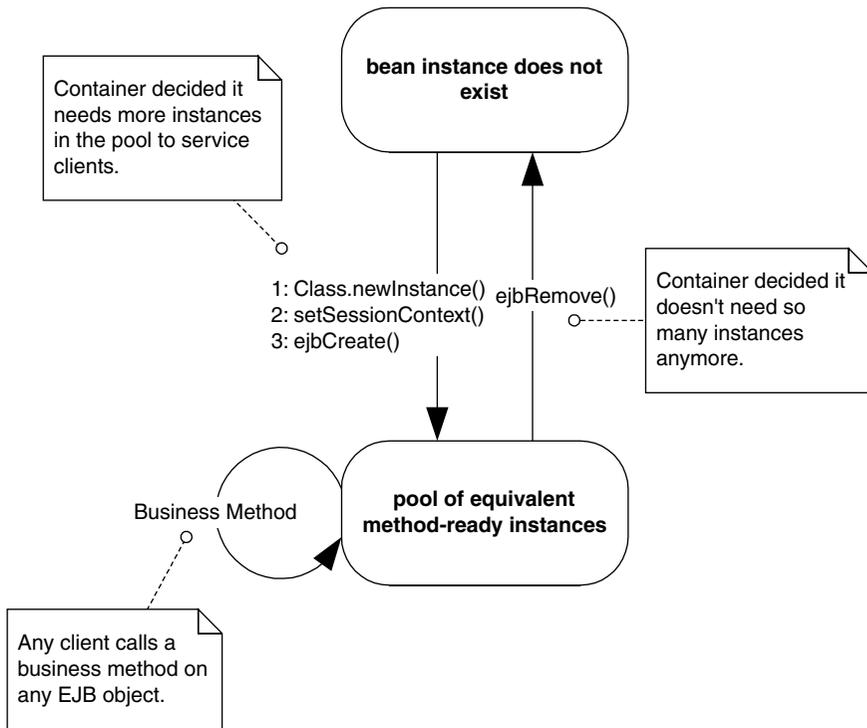


Figure E.2 The life cycle of a stateless session bean. Each method call shown is an invocation from the container to the bean instance.

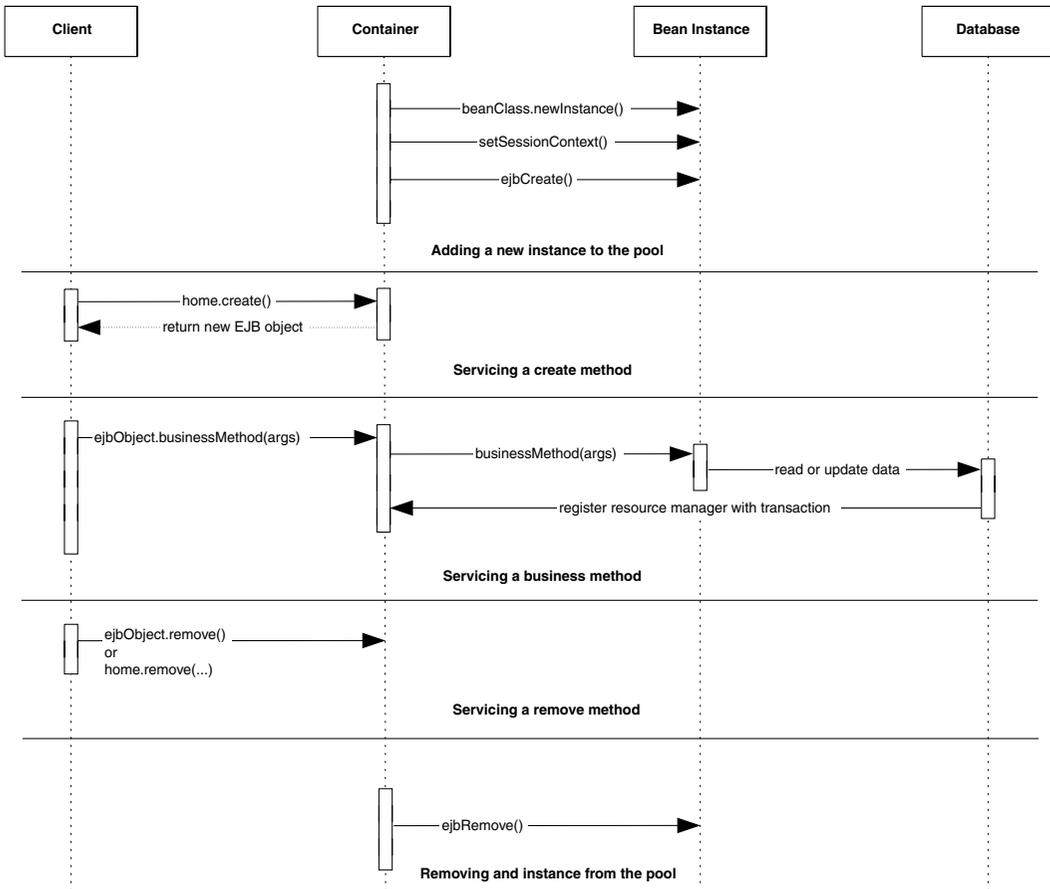


Figure E.3 Sequence diagram for stateless session beans. For simplicity, the Container object represents all container subsystems, including EJB objects, home objects, transaction services, and so on.

Stateful Session Bean Diagrams

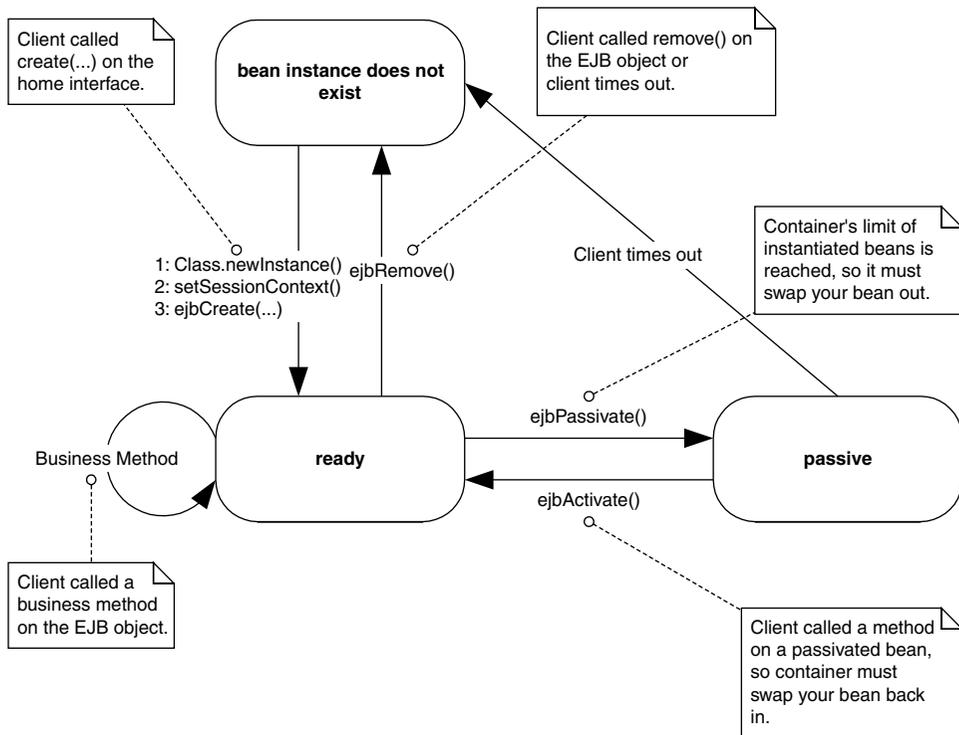


Figure E.4 The life cycle of a stateful session bean (does not implement *javax.ejb.SessionSynchronization*). Each method call shown is an invocation from the container to the bean instance.

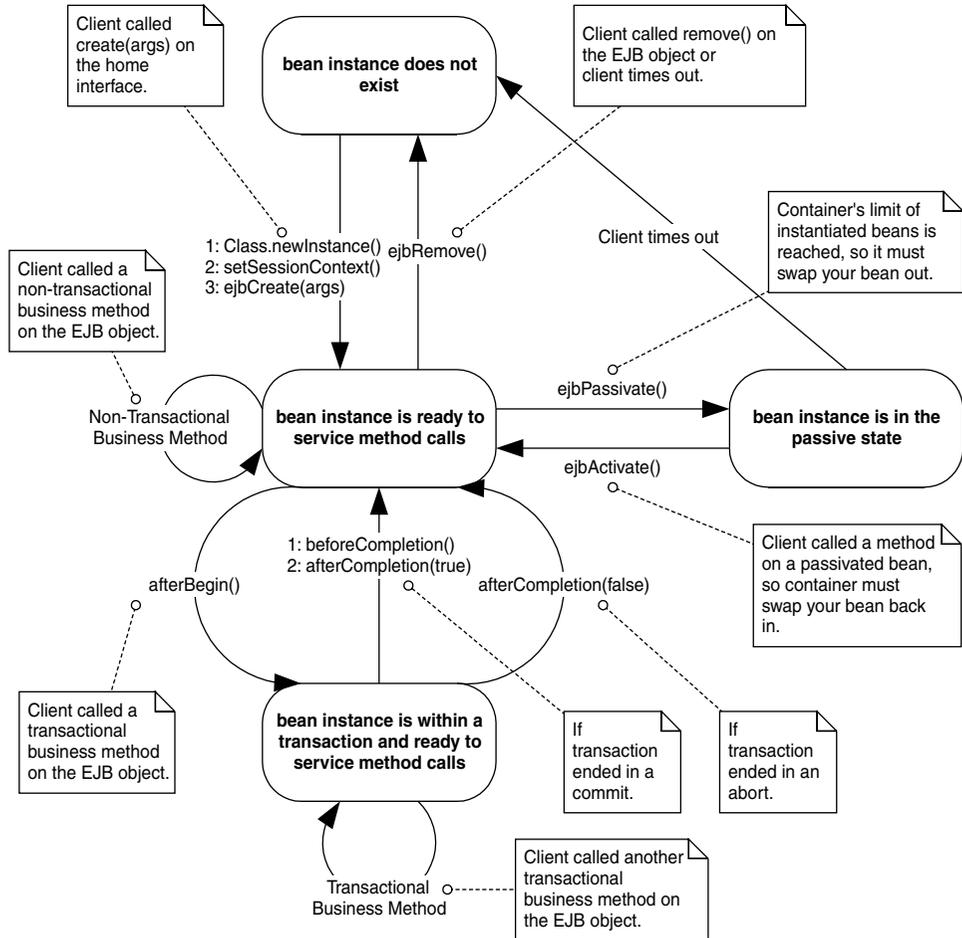


Figure E.5 The life cycle of a stateful session bean (implements *javax.ejb.SessionSynchronization*). Each method call shown is an invocation from the container to the bean instance.

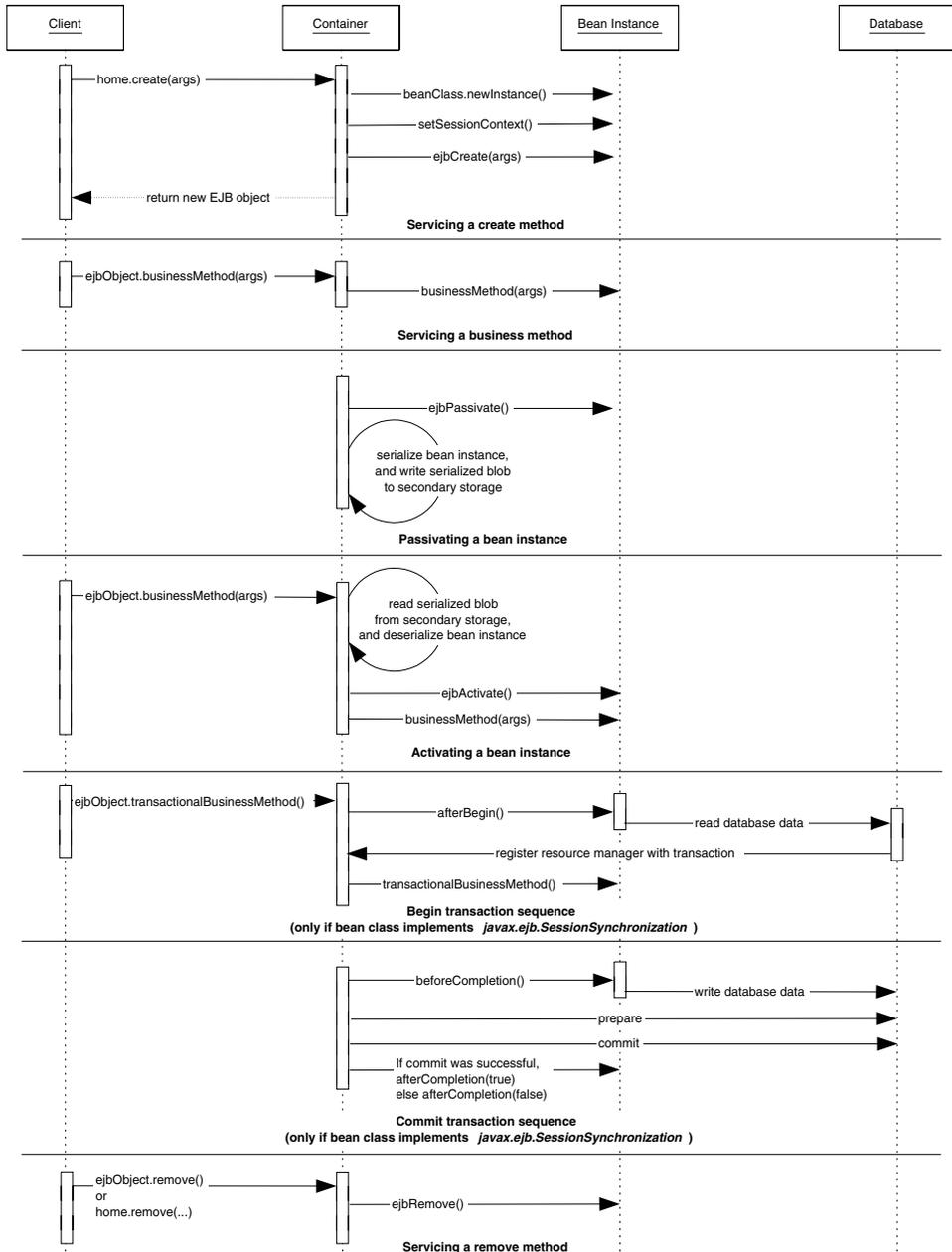


Figure E.6 Sequence diagram for stateful session beans. For simplicity, the Container object represents all container subsystems, including EJB objects, home objects, transaction services, and so on.

Entity Bean Diagrams

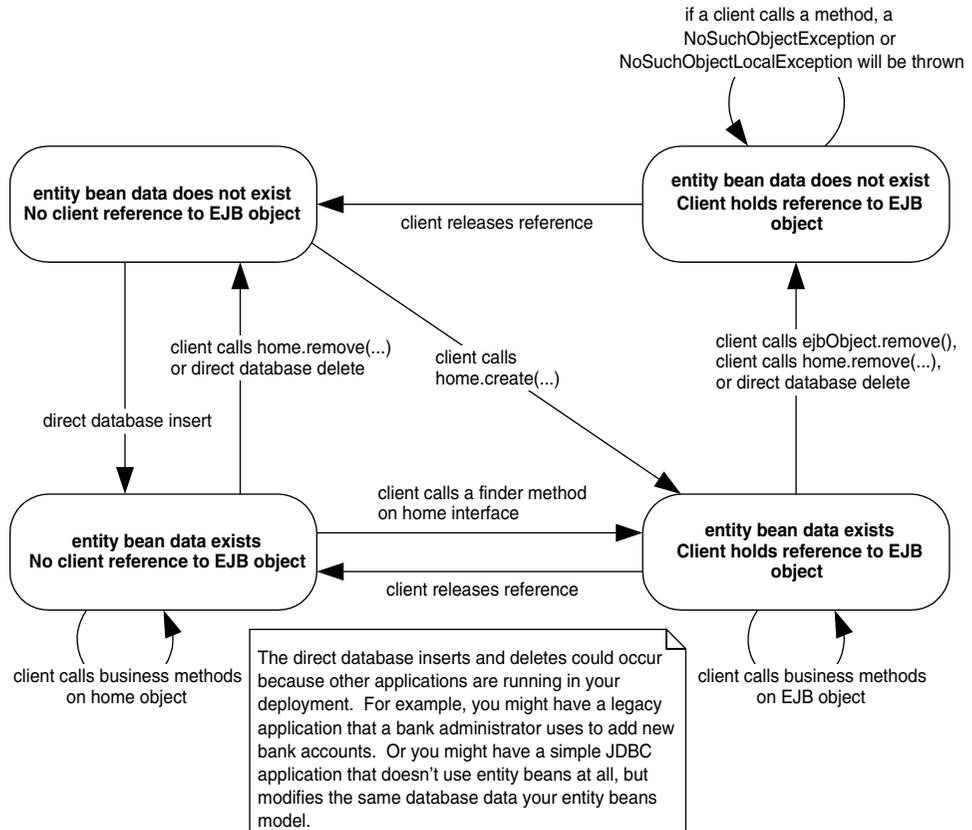


Figure E.7 The client's view of an entity bean object life cycle.

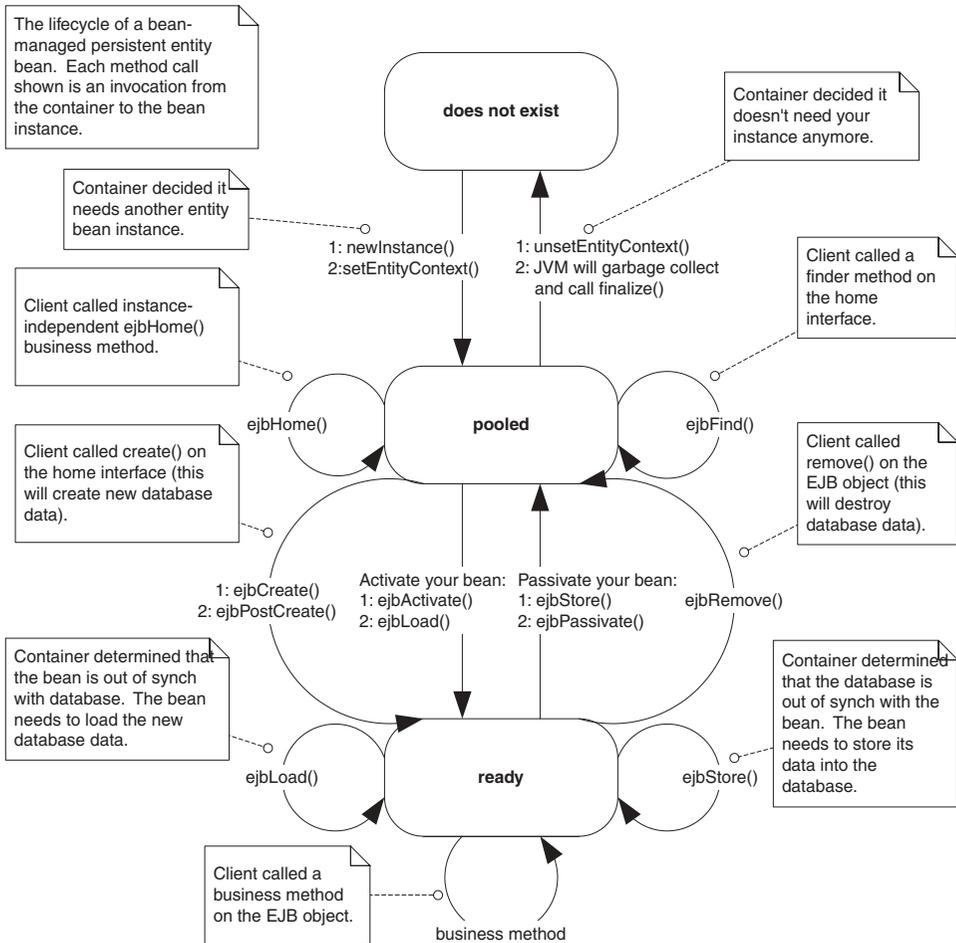


Figure E.8 The life cycle of a bean managed persistent entity bean. Each method call shown is an invocation from the container to the bean instance.

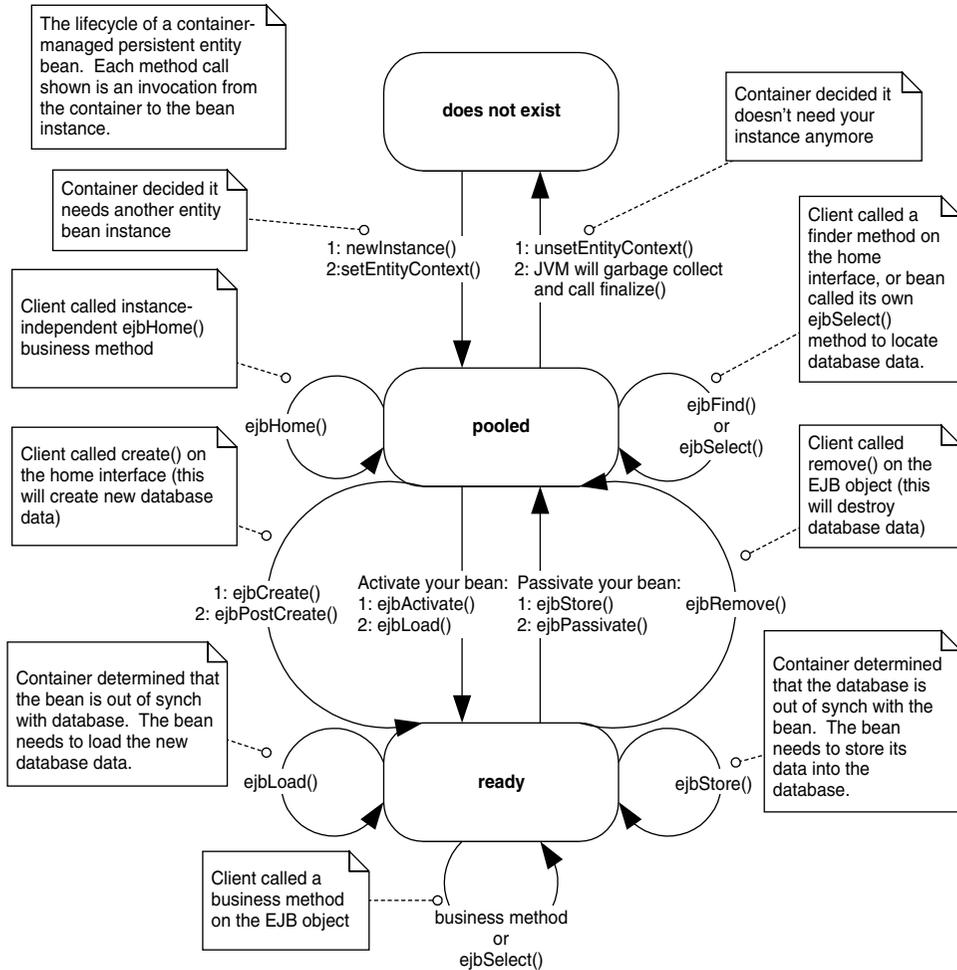


Figure E.9 The life cycle of a container managed persistent entity bean. Each method call shown is an invocation from the container to the bean instance.

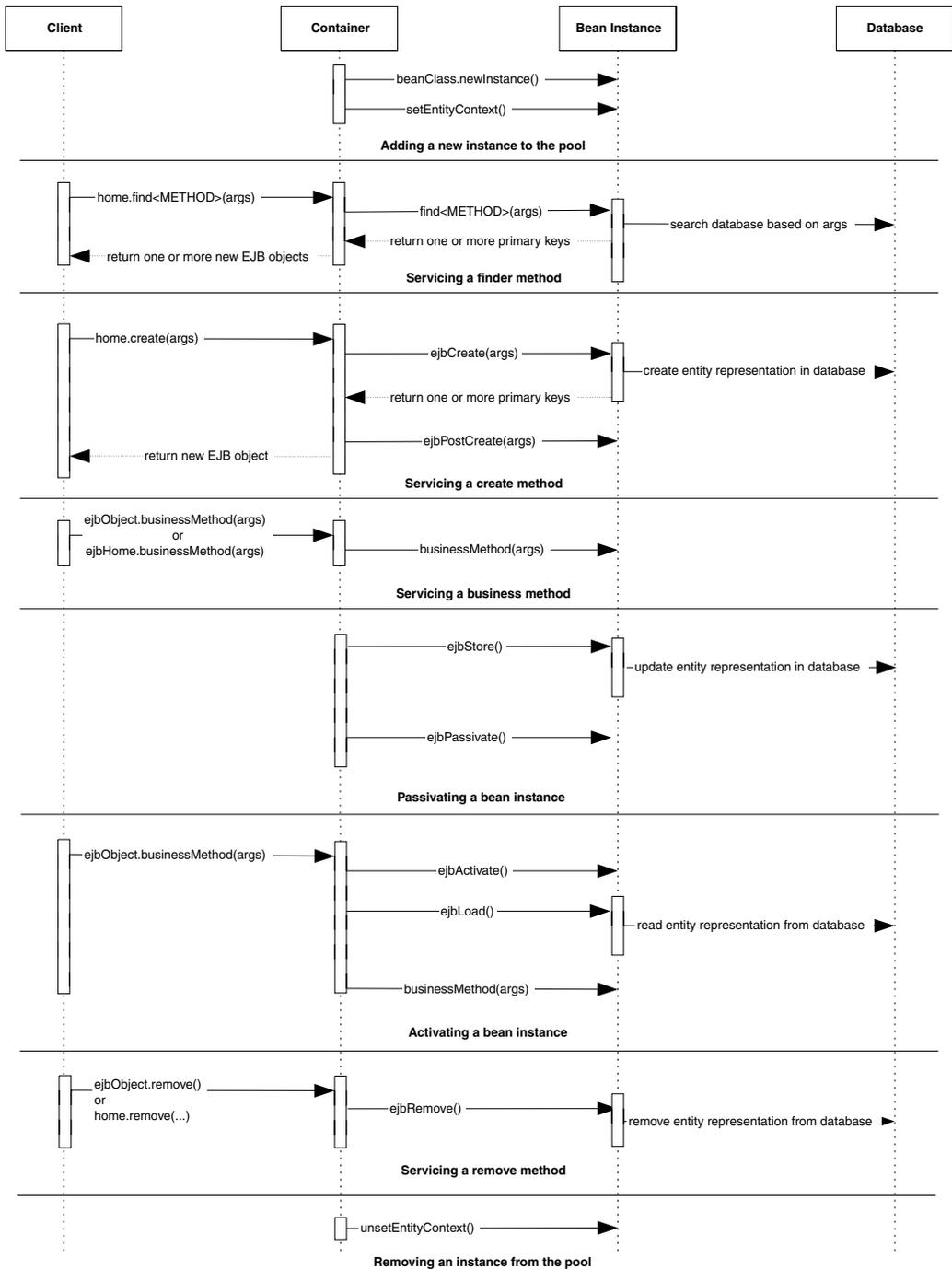


Figure E.10 Sequence diagram for bean managed persistent entity beans. For simplicity, the Container object represents all container subsystems, including EJB objects, home objects, transaction services, and so on.

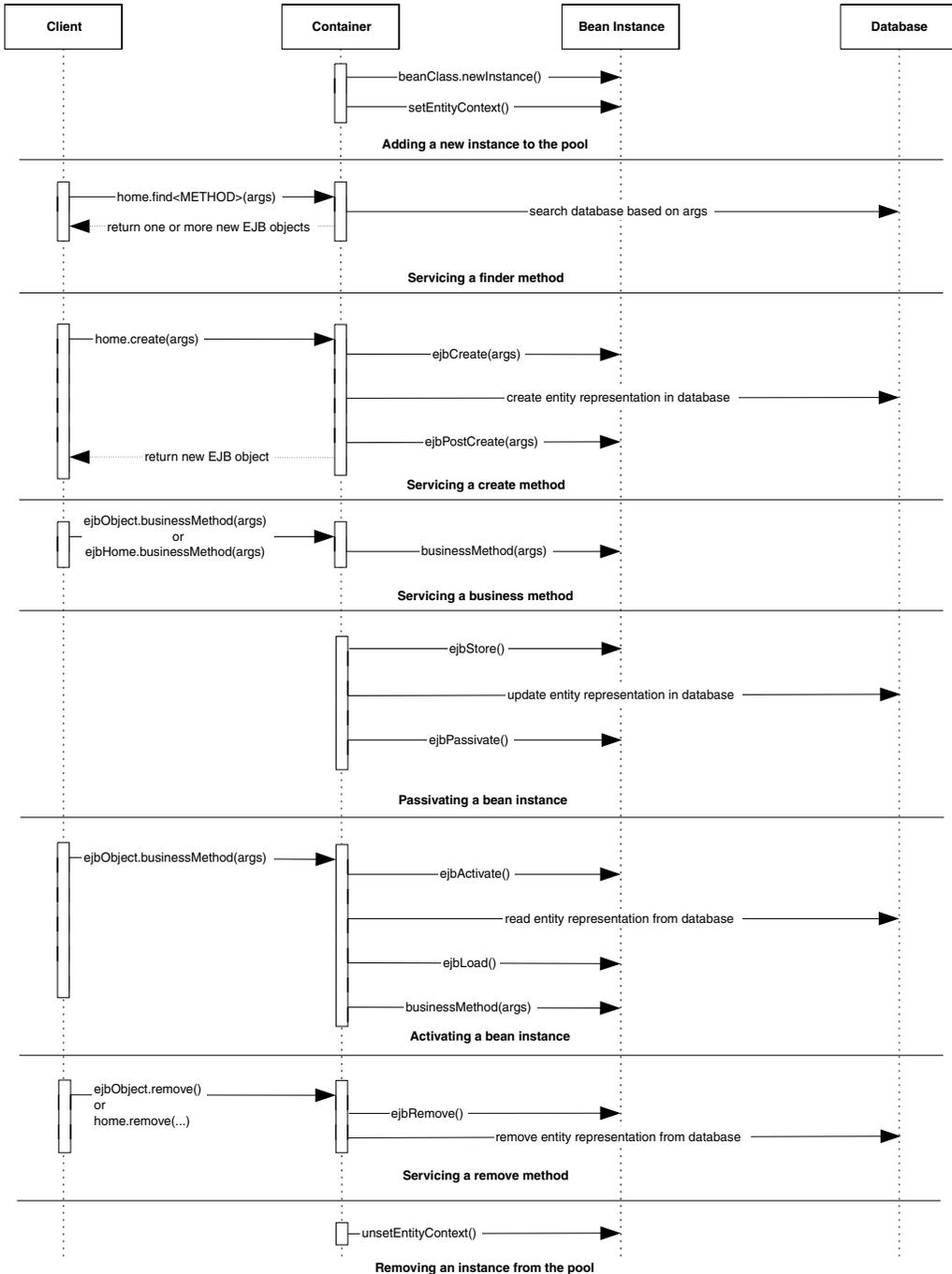


Figure E.11 Sequence diagram for container managed persistent entity beans. For simplicity, the Container object represents all container subsystems, including EJB objects, home objects, transaction services, and so on.

Message-Driven Bean Diagrams

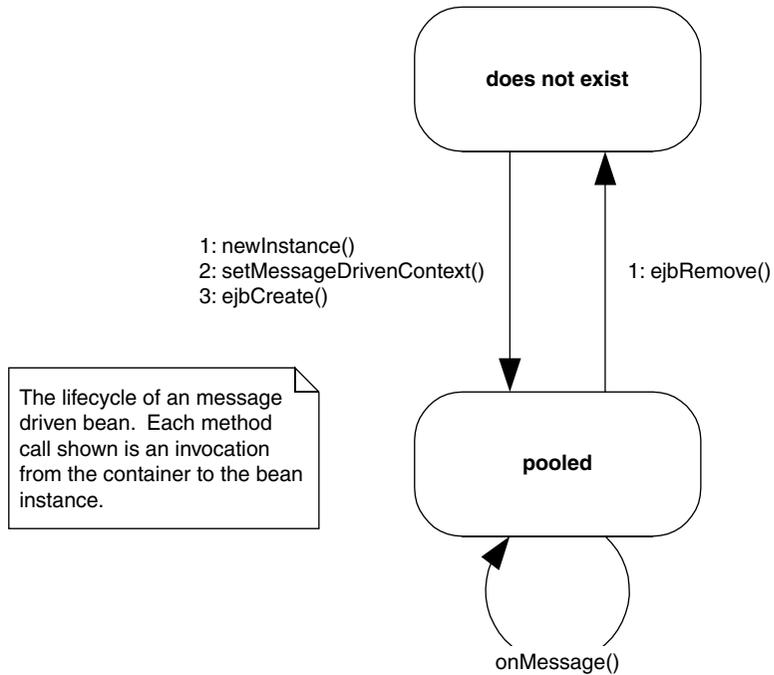


Figure E.12 The life cycle of a message-driven bean. Each method call shown is an invocation from the container to the bean instance.

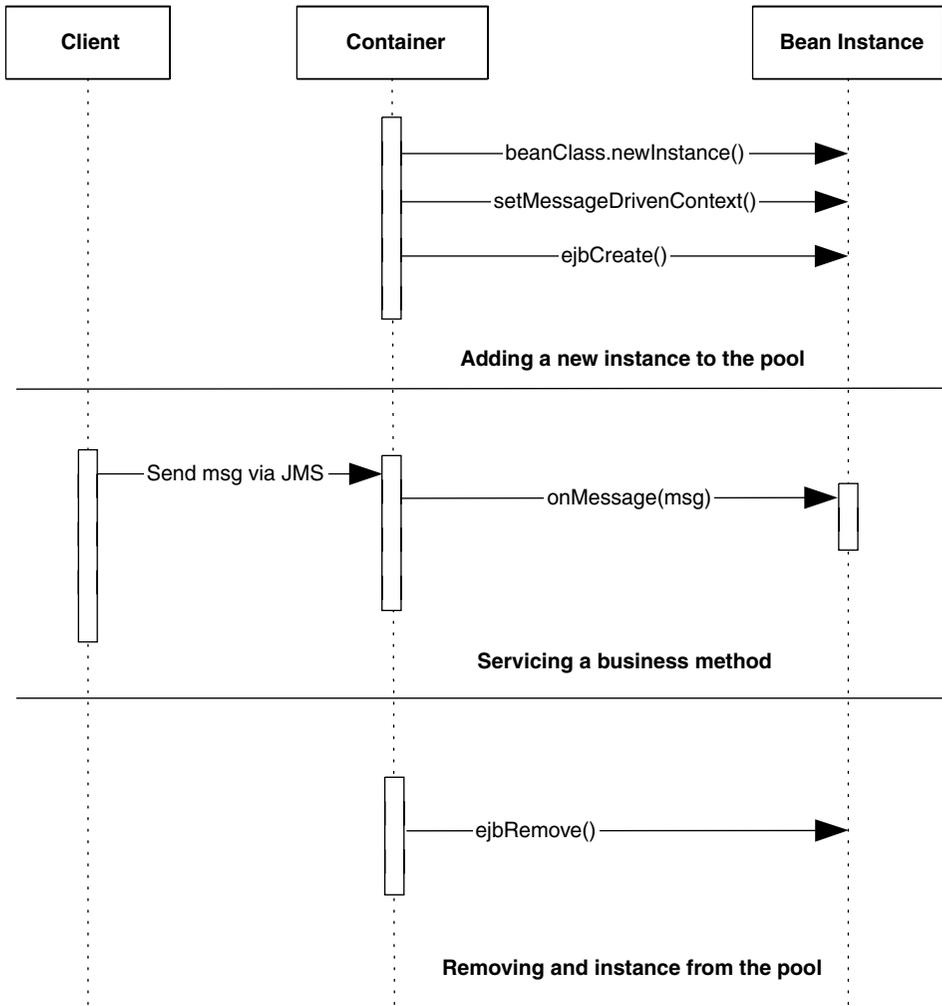


Figure E.13 Sequence diagram for message-driven beans. For simplicity, the Container object represents all container subsystems, including home objects, transaction services, and so on.

EJB API Reference

The following section explains the Enterprise JavaBeans API, which is the *javax.ejb* package. This API is the essence of EJB and defines the specific signature contracts between clients, enterprise beans, and containers.

EJBContext

An EJBContext object is a container-implemented object (see Table E.1). Your bean can use an EJB context to perform callbacks to the container. These callbacks help your bean determine its current transactional status, security status, and more. Your container must make an EJB context available to your enterprise bean at runtime. Both *SessionContext*, *EntityContext*, and *MessageDrivenContext* extend this interface.

```
public interface javax.ejb.EJBContext
{
    public javax.ejb.EJBHome getEJBHome();
    public javax.ejb.EJBLocalHome getEJBLocalHome();

    public boolean getRollbackOnly();
    public void setRollbackOnly();
    public javax.transaction.UserTransaction getUserTransaction();

    public boolean isCallerInRole(java.lang.String);
    public java.security.Principal getCallerPrincipal();
}
```

Table E.1 javax.ejb.EJBContext

METHOD	EXPLANATION
getEJBHome()	Call this from within your bean to access your own home object. You can use this home object to create, destroy, or find EJB objects of your own bean class type. This method is not used very often.
getEJBLocalHome()	Same as getEJBHome() except this retrieves the local interface version.
getRollbackOnly()	Asks the container if the transaction is doomed to rollback. If it's doomed, you can avoid performing computer-intensive operations (see Chapter 10).
setRollbackOnly()	If something goes horribly wrong inside your bean, you can call this method to force the current transaction to rollback (see Chapter 10).
getUserTransaction()	Retrieves the JTA UserTransaction interface to perform programmatic transactions (see Chapter 10).
isCallerInRole(String)	Asks the container if the current logged-in user is in the proper security role to perform a desired operation. Useful for programmatic security (see Chapter 9).
getCallerPrincipal()	Retrieves the current logged-in user's security principal. You can use this principal to query a database or perform other operations (see Chapter 9).

EJBHome

Remote clients create, find, and remove EJB objects through home interfaces. Your home interfaces extend *javax.ejb.EJBHome*. The container will implement the methods in *javax.ejb.EJBHome* when it implements your home interface as a concrete home object (see Table E.2).

```
public interface javax.ejb.EJBHome
    extends java.rmi.Remote
{
    public EJBMetaData getEJBMetaData()
        throws java.rmi.RemoteException;

    public void remove(Handle handle)
        throws java.rmi.RemoteException,
            javax.ejb.RemoveException;

    public void remove(Object primaryKey)
        throws java.rmi.RemoteException,
            javax.ejb.RemoveException;

    public javax.ejb.HomeHandle getHomeHandle()
        throws java.rmi.RemoteException;
}
```

Table E.2 javax.ejb.EJBHome

METHOD	EXPLANATION
<code>getEJBMetaData()</code>	Returns metadata about the enterprise bean you're working with. Useful if your client code is written in a scripting language, or if you're writing EJB development tools.
<code>getHomeHandle()</code>	Retrieves a serializable handle to the bean's home object. You can tuck this handle away somewhere (such as writing it to disk) and then use it again later to retrieve the home without performing a JNDI lookup.
<code>remove()</code>	This method destroys an EJB object based upon an EJB object handle or primary key you pass in.

Note: These methods are called by remote clients; for entity beans, `remove()` also deletes the bean from the underlying persistent store.

EJBLocalHome

Local clients create, find, and remove local EJB objects through local home interfaces. Your local home interfaces extend *javax.ejb.EJBLocalHome*. The container will implement the methods in *javax.ejb.EJBLocalHome* when it implements your local home interface as a concrete local home object (see Table E.3).

```
public interface javax.ejb.EJBLocalHome
{
    public void remove(Object primaryKey)
        throws javax.ejb.EJBException,
            javax.ejb.RemoveException;
}
```

Table E.3 javax.ejb.EJBLocalHome

METHOD	EXPLANATION
remove()	This method destroys an EJB local object based upon a primary key you pass in. Applies only to entity beans. This will also delete the bean data from the underlying persistent store.

Note: These methods are called by local clients.

EJBLocalObject

A local client accesses a bean through an EJB local object, which implements a local interface. Your local interface must extend *javax.ejb.EJBLocalObject*. The container will implement the methods in *javax.ejb.EJBLocalObject* when it implements your local interface as a concrete EJB local object (see Table E.4).

```
public interface javax.ejb.EJBLocalObject
{
    public javax.ejb.EJBLocalHome getEJBLocalHome()
        throws javax.ejb.EJBException;

    public java.lang.Object getPrimaryKey()
        throws javax.ejb.EJBException;

    public void remove()
        throws javax.ejb.EJBException,
            javax.ejb.RemoveException;
}
```

```

    public boolean isIdentical(javax.ejb.EJBObject)
        throws java.rmi.RemoteException;
}

```

Table E.4 javax.ejb.EJBObject

METHOD	EXPLANATION
getEJBLocalHome()	Gets the local home object for this EJB local object.
getPrimaryKey()	Returns the primary key for this EJB local object. A primary key is only used for entity beans (see Chapter 5).
remove()	Destroys this EJB local object. When your client code is done using an EJB local object, you should call this method. The system resources for the EJB local object can then be reclaimed.
isIdentical()	Tests whether two EJB local objects are identical.

Note: For entity beans, `remove()` also deletes the bean from the underlying persistent store.

EJBMetaData

This interface encapsulates metadata about an enterprise bean. Metadata is not very useful for typical client code; it is more suited towards clients that need to dynamically discover information about an enterprise bean, such as scripting languages or EJB development tools. Your client code can retrieve this metadata by calling `homeObject.getEJBMetaData()`. The client code will get back a serializable implementation of `javax.ejb.EJBMetaData`.

```

public interface javax.ejb.EJBMetaData
{
    public javax.ejb.EJBHome getEJBHome();
    public java.lang.Class getHomeInterfaceClass();
    public java.lang.Class getPrimaryKeyClass();
    public java.lang.Class getRemoteInterfaceClass();
    public boolean isSession();
    public boolean isStatelessSession();
}

```

EJBObject

A remote client accesses a bean through an EJB object, which implements a remote interface. Your remote interface must extend `javax.ejb.EJBObject`. The container will implement the methods in `javax.ejb.EJBObject` when it implements your remote interface as a concrete EJB object (see Table E.5).

```

public interface javax.ejb.EJBObject
    extends java.rmi.Remote
{
    public javax.ejb.EJBHome getEJBHome()
        throws java.rmi.RemoteException;

    public java.lang.Object getPrimaryKey()
        throws java.rmi.RemoteException;

    public void remove()
        throws java.rmi.RemoteException,
            javax.ejb.RemoveException;

    public javax.ejb.Handle getHandle()
        throws java.rmi.RemoteException;

    public boolean isIdentical(javax.ejb.EJBObject)
        throws java.rmi.RemoteException;
}

```

Table E.5 javax.ejb.EJBObject

METHOD	EXPLANATION
getEJBHome()	Gets the home object for this EJB object.
getPrimaryKey()	Returns the primary key for this EJB object. A primary key is used only for entity beans (see Chapters 5–7).
remove()	Destroys this EJB object. When your client code is done using an EJB object, call this method. The system resources for the EJB object can then be reclaimed.
getHandle()	Acquires a <i>handle</i> for this EJB object. An EJB handle is a persistent reference to an EJB object that the client can stow away somewhere. Later, the client can use the handle to reacquire the EJB object and start using it again.
isIdentical()	Tests whether two EJB objects are identical. Should be used instead of equals() or the == operator, since those test whether two stubs are the same—stubs are not EJB objects.

Note: For entity beans, remove() also deletes the bean from the underlying persistent store.

EnterpriseBean

This interface serves as a marker interface; implementing this interface indicates that your class is indeed an enterprise bean class. You should not implement this interface; rather, implement either javax.ejb.EntityBean, javax.ejb.SessionBean, or javax.ejb.MessageDrivenBean, each of which extends this interface.

```
public interface javax.ejb.EnterpriseBean
    extends java.io.Serializable
{
}
```

EntityBean

To write an entity bean class, your class must implement the *javax.ejb.EntityBean* interface. This interface defines a few required methods that you must fill in. These are management methods that the EJB container calls to alert your bean to life-cycle events. Clients of your bean will never call these methods, because these methods are not made available to clients via the EJB object (see Table E.6). Each of the following methods can throw a *java.rmi.RemoteException* or *javax.ejb.EJBException*.

```
public interface javax.ejb.EntityBean
    implements javax.ejb.EnterpriseBean
{
    public void setEntityContext(javax.ejb.EntityContext);
    public void unsetEntityContext();
    public void ejbRemove();
    public void ejbActivate();
    public void ejbPassivate();
    public void ejbLoad();
    public void ejbStore();
}
```

Table E.6 Required Methods for Entity Bean Classes

METHOD	DESCRIPTION	TYPICAL IMPLEMENTATION (BEAN-MANAGED PERSISTENT ENTITIES)	TYPICAL IMPLEMENTATION (CONTAINER-MANAGED PERSISTENT ENTITIES)
SetEntityContext (EntityContext ctx)	Associates your bean with an entity context. You can query the entity context about your current transactional state, your current security state, and more.	Store the context away in a member variable so the context can be queried later.	Store the context away in a member so the context can be queried later.
ejbFind... (...) <i>Note: You only use ejbFind() methods with bean- managed persistent entity beans.</i>	Finds an existing entity bean in storage. You can have many different finder methods, which all perform different operations.	Search through a data store using a storage API such as JDBC or SQL/J. For example, you might perform a relational query such as SELECT id FROM accounts WHERE balance > 0. Return the resulting primary key set.	Do not implement these methods for container-managed persistent entity beans. The EJB container will handle <i>all</i> issues relating to finding data for you. Use your container tools to describe your finder method needs.
ejbSelect... (...) <i>Note: You only use ejbSelect() methods with container- managed persistent entity beans.</i>	ejbSelect() methods are CMP helper methods that perform queries internally by your bean but that are not accessible to clients of your bean.	There are no ejbSelect() methods for BMP entity beans. You can write your own helper methods and name them whatever you wish.	Define this method as abstract. Then write EJB-QL in the deployment descriptor to set up the query.

continued

Table E.6 Required Methods for Entity Bean Classes (*continued*)

METHOD	DESCRIPTION	TYPICAL IMPLEMENTATION	
		(BEAN-MANAGED PERSISTENT ENTITIES)	(CONTAINER-MANAGED PERSISTENT ENTITIES)
<code>ejbHome ... (...)</code>	Sometimes you need methods on an entity bean that are not specific to any given data instance (or row)—for example, counting the total number of accounts in a table. You can write <code>ejbHome</code> methods to perform these operations. The <code>ejbHome</code> methods are special business methods because they are called from a bean in the pool before the bean is associated with any specific data. Clients call these methods from the home interface or local home interface.	Perform your global operations, such as counting the rows in a database and returning the results to the client. The fast-and-easy way to achieve this is to use JDBC. The cleaner (but lower-performing way if you're not careful) is to call <code>ejbSelect()</code> and perhaps other entity bean methods.	Perform your global operations, such as counting the rows in a database and returning the results to the client. The fast-and-easy way to achieve this is to use JDBC. The cleaner (but lower-performing way if you're not careful) is to call <code>ejbSelect()</code> and perhaps other entity bean methods.
<code>ejbCreate ... (...)</code>	Initializes a bean for a particular client and creates underlying database data. Each <code>ejbCreate()</code> method you define gives clients a different way to create your entity beans.	Validate the client's initialization parameters. Explicitly create the database representation of the data via a storage API such as JDBC or SQL/J.	Validate the client's initialization parameters. Call your abstract <code>set()</code> methods to initialize the generated bean subclass to the parameters passed in. The container will then use these values in the subclass to create the database data for you.

	TYPICAL IMPLEMENTATION (BEAN-MANAGED)		TYPICAL IMPLEMENTATION (CONTAINER-MANAGED)	
METHOD	DESCRIPTION	PERSISTENT ENTITIES	PERSISTENT ENTITIES	
ejbPostCreate (... (...))	Your bean class must define <code>ejbPostCreate()</code> for each one <code>ejbCreate()</code> . Each pair must accept the same parameters. The container calls <code>ejbPostCreate()</code> right after <code>ejbCreate()</code> .	Perform any initialization you need to that requires a reference to your own EJB object, such as passing your bean's EJB object reference to other beans. You can get your EJB object via <code>entityContext.getEJBObject()</code> .	Perform any initialization you need to that requires a reference to your own EJB object, such as passing your bean's EJB object reference to other beans. You can get your EJB object via <code>entityContext.getEJBObject()</code> .	
ejbPassivate()	Called immediately before your bean is passivated (swapped out to disk because too many beans are instantiated).	Release any resources your bean may be holding.	Release any resources your bean may be holding.	
ejbStore()	Called when the container needs to update the database with your bean's state. The current transactional state dictates when this method is called. This method is also called during passivation, directly before <code>ejbPassivate()</code> .	Explicitly update the database representation of the data via a storage API such as JDBC. Typically, you'll write a number of your member variable's fields out to disk.	Do not update the database in this method. Rather, the EJB container will update the database for you automatically right after calling your <code>ejbStore()</code> method. It does this in the subclass by taking your bean state and writing it to the database. Thus, you should prepare your state to be written to the database, such as compressing fields, by calling your abstract <code>get/set</code> methods.	<i>continued</i>

Table E.6 Required Methods for Entity Bean Classes (*continued*)

METHOD	DESCRIPTION	TYPICAL IMPLEMENTATION	
		(BEAN-MANAGED PERSISTENT ENTITIES)	(CONTAINER-MANAGED PERSISTENT ENTITIES)
<code>ejbLoad()</code>	Called when the container needs to update your bean with the database's state. The current transactional state dictates when this method is called. This method is also called during activation, directly before <code>ejbActivate()</code> .	First your bean instance must figure out what data to load. Call the <code>getPrimaryKey()</code> method on the entity context; that will tell your bean what data it should load. Next read database data into your bean via a storage API such as JDBC or SQL/J.	<i>Do not read data from the database in this method.</i> The EJB container will read in data from the database for you automatically right before calling your <code>ejbLoad()</code> method. It does this in the subclass by querying the database and populating your bean state. Thus, you should perform any necessary post-load operations, such as decompressing fields, by calling your abstract <code>get/set</code> methods.
<code>ejbActivate()</code>	Called immediately before your bean is activated (swapped in from disk because a client needs your bean).	Acquire any resources your bean needs, such as those released during <code>ejbPassivate()</code> .	Acquire any resources your bean needs, such as those released during <code>ejbPassivate()</code> .

continued

Table E.6 Required Methods for Entity Bean Classes (continued)

METHOD	DESCRIPTION	TYPICAL IMPLEMENTATION (BEAN-MANAGED PERSISTENT ENTITIES)	TYPICAL IMPLEMENTATION (CONTAINER-MANAGED PERSISTENT ENTITIES)
ejbRemove()	To destroy an entity bean's data in a database, the client must call <code>remove()</code> on the EJB object or home object. This method causes the container to issue an <code>ejbRemove()</code> call on the bean. <code>ejbRemove()</code> is a required method of all beans and takes no parameters. <i>Note: <code>ejbRemove()</code> does not mean the in-memory entity bean instance is going to be destroyed. <code>ejbRemove()</code> destroys only database data. The bean instance can be recycled to handle different database data, such as a bank account bean representing different bank accounts.</i>	First figure out what data you should destroy via <code>getPrimaryKey()</code> on the <code>EntityContext</code> . Then explicitly delete the database representation of the data via a storage API such as JDBC or SQL/J.	<i>Do not destroy database data in this method.</i> Simply perform any operations that must be done before the data in the database is destroyed. The EJB container will destroy the data for you right after <code>ejbRemove()</code> is called.
unsetEntityContext()	Called right before your entity bean instance is destroyed (when the container wants to reduce the pool size).	Release any resources you allocated during <code>setEntityContext()</code> and get ready for garbage collection.	Release any resources you allocated during <code>setEntityContext()</code> , and get ready for garbage collection.

EntityContext

An *entity context* is a specific EJB context used only for entity beans (see Table E.7).

```
public interface javax.ejb.EntityContext implements
    javax.ejb.EJBContext
{
    public javax.ejb.EJBObject getEJBLocalObject()
        throws IllegalStateException;

    public javax.ejb.EJBObject getEJBObject()
        throws IllegalStateException;

    public java.lang.Object getPrimaryKey();
        throws IllegalStateException;
}
```

Handle

An EJB object handle is a persistent reference to an EJB object. Handles allow you to disconnect from your EJB server, shut down your application, and later resume your application while preserving the conversational state in the beans you've been working with. Home handles are also useful when your client

Table E.7 javax.ejb.EntityContext

METHOD	DESCRIPTION	USEFULNESS
getEJBLocalObject()	Returns a reference to your bean's own local EJB object.	Useful if your bean needs to call another local bean and you want to pass a reference to yourself.
getEJBObject()	Returns a reference to your bean's own remote EJB object.	Useful if your bean needs to call another remote bean and you want to pass a reference to yourself.
getPrimaryKey()	Retrieves the primary key that is currently associated with this entity bean instance.	Call to determine what database data your instance is associated with. You need to use this in <i>ejbLoad()</i> to determine what database data to load and in <i>ejbRemove()</i> to determine what database data to remove.

code needs to store a reference to an EJB object in stable storage and reconnect to that EJB object later.

```
public interface javax.ejb.Handle
{
    public javax.ejb.EJBObject getEJBObject();
}
```

HomeHandle

Just as an EJB object handle is a persistent reference to an EJB object, a home handle is a persistent reference to a home object. Home handles are useful when your client code needs to store a reference to a home object in stable storage and reconnect to that home object later. Home handles allow you to avoid doing a JNDI lookup when reconnecting to a home object.

```
public interface javax.ejb.HomeHandle
{
    public javax.ejb.EJBHome getEJBHome();
}
```

MessageDrivenBean

To write a message-driven bean class, your class must implement the *javax.ejb.MessageDrivenBean* interface. This interface defines a few required methods that you must fill in. These are management methods that the EJB container calls to alert your bean about life-cycle events. Clients of your bean will never call these methods because clients do not call message-driven beans directly; rather, they send messages using the Java Message Service (JMS). Each of these methods can throw a *javax.ejb.EJBException* (see Table E.8).

```
public interface javax.ejb.MessageDrivenBean
    extends javax.ejb.EnterpriseBean
{
    public void setMessageDrivenContext(MessageDrivenContext ctx);
    public void ejbRemove();
}
```

MessageDrivenContext

A *message-driven context* is a specific EJB context used only for message-driven beans. This interface serves as a *marker* interface. There are no specific additional methods that message-driven beans get on their context objects.

```
public interface javax.ejb.MessageDrivenContext
    extends javax.ejb.EJBContext
{
}
```

Table E.8 Required Methods for Message-Driven Bean Classes

METHOD	DESCRIPTION	TYPICAL IMPLEMENTATION
setMessageDrivenContext (MessageDrivenContext ctx)	Associates your bean with a message-driven context. Your bean can query the context about its current transactional state, retrieve its own home object, and more.	Store the context away in a member variable so the context can be queried later.
ejbCreate()	Initializes your message-driven bean. You can only define a single <code>ejbCreate()</code> method that takes no parameters. This is called directly after you are associated with a context object.	Perform any initialization your bean needs, such as locating external resources or looking up other EJB home objects to be used later.
onMessage (message)	Your bean has received a new message.	Crack open the message, figure out what it means to you, process the message, and perform any logic you desire.
ejbRemove()	Called by the container immediately before your bean is removed from memory.	Prepare your bean for destruction. Free all resources you may have allocated.

SessionBean

To write a session bean class, your class must implement the *javax.ejb.SessionBean* interface. This interface defines a few required methods that you must fill in. These are management methods that the EJB container calls to alert your bean about life-cycle events. Clients of your bean will never call these methods because these methods are not made available to clients via the EJB object (see Table E.9). Each of these methods can throw a *java.rmi.RemoteException* or *javax.ejb.EJBException*.

```
public interface javax.ejb.SessionBean
    extends javax.ejb.EnterpriseBean
{
    public void setSessionContext(SessionContext ctx);
    public void ejbPassivate();
    public void ejbActivate();
    public void ejbRemove();
}
```

Table E.9 Required Methods for Session Bean Classes

METHOD	DESCRIPTION	TYPICAL IMPLEMENTATION (STATEFUL SESSION BEANS)	TYPICAL IMPLEMENTATION (STATELESS SESSION BEANS)
setSessionContext (SessionContext ctx)	Associates your bean with a session context. Your bean can query the context about its current transactional state, current security state, and more.	Store the context away in a member variable so the context can be queried later.	Store the context away in a member variable so the context can be queried later.
ejbCreate ... (...)	Initializes your session bean.	Perform any initialization your bean needs, such as setting member variables to the argument values passed in. Note: You can define several <i>ejbCreate ... (...)</i> methods, and each can take different arguments. You must provide at least one <i>ejbCreate ... (...)</i> method in your session bean.	Perform any initialization your bean needs, such as setting member variables to the argument values passed in. Note: You can only define a single empty <i>ejbCreate()</i> method with no parameters. After all, if it had parameters and the bean initialized itself to those parameters, the bean would never remember what it initialized itself to upon subsequent calls, since it is stateless!
ejbPassivate()	Called immediately before your bean is passivated (swapped out to disk because there are too many instantiated beans).	Release any resources your bean may be holding.	Unused because there is no conversational state; leave empty.
ejbActivate()	Called immediately before your bean is activated (swapped in from disk because a client needs your bean).	Acquire any resources your bean needs, such as those released during <i>ejbPassivate()</i> .	Unused because there is no conversational state; leave empty.
ejbRemove()	Called by the container immediately before your bean is removed from memory.	Prepare your bean for destruction. Free all resources you may have allocated.	Prepare your bean for destruction. Free all resources you may have allocated.

Table E.10 javax.ejb.SessionContext

METHOD	DESCRIPTION	USEFULNESS
getEJBLocalObject()	Returns a reference to your bean's own local EJB object.	Useful if your bean needs to call another local bean and you want to pass a reference to yourself.
getEJBObject()	Returns a reference to your bean's own EJB object.	Useful if your bean needs to call another bean, and you want to pass a reference to yourself.

SessionContext

A *session context* is a specific EJB context used only for session beans (see Table E.10).

```
public interface javax.ejb.SessionContext
    extends javax.ejb.EJBContext
{
    public javax.ejb.EJBLocalObject getEJBLocalObject()
        throws IllegalStateException;

    public javax.ejb.EJBObject getEJBObject()
        throws IllegalStateException;
}
```

SessionSynchronization

If your stateful session bean is caching database data in memory or needs to roll back in-memory conversational state upon a transaction abort, you should implement this interface (see Table E.11). The container will call each of the methods in this interface automatically at the appropriate times during transactions, alerting you to important transactional events. Each of these methods can throw a *java.rmi.RemoteException* or *javax.ejb.EJBException*.

```
public interface javax.ejb.SessionSynchronization
{
    public void afterBegin();
    public void beforeCompletion();
    public void afterCompletion(boolean);
}
```

Table E.11 javax.ejb.SessionSynchronization

METHOD	DESCRIPTION
afterBegin()	Called by the container directly after a transaction begins. You should read in any database data you want to cache in your stateful session bean during the transaction. You should also create a backup copy of your state in case the transaction rolls back.
beforeCompletion()	Called by the container right before a transaction completes. Write out any database data you've cached during the transaction.
afterCompletion (boolean)	Called by the container when a transaction completes either in a commit <i>or</i> an abort. True indicates a successful commit; false indicates an abort. If an abort happened, revert to the backup copy of your state to preserve your session bean's conversation.

Exception Reference

Table E.12 describes the purpose of each exception class in EJB.

Table E.12 EJB Exception Explanations

EXCEPTION	DESCRIPTION
AccessLocalException	Indicates the client does not have permission to call this method. Used only for local clients.
CreateException	This exception type indicates failure to create an enterprise bean. You should throw this exception in your home interface's <i>create(...)</i> methods.
DuplicateKey Exception	This exception type indicates failure to create an entity bean because an entity bean with the same primary key already exists. This is a subclass of <i>CreateException</i> and is thrown in an entity bean home interface's <i>create(...)</i> methods.
EJBException	Your enterprise bean class should throw this exception to indicate an unexpected error, such as a failure to open a database connection, or a JNDI exception. Your container treats this exception as a serious problem and may take action such as logging the event or paging a system administrator, depending upon your container's policy. The container then rethrows a <i>java.rmi.RemoteException</i> if the client is remote, or a <i>javax.ejb.EJBException</i> if the client is local. Since <i>EJBException</i> is a <i>RuntimeException</i> , it does not need to be declared in <i>throws</i> clauses.
FinderException	This exception indicates a failure to locate an existing entity bean. You should throw this method from your home interface's finder methods.
NoSuchEntity Exception	Your entity bean class should throw this exception to indicate that the database data corresponding to the in-memory entity bean instance has been removed. You can throw this exception from any of your entity bean class' business methods, and from your <i>ejbStore</i> and <i>ejbLoad</i> methods.
NoSuchObjectLocal Exception	Thrown when a client tries to call a method on a bean that has been destroyed.
ObjectNotFound Exception	When you're writing finder methods in your entity bean's home interface, throw this exception to indicate that the specified EJB object was not found. You should use this exception only when your finder method is returning a single EJB object. If you're returning more than one EJB object, a null collection is returned instead.

continued

Table E.12 EJB Exception Explanations (*continued*)

EXCEPTION	DESCRIPTION
RemoveException	Your enterprise bean should throw this exception when an error occurs during <i>ejbRemove()</i> . The container will rethrow this exception back to the client. This is considered a normal, run-of-the-mill application-level exception and does not indicate a system-level problem. When your client code receives this exception, you do not know for sure whether the entity bean has been removed or not.
TransactionRequiredLocalException	Indicates the receiving object needed a transaction to run but was not supplied with a transaction context.
TransactionRolledBackLocalException	Indicates the request's transaction was rolled back or marked for rollback. Thus, the request could not be completed.

Transaction Reference

The following section offers reference information on transactions as outlined in Tables E.13 through E.18.

Table E.13 The Effects of Transaction Attributes

TRANSACTION ATTRIBUTE	CLIENT'S TRANSACTION	BEAN'S TRANSACTION
Required	none T1	T2 T1
RequiresNew	none T1	T2 T2
Supports	none T1	none T1
Mandatory	none T1	error T1
NotSupported	none T1	none none
Never	none T1	none error

Table E.14 Transaction Attributes

CONSTANT	MEANING
<i>NotSupported</i>	Your bean <i>cannot</i> be involved in a transaction at all. When a bean method is called, any existing transaction is suspended.
<i>Never</i>	Your bean <i>cannot</i> be involved in a transaction at all. When a bean method is called, if a transaction is in progress, an exception is thrown back to the client (<i>java.rmi.RemoteException</i> if remote, <i>javax.ejb.EJBException</i> if local).
<i>Required</i>	Your bean must <i>always</i> run in a transaction. If a transaction is already running, your bean joins that transaction. If no transaction is running, the EJB container starts one for you.
<i>RequiresNew</i>	Your bean must always run in a <i>new</i> transaction. Any current transaction is suspended.
<i>Supports</i>	If a transaction is already underway, your bean joins that transaction. Otherwise, the bean runs with no transaction at all.
<i>Mandatory</i>	Mandates that a transaction <i>must be already running</i> when your bean method is called, or a <i>javax.ejb.TransactionRequired</i> exception is thrown back to the caller.

Table E.15 Permissible Transaction Attributes for Each Bean Type

TRANSACTION ATTRIBUTE	STATEFUL SESSION BEAN			
	STATELESS SESSION BEAN	IMPLEMENTING SESSION SYNCHRONIZATION	ENTITY BEAN	MESSAGE- DRIVEN BEAN
Required	Yes	Yes	Yes	Yes
RequiresNew	Yes	Yes	Yes	No
Mandatory	Yes	Yes	Yes	No
Supports	Yes	No	No	No
NotSupported	Yes	No	No	Yes
Never	Yes	No	No	No

Table E.16 Transaction Isolation Levels

ISOLATION LEVEL	DIRTY READS?	UNREPEATABLE READS?	PHANTOM READS?
<i>READ UNCOMMITTED</i>	Yes	Yes	Yes
<i>READ COMMITTED</i>	No	Yes	Yes
<i>REPEATABLE READ</i>	No	No	Yes
<i>SERIALIZABLE</i>	No	No	No

Table E.17 The *javax.transaction.Status* Constants for Transactional Status

CONSTANT	MEANING
STATUS_ACTIVE	A transaction is currently happening and is active.
STATUS_NO_TRANSACTION	No transaction is currently happening.
STATUS_MARKED_ROLLBACK	The current transaction will eventually abort because it's been marked for rollback. This could be because some party called <i>setRollbackOnly()</i> .
STATUS_PREPARING	The current transaction is preparing to be committed (during Phase One of the two-phase commit protocol).
STATUS_PREPARED	The current transaction has been prepared to be committed (Phase One is complete).
STATUS_COMMITTING	The current transaction is in the process of being committed right now (during Phase Two).
STATUS_COMMITTED	The current transaction has been committed (Phase Two is complete).
STATUS_ROLLING_BACK	The current transaction is in the process of rolling back.
STATUS_ROLLEDBACK	The current transaction has been rolled back.
STATUS_UNKNOWN	The status of the current transaction cannot be determined.

Table E.18 The `javax.transaction.UserTransaction` Methods for Transactional Boundary

METHOD	DESCRIPTION
<code>begin()</code>	Begin a new transaction. This transaction becomes associated with the current thread.
<code>commit()</code>	Run the two-phase commit protocol on an existing transaction associated with the current thread. Each resource manager will make its updates durable.
<code>getStatus()</code>	Retrieve the status of the transaction associated with this thread.
<code>rollback()</code>	Force a rollback of the transaction associated with the current thread.
<code>setRollbackOnly()</code>	Call this to force the current transaction to roll back. This will eventually force the transaction to abort. One interesting use of this is to test out what your components will do without having them perform any permanent resource updates.
<code>setTransactionTimeout(int)</code>	The <i>transaction timeout</i> is the maximum amount of time that a transaction can run before it's aborted. This is useful to avoid deadlock situations, when precious resources are being held by a transaction that is currently running.

SYMBOLS

2PC (two-phase commit protocol), 318
 3-tier architecture, 415
 4-tier architecture, 415–416

A

abort statements,
 transactions, 287
 abort() method, 250
 aborted transactions,
 282–283
 abstract entities, 41
 abstract methods, 171
 abstract persistence, 172
 abstract roles, 260–261
 abstract schema
 names, 173
 account remote interface,
 138, 143
 account.java interface, 137
 AccountBean.java,
 143–154
 AccountException.java
 class, 156
 AccountExceptions, 143
 AccountHome.java
 interface, 138
 AccountLocal.java local
 interface, 138
 AccountLocalHome.java
 interface, 138

AccountPK.java, 139
 ACID (atomicity,
 consistency, isolation,
 and durability)
 properties,
 transactions, 280
 acknowledgement modes,
 deployment
 descriptors, 293
 actions, 250
 activation, 88
 entity bean
 instances, 117
 stateful session
 beans, 85
 administrative support,
 EJB servers, 456
 administrator's role,
 beans, 260
 AfterBegin() method, 321
 AfterCompletion()
 method, 321–322
 aggregation relationships,
 350
 aggressive loading, entity
 beans, 349
 alerts, session beans, 321
 algorithms, caching, 426
 all or nothing paradigm,
 transactions, 281–283
 APIs, JMS, 203, 206
 application assemblers,
 EJB Ecosystem, 17–18

application servers, 5, 442
 application-level
 exceptions, 60–61
 applications
 context switching, 85
 dynamic discovery, 395
 logic components, 109
 APS (Arrivals Per
 Second), 414
 architecture
 components, 12
 JAAS, 249–250
 assembly-descriptor,
 entity bean
 deployment
 descriptor, 161
 association tables, many-
 to-many relationships,
 336
 asynchronous method
 invocations, 235
 asynchronous
 programming,
 message-driven
 beans, 202
 atomicity, transactions,
 276, 279–281
 attributes, transactions,
 292, 298
 declaring, 294
 Mandatory, 296–297
 Never, 297

- attributes, transactions,
 - (Continued)
 - NotSupported, 297
 - Required, 294–295, 400
 - RequiresNew, 296
 - Supports, 296
- audit logs, object changes, 378
- auditing, 6
- authentication
 - bean security, 246–248
 - security identities, 246
- authorization
 - declarative, 262
 - instance level, 266
 - JAAS, 257
 - programmatic, 258
- auto-component
 - generation, EJB servers, 457
- automatic fail-over,
 - stateful session beans, 425
- automatic session fail-over, 414
- avoiding stored procedures, 373
- B**
- back-end integration, 6
- bank account client, 161–163
- bank account
 - database, 161
- bank account entity bean, 136–139, 143, 159–160
- base prices, 5
- basic authentication, 248
- bean class, 62
- bean instance pool, 397
- bean providers, EJB Ecosystem, 17
- bean security,
 - authentication, 246–248
- bean-managed persistent
 - entity beans, 127, 163
- bean-managed state
 - fields, 143
- bean-managed
 - transactions, 306
- beans
 - activation, 88
 - administrator's role, 260
 - calling, 69
 - JNDI lookups, 238
 - other beans, 237
 - conversational states, 86
 - counter, 89
 - client code, 96
 - client-side output, 99
 - deployment
 - descriptor, 97–98
 - home interface, 94
 - proprietary
 - descriptor, 96
 - remote interface, 91–92
 - running client, 99
 - server-side output, 100
 - session context
 - interface, 94
 - val member variable, 91
- declarative
 - authorization, 262
- delegating home objects, 135
- deploying in
 - containers, 68
- enterprise, client
 - conversations, 82
- entity, 105, 119
- failure, 394
- message driven,
 - 201–202, 212
- method permissions, 262
- passivation, 87
- programmatic
 - authorization, 258
- security
 - policies, 257, 262–263
 - propagation, 267
 - roles, 258–264
- session, 81–82
- stateful session, 82, 85
- stateless session, 83–84
- threads, 210
- BeforeCompletion()
 - method, 321
- begin statements,
 - transactions, 287
- begin() method, 291, 305
- behavior, versioning, 379
- best practices
 - hard-coded SQL, 370
 - object-to-relational mapping, 370, 375
 - persistence, 361–365
 - soft-coded SQL, 370
 - stored procedures, 371–372
- bidirectional relationships,
 - entity beans, 344–347

- BluePrints document, J2EE, 23
- BMP entity beans
 - bidirectional relationships, 344–345
 - cascading deletes, 351
 - lazy loading, 349
 - many-to-many relationships, 338–339
 - one-to-many relationships, 333–334
 - one-to-one relationships, 328–329
 - persistence, 366–368
 - referential integrity, 356
 - true many-to-many relationships, 341–342
- boundaries, transactions, 304
- building components, 55–56
- bundling discounts, 7
- business interface, 76
- business logic
 - methods, 143
 - tier, Jasmine's Computer Parts, 467–475
- C**
- caching, 7, 426
- CallHelloWorld.java, 256
- calling beans, 69
- cardinality, relationships, 326
- cart line items, Jasmine's Computer Parts, 472
- cascading deletes, 351
- Catalog JSP, 480
- CDATA section, CMP entity bean deployment descriptors, 194
- centralized naming servers, 430
- certificate authentication, 248
- chained transactions, 286
- circular relationships, 354
- class keyword, 14
- classes
 - AccountException.java, 156
 - bean, 62
 - enterprise bean, 37
 - entity bean, 112, 127
 - java.lang.Exception class, 156
 - javax.jms.Queue Requestor, 235
 - javax.jms.Topic Requestor, 235
 - ProductPK, 184, 195
- clean shutdowns, 6, 457
- client code
 - counter bean, 96
 - transactions, 306
- client-controlled transactions, 292
- client program
 - CMP entity beans, 196
 - message-driven beans, 223
- client-initiated transactions, 290
- client-side components, 15
- client-side output
 - bank account program, 163
 - HelloClient program, 75
- client-side proxy objects, 32
- client.java, 156–159, 196–198
- clients
 - bank account, 161
 - CORBA, 70
 - counter bean, 99
 - RMI-IIOP, 70
 - stateful session bean conversations, 85
- clustering, 6, 411–413
 - EJBs, 419, 455
 - entity beans, 425–426
 - J2EE, naming servers, 430
 - message-driven beans, 429
 - partitions, 227, 415
 - stateless session beans, 421
- CMP (Container-Managed Persistence) entity beans, 167
- abstract persistence schema, 172
- bidirectional relationships, 345–347
- client program, 196
- composition relationships, 351
- custom primary key classes, 195

- CMP entity beans,
 - (Continued)
 - dependent value
 - classes, 369
 - deployment
 - descriptors, 191, 195
 - ejbSelect() methods,
 - 175–176
 - finder methods, 173
 - lack of declared fields,
 - 168–169
 - lazy loading, 350
 - life cycle, 200
 - many-to-many
 - relationships,
 - 339–340
 - one-to-many
 - relationships, 335
 - one-to-one
 - relationships,
 - 330–332
 - persistence, 176
 - application
 - server/database
 - independence,
 - 367
 - bugs, 366
 - code reduction, 365
 - control, 367
 - fields, 170
 - learning curve, 368
 - performance, 366
 - product line
 - example, 180
 - relationship
 - features, 368
 - queries, 173–174
 - referential integrity, 356
 - subclasses, 167–170
 - superclass, 171
 - true many-to-many
 - relationships,
 - 342–343
 - cmp-field, CMP entity
 - beans, 173
 - coding interfaces, 57–59
 - collections
 - EJB objects, 135
 - primary keys, 135
 - relationships, 357–358
 - returning via finder
 - methods, 135
 - commit statements,
 - transactions, 287
 - Commit() method, 250,
 - 291, 305
 - committed
 - transactions, 282
 - compatibility test suite,
 - EJB servers, 450
 - complex mappings, EJB
 - servers, 451
 - component
 - architecture, 12
 - component assembly, EJB
 - best practices, 405
 - component interface, 47,
 - 75–76
 - components, 5
 - accessing via local
 - interfaces, 241
 - application logic, 109
 - building, 55–56
 - client-side, 15
 - implicit location
 - transparency, 40
 - persistent data, 110
 - pricing, 5
 - reusable, 9, 406
 - server-side, 16
 - composition
 - relationships, 351
 - concurrency control,
 - 308, 316
 - connection pooling,
 - 147, 242
 - ConnectionFactory,
 - 206, 430
 - connections, JMS, 206
 - constants,
 - transactions, 302
 - consumers, JMS, 206–207
 - container-managed
 - transactions, 292, 306
 - container providers, EJB
 - Ecosystem, 19
 - container-specific
 - deployment
 - descriptors, 195
 - container-specific
 - deployment
 - descriptors, 161
 - containers
 - bean management, 41
 - EJB clustering, 419
 - glue-code tools, 41
 - implicit monitoring, 40
 - instance pooling, 43
 - request interception, 38
 - swapping out stateful
 - session beans, 85
 - context objects, 64–65
 - context switching, 85
 - contexts, transactions, 319
 - conversational states,
 - 86, 320
 - conversations
 - enterprise beans and
 - clients, 82
 - stateful session beans
 - and clients, 85

- transactions, 320
- conversion tools, J2EE, 450
- CORBA
 - clients, 70
 - interoperability,
 - 523–531, 534, 539, 543
 - OTS, 70, 300
- COS Naming (CORBA Naming Service), 70
- CosTransactions interface, OTS, 301
- CosTSPortability interface, OTS, 301
- Count() method, 91, 322
- counter beans, 89
 - client code, 96
 - client-side output, 99
 - deployment descriptor, 97–98
 - home interface, 94
 - proprietary
 - descriptor, 96
 - remote interface, 91–92
 - running client, 99
 - server-side output, 100
 - session context
 - interface, 94
 - val member variable, 91
- Create() method, 58, 62, 162, 165
- CreateExceptions, 184
- cross-platform
 - compatibility, 14
- CurrentVersion
 - column, 378
- custom exception
 - classes, 156
- custom primary key
 - classes, CMP entity beans, 195
- customer abstractions, 470
- customizing beans,
 - environment
 - properties, 244
- D**
- data denormalization , 374
- data design process, 375
- data normalization, 373–374
- data transfers, entity beans, 113
- database updates, 377
- databases
 - bank account, 161
 - dirty reads, 311
 - locks, 309
 - objects, 109
 - phantoms, 313–314
 - schema mapping,
 - directionality, 344, 347
 - directionality
 - unrepeatable reads, 312–313
- DataSource objects, 430
- DDL (Data Definition Language), 161
- deadlock,
 - transactions, 310
- debugged libraries, 14
- debugging, EJB best practices, 402–403
- declarative
 - authorization, 262
- declarative middleware.
 - See implicit middleware.
- declarative transactions, 288–289
 - advantages, 292
- deployment
 - descriptors, 289
 - writing, 304
- declared fields, CMP
 - entity beans, 168–169
- declaring environment
 - properties, 245
- default initial contexts, JNDI, 238
- deferred database writes, EJB servers, 451
- delegating home objects to beans, 135
- demarcating transactional boundaries
 - declarative
 - transactions, 288–289
 - programmatic
 - transactions, 287
- denormalization of data, 374
- dependent value classes, CMB entity beans, 369
- deployment descriptors, 50, 53, 66, 545, 549, 553, 558, 562, 565
 - acknowledgement
 - modes, 293
 - bank account entity bean, 159–160
 - bean requirement specs, 50
 - CMP entity beans, 172–173, 191, 195
 - container-specific, 161
 - counter bean, 97–98
 - declarative
 - transactions, 289
 - entity beans, 156

- deployment descriptors,
(*Continued*)
 - message-driven beans,
219–220
 - transactions, declaring
attributes, 294
- Deposit() method, 304
- Destination objects, 430
- destinations, JMS, 206
- destroying entity
beans, 120
- development
 - components, 17
- development tools, project
design, 445
- digest authentication, 248
- directory services, 70
- dirty reads, 311
- distributed application
 - advantages of EJB, 4
- distributed application
deployment, 277–278
- distributed objects, 32–36
- distributed shared object
caches
 - consistency, 427
 - lock managers, 428
- distributed
 - transactions, 317
 - coordinator, 318
 - EJB servers, 458
- distributed two-phase
commit protocol, 458
- DoAs() method, 250
- does not exist state,
163, 215
- domains, messaging, 204
- dooming transactions, 305
- Double parameter, 191
- double values, select()
methods, 176
- drivers, JMS, 206
- duplicate consumption,
message-driven beans,
224–225
- durability, transactions,
282, 317
- durable subscriptions,
message-driven
beans, 214
- dynamic discovery of
applications, 395
- dynamic redeployment, 6
- E**
- Ear files, 482
- EffectiveEnd column, 378
- EffectiveStart column, 378
- EJB (Enterprise Java
Beans), 3, 23
 - abstract entities, 41
 - advanced concepts,
272, 275
 - agreed-upon standards,
13
 - bean class, 62
 - beans
 - calling, 69
 - deploying in
containers, 68
 - business interface, 76
 - business solutions, 14
 - component interface,
47, 75–76
 - components, 55
 - containers
 - bean management,
41
 - instance pooling, 43
 - request interception,
38
- context objects, 64–65
- deployment
 - descriptors, 50,
53, 66
- distributed application
advantages, 4
- distributed objects,
32–36
- Ejb-jar files, 54
- enterprise bean class, 37
- enterprise beans, 13, 53,
29–30
- entity beans, 30, 110,
120
- exceptions, 60
- glue-code tools, 41
- home handles, 269
- home interface, 45, 53,
58
- home objects, 44, 48,
53, 70
- implicit monitoring, 40
- implicit persistence, 39
- implicit remote
 - accessibility, 39
- isolation, 315
- javax.ejb.EJBContext
interface, 64
- JMS integration, 211
- local home interface,
47, 53, 59
- local home objects,
47, 53
- local interface, 58
- local interfaces,
46–49, 53
- local objects, 53

- location transparency,
 - 45, 70
- message-driven
 - beans, 31
- objects, 38–40, 53
 - factories, 44
 - handles, 268–269, 474
- persistence manager, 21
- portability, 13
- quick reference guide, 585–614
- rapid application
 - development, 13
- remote exceptions, 44
- remote interface, 42, 53, 57
- remote method
 - invocations, 44
- roles, 20
- servers, hosting Ejb-jar files, 56
- session beans, 30, 66, 81
- startup class, 211
- thick clients, 15
- transactional
 - conversations, 320
- vendor specific files, 51–53, 67
- EJB best practices
 - component assembly, 405
 - debugging, 402–403
 - guaranteeing response time via capacity planning, 397
 - legacy interaction, 408–410
 - limiting thread pool, 397
 - local or remote
 - interfaces, 401–402
 - messaging or RMI-IIOP, 393–396
 - performance tuning
 - entity beans, 400–401
 - resource partitioning, 404
 - reusable components, 406
 - singletons, 398
 - using stateful or stateless, 391–393
 - XML usage, 407–408
- EJB clustering, 419
- EJB components, 379
- EJB Ecosystem, 16
 - application assemblers, 17–18
 - bean providers, 17
 - container providers, 19
 - deployers, 18
 - IDEs, 20
 - server providers, 20
 - system administrators, 19
- EJB object model, 375
- EJB objects
 - collections, 135
 - versioning, 377–378
- EJB projects
 - additional vertical slices, 446
 - application servers, 442
 - business and presentation logic separation, 436
- EJB appropriateness, 434–435
- EJB server costs, 436
- horizontal team
 - organization, 443
- initial vertical slices, 439–441
- J2EE appropriateness, 434
- object model
 - design, 439
- reasons to use EJB, 437
- resource pooling, 436
- staffing, 438
- standard build
 - process, 446
- EJB-QL (EJB Query Language), 174, 352, 569, 573–575, 579–583
- EJB references, 239–240
- EJB servers, 449
 - administrative support, 456
 - auto-component generation, 457
 - clean shutdowns, 457
 - clustering, 455
 - compatibility test suite, 450
 - complex mappings, 451
 - conversion tools, 450
 - deferred database writes, 451
 - distributed
 - transactions, 458
 - high availability, 453
 - hot deployment, 456
 - IDE integration, 454
 - in-memory data cache, 452
 - instance pooling, 456

- EJB servers, (*Continued*)
 - integrated tier
 - support, 452
 - intelligent load
 - balancing, 455
 - JCA, 459
 - JMX, 456
 - lazy loading, 451
 - messaging, 458
 - open sourcecode, 460
 - pass-through mode, 452
 - pluggable persistence
 - providers, 451
 - provided
 - components, 458
 - realtime deployment, 457
 - reputable vendors, 461
 - scalability, 452
 - security, 453
 - specialized services, 460
 - state replication, 423–424
 - stateless transparent
 - fail-over, 455
 - sustituting JDBC
 - drivers, 451
 - UML editors, 454
 - Web services, 459
 - workflow engines, 459
- Ejb-client jar files, 69
- Ejb-jar files, 51, 54, 67
 - deploying in
 - containers, 68
 - folder structure, 67
- ejb-jar.xml file, 68, 192–194, 220
- EJB-required methods, 143
- EJB/J2EE application, 463–464
 - ejbActivate() method, 64, 88, 117, 166, 404
 - ejbCreate() method, 64, 119–120, 147, 163–165, 190, 215, 219, 404, 422
 - ejbFind() method, 129, 147
 - ejbHome() method, 138, 147, 165, 426
 - ejbHomeGetTotalBankValue() method, 175
 - ejbLoad() method, 113, 125, 147, 163–166, 291, 330, 334, 349–350, 366, 426
 - ejbPassivate() method, 64, 88, 117, 166, 404
 - ejbRemove() method, 64, 103, 119–120, 125, 147, 165, 215, 219, 404
 - ejbSelect() methods, 175–176
 - ejbStore() method, 113, 147, 163–166, 291, 330, 334, 426
- Enterprise bean class, 37
- Enterprise beans, 13, 29–30
 - conversations, 82
 - instances, 53
 - transactions, 286
- entity bean classes, 112, 127
- entity beans, 30, 110, 120
 - aggressive loading, 349
 - bank account, 136–139
 - bean-managed
 - persistent, 127
 - bean-managed
 - persistent, 118
- caching, 426
- clustering, 425
- CMP, 168–171
- container-managed
 - persistence, 119
- creating, 119–120
- data transfers, 113
- deployment
 - descriptors, 156
- destroying, 120
- directionality, 347
- EntityContext field, 143
- failure survival, 112
- finding, 121
- granularity, 368
- instances, 111–113, 117
- Java classes, 369
- multi-client data access, 114–115
- performance tuning, 400–401
- persistence, 82, 105, 118, 362–365
- persistence best
 - practices, 362
- pooling instances, 116
- primary key class, 112
- read-mostly caches, 428
- read-only caches, 427
- retrieving current
 - associated object, 124
- reusing, 399
- Session
 - Synchronization, 299
- synchronizing instances
 - with underlying storage, 115
- transactions, 291

- wrapping with session beans, 398–399
- entity contexts, 124
- EntityContext field, 143
- environment properties, 244–245
- Equals() method, 143
- exceptions, 60–61, 276
- exclusive locks, 309
- explicit middleware, 34

F

- fail-over, 414
 - clustering, 426
 - message-driven beans, 429
 - method invocations, 422
 - stateful session beans, 423–425
 - stateless session beans, 422
- failed requests, 420
- failure, beans, 394
- failure survival, entity beans, 112
- false parameter, JMS sessions, 208
- fat key pattern, 366
- files
 - ejb-client jar, 69
 - ejb-jar, 51, 67
 - ejb-jar.xml, 68, 192–194
 - MANIFEST.MF, 68
- Finally() clause, 196
- findBigAccounts() home method, 174
- findByOwnerName() method, 138
- findByPrimaryKey() method, 138, 184

- finder methods, 129, 165
 - CMP entity beans, 173
 - idempotency, 426
 - returning collections, 135
 - rules, 134
- findExpensiveProducts() method, 191
- finding entity beans, 121
- flat transactions, 282–283
- folder structure, Ejb-jar files, 67
- foreign key relationships, 328
- form-based authentication, 248

G

- generalists, team organization, 443
- get() accessor methods, idempotency, 425
- get() methods, idempotency, 426
- getAccountBalance() method, 110
- getBankAccountNumber() method, 110
- getCallerPrincipal() method, 259
- getConnection() method, 147
- getEJBLocalObject() method, 94, 124
- getEJBObject() method, 94, 124
- getPrimaryKey() method, 125, 330
- getRollbackOnly() method, 305

- getShipment() method, 330
- getStatus() method, 305
- getSubtotal() method, 171
- getTaxes() method, 171
- getTotal() method, 171
- getTotalBankValue() method, 138
- glue-code tools, EJB, 41
- granularity, entity beans, 368
- guaranteed message delivery, 203–204

H

- halting problem, 293
- handles
 - home, 269
 - objects, 268–269
- handling large result sets, 390
- HashCode() method, 143
- Hello() method, 57
- HelloClient program, 73–75
- HelloClient.java, 251
- high availability, EJB servers, 453
- high availability systems, 414
- historical tables, 378
- home handles, 269
- home interface, 45, 53
 - bank account entity bean, 138
 - coding, 58
 - counter bean, 94
- home objects, 48, 53
 - delegating to beans, 135
 - EJB, 44

- home objects, (*Continued*)
 - locating via JNDI, 71
 - lookups, 70
- home stubs, EJB
 - clustering, 419
- horizontal team
 - organization, project design, 443
- hot deployment, EJB
 - servers, 456
- HttpSession object, 393
- I**
- IDE, EJB servers, 454
- idempotency, 420, 425
- IDEs (Integrated Development Environments), 20
- IIOP (Internet Inter-ORB Protocol), 319
- immutable object design model, 378
- impedance mismatch, 370
- implementation classes, 143, 148–154
- implicit component
 - location transparency, 40
- implicit distributed transaction management, 39
- implicit middleware, 35–36
- implicit persistence, 39
- implicit remote
 - accessibility, 39
- implicit security, 39
- implicit thread support, 39
- in-memory data cache, EJB servers, 452
- infinite block problem, 293
- initial vertical slice, project design, 439–441
- initial vertical slices, project design, 442
- InitialContext requests, 430–431
- initialization parameters, JNDI, 238
- instance level
 - authorization, 266
- instance pooling, 43, 456
- Instanceof operator, 213
- instances, entity beans, 111–113, 116
- integrated tier support, EJB servers, 452
- intelligent load balancing, EJB servers, 455
- intercepting exceptions, 61
- interface keyword, 14
- interfaces
 - AccountHome.java, 138
 - AccountLocal.java, 138
 - AccountLocalHome.java, 138
 - component, 75–76
 - java.rmi.Remote, 44
 - javax.ejb.EntityBean, 127
 - javax.ejb.EJBContext, 64–124
 - javax.ejb.EJBObject, 42
 - javax.ejb.EnterpriseBean, 38, 127
 - javax.ejb.EntityBean, 128
 - javax.ejb.SessionSynchronization, 321
- javax.transaction.UserTransaction, 302
- message-driven beans, 214
- UserTransaction, 302
- interoperability, CORBA, 523–543
- invalidation strategies, read-only entity beans, 427
- invoking stored procedures, 371
- IPS (Invocations Per Second), 415
- IsCallerInRole() method, 259–260
- isolation
 - EJB, 315
 - portability, 315
 - transactions, 281, 307–310
- J**
- J2EE (Java 2 Platform, Enterprise Edition), 2, 22
- BluePrints document, 23
- clustering, 415, 430
- conversion tools, 450
- EJB, 23
- JAAS, 26
- Java IDL, 25
- JavaMail, 25
- JAXP, 26
- JCA, 26
- JDBC, 24
- JMS, 25
- JNDI, 24
- JSPs, 25

- JTA, 25
 - packages, 481
 - reference
 - implementation, 23
 - RMI, 23
 - RMI-IIOP, 23
 - servlets, 25
 - specifications, 22
 - technologies, 23
 - test suite, 23
 - J2ME (Java 2 Platform, Micro Edition), 22
 - J2SE (Java 2 Platform, Standard Edition), 22
 - JAAS (Java Authentication and Authorization Service), 26, 246
 - architecture, 249–250
 - authorization, 257
 - “Hello, World” bean, 251
 - Jad decompiler, 404
 - Jakarta Struts, 480
 - jar files, 481
 - jar utility, 56
 - Jasmine’s Computer Parts
 - application, 468–476, 482
 - Java, 14
 - Java classes, 369
 - Java IDL, 25
 - Java servlets, 15, 25
 - java.lang.Exception
 - class, 156
 - java.lang.Security
 - Exceptions, 264
 - java.rmi.Remote
 - interface, 44
 - java.rmi.Remote
 - Exceptions, 44
 - java.util.Collection, 184
 - java.util.Iterator.remove()
 - method, 358
 - JavaBeans, 17
 - JavaMail, 25
 - javax.ejb.Create
 - Exceptions, 138, 184
 - javax.ejb.EJBContext
 - interface, 64, 124
 - javax.ejb.EJBObject
 - interface, 42
 - javax.ejb.EnterpriseBean
 - interface, 38, 127
 - javax.ejb.EntityBean
 - interface, 127–128
 - javax.ejb.Finder
 - exceptions, 138
 - javax.ejb.MessageDriven
 - Bean interface, 214, 217
 - javax.ejb.Session
 - Synchronization
 - interface, 321
 - javax.jms.MessageListener
 - interface, 214, 217
 - javax.jms.QueueRequestor
 - class, 235
 - javax.jms.Topic
 - ConnectionFactory, 208
 - javax.jms.TopicRequestor
 - class, 235
 - javax.transaction.User
 - Transaction interface, 302
 - Javlin, 452
 - JAXP (Java API for XML Parsing), 26
 - JCA (J2EE Connector Architecture), 15, 26, 459
 - JDBC (Java Database Connectivity), 24, 128
 - JDBC API, 15
 - JMS (Java Message Service), 25, 203
 - API, 203, 206
 - consumers, 207
 - Messages, 213
 - producers, 207
 - sessions, 208
 - SPI, 203
 - TextMessages, 208
 - JMX (Java Management Extension), 19, 456
 - JNDI (Java Naming and Directory Interface)
 - client code
 - transactions, 306
 - default initial
 - contexts, 238
 - drivers, 419
 - locating home objects, 71
 - lookups, 238
 - running system, 74
 - JSPs (JavaServer Pages), 15, 25, 477
 - calling EJB components
 - via Java code, 478
 - Jasmine’s Computer Parts, 468, 480
 - login, 483
 - JTA (Java Transaction API), 25, 301
 - JTS (Java Transaction Service), 301
 - just-in-time algorithm, 86
- K**
- keepgenerated option, WebLogic, 404

- keywords
 - class, 14
 - interface, 14
 - synchronized, 310
 - this, 94
- L**
- large result sets, 390
- large-scale systems, 411–413
- lazy loading, 349–350, 451
- legacy database designs, 380, 387
- legacy interaction, 408–410
- levels of transaction
 - isolation, 309–310
- libraries, debugged, 14
- life cycles
 - alerts, 321
 - BMP entity beans, 163
 - CMP entity beans, 200
 - message-driven
 - beans, 215
 - session beans, 100–101
- limiting thread times, EJB
 - best practices, 397
- load balancing, 6, 414
 - EJB servers, 455
 - entity bean
 - clustering, 425
 - message-driven
 - beans, 224
 - stateful session
 - beans, 424
 - stateless session
 - beans, 421
- local home interface, 47, 53, 59, 138
- local home objects, 47, 53
 - local interface, 46–49, 53, 58, 138, 347, 401–402
 - local objects, 53
 - local/remote
 - transparency, 33
 - locale discounts, 7
 - locating home objects, 71
 - location transparency, 45, 70
 - lock managers,
 - distributed shared
 - object cache, 428
 - locks, 309–310
 - LogBean.java, 218
 - logging, 6
 - login JSP, 480, 483
 - login servlet, 484–487
 - Login() method, 250
 - look ups, home objects, 70
 - lost updates, 308–309
 - LRU (Least Recently Used) passivation, 86
 - M**
 - M1() method, 420
 - main transactions, 285
 - Mandatory attribute,
 - transactions, 296–297
 - MANIFEST.MF file, 68
 - many-to-many
 - relationships, 327, 336–343, 350
 - mapping abstract roles to
 - actual roles, 260–261
 - market interface, 38
 - marketplace for
 - components, 11
 - message consumers, 202
 - message-driven beans, 201–202, 212, 217
 - asynchronous
 - programming, 202
 - characteristics, 213
 - client program, 223
 - deployment
 - descriptors, 219–220
 - duplicate consumption, 224–225
 - durable subscriptions, 214
 - instanceof operator, 213
 - interfaces, 214
 - life cycle, 215
 - load balancing, 224
 - message logging, 215
 - message order, 225
 - message responses, 230–234
 - methods, 215–216
 - multiple senders/
 - receivers, 203
 - performance, 202
 - poison messages, 228–230
 - reliability, 203
 - security, 224
 - semantics, 214
 - transactions, 223, 293
 - message order, message
 - driven beans, 225
 - message producers, 202
 - message-oriented
 - middleware, 6
 - messageDrivenContext.set
 - RollbackOnly()
 - method, 229
 - messaging, 213
 - domains, 204
 - EJB best practices, 393–396

- EJB servers, 458
- PTP domains, 205
- method invocations,
 - fail-over, 422
- method permissions,
 - beans, 262
- method signatures,
 - preserving compile time checks, 76–77
- methods
 - findByOwnerName(), 138
 - abort(), 250
 - abstract, 171
 - afterBegin(), 321
 - afterCompletion(), 321–322
 - beforeCompletion(), 321
 - begin(), 291, 305
 - commit(), 250, 291, 305
 - count(), 91, 322
 - create(), 58, 162, 165
 - deposit(), 304
 - doAs(), 250
 - ejbActivate(), 64, 88, 117, 166
 - ejbCreate(), 64, 119–120, 147, 163–165, 190, 215, 219, 422
 - ejbFind(), 129, 147
 - ejbHome(), 138, 147, 165
 - ejbHomeGetTotalBankValue(), 175
 - ejbLoad(), 113, 125, 147, 163–166, 291, 330, 334, 349–350, 366
 - ejbPassivate(), 64, 88, 117, 166
 - ejbRemove(), 64, 103, 119–120, 125, 165, 215, 219
 - ejbRemove(), 147
 - ejbSelect(), 175–176
 - ejbStore(), 113, 147, 163–166, 291, 330, 334
 - equals(), 143
 - findBigAccounts(), 174
 - findByPrimaryKey(), 138–184
 - finder, 134, 165
 - findExpensiveProducts(), 191
 - getAccountBalance(), 110
 - getBankAccountNumber(), 110
 - getCallerPrincipal(), 259
 - getConnection(), 147
 - getEJBLocalObject(), 94–124
 - getEJBObject(), 94–124
 - getPrimaryKey(), 125, 330
 - getRollbackOnly(), 305
 - getShipment(), 350
 - getStatus(), 305
 - getSubtotal(), 171
 - getTaxes(), 171
 - getTotal(), 171
 - getTotalBankValue(), 138
 - hashCode(), 143
 - hello(), 57
 - idempotent, 420
 - isCallerInRole(), 259–260
 - java.util.Iterator.
 - remove(), 358
 - login(), 250
 - m1(), 420
 - message-driven beans, 215–216
 - MessageDrivenContext.
 - setRollbackOnly(), 229
 - newInstance(), 162–164
 - onMessage(), 213, 219
 - outprintin(), 478
 - PortableRemoteObject.
 - narrow(), 402
 - remove(), 165
 - rollback(), 305
 - run(), 250
 - service(), 476
 - SessionSynchronization, 321–322
 - setEntityContext(), 162, 165, 404
 - setEntityContext(EntityContext ctx), 164
 - setMessageDrivenContext(), 217
 - setRollbackOnly(), 305
 - setSessionContext(), 64, 94
 - toString(), 142
 - unsetEntityContext(), 165, 404
- middleware, 7, 34–36
- migrating older J2EE implementations, 450
- minimum variable, 174
- missed ejbRemove() calls, 227
- model-view-controller paradigm, 479

modifying entity bean
data, 123

MOM (Message-Oriented
Middleware), 203

multi-client data access,
114–115

multithreaded beans, 210

multiuser data sharing,
278

N

n-tier architecture, 416

naming and directory
services, 70

naming servers, 430

natural keys, 376

navigatability of
relationships, 353

NDI (Java Naming and
Directory Interface),
71

nested transactions,
284–285

network failures,
distributed
application
deployments, 277

Never attribute,
transactions, 297, 427

NewInstance() method,
162–164

nodes, 414

nonpersistence, session
beans, 82

normalization of data,
373–374

NotSupportedException
transactional attribute,
297, 427

NullPointerException, 61

O

object databases, 109

object factories, 44

object handles, 268–269,
474

object model design, 439

object serialization, 106

object-to-relational
mapping, 370, 375

objects, 40, 53

collections, 135

ConnectionFactory, 430

context, 64–65

Destination, 430

distributed, 32

EJB, 38

HttpSession, 393

life cycles, 6

relational mapping,
106–107

relationships, 109

remote interface, 42

transactional, 280

versioning, 377–378

ODBMS, 109

oldVal conversational
state, 322

one-to-many
relationships, 326,
332–335

one-to-one relationships,
326–332

onMessage() method,
213, 219

open source code, EJB
servers, 460

operators, instanceof, 213

optimistic concurrency
control, 316

OQL (object query
language), 109

order abstractions, 473

order confirmation
JSP, 480

order line items, 473–474

order processors, 474

OTS (Object Transaction
Service), CORBA, 70,
300–301

outprintln() method, 478

overhead costs, 7

P

packages, 481

parameters, double, 191

partitioning resources, EJB
best practices, 404

partitions, clusters, 415

pass-through mode, EJB
servers, 452

passivation

- alerts, 321
- beans, 87
- entity bean
instances, 117
- stateful session
beans, 85

PasswordConfig.java, 252

PasswordLoginModule.
java, 254–255

performance

- entity beans, 400–401
- message-driven beans,
202

persistence

- best practices, 361–365
- bugs, 366
- entity beans, 82, 105,
118–119, 176, 180,
366–368

- JDBC control, 367
 - persistence manager,
 - EJB, 21
 - persistence-type
 - element, 160
 - persistent data
 - components, 110
 - persistent fields, CMP
 - entity beans, 170
 - pessimistic concurrency
 - control, 316
 - phantoms, databases,
 - 313–314
 - phases, transaction two-
 - phase commit,
 - 317–318
 - pluggable persistence
 - providers, EJB
 - servers, 451
 - PoisonBean.java, 229
 - poison messages, 228–230
 - pooling
 - entity bean
 - instances, 116
 - message-driven
 - beans, 215
 - stateful session
 - beans, 85
 - portability
 - EJB, 13
 - isolation, 315
 - PortableRemoteObject.
 - narrow() method, 402
 - preferred customer
 - discounts, 7
 - Presentation tier, 475–476,
 - 480–482
 - preserving compile time
 - checks, method
 - signatures, 76–77
 - pricing components, 5
 - pricing rules, 5–7
 - Prim-key-class element,
 - entity bean
 - deployment
 - descriptor, 160
 - Primary key class, 112, 139
 - processing queues, 414
 - producers, JMS, 206–207
 - product abstractions, 470
 - product detail JSP, 480
 - product IDs,
 - ProductPK.java, 184
 - product line example,
 - CMP entity bean
 - persistence, 180
 - Product.java, 181
 - ProductBean.java, 187–189
 - ProductHome.java,
 - 182–183
 - ProductLocal.java, 182
 - ProductLocalHome.java,
 - 184–185
 - ProductPK class, 184, 195
 - ProductPK.java, 184, 186
 - programmatic
 - authorization, 258
 - programmatic
 - invalidation, 427
 - programmatic
 - transactions, 287, 300
 - advantages, 291
 - writing, 304
- programming model,
 - JMS, 206
 - programs, HelloClient,
 - 73–75
 - projects
 - additional vertical
 - slices, 446
 - application servers, 442
 - business and
 - presentation logic
 - separation, 436
 - business requirements,
 - 433–434
 - development tools, 445
 - EJB appropriateness,
 - 434–435
 - EJB server costs, 436
 - horizontal team
 - organization, 443
 - initial vertical slices,
 - 439–441
 - object model design,
 - 439
 - reasons to use EJB, 437
 - resource pooling, 436
 - staffing, 438
 - standard build
 - process, 446
 - vertical team
 - organization, 443
- proprietary descriptors,
 - counter bean, 96
 - provided components, EJB
 - servers, 458
 - PTP domains,
 - messaging, 205
 - Pub domains,
 - messaging, 204
 - pull model, messages, 224
 - push model, client
 - requests, 224
- Q**
- quantity discounts, 5
 - queries, CMP entity beans,
 - 173–174

quick reference guide,
EJB, 585–614

R

rapid application

development, EJB, 13

RAS (Reliability,

Availability,

Serviceability), 412

Rational Rose, 454

read locks,

transactions, 310

read-mostly caches, entity

beans, 428

read-only caches, 427

READ COMMITTED

mode, 309, 312

READ UNCOMMITTED

mode, 309–311

recursive

relationships, 353

reentrant element, 160

references

EJB, 239–240

resource factories, 243

referential integrity,

355–356

relational mapping of

objects, 106–107

relationships

aggregation, 350

BMP entity beans, 326

cardinality, 326

circular, 354

CMP entity beans, 326

collections, 357–358

composition, 351

directionality, 344

EJB-QL, 352

local interfaces, 347

many-to-many, 336–337

navigability, 353

objects, 109

one-to-many, 332

one-to-one, 328

recursive, 353

session beans, 327

traversing, 352

reliability

large-scale systems, 412

message-driven

beans, 203

remote exceptions, 44

remote home stubs,

automatic

fail-over, 422

remote interface, 42, 33, 53

bank account entity

bean, 137

coding, 57

counter bean, 91–92

entity beans, 401–402

remote stubs

automatic fail-over, 422

EJB clustering, 419

failed requests, 421

RemoteExceptions,

182, 402

remove() method, 62, 165

REPEATABLE READ

mode, 310, 313

request interception, 38

request-level fail-over, 414

requests, servlets, 475

Required attribute,

294–295, 400

RequiresNew

attribute, 296

res-auth entry, JDBC

connections, 242

resource factories, 241–243

resource managers, 280

resource partitioning, 404

resource pooling, 7

resource-ref element, 161

resources,

transactions, 280

responses

message-driven beans,

230–234

servlets, 475

returning collections via

finder methods, 135

reusable components,

9, 406

reusing entity beans, 399

RMI (Remote Method

Invocation), 23

RMI-IIOP (Remote

Method Invocation

over the Internet

Inter-ORB Protocol),

23, 44, 393–396, 491,

495, 499, 503, 507,

511, 515, 518–520

roll backs, 283–284

rollback() method, 305

root transactions, 285

RPS (Requests Per

Second), 414

run() method, 250

S

sagas, 286

security, 7

4-tier architecture, 416

EJB servers, 453

Java, 14

message-driven

beans, 224

- security contexts, 266
- security identities, 246
- security policies, 257, 262–263
- security propagation, 266–267
- security roles, 258, 260–264
- semantics, message driven beans, 214
- serializability, transactions, 310
- SERIALIZABLE mode, 310, 314
- serializing objects, 106
- server providers, EJB Ecosystem, 20
- server-side
 - components, 16
- Server-side output
 - bank account program, 162
 - HelloClient program, 75
- server-side proxy objects, 33
- servers
 - counter bean, 100
 - EJB, 449
 - administrative support, 456
 - auto-component generation, 457
 - clean shutdowns, 457
 - clustering, 455
 - compatibility test suite, 450
 - complex mappings, 451
 - conversion tools, 450
 - deferred database
 - writes, 451
 - distributed
 - transactions, 458
 - high availability, 453
 - hot deployment, 456
 - IDE integration, 454
 - in-memory data cache, 452
 - instance pooling, 456
 - integrated tier support, 452
 - intelligent load balancing, 455
 - JCA, 459
 - JMX, 456
 - lazy loading, 451
 - messaging, 458
 - open sourcecode, 460
 - pass-through mode, 452
 - pluggable
 - persistence providers, 451
 - provided
 - components, 458
 - realtime
 - deployment, 457
 - reputable vendors, 461
 - scalability, 452
 - security, 453
 - specialized services, 460
 - stateless transparent fail-over, 455
 - substituting JDBC drivers, 451
 - UML editors, 454
 - Web services, 459
 - workflow engines, 459
 - hosting Ejb-jar files, 56
- Service() method, 476
- services, naming and directory, 70
- servlets, 15, 25
 - calling EJB components and passing results to JSPs, 479
 - Jasmine's Computer Parts, 468
 - login, 484–487
 - requests, 475
 - responses, 475
- session beans, 30, 81–85
 - alerts, 321
 - descriptors, 66
 - life cycles, 100–101
 - relationships, 327
- session context interface, 94
- sessions, JMS, 206
- SessionSynchronization, entity beans, 299
- SessionSynchronization methods, 321–322
- SetEntityContext() method, 162–165, 404
- SetMessageDrivenContext() method, 217
- SetRollbackOnly() method, 305
- SetSessionContext() method, 64, 94
- single access point simplicity, 414
- single-threaded beans, 210
- singletons, 398
- skeletons, stubs, 32

- specialists, team
 - organization, 443
 - SPI (Service Provider Interface), 203
 - staffing project, 438
 - standard build process,
 - project design, 446
 - starting JNDI, 74
 - startup classes, 211
 - state replication, 423–424
 - stateful session beans, 82
 - activation, 85
 - automatic fail-over, 425
 - client conversations, 85
 - conversational
 - state, 320
 - counter beans, 89
 - client code, 96
 - client-side output, 99
 - deployment
 - descriptor, 97–98
 - home interface, 94
 - proprietary
 - descriptor, 96
 - remote interface,
 - 91–92
 - running client, 99
 - server-side
 - output, 100
 - session context
 - interface, 94
 - val member
 - variable, 91
 - fail-over, 423–425
 - failure, 394
 - load balancing, 424
 - passivation, 85
 - pooling, 85
 - swapping out, 85
 - stateless session beans,
 - 83–84
 - clustering, 421
 - fail-over, 422
 - life cycles, 100
 - load balancing, 421
 - stateless transparent fail-over, 455
 - static data delivery, 416
 - stored procedures,
 - 371–373
 - storing objects, 106–107
 - stubs, skeletons, 32
 - sub domains,
 - messaging, 204
 - subclasses, 167–170
 - subtransactions, 285
 - superclass, CMP entity
 - beans, 171
 - Supports attribute, 296
 - surrogate keys, 376–377
 - swapping out stateful
 - session beans, 85
 - switching context,
 - applications, 85
 - synchronization protocols,
 - transactions, 281
 - synchronized
 - keyword, 310
 - synchronizing entity
 - beans, 115
 - system abstraction, 287
 - system administrators,
 - EJB Ecosystem, 19
 - system wide invalidation
 - notification, 427
 - system-level
 - exceptions, 60
 - systems management, 6
- T**
- tables, versioning
 - columns, 378
 - tag-libraries, JSP, 478
 - TextMessage, 208, 219
 - thick clients, 15
 - this keyword, 94
 - threads, 6, 397
 - beans, 210
 - implicit support, 39
 - throughput, 414
 - timeouts, read only
 - caches, 427
 - Togethersoft Together/J,
 - 454
 - TopicClient.java, 209
 - TopicConnectionFactory,
 - 208
 - toString() method, 142
 - TPS (Transactions Per Second), 414
 - transaction contexts, 319
 - transaction managers, 280
 - transactional alerts, 321
 - transactional
 - communications
 - protocol, 318
 - transactional
 - conversations, 320
 - transactional models,
 - 282–285
 - transactional objects, 280
 - transactions, 6, 275
 - aborted, 282, 287
 - ACID properties, 280
 - all or nothing
 - paradigm, 281–283
 - atomic operations,
 - 276, 279
 - atomicity, 281
 - attributes, 298
 - bean managed, 306
 - begin statements, 287
 - boundaries, 304
 - client code, 306
 - client-controlled, 292

- client-initiated, 290
- commit statements, 287
- committed, 282
- consistency, 281
- constants, 302
- container managed,
 - 292, 306
- deadlock, 310
- declarative, 288–289,
 - 292, 304
- demarcating
 - boundaries,
 - 287–289
- deployment
 - descriptors, 294
- distributed, 317
- dooming, 305
- durability, 282, 317
- enterprise beans, 286
- entity beans, 291
- flat, 282–284
- isolation, 281, 307–310
- locks, 310
- lost updates, 308–309
- Mandatory attribute,
 - 296–297
- message-driven beans,
 - 223, 293
- nested, 284–285
- Never attribute, 297
- NotSupported
 - attribute, 297
- programmatic, 287, 291,
 - 300
- READ COMMITTED
 - mode, 312
- read locks, 310
- READ UNCOMMITTED
 - mode, 311

- REPEATABLE READ
 - mode, 313
- Required attribute,
 - 294–295
- RequiresNew
 - attribute, 296
- roll backs, 283
- selecting style, 291
- serializability, 310
- SERIALIZABLE
 - mode, 314
- Supports attribute, 296
- synchronization
 - protocols, 281
- system abstraction, 287
- two-phase commit, 317
- write locks, 310
- transferring data, entity
 - beans, 113
- transparent fail-over, 6,
 - 61, 414
- traversing
 - relationships, 352
- tree of transactions, 285
- trip planning problem,
 - nested
 - transactions, 285
- two-phase commit,
 - transactions, 317

U

- UML (Unified Modeling Language), 20, 454
- unchecked exceptions, 61
- units of work, flat
 - transactions, 282
- unrepeatable reads,
 - 312–313
- UnsetEntityContext()
 - method, 165, 404
- unverified order
 - status, 474

- UserTransaction
 - interface, 302
- utilities, jar, 56

V

- val conversational
 - state, 322
- val member variable, 91
- variables, 174
- vendor-specific files,
 - 51–53, 67
- versioning behavior,
 - 377–379
- vertical slices, project
 - design, 440
- vertical team organization,
 - project design, 443
- view cart JSP, 480

W

- War files, 481
- Web services,
 - EJB servers, 459
- Web storefront JSP, 480
- WebLogic, keepgenerated
 - option, 404
- workflow engines, EJB
 - servers, 459
- workflow solutions, 8
- write locks,
 - transactions, 310

X

- XML
 - deployment
 - descriptors, 545,
 - 549, 553, 558, 562,
 - 565
 - EJB best practices,
 - 407–408