

**Business Component
Development with EJB
Technology, Java EE 5**

Student Guide

SL-351-EE5 REV D.2

D61838GC10
Edition 1.0
D62447

ORACLE®

Copyright © 2008, 2009, Oracle and/or its affiliates. All rights reserved.

Disclaimer

This document contains proprietary information, is provided under a license agreement containing restrictions on use and disclosure, and is protected by copyright and other intellectual property laws. You may copy and print this document solely for your own use in an Oracle training course. The document may not be modified or altered in any way. Except as expressly permitted in your license agreement or allowed by law, you may not use, share, download, upload, copy, print, display, perform, reproduce, publish, license, post, transmit, or distribute this document in whole or in part without the express authorization of Oracle.

The information contained in this document is subject to change without notice. If you find any problems in the document, please report them in writing to: Oracle University, 500 Oracle Parkway, Redwood Shores, California 94065 USA. This document is not warranted to be error-free.

Sun Microsystems, Inc. Disclaimer

This training manual may include references to materials, offerings, or products that were previously offered by Sun Microsystems, Inc. Certain materials, offerings, services, or products may no longer be offered or provided. Oracle and its affiliates cannot be held responsible for any such references should they appear in the text provided.

Restricted Rights Notice

If this documentation is delivered to the U.S. Government or anyone using the documentation on behalf of the U.S. Government, the following notice is applicable:

U.S. GOVERNMENT RIGHTS

The U.S. Government's rights to use, modify, reproduce, release, perform, display, or disclose these training materials are restricted by the terms of the applicable Oracle license agreement and/or the applicable U.S. Government contract.

Trademark Notice

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

AMD, Opteron, the AMD logo, and the AMD Opteron logo are trademarks or registered trademarks of Advanced Micro Devices. Intel and Intel Xeon are trademarks or registered trademarks of Intel Corporation. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. UNIX is a registered trademark licensed through X/Open Company, Ltd.

This page intentionally left blank.

Hong Lu (honglu11@hotmail.com) has a non-transferable license to
use this Student Guide.

This page intentionally left blank.

Hong Lu (honglu11@hotmail.com) has a non-transferable license to
use this Student Guide.

Table of Contents

About This Course	Preface-xiii
Course Goals	Preface-xiii
Course Map	Preface-xiv
Topics Not Covered	Preface-xv
How Prepared Are You?	Preface-xvi
Introductions	Preface-xvii
How to Use Course Materials	Preface-xviii
Conventions	Preface-xix
Icons	Preface-xix
Typographical Conventions	Preface-xx
Additional Conventions	Preface-xxi
Examining Enterprise JavaBeans (EJB) Applications	1-1
Objectives	1-1
Additional Resources	1-2
Introducing the Java Platform Enterprise Edition (Java EE)	1-3
Examining the Java EE Application Architecture	1-6
Examining the Component-Container Architecture	1-6
Examining the Java EE Implementation of the Container-Component Architecture	1-7
Examining Java EE Container Services	1-9
Examine the EJB Application Creation Process	1-14
Comparing Java EE Application Development With Traditional Enterprise Application Development	1-18
Introducing the Auction Application	2-1
Objectives	2-1
Additional Resources	2-2
Auction Application Use Cases	2-3
Analyzing the Auction System	2-5
Auction Application Domain Objects	2-5
The Auction Domain Object	2-7
The AuctionUser Domain Object	2-9
The Bid Domain Object	2-10

The Item Domain Object	2-11
The BookItem Domain Object.....	2-12
Examining the Implementation Model.....	2-13
Middle-Tier Subsystems	2-13
Implementing EJB 3.0 Session Beans.....	3-1
Objectives	3-1
Additional Resources	3-2
Comparison of Stateless and Stateful Behaviors	3-3
Stateless Session Bean Operational Characteristics	3-4
Stateful Session Bean Operational Characteristics	3-6
Creating Session Beans: Essential Tasks	3-7
Declaring a Business Interface for the Session Bean	3-7
Creating the Session Bean Class That Implements the Business Interface.....	3-9
Annotating the Session Bean Class.....	3-10
Creating Session Beans: Adding Life-Cycle Event Handlers	3-12
Defining Life-Cycle Event Handlers	3-13
The SessionContext Object.....	3-14
Session Bean Packaging and Deployment.....	3-15
Introducing DDs.....	3-15
Example of a DD for an Enterprise Bean.....	3-16
Creating a Session Bean Component Archive	3-17
Deploying a Session Bean Component Archive	3-18
Creating a Session Bean Client.....	3-19
Creating a Client Using Container Services.....	3-20
Creating a Client Without Using Container Services	3-20
Reviewing Session Beans	3-22
Implementing Entity Classes: The Basics.....	4-1
Objectives	4-1
Additional Resources	4-2
Examining Java Persistence	4-3
Object Tier / Data Tier Static and Dynamic Mapping Example	4-5
Defining Entity Classes: Essential Tasks	4-9
Declare the Entity Class	4-9
Verifying and Overriding the Default Mapping	4-12
Examining Managing Entity Instance Life-Cycle States	4-14
Using Entities to Interact With the Database	4-18
Using an Extended Persistence Context Entity Manager..	4-22
Deploying Entity Classes	4-25
Creating a Persistence Unit Using Default Settings.....	4-27
Examining a Persistence Unit Using Non Default Settings	4-29

Implementing Entity Classes: Modelling Data Association Relationships	5-1
Objectives	5-1
Additional Resources	5-2
Examining Association Relationships in Data and Object Models	5-3
Using Relationship Properties to Define Associations	5-5
Examples of Association Relationships	5-6
Implementing One-to-One Unidirectional Association	5-10
Implementing One-to-One Bidirectional Association	5-12
Implementing One-to-Many/Many-to-One Bidirectional Association.....	5-14
Implementing Many-to-Many Bidirectional Association	5-17
Examining Fetch and Cascade Mode Settings	5-19
Fetch Mode Attribute	5-19
Cascade Mode Attribute	5-20
Implementing Entity Classes: Modelling Inheritance and Embedded Relationships.....	6-1
Objectives	6-1
Additional Resources	6-2
Examining Entity Inheritance	6-3
Examining Object/Relational Inheritance Hierarchy Mapping Strategies	6-4
Mapping Inheritance Using the Single Table per Class Hierarchy Strategy	6-5
Mapping Inheritance Using a Table per Class Strategy	6-7
Mapping Inheritance Using the Joined Subclass Strategy	6-8
Inheriting From an Entity Class.....	6-9
Inheriting Using a Mapped Superclass.....	6-11
Inheriting From a Non-Entity Class	6-14
Entity Classes: Using an Embeddable Class	6-16
Defining Entity Classes: Using an Embeddable Class.....	6-17
Entity Classes: Using a Composite Primary key	6-18
Defining a Composite Primary Key Using the EmbeddedId Annotation	6-19
Defining a Composite Primary Key Using the IdClass Annotation	6-20
Using the Java Persistence Query Language.....	7-1
Objectives	7-1
Additional Resources	7-2
Introducing Querying Over Entities and Their Persistence State	7-3
Examining Query Objects	7-4
Creating and Using Query Objects.....	7-6
Declaring and Using Positional Parameters	7-7

Introducing the Java Persistence Query Language.....	7-8
Declaring Query Strings: The SELECT Statement.....	7-9
Examining the FROM Clause	7-10
Examining the WHERE Clause.....	7-14
Examining the SELECT Clause	7-18
Examining the GROUP BY and HAVING Clauses	7-18
Examining the ORDER BY Clause	7-20
Declaring Query Strings: The BULK UPDATE Statement	7-21
Declaring Query Strings: The DELETE Statement.....	7-22
Developing Java EE Applications Using Messaging.....	8-1
Objectives	8-1
Additional Resources	8-2
Reviewing JMS Technology.....	8-3
Administered Objects.....	8-4
Messaging Clients	8-5
Messages.....	8-6
Point-to-Point Messaging Architecture	8-8
Publish/Subscribe Messaging Architecture	8-9
Creating a Queue Message Producer.....	8-10
Queue Message Producer Code Example	8-11
Creating a Synchronous Message Consumer	8-14
Synchronous Queue Consumer Code Example	8-15
Creating an Asynchronous Queue Consumer.....	8-18
Asynchronous Queue Consumer Code Example	8-19
Evaluating the Capabilities and Limitations of EJB Components as Messaging Clients	8-22
Using EJB Components as Message Producers	8-22
Using EJB Components as Message Consumers	8-22
Developing Message-Driven Beans	9-1
Objectives	9-1
Additional Resources	9-2
Introducing Message-Driven Beans	9-3
Java EE Technology Client View of Message-Driven Beans	9-3
Life Cycle of a Message-Driven Bean.....	9-4
Types of Message-Driven Beans	9-5
Creating a JMS Message-Driven Bean: Essential Tasks.....	9-6
Creating a JMS Message-Driven Bean: Adding Life-Cycle Event Handlers	9-10
Creating a Non-JMS Message-Driven Bean	9-12
Implementing Interceptor Classes and Methods	10-1
Objectives	10-1
Additional Resources	10-2
Introducing Interceptors and Interceptor Classes.....	10-3

Creating a Business Interceptor Method in the Enterprise Bean	
Class	10-6
Creating an Interceptor Class.....	10-8
Associating Multiple Business Interceptor Methods With an Enterprise Bean	10-10
Including Life-Cycle Callback Interceptor Methods in an Interceptor Class	10-13
Creating a Life-Cycle Callback Interceptor Method	10-14
Defining Entity Classes: Adding Life-Cycle Event Handlers..	10-16
Defining Life-Cycle Event Handlers	10-17
Reviewing Interceptors	10-20
Implementing Transactions.....	11-1
Objectives	11-1
Additional Resources	11-2
Introducing Transaction Demarcation Management	11-3
Transaction Demarcation Task	11-3
Guidelines for Selecting a Transaction Demarcation Policy.....	11-5
Implementation Options for Transaction Demarcation	11-7
Using CMT	11-9
CMT Transaction Attributes.....	11-9
Implementing CMT	11-14
Interacting Programmatically With an Ongoing CMT Transaction.....	11-16
Getting or Setting the Rollback Status During CMT	11-16
Enabling a Stateful Session Bean Instance to Monitor a CMT Transaction.....	11-18
Using BMT	11-22
Restrictions on Transaction Demarcation Policies for Enterprise Beans Using BMT.....	11-24
Applying Transactions to Messaging	11-29
Examining Transactions and JMS API Messages	11-29
Applying Transactions to Message-Driven Beans	11-30
Handling Exceptions	12-1
Objectives	12-1
Additional Resources	12-2
Introducing Exceptions in Java EE Applications	12-3
Java EE Platform Perspective of Exceptions	12-5
Examining the Exception Path in a Java EE Application Environment	12-11
Examining Exception Handling by Container.....	12-12
Application Exception Handling by Container	12-13
System Exception Handling by Containers	12-15
Handling Exceptions in Enterprise Bean Methods	12-17
Handling Exceptions in Enterprise Bean Client Code.....	12-20

Handling Application Exceptions in Bean Client Code ..	12-20
Handling System Exceptions in Bean Client Code	12-22
Reviewing Exception Handling in EJB Applications.....	12-27
Using Timer Services	13-1
Objectives	13-1
Additional Resources	13-2
Introducing Timer Services	13-3
Creating a Timer Callback Notification.....	13-4
Examining Types of Timers.....	13-4
Creating a Timer Object	13-7
Processing a Timer Callback Notification Object.....	13-9
Creating the Timer Handler: Implementing the TimedObject Interface	13-9
Creating the Timer Handler: Using the Timeout Annotation	13-10
Guidelines for Coding Timer Handler Method.....	13-11
Managing Timer Objects.....	13-12
Interrogating a Timer Callback Notification.....	13-13
Obtaining a List of Outstanding Timer Notifications....	13-15
Implementing Security	14-1
Objectives	14-1
Additional Resources	14-2
Understanding the Java EE Security Architecture	14-3
Authenticating the Caller.....	14-7
Establishing User Identities	14-7
Caller Authentication	14-10
Examining the Java EE Authorization Strategies	14-12
Using Declarative Authorization.....	14-14
Declaring Security Roles	14-14
Declaring Method Permissions	14-16
Declaring Security Identity	14-19
Using Programmatic Authorization.....	14-23
Using the isCallerInRole Method.....	14-23
Using the getCallerPrincipal Method	14-27
Examining a Complete Example of the DD Security Section.....	14-29
Examining the Responsibilities of the Deployer	14-31
Reviewing the Java EE Application Security Tasks.....	14-32
Using EJB Technology Best Practices	15-1
Objectives	15-1
Additional Resources	15-2
Defining Best Practices	15-3
Choosing Between EJB and Other Technologies.....	15-4
Using and Applying Known Patterns.....	15-7

Selecting and Applying Java EE Application Design	
Decisions	15-12
Matching the EJB Components to the Tasks	15-12
Maintaining Session-Specific Data	15-15
Minimizing Network Traffic	15-16
Introducing Transactions	A-1
Additional Resources	A-2
Examining Transactions.....	A-3
Types of Transaction	A-5
Transaction-Related Concurrency Issues	A-9
Handling Distributed Transactions.....	A-13
Two-Phase Commit	A-14
Java Transaction API (JTA).....	A-15
Integrating With Legacy Systems	B-1
Objectives	B-1
Additional Resources	B-2
Introducing Integration With Legacy Systems	B-3
Examining the EIS Connectivity Module	B-4
Requirements.....	B-4
Implementation Alternatives	B-6
Examining JCA Resource Adapters	B-10
Resource Adapter System Contracts.....	B-10
Application Contract	B-11
Using the CCI API Interfaces	B-12
Creating the ConnectionSpec Object.....	B-16
Using the ConnectionFactory Object.....	B-17
Using the Connection Object	B-18
Using the Interaction Object.....	B-19
Examining a CCI Example.....	B-21
Using a Message-Driven Bean Resource Adapter: An Example	B-23
Integrating With Legacy Systems Using the CORBA Protocol.....	B-26
Examining the Interoperability Between the EJB Technology and CORBA Protocols	B-27
Accessing an Enterprise Bean With a CORBA Client.....	B-28
Accessing a CORBA EIS From an EJB Component.....	B-30
Quick Reference for UML.....	C-1
Additional Resources	C-2
UML Basics	C-3
General Elements	C-6
Packages	C-6
Stereotypes.....	C-8
Annotation	C-8
Constraints.....	C-9

Tagged Values	C-9
Use Case Diagrams	C-10
Class Diagrams	C-11
Class Nodes	C-11
Inheritance	C-14
Interface Implementation	C-15
Association, Roles, and Multiplicity	C-16
Aggregation and Composition	C-17
Association Classes	C-18
Other Association Elements	C-21
Object Diagrams	C-22
Collaboration Diagrams	C-24
Sequence Diagrams	C-26
Statechart Diagrams	C-28
Transitions	C-29
Activity Diagrams	C-30
Component Diagrams	C-34
Deployment Diagrams	C-36

Preface

About This Course

Course Goals

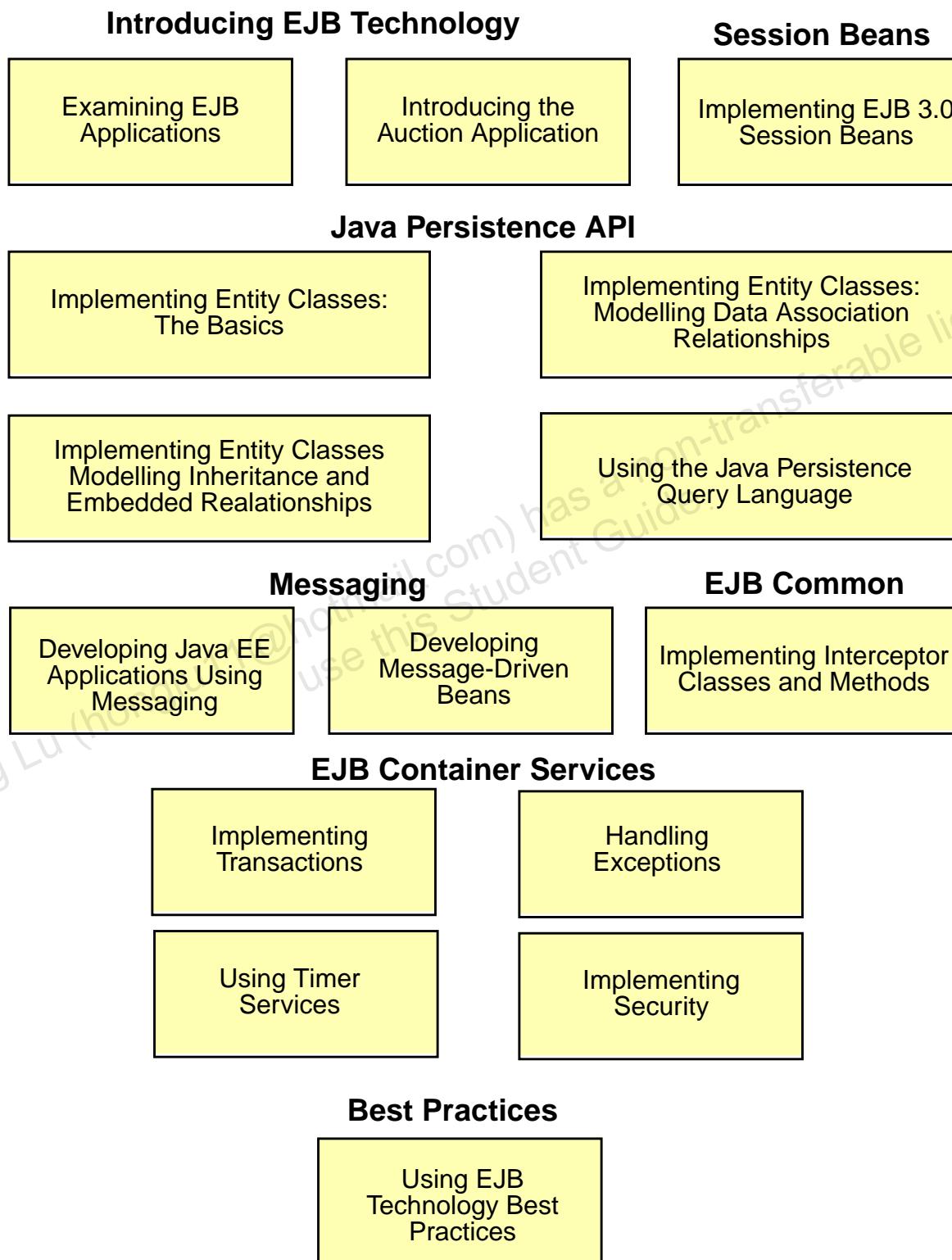
Upon completion of this course, you should be able to:

- Examine the Java™ Platform, Enterprise Edition (Java™ EE)
- Examine a Java EE technology application
- Implement Enterprise JavaBeans™ (EJB™) 3.0 session beans
- Implement Java Persistence API entity classes
- Implement entity composition, association, and inheritance
- Use the Java Persistence API query language
- Develop Java EE technology applications using messaging
- Create message-driven beans
- Implement interceptor classes and methods
- Implement transactions
- Implement exception handling for EJB technology
- Add timer functionality to EJB components
- Implement security for Java EE technology
- Evaluate best practices for EJB technology

Course Map

Course Map

The following course map enables you to see what you have accomplished and where you are going in reference to the course goals.



Topics Not Covered

This course does not cover the following topics. Many of these topics are covered in other courses offered by Sun Educational Services:

- Object-oriented concepts – Covered in OO-226: *Object-Oriented Analysis and Design for Java™ Technology (UML)*
- Distributed programming concepts and support technologies, such as remote procedure call (RPC), Remote Method Invocation (RMI), Internet Inter-ORB Protocol (IIOP), Common Object Request Broker Architecture (CORBA), Lightweight Directory Access Protocol (LDAP), Java Naming and Directory Interface™ API (JNDI API)

Refer to the Sun Educational Services catalog for specific information and registration.

How Prepared Are You?

To be sure you are prepared to take this course, can you answer yes to the following questions?

- Can you write a Java technology class?
- Can you implement composition, association, and inheritance relationships?
- Can you handle events and exceptions generated in Java technology classes?
- Can you describe the issues associated with transaction management?

Introductions

Now that you have been introduced to the course, introduce yourself to the other students and the instructor, addressing the following items:

- Name
- Company affiliation
- Title, function, and job responsibility
- Experience related to topics presented in this course
- Reasons for enrolling in this course
- Expectations for this course

How to Use Course Materials

To enable you to succeed in this course, these course materials contain a learning module that is composed of the following components:

- Goals – You should be able to accomplish the goals after finishing this course and meeting all of its objectives.
- Objectives – You should be able to accomplish the objectives after completing a portion of instructional content. Objectives support goals and can support other higher-level objectives.
- Lecture – The instructor presents information specific to the objective of the module. This information helps you learn the knowledge and skills necessary to succeed with the activities.
- Activities – The activities take various forms, such as an exercise, self-check, discussion, and demonstration. Activities help you facilitate the mastery of an objective.
- Visual aids – The instructor might use several visual aids to convey a concept, such as a process, in a visual form. Visual aids commonly contain graphics, animation, and video.

Conventions

The following conventions are used in this course to represent various training elements and alternative learning resources.

Icons



Additional resources – Indicates other references that provide additional information on the topics described in the module.



Discussion – Indicates a small-group or class discussion on the current topic is recommended at this time.



Note – Indicates additional information that can help students but is not crucial to their understanding of the concept being described. Students should be able to understand the concept or complete the task without this information. Examples of notational information include keyword shortcuts and minor system adjustments.

Conventions

Typographical Conventions

Courier is used for the names of commands, files, directories, programming code, and on-screen computer output; for example:

Use `ls -al` to list all files.

`system%` You have mail.

Courier is also used to indicate programming constructs, such as class names, methods, and keywords; for example:

The `getServletInfo` method is used to get author information.

The `java.awt.Dialog` class contains `Dialog` constructor.

Courier bold is used for characters and numbers that you type; for example:

To list the files in this directory, type:

`# ls`

Courier bold is also used for each line of programming code that is referenced in a textual description; for example:

```
1 import java.io.*;
2 import javax.servlet.*;
3 import javax.servlet.http.*;
```

Notice the `javax.servlet` interface is imported to allow access to its life-cycle methods (Line 2).

Courier italics is used for variables and command-line placeholders that are replaced with a real name or value; for example:

To delete a file, use the `rm filename` command.

Courier italic bold is used to represent variables whose values are to be entered by the student as part of an activity; for example:

Type `chmod a+rwx filename` to grant read, write, and execute rights for `filename` to world, group, and users.

Palatino italics is used for book titles, new words or terms, or words that you want to emphasize; for example:

Read Chapter 6 in the *User's Guide*.

These are called *class* options.

Additional Conventions

Java programming language examples use the following additional conventions:

- Method names are not followed with parentheses unless a formal or actual parameter list is shown; for example:
“The `doIt` method...” refers to any method called `doIt`.
“The `doIt()` method...” refers to a method called `doIt` that takes no arguments.
- Line breaks occur only where there are separations (commas), conjunctions (operators), or white space in the code. Broken code is indented four spaces under the starting code.
- If a command used in the Solaris™ Operating System (Solaris OS) is different from a command used in the Microsoft Windows platform, both commands are shown; for example:

In the Solaris OE:

```
$CD SERVER_ROOT/BIN
```

In Microsoft Windows:

```
C:\>CD SERVER_ROOT\BIN
```


Module 1

Examining Enterprise JavaBeans (EJB) Applications

Objectives

Upon completion of this module, you should be able to:

- Introduce the Java Platform, Enterprise Edition (Java EE)
- Examine the Java EE application architecture
- Examine the EJB application creation process
- Compare the Java EE application development with traditional enterprise application development

Additional Resources

Additional resources – The following references provide additional information on the topics described in this module:

- Sun Microsystems, “JSR 220: Enterprise JavaBeans™, Version 3.0 EJB Core Contracts and Requirements.”
[<https://sdlc3e.sun.com/ECom/EComActionServlet;jsessionid=CEAAE57A3BAB8A76D4555E3C5A1F4031>], accessed July 25, 2006.
- Sun Microsystems, “JSR 220: Enterprise JavaBeans™, Version 3.0 EJB 3.0 Simplified API.”
[<https://sdlc3e.sun.com/ECom/EComActionServlet;jsessionid=CEAAE57A3BAB8A76D4555E3C5A1F4031>], accessed July 25, 2006.
- Sun Microsystems, “Java 2™ Platform Enterprise Edition Specification, v5.0”
[<https://sdlc3e.sun.com/ECom/EComActionServlet;jsessionid=CEAAE57A3BAB8A76D4555E3C5A1F4031>], accessed July 25, 2006.

Introducing the Java Platform Enterprise Edition (Java EE)

The Java EE platform consists of the following elements.

- A set of specifications

The primary specification for the Java EE platform is Java Platform, Enterprise Edition 5 (Java EE 5) Java Specification Request 244 (JSR 244).

Secondary specifications for the Java EE 5 platform cover specific technology areas of the platform, such as

- Web services technologies
- Web application technologies
- Enterprise application technologies
- Management and security technologies

Table 1-1 lists the specifications for the different technology areas of the Java EE 5 platform.

Table 1-1 Java EE 5 Technology Specifications

Technology Area	Specification Name	JSR
Web services technologies	Implementing Enterprise Web Services	JSR 109
	Java API for XML-Based Web Services (JAX-WS) 2.0	JSR 224
	Java API for XML-Based RPC (JAX-RPC) 1.1	JSR 101
	Java Architecture for XML Binding (JAXB) 2.0	JSR 222
	SOAP with Attachments API for Java (SAAJ)	JSR 67
	Streaming API for XML	JSR 173
	Web Service Metadata for the Java Platform	JSR 181
Web application technologies	Java Servlet 2.5 JavaServer™ Faces 1.2 JavaServer Pages™ 2.1 JavaServer Pages Standard Tag Library	JSR 154
	JavaServer Faces 1.2	JSR 252
	JavaServer Pages 2.1	JSR 245
	JavaServer Pages Standard Tag Library	JSR 52

Introducing the Java Platform Enterprise Edition (Java EE)

Table 1-1 Java EE 5 Technology Specifications

Technology Area	Specification Name	JSR
Enterprise Application Technologies	Enterprise JavaBeans 3.0	JSR 220
	J2EE™ Connector Architecture 1.5	JSR 112
	Common Annotations for the Java Platform	JSR 250
	Java Message Service API	JSR 914
	Java Persistence API	JSR 220
	Java Transaction API (JTA)	JSR 907
	JavaBeans Activation Framework (JAF) 1.1	JSR 925
	JavaMail™	JSR 919
Management and Security Technologies	J2EE Application Deployment	JSR 88
	J2EE Management	JSR 77
	Java Authorization Contract for Containers	JSR 115

- Java EE 5 Software Development Kit (Java EE 5 SDK)

The Java EE 5 SDK consists of the following items:

- Sun Java™ System Application Server Platform Edition 9 (Application Server 9 PE)

The Application Server 9 PE is the free implementation of the Java EE Application Server defined in the Java EE 5 specification.

- Java EE 5 Samples

Java EE 5 samples contains example code that demonstrates the use of Java EE 5 technologies.

- Java Platform, Standard Edition 5.0 (Java SE 5)

The Java EE 5 classes and interfaces use the Java SE 5 classes and interfaces.

- Java™ BluePrints

The blueprints provide guidelines and best practice information on the application of Java EE 5 technologies.

- Application Programmer Interface (API) documentation (Javadocs)
The API documents the Java EE 5 technology classes, interfaces, annotations and exceptions.
- Commercial and open source Java EE application servers and tools
Commercial and open source Java EE servers are available as alternatives to the SDK's Application Server 9 PE. The availability of these commercial or open source servers prevents vendor lock in. That is, any application developed to the specification can be easily ported to execute on a different application server.
- Java EE components and applications
The end purpose of providing the Java EE 5 platform is to enable the creation of software components that can be packaged to create enterprise applications.
The pool of available third-party commercial and open source Java EE applications and component libraries is increasing.

Examining the Java EE Application Architecture

This section describes the Java EE application architecture.

Examining the Component-Container Architecture

The underlying architecture of Java EE applications is the component-container architecture. Figure 1-1 illustrates the component-container architecture.

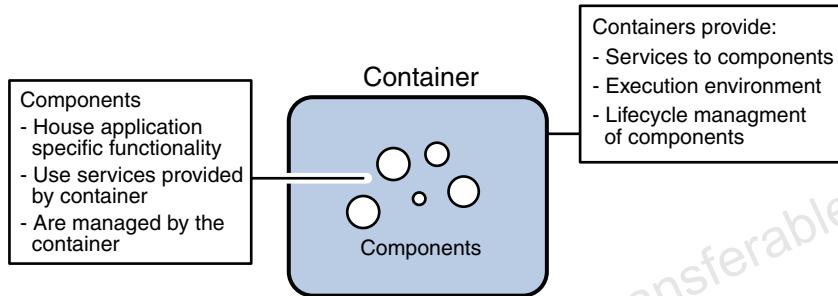


Figure 1-1 Generic Component-Container Architecture

The key features of the component-container architecture are:

- The separation of application specific functionality and application generic functionality.
- Components house application specific functionality.
- Containers house the generic functionality common to all enterprise applications.
- Containers provide:
 - An execution environment for components
 - Services for components

Examining the Java EE Implementation of the Container-Component Architecture

Figure 1-2 shows the Java EE containers associated with the Java EE platform.

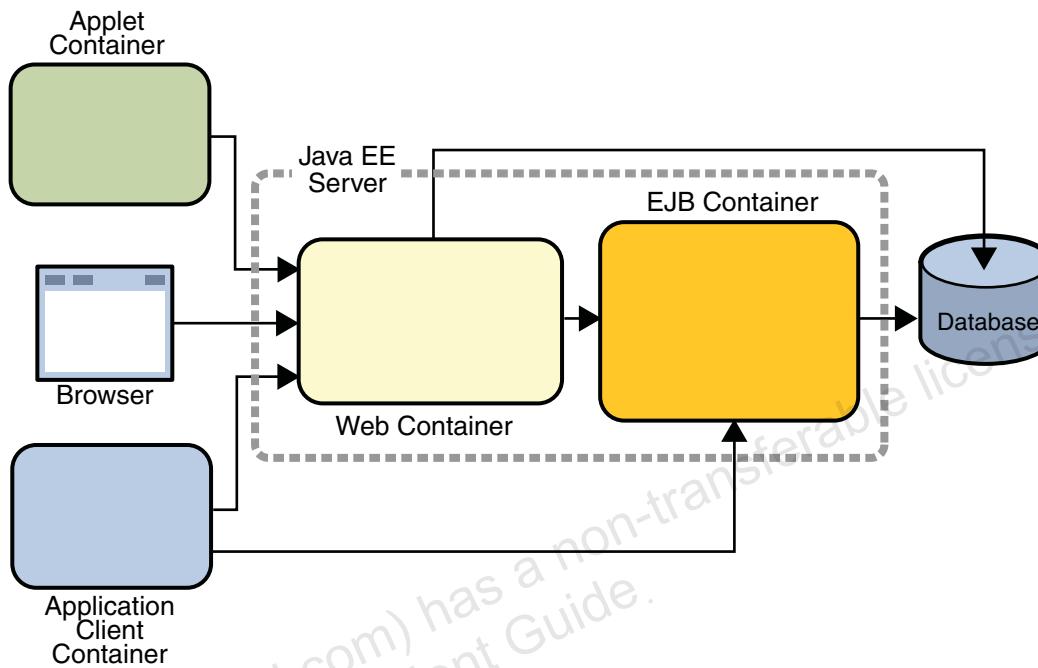


Figure 1-2 Java EE Application Architecture: Generic Elements

Table 1-2 lists the containers used by Java EE applications, the components hosted by each container, and the machines (client or server) that host the containers.

Table 1-2 Java EE Platform Containers

Container	Component Type	Container Host
EJB container	EJBs	EJB container is part of the Java EE application server.
Web container	Servlets and JavaServer Pages (JSP™)	EJB container is part of the Java EE application server.
Applet container	Applets	Applet container can be a web browser or other application that supports the applet programming model. The applet container is hosted on a client machine.
Application client container	Application client component	Application client container is hosted by the client machine.

Examining the Java EE Application Architecture

Table 1-2 Java EE Platform Containers

Container	Component Type	Container Host
Web browser	Web pages	Web browser is hosted by the client machine.



Note – Although web browsers are not a Java EE specification-mandated container, the web browser-to-web page relationship is a container-to-component relationship,

Figure 1-3 shows the Java EE application architecture with the inclusion of the application function-specific components.

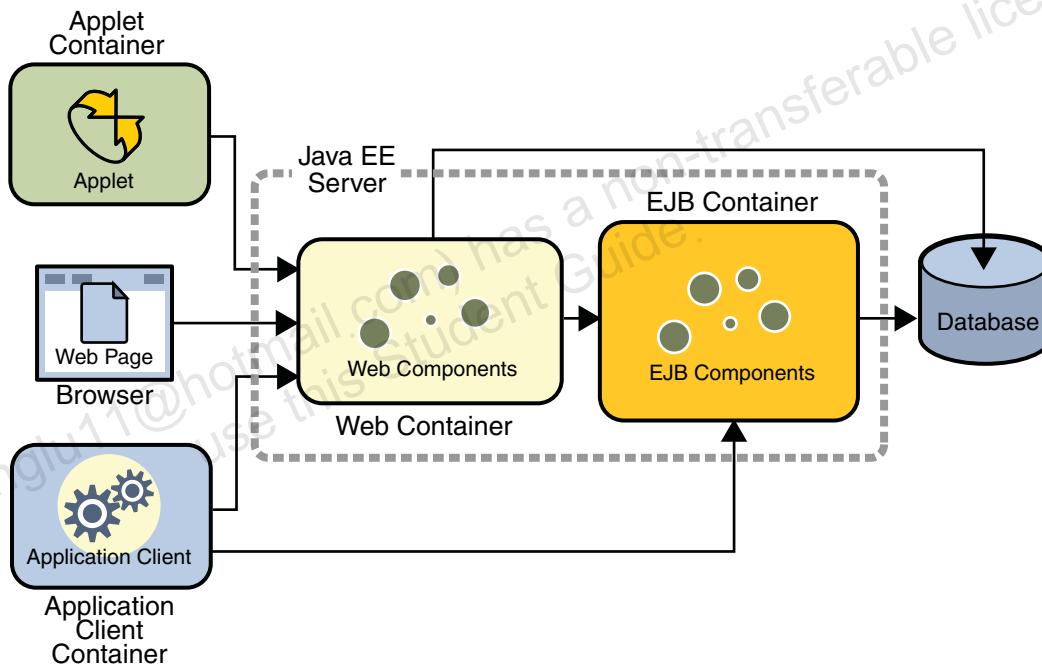


Figure 1-3 Complete EJB Application Architecture

The application-specific parts of a Java EE application are embedded in Java EE application components. Table 1-3 lists the components used in Java EE applications.

Table 1-3 Java EE Platform Components

Component	Container	Primary Application Functionality
Session bean	EJB container	Session beans model business processes. Stateful session beans maintain client session state. Stateless session beans do not maintain client session state.
Message-driven bean	EJB container	Message-driven beans model message system queue event listeners. That is, message-driven beans model asynchronous message receivers.
Servlet	Web container	Servlets are Java programming language classes that dynamically process requests from web clients and construct responses for display on a browser.
JSP	Web container	JSP pages are text-based documents that embed Java technology code and execute as servlets to create dynamic content for display on a browser.
Application client component	Application client container	Application client components model client code and execute in an application client container.
Applet	Applet Container	Applets model client code and execute in an applet container.

Examining Java EE Container Services

The Java EE component model relies on both container-based and platform services for ancillary functionality that is not directly related to the application business logic. The Java EE platform goes beyond the traditional middleware server in terms of the range of services that it offers and the generality of application(s) that can be supported.

Examining the Java EE Application Architecture

Java EE services can be grouped into the following categories.

- Deployment-based services – You request deployment-based services declaratively, using either metadata annotations embedded in code or using XML in a file called a deployment descriptor. Deployment-based services might include:
 - Persistence
 - Transaction
 - Security
 - Injection
- Inherent services – The container automatically supplies inherent services to components on an as-needed basis. Inherent services include:
 - Life-cycle
 - Threading
 - Remote object communication, such as RMI and CORBA
- Vendor-specific functionality – Vendor-specific functionality is added to the Java EE specification, and it is vendor dependent. Vendor-specific functionality can include clustering, which addresses:
 - Scalability
 - Failover
 - Load balancing

Note – You should consider vendor-specific functionality, such as scalability and load balancing, more as a feature of the server than as a service.

- API-based services – You request API-based services programmatically. You must include code in the component to request these services. The following list identifies the most important supporting services and APIs that are included in the Java EE platform:

- Java DataBase Connectivity™ (JDBC™) API for database connectivity

This API provides a vendor-neutral way for applications to complete relational database operations in the Java programming language, with SQL as the query medium.



- JNDI API

This API is used for vendor-neutral access to directory services, such as Network Information Service Plus (NIS+) and Lightweight Directory Access Protocol (LDAP). Java EE applications also make use of the JNDI API to locate components and services using a central lookup service.

- RMI over Internet Inter-Object Request Broker (ORB) Protocol (IIOP) and the interface definition language (IDL) for the Java application

Together, these services form a CORBA-compliant remote method invocation strategy. The strength of this strategy over Java RMI schemes is that it is programming language-independent, so not all clients of a particular enterprise application need to be written in the Java programming language.

- JavaMail API and JavaBeans™ Activation Framework (JAF) API

These APIs allow a Java EE software application to send email messages in a vendor-independent way.

- Java EE Connector Architecture

This API allows for the provision of integration modules, called *resource adapters*, for legacy systems in a way that is independent of the application server vendor.

- JMS API

This is an API for sending and receiving asynchronous messages.

- JTA

This is an API by which software components can initiate and monitor distributed transactions. Java Transaction Service (JTS) specifies the implementation of a transaction manager, which supports the JTA Specification at the high-level, and implements the Java programming language mapping of the Object Management Group (OMG) Object Transaction Service (OTS) Specification at the low-level.

- Java Authentication and Authorization Service (JAAS)

In the Java EE platform JAAS might be used to integrate an application server with an external security infrastructure.

Examining the Java EE Application Architecture

- Java API for XML Processing (JAXP)
This API provides access to XML parsers. The parsers themselves might be vendor-specific, but as long as they implement the JAXP interfaces, vendor distinctions should be invisible to the application programmer.
- Web services integration features
These services include, Simple Object Access Protocol (SOAP) for the Java application, SOAP with Attachments API for Java™ (SAAJ), Java API for XML Registries (JAXR), and JAX-RPC. Together these services allow Java EE software applications to respond to, and to initiate XML-based RPC and messaging operations, which provides a full web services platform.
- JMX
This API exposes the internal operation of the application server and its components for control and monitoring vendor-neutral management tools.
- Timer services
These services provide the ability to run scheduled background tasks.

Figure 1-4 shows how the Java EE containers have access to a range of important API-based services, as defined by the Java EE specification.

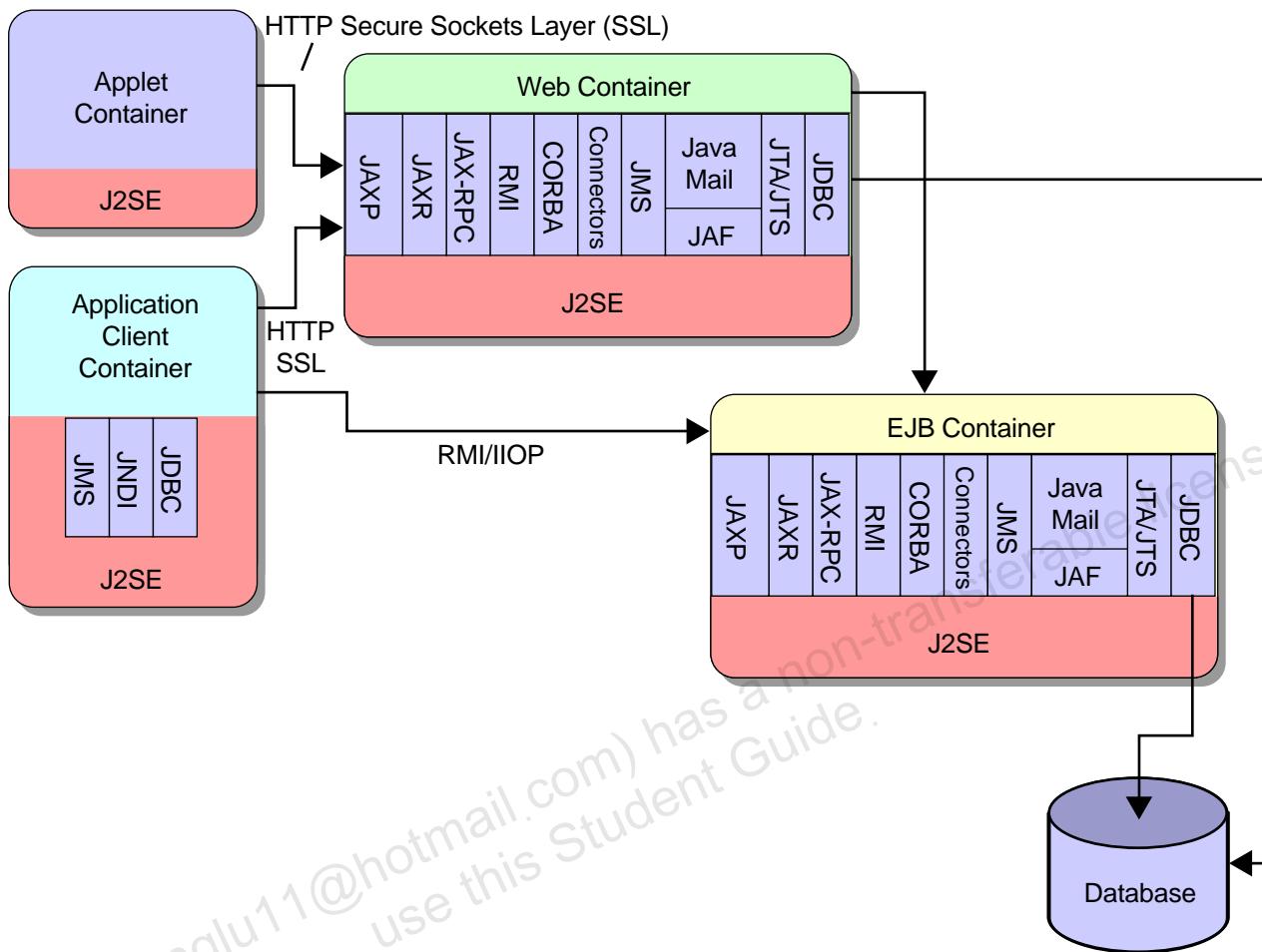


Figure 1-4 Java EE Service Infrastructure

Note – In addition to the API-based services shown in Figure 1-4, the application client, web, and EJB containers also provide deployment-based services, such as security and injection, and inherent services, such as life-cycle management.



Examine the EJB Application Creation Process

Examine the EJB Application Creation Process

The following steps provide a high level overview of a process you could follow to create an EJB application. This process assumes that you have already selected the Java EE application server that will host your application.

1. Use your preferred object oriented methodology to analyze the business problem. The analysis should identify the following:
 - The functional requirements of the application that is the business use cases.
 - The non-functional requirements of the application, such as scalability requirements
 - The business logic that services the use cases
 - Persistence data required by the applicationThis is a design step and the outputs of this step are normally expressed as UML notations.
2. Analyze the inter-tier communications requirements:
 - Identify the synchronous communication requirements of the application
 - Identify the asynchronous communication requirements of the application
3. Use entity classes to model the persistence data.

This is an implementation step. The output of this step is a set of Java technology classes and associated configuration data, such as object-relational mapping information.

Note – As of EJB 3.0 specification, the configuration data is usually included in the entity classes using metadata annotations. This is true for both session beans and message-driven beans. Before the EJB 3.0 specification, configuration data was contained in an XML file. This file was referred to as a deployment descriptor file.



4. Use session beans to model the server-side business logic and synchronous service façade for the server-side components.

In practice, this step requires an in-depth analysis of the server-side business logic requirements and includes the application of Java EE patterns to meet the non-functional requirements of the application.

This is an implementation step. The output of this step is a set of session bean components and associated helper classes. Session bean components consist of (session bean) business interfaces, corresponding implementation cases and associated configuration data. With EJB 3.0, the configuration data is usually included in the enterprise bean classes using metadata annotations.

5. Use the JMS API and message-driven beans to model the asynchronous communication requirements of the application.

This is an implementation step. The output of this step is a set of message-driven bean components and associated helper classes.

6. Create the client.

This is an implementation step. The output of this step is the client code.

7. Assemble and package the application.

Application assembly is the process of configuring the individual Java EE components (enterprise beans) created in the previous steps to work together as an application.

Packaging is the process of combining individual enterprise beans and their associated classes into one or more Java EE modules. A Java EE module represents the basic unit of composition of a Java EE application. In some circumstances, a single Java EE module can constitute a complete application. A Java EE module is also the smallest unit you can deploy in an application server.

8. Deploy the server-side components.

Deployment is the process of configuring Java EE modules to execute in a specific application server.

9. Package and distribute the client.

The objectives of this step is to produce a client in a format that is ready for distribution and deployment. The actual activities performed depend on the type of client produced (standalone Java SE client, application client, web client, web service client).

Examine the EJB Application Creation Process

In practice, enterprise bean components are combined with other Java EE components, such as servlets and JSPs to form a Java EE application. Figure 1-5 illustrates the creation of Java EE components and their assembly into Java EE modules that are then deployed as a Java EE application in an application server.

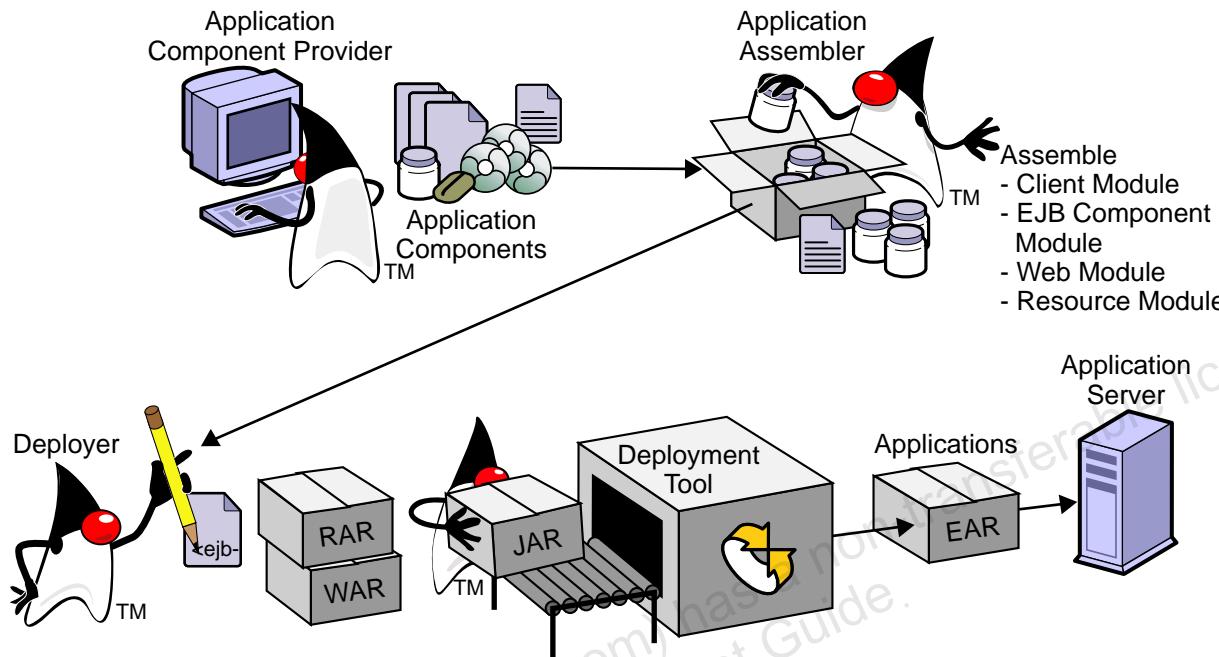


Figure 1-5 Java EE Application Development Process

The roles shown in Figure 1-5 are described in Table 1-4.

Table 1-4 Java EE Platform Roles

Role	Primary Function
Application Component Provider	There are multiple roles for Application Component Providers, including HTML document designers, document programmers, and enterprise bean developers. These roles use tools to produce Java EE applications and components.
Application Assembler	The Application Assembler takes a set of components developed by Application Component Providers and assembles them into a complete Java EE application delivered in the form of an Enterprise Archive (.ear) file.

Table 1-4 Java EE Platform Roles

Role	Primary Function
Deployer	The Deployer is responsible for deploying application clients, web applications, and Enterprise JavaBeans components into a specific operational environment. The deployer's primary tasks are installation into a specific server, configuration, and application execution start up.
System Administrator	The System Administrator is responsible for the configuration and administration of the enterprise's computing and networking infrastructure. The System Administrator is also responsible for overseeing the runtime well-being of the deployed Java EE applications.
Java EE Product Provider	A Java EE Product Provider is the implementer and supplier of a Java EE product that includes the component containers, Java EE platform APIs, and other features defined in this specification.
Tool Provider	A Tool Provider provides tools used for the development and packaging of application components.

Comparing Java EE Application Development With Traditional Enterprise Application Development

A key feature of the Java EE platform is the strict separation of the application components from the general services and infrastructure. One of the main benefits of the Java EE platform for the application developer is that it makes it possible for developers to focus on the application business logic while leveraging the supporting services and platform infrastructure provided by the container and application server vendor. For example, in an online banking application, the application component developers need to code the logic that underlies the transfer of funds from one account to another, but they do not need to be concerned about managing database concurrency or data integrity in the event of a failure. The application server infrastructure and services are responsible for these functions.

Figure 1-6 contrasts the tasks that are required of an application developer who builds an application and supporting services from the ground up to those of a developer who relies on an application component server for service and platform level functions.

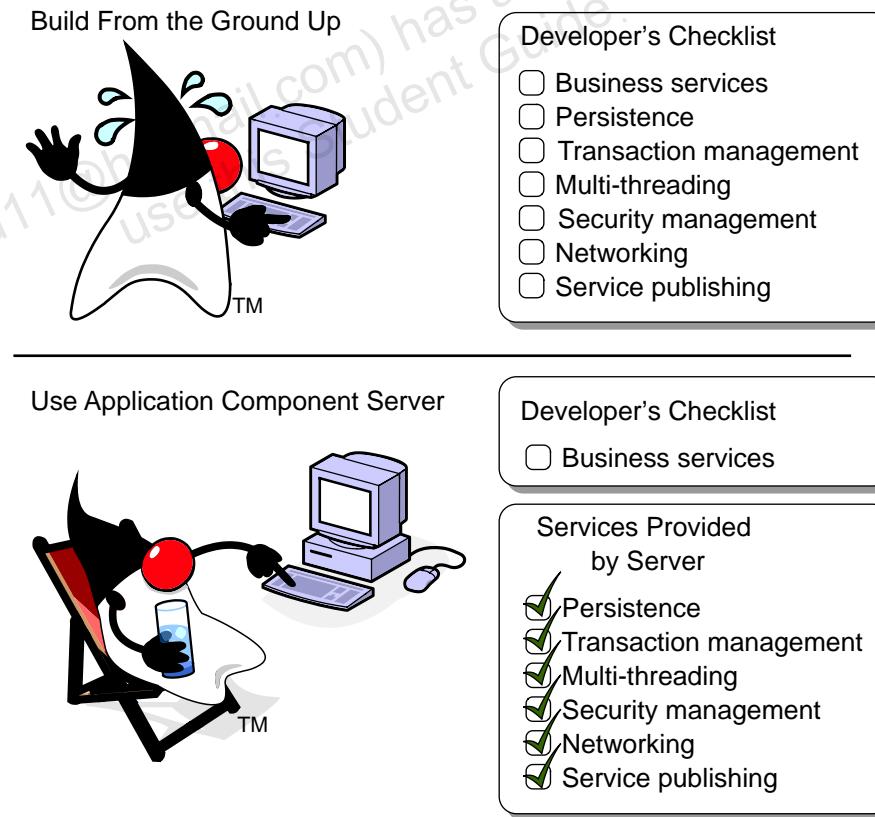


Figure 1-6 Advantages of Using Server-Provided Services

As you can see, relying on a component server for application support services dramatically reduces the amount of coding required of the application component developer.

Module 2

Introducing the Auction Application

Objectives

Upon completion of this module, you should be able to:

- Describe the auction application
- Define the domain objects of the auction application
- Describe the implementation model for the auction system

Additional Resources



Additional resources – The following reference provides additional information on the topics described in this module:

- Module 15, “Using EJB Technology Best Practices”

Auction Application Use Cases

The auction system provides a problem domain that is easily understandable, allows for end-to-end development during labs, and still has enough complexity to illustrate complex activities associated with creating an EJB application. The auction application is similar to many existing online auction systems.

The auction system views users as one of the following user types (actors):

- Seller
- Bidder
- Administrator

Figure 2-1 shows the actors and the use cases associated with each actor.

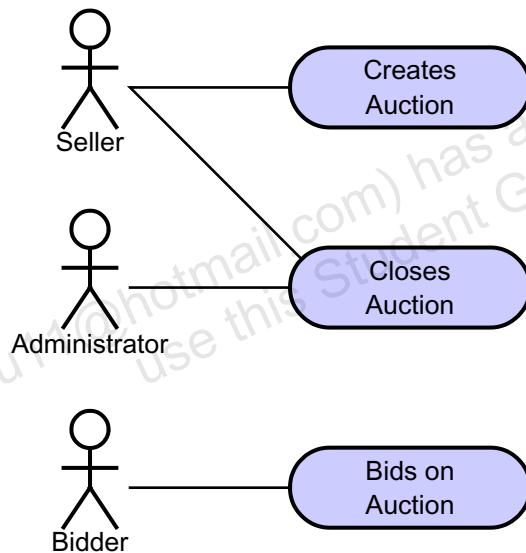


Figure 2-1 Auction Application Use Cases

The auction application must implement the following use cases:

- Create-auction use case – The create-auction use case is executed by the seller. The seller creates an auction by placing an item for sale. The seller specifies the start amount. In addition, each auction has a start time, close time, and bid increment.

Auction Application Use Cases

- Close-auction use case – The close-auction use case is executed by the seller and administrator. The administrator can withdraw any auction. Sellers can withdraw only auctions they created.
- Bid-on-auction use case – The bid-on-auction use case is executed by the bidder. For a bid to be accepted, it must be greater than the sum of the current highest bid by at least the bid increment amount.

Analyzing the Auction System

This section analyzes the auction system and describes the domain objects in the simplified auction system used in the lab exercises. It also describes the relationships among the domain objects.

Auction Application Domain Objects

Figure 2-2 illustrates the domain objects of the auction application and the relationships among the domain objects.

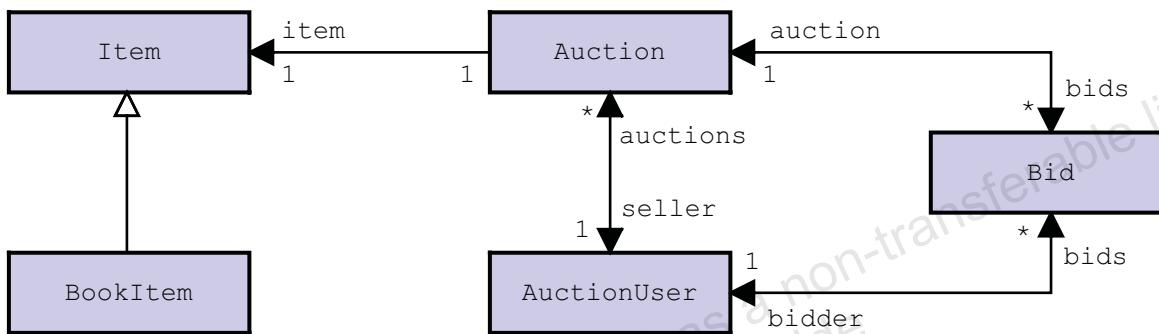


Figure 2-2 Auction Application's Data Relationship Model

The auction system has many design decisions made with the idea of extensibility in mind. For example, the **Auction** object and **Item** object have a one-to-one relationship, and they could feasibly be combined. In a more robust auction system, an unsold item could be re-auctioned. Keeping the item as a separate entity allows you to do that more easily.



Note – A more realistic domain model of an auction system is significantly more complex and includes some kind of credit authorization model and corresponding domain objects, such as credit cards or bank accounts.

Analyzing the Auction System

The auction application is modeled using the following domain objects:

- The Auction domain object – This object represents a single online auction within the auction system. Each Auction object has an association with a seller, an item, and a collection of bids.
- The AuctionUser domain object – This object represents either a seller or a bidder within the auction system. Each AuctionUser object has an association with a credit card, a shipping address, a collection of auctions, and bids that the auction users have placed.
- The Bid domain object – This object represents a purchase offer on an auction sale item. Each Bid object is associated with a bidder (AuctionUser) and an auction.
- The Item domain object – This object represents the article offered for sale in an auction.

The Auction Domain Object

Figure 2-3 illustrates the Auction domain object.

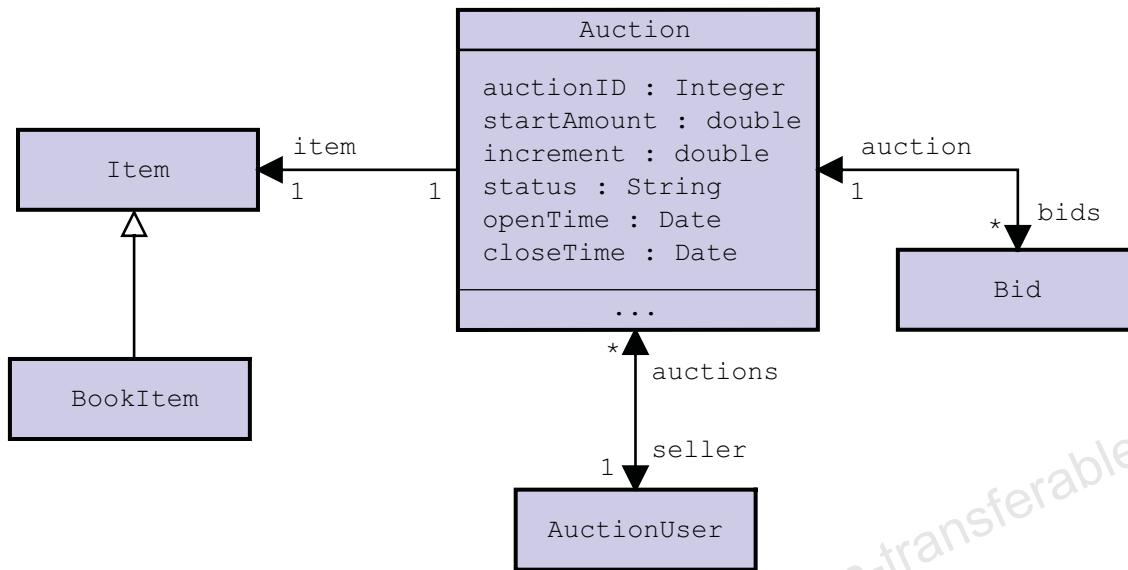


Figure 2-3 The Auction Domain Object

The Auction domain object represents a single online auction within the auction application. The Auction domain object contains the following attributes:

- The **auctionID** attribute – This field represents a unique identity for the Auction object.
- The **startAmount** attribute – This field represents the start amount specified by the seller.
- The **increment** attribute – This field represents the minimum bid increment.
- The **status** attribute – This field represents the auction status. Status values are OPEN, CLOSED, CANCELLED.
- The **openTime** attribute – This field represents the start time of the auction.
- The **closeTime** attribute – This field represents the close time of the auction.

Analyzing the Auction System

Each Auction domain object contains relationships with other domain objects. These relationships are represented by the following associations:

- The item relationship – This field represents the item for sale in the auction.
- The seller relationship – This field represents the AuctionUser object who is selling the item.
- The bids relationship – This field represents the collection of bids placed on the auction.

The AuctionUser Domain Object

Figure 2-4 illustrates the AuctionUser domain object.

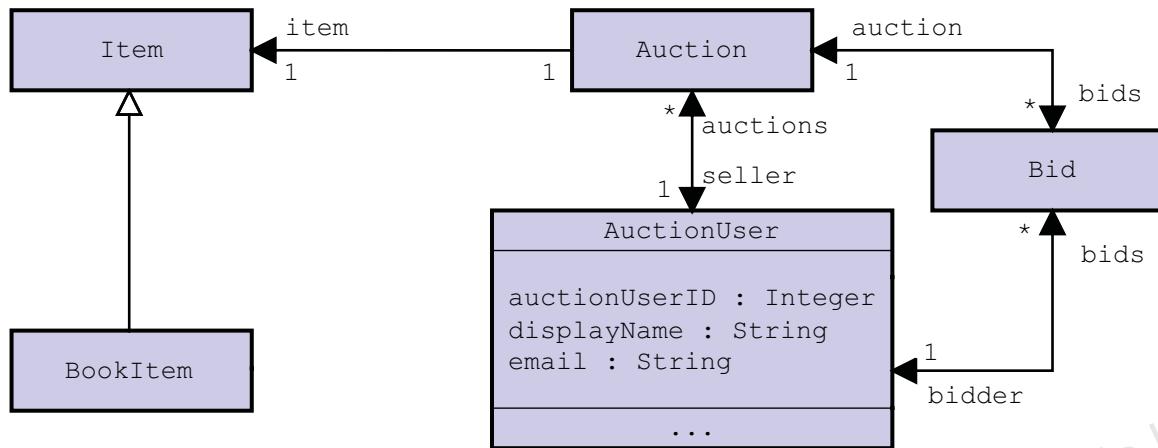


Figure 2-4 The AuctionUser Domain Object

The AuctionUser domain object represents either a seller or a bidder within the auction application. The AuctionUser domain object maps to both the seller and bidder roles in the auction application use cases that are illustrated in Figure 2-1. The AuctionUser domain object contains the following attributes:

- The auctionUserID attribute – This field represents a unique identity for the auction user.
- The displayName attribute – This field represents the display name of the auction user.
- The email attribute – This field represents the email address of the auction user.

Each AuctionUser domain object contains relationships with other domain objects. These relationships are represented by the following associations.

- The auctions relationship – This field represents the auctions that the auction user has participated in as a seller.
- The bids relationship – This field represents the collection of bids placed by the auction user.

The Bid Domain Object

Figure 2-5 illustrates the Bid domain object.

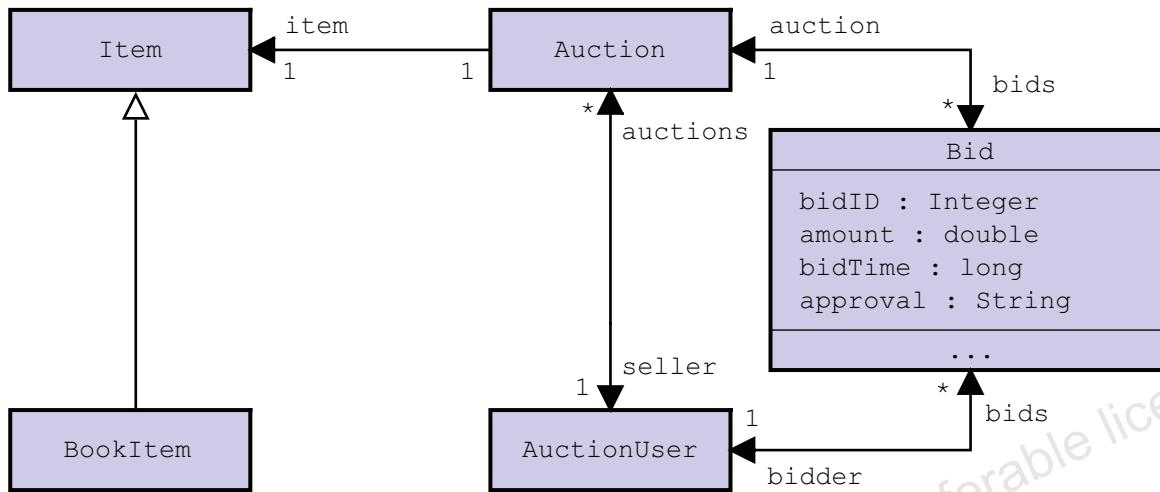


Figure 2-5 The Bid Domain Object

The Bid domain object represents a purchase offer on an auction sale item. Each Bid domain object is associated with a bidder (AuctionUser) and an auction. The Bid domain object contains the following attributes.

- The `bidID` attribute – This field represents a unique identity for the bid.
- The `amount` attribute – This field represents the bid amount.
- The `bidTime` attribute – This field represents the time the bid was made.
- The `approval` attribute – This field represents the (sufficient funds availability check) authorization code associated with the bid.

Each Bid domain object contains relationships with other domain objects. These relationships are represented by the following associations.

- The `auction` relationship – This field represents the auction associated with the bid.
- The `bidder` relationship – This field represents the auction user who placed the bid.

The Item Domain Object

Figure 2-6 illustrates the Item domain object.

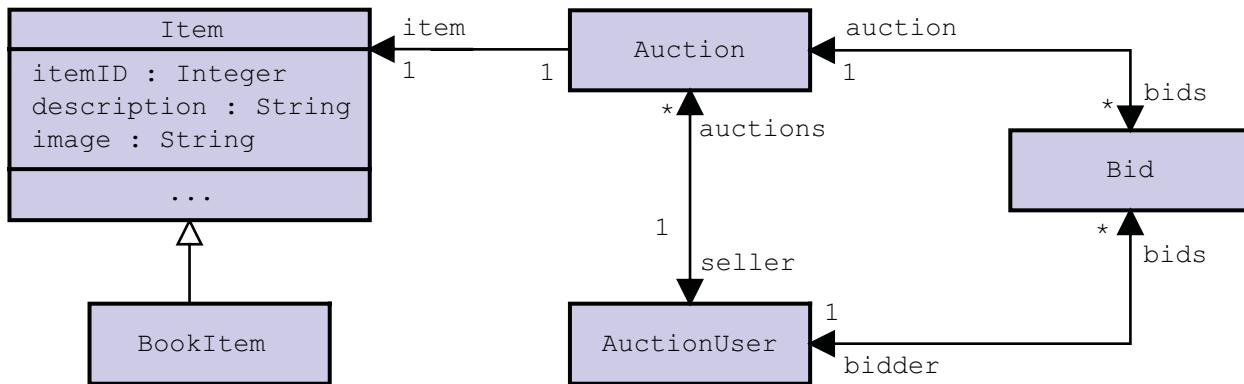


Figure 2-6 The Item Domain Object

The Item domain object represents the article offered for sale in an auction. The Item domain object contains the following attributes:

- The itemID attribute – This field represents a unique identity for the item.
- The description attribute – This field represents the item's description.
- The image attribute – This field represents the location of an image of the item.

The BookItem Domain Object

Figure 2-7 illustrates the BookItem domain object.

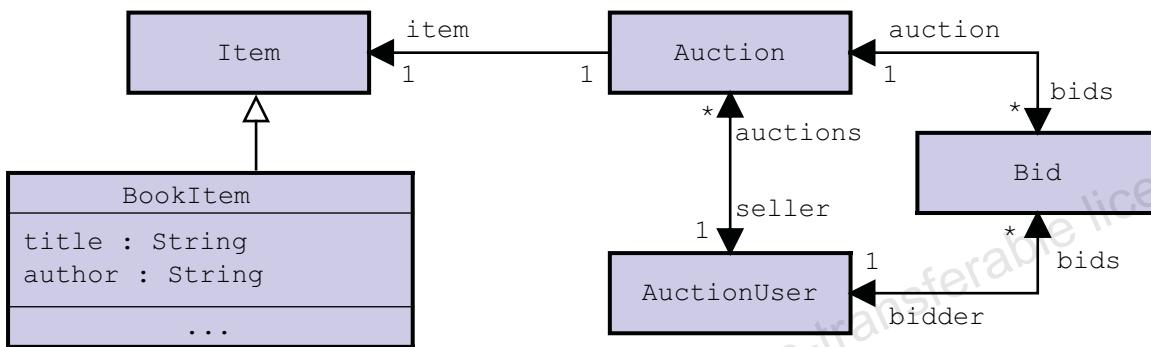


Figure 2-7 The BookItem Domain Object

The BookItem domain object represents the book item. It inherits from the Item domain object. The BookItem domain object contains the following attributes:

- The title attribute – Represents the title of the book.
- The author attribute – Represents the name of the author of the book.

Examining the Implementation Model

Figure 2-8 illustrates the preliminary design of the auction system implementation.

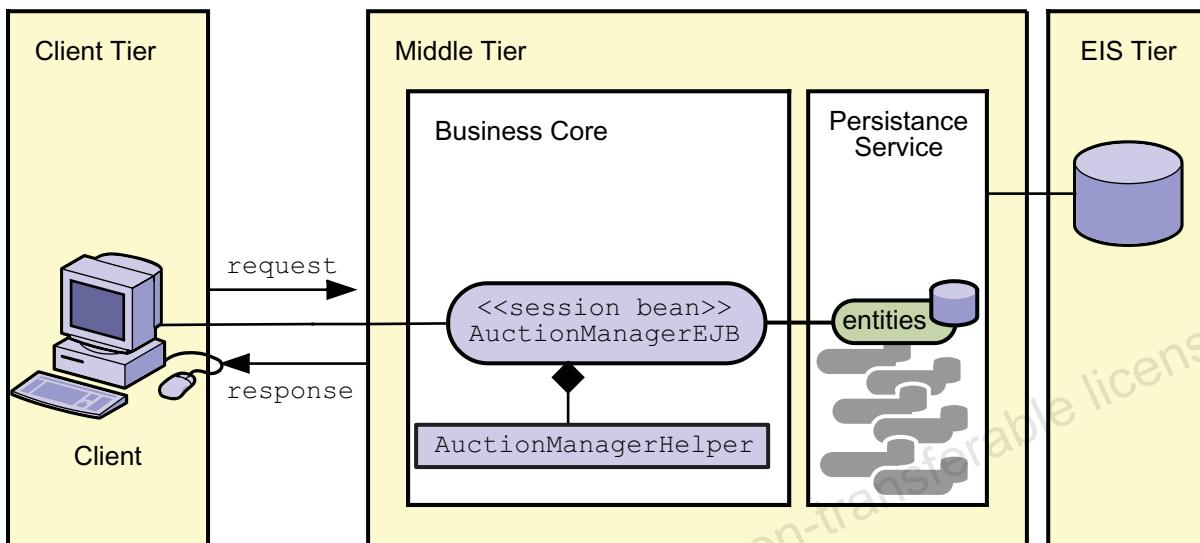


Figure 2-8 Auction System Preliminary Design

The auction system is based on the Java EE technology and the EJB technology architecture. The middle tier consists of a business core and persistence service. The business core consists of EJBs and helper classes that implement the auction application's business logic. The persistence service consists of persistence entities. Persistence entities are objects that represent data stored in the EIS tier (database).

Middle-Tier Subsystems

The primary client interface object in the business core is the `AuctionManager` session bean. The `AuctionManager` functions as a session facade. It receives requests, such as `add auction`, `place bid` place from the client tier. These requests are processed with the assistance of helper objects, such as the `AuctionManagerHelper` and persistence entities. The results are then returned to the client.

Examining the Implementation Model

Figure 2-9 illustrates the persistence entities used by the auction application.

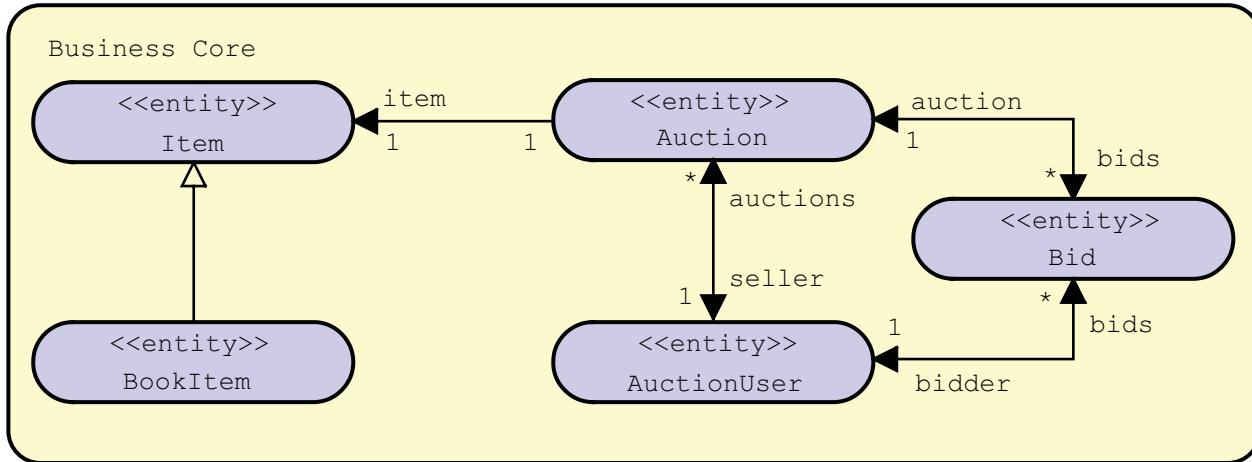


Figure 2-9 Business Core Implementation

Each persistence entity maps to the corresponding domain object in the domain model. Each entity instance is a representation in the object model of a row of data from a database table located in the EIS tier. The EJB container uses a persistence provider to manage entity instances. Each EJB container is associated with a persistence provider. The persistence provider is responsible for synchronizing the data held in entity instances it manages with data in the EIS tier.

Module 3

Implementing EJB 3.0 Session Beans

Objectives

Upon completion of this module, you should be able to:

- Examine session beans
- Create session beans: Essential tasks
- Create session beans: Add life-cycle event handlers
- Package and deploy session beans
- Create a session bean client

Additional Resources



Additional resources – The following reference provides additional information on the topics described in this module:

- Sun Microsystems, “JSR 220: Enterprise JavaBeans™, Version 3.0 EJB Core Contracts and Requirements, Chapter 4 and 18.” [<https://sdlc3e.sun.com/ECom/EComActionServlet;jsessionid=CEAAE57A3BAB8A76D4555E3C5A1F4031>], accessed July 25, 2006.
- Sun Microsystems, “JSR 220: Enterprise JavaBeans™, Version 3.0 EJB 3.0 Simplified API.” [<https://sdlc3e.sun.com/ECom/EComActionServlet;jsessionid=CEAAE57A3BAB8A76D4555E3C5A1F4031>], accessed July 25, 2006.
- Sun Microsystems, “Java 2™ Platform Enterprise Edition Specification, v5.0, Chapter 8.” [<https://sdlc3e.sun.com/ECom/EComActionServlet;jsessionid=CEAAE57A3BAB8A76D4555E3C5A1F4031>], accessed July 25, 2006.

Comparison of Stateless and Stateful Behaviors

A session bean is a server-side client resource. Session beans essentially function as a service façade making available server-side services to a client. Session beans are classified as stateful and stateless. Stateful session beans retain client state in between client invocation of the session bean methods. Stateless session beans do not retain client state.

Most Java EE applications that have user interfaces require the retention of state data. Typical examples of data that need to be retained are multi-page forms and shopping carts. State can be stored in any number of places including:

- User tier – State could be stored in objects in a Swing-based application or in cookies with web-based applications. This is typically avoided because of security and performance reasons. Large amounts of data would have to be frequently transferred between the client and server tiers.
- Web tier – Web clients can have their state stored on the web tier in `javax.servlet.http.HttpSession` objects. If the web and EJB business tiers are running in the same application server, there will not be a drastic performance penalty if local business components are used. State management should be done in a centralized business tier if non-web clients are used for reusability.
- Database tier – Sometimes used to achieve increased retention time of state. This option can provide increased failover capabilities when a vendor's Java EE application server lacks or has not been configured with advanced clustering and failover features.
- Business or EJB tier – Session EJBs can hold client state. Since both web and Swing UI applications can store state in the EJB tier, this allows most state management code to be written only once. This is typically easier and faster than storing state in a database.

In a Java EE 5 application, the business tier contains session beans. Session beans can be configured as stateless or stateful. The statefulness of a bean depends on the type of business function it performs:

- In a stateless client-service interaction, no client-specific information is maintained beyond the duration of a single method invocation.
- Stateful services require that information obtained during one method invocation be available during subsequent method calls:
 - Shopping carts
 - Multi-page data entry
 - Online banking

Stateless Session Bean Operational Characteristics

A stateless session bean has the following characteristics:

- The bean does not retain client-specific information.
- A client might not get the same session bean instance.
- Any number of client requests can be handled by the same session bean instance. This has profound performance advantages.

Memory usage is decreased and performance increased when using stateless session beans when compared to unnecessarily configuring a session bean as stateful.

Note – A common misconception is that stateless session beans cannot have instance variables. It is valid for stateless session beans to have instance variables, however, clients cannot get the same bean on every method call, so from a client's perspective, the value of the variables can appear to change.

Figure 3-1 illustrates the allocation of a pooled instance of a stateless session bean to service client requests made through the session bean's business interface.

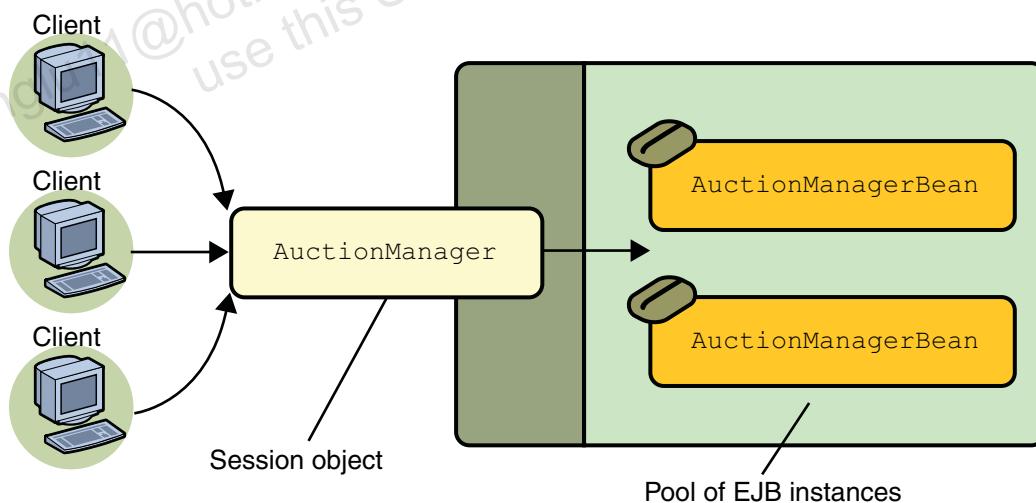


Figure 3-1 Stateless Session Bean Cardinality



Note – You can view a session object as an EJB container-created wrapper object of the session bean instance. It enables the container to provide container services (such as transaction services, security services, and so on) to the session bean instance. It does so by interposing itself between the client and the session bean instance.

Stateful Session Bean Operational Characteristics

A stateful session bean has the following characteristics:

- The bean belongs to a particular client for an entire conversation or session.
- The client connection exists until the client removes the bean or the session times out. Session bean time outs are typically based on a vendor-specific inactivity timeout threshold.
- The container maintains a separate EJB instance for each client. Stateful session beans require more memory per client than stateless session beans.

Note – Another common misconception is that stateless session beans scale better than stateful session beans. There is always a cost to maintain client state, maintaining more state than is needed or using stateful session beans when a stateless bean would be adequate negatively impact performance. If an application must store client state, then stateful session beans are an effective way to achieve it.

Figure 3-2 illustrates the use of dedicated stateful session bean instances to service clients.

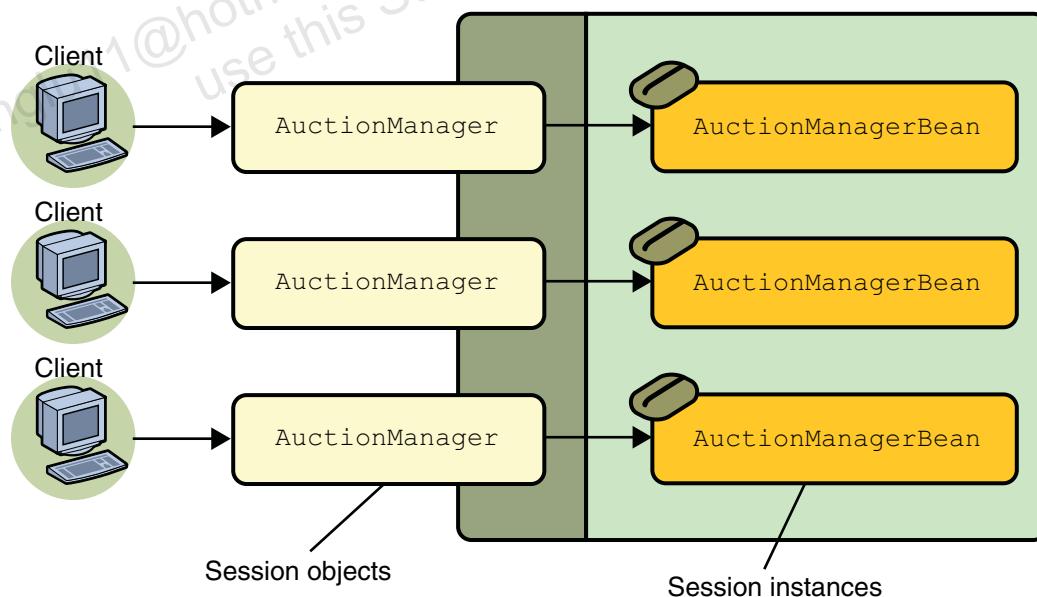


Figure 3-2 Stateful Session Bean Cardinality

Creating Session Beans: Essential Tasks

This section provides an overview of the essential tasks you need to perform to create a EJB 3.0 session bean. To create a session bean, you are required to perform the following tasks:

- Declare a business interface for the session bean.
- Create the session bean class that implements the business interface.
- Configure the session bean by either annotating the session bean class or providing a deployment descriptor (DD).

The following subsections describe these tasks in more detail.

Declaring a Business Interface for the Session Bean

An EJB 3.0 session bean component must use an interface to specify the business services it provides. Code 3-1 contains an example of a business service interface.

Code 3-1 Session Bean Business Interface

```
1 public interface Calculator {  
2     int add(int a, int b);  
3     int subtract(int a, int b);  
4 }
```

Code 3-1 shows a plain Java technology interface that you can use as a business interface to create a session bean. The following frequently asked questions (FAQ) provide additional guidelines to assist you with the task of declaring session bean business interfaces.

- What are the requirements for the session bean business interface?

A session beans business interface is a plain Java technology interface. The business interface can have super interfaces. The business methods can throw arbitrary application exceptions.

- How are business interfaces classified?

A business interface to a session bean can be:

- A local interface

A local interface provides access to code that executes on the same JVM™ machine as the session bean component.

Creating Session Beans: Essential Tasks

- A remote interface

A remote interface provides access to code that executes on a JVM that is different (remote) to that of the session bean component.

A session bean can define both a local and a remote business interface.

- How can a business interface be designated a local interface?

To specify an interface as a local interface, use the `Local` annotation on the interface or the bean class. For example:

```
@Local public interface Calculator { ... }
```

Alternatively, you can use the `Local` annotation on the bean class. For example:

```
@Local public class CalculatorBean implements Calculator
```

- How can a business interface be designated a remote interface?

To specify an interface as a remote interface use the `Remote` annotation on the interface or the bean class. For example:

```
@Remote public interface Calculator { ... }
```

Alternatively you can use the `Remote` annotation on the bean class. For example,

```
@Remote public class CalculatorBean implements Calculator
```

- Are there any additional requirements for remote interfaces?
 - The remote business interface is not required or expected to be a `java.rmi.Remote` interface.
 - A remote interface method must not expose local interface types, timers, or timer handles.
 - The arguments and return types of the methods must be of valid types for RMI/IOP.
 - The throws clauses of the methods of the interface should not include the `java.rmi.RemoteException`.
- Can a session bean have more than one business interface?

A session bean class can implement multiple interfaces. To designate an interface as the business interface of a bean class, use any of the following options:

- Local or remote annotation on the interface
- Local or remote annotation on the implementing bean class
- Local or remote designation in the deployment descriptor associated with the bean class

Creating the Session Bean Class That Implements the Business Interface

The EJB 3.0 specification requires a session bean class to implement the methods declared in its associated business interface. Code 3-3 shows a stateless session bean class implementing the remote business interface shown in Code 3-2.

Code 3-2 Session Bean Remote Business Interface

```
1 @Remote public interface Calculator {  
2     int add(int a, int b);  
3     int subtract(int a, int b);  
4 }
```

Code 3-3 A Stateless Session Bean Class

```
1 @Stateless public class CalculatorBean implements Calculator {  
2     public int add(int a, int b) {  
3         return a + b;  
4     }  
5  
6     public int subtract(int a, int b) {  
7         return a - b;  
8     }  
9 }
```

The answers associated with the following questions provide additional guidelines to assist you with the task of declaring session bean classes.

- What are the requirements for the session bean class?

The session bean class must comply with the following design rules:

- The class must be a top-level class. In addition, the class must be defined as public, must not be final, and must not be abstract.
- The class must have a public constructor that takes no parameters. The container uses this constructor to create instances of the session bean class.
- The class must not define the `finalize` method.
- The class must implement the methods of the bean's business interface(s), if any.

Annotating the Session Bean Class

The EJB3.0 specification provides a number of annotations you can use to supply metadata about the session bean class. Code 3-4 shows an example of a stateless session bean class that demonstrates the use of annotations. You can identify the annotations by the leading @ character that prepends each annotation.

Code 3-4 Session Bean Code With Annotation

```

1  @Stateless public class CalculatorBean implements Calculator {
2      public int add(int a, int b) {
3          return a + b;
4      }
5
6      public int subtract(int a, int b) {
7          return a - b;
8      }
9 }
```

Table 3-1 contains a brief explanation of the most commonly used annotations applicable to a session bean.

Table 3-1 Common Annotations Applicable to Session Beans

Annotation	Applies To	Description
Stateful Stateless	Session bean class	The Stateful or Stateless annotation is used to denote the type of session bean. You are required to supply the session bean type by either annotating the bean class or by supplying the corresponding deployment descriptor (DD) element.
Local Remote	Session bean class or its business interface	The Local annotation designates a local interface of the bean. The Remote annotation designates a remote interface of the bean. The absence of Local or Remote implies that the bean class implements a single local interface.
Remove	Stateful session bean class	The Remove annotation is used to denote a remove method of a stateful session bean.

The following types of annotations are also applicable to session beans.

- Transaction-related annotations
Transaction-related annotations are described in Module 11, "Implementing Transactions".
- Security-related annotations
Security-related annotations are described in Module 14, "Implementing Security".
- Life-cycle callback handler method annotations
Life-cycle callback handler method annotations are described in "Creating Session Beans: Adding Life-Cycle Event Handlers" on page 3-12.

Creating Session Beans: Adding Life-Cycle Event Handlers

With EJB 3.0, you are not required to provide life-cycle event handlers for session beans. However, session beans do generate life-cycle events and you can optionally provide event handlers for these methods.

A stateless session bean generates the following life-cycle events:

- PostConstruct callback
- PreDestroy callback

The stateless session bean life-cycle diagram shown in Figure 3-3 provides the context for the stateless session beans life-cycle events.

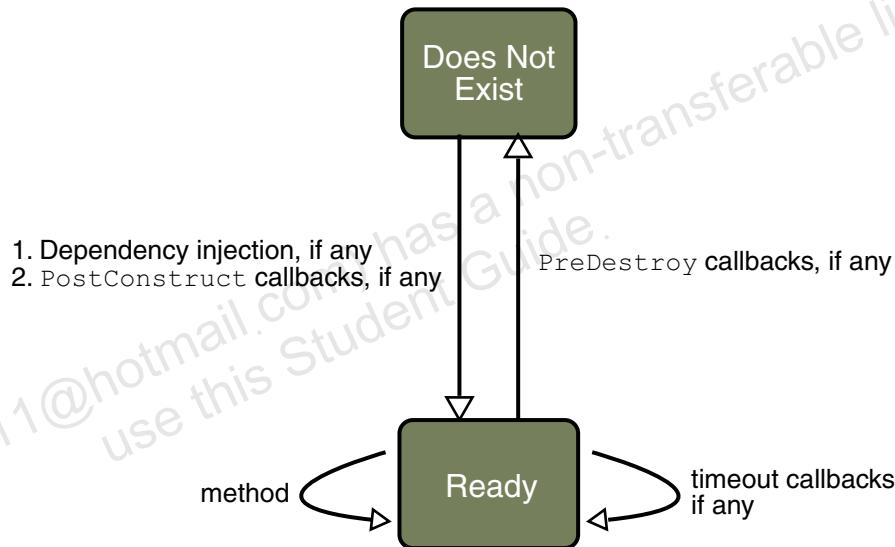


Figure 3-3 Life Cycle of a Stateless Session Bean

A stateful session bean generates the following life-cycle events:

- PostConstruct callback
- PreDestroy callback
- PostActivate callback
- PrePassivate callback

The stateful session bean life-cycle diagram shown in Figure 3-4 provides the context for the stateful session beans life-cycle events.

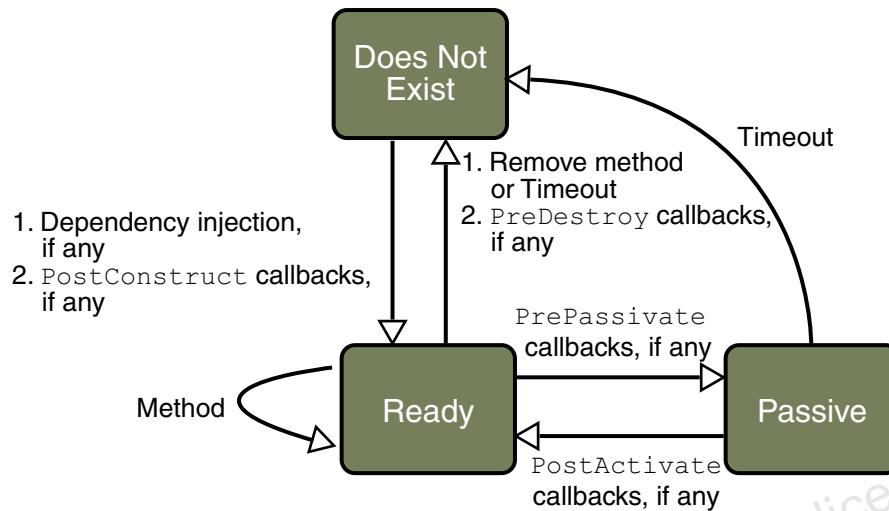


Figure 3-4 Life Cycle of a Stateful Session Bean

Defining Life-Cycle Event Handlers

The rules to define lifecycle event handlers for both stateless and stateful session beans are the same. You can define the event handler in the bean class.

The following rules apply to defining a callback method in the bean class.

- A callback method is designated using the appropriate callback annotation (`PostConstruct`, `PreDestroy`, `PostActivate`, `PreActivate`)
- The same method can be designated to handle multiple callback events.
- Each callback event can only have one callback method designated to handle it.
- A callback method can access entries in the bean's environment.
- Callback methods can throw runtime exceptions. A runtime exception thrown by a callback method that executes within a transaction causes that transaction to be rolled back.
- Callback methods must not throw application exceptions.
- Callback methods can have any type of access modifier (`public`, `default`, `protected` or `private`).

Creating Session Beans: Adding Life-Cycle Event Handlers

Code 3-5 shows an example of a life-cycle event handler method (`endShoppingCart`) defined within a bean class.

Code 3-5 Example of Callback Method in Bean Class

```

1 import javax.ejb.*;
2 import java.util.*;
3
4 @Stateful public class ShoppingCartBean implements ShoppingCart {
5     private float total;
6     private Vector productCodes;
7     public int someShoppingMethod(){...};
8     //...
9     @PreDestroy private void endShoppingCart() {...};
10 }
```

A callback method defined in a bean class is required to have the following signature:

anyAccessModifier void methodName()

Note – The method can have any name and any access modifier. The return type must be void and it must have zero arguments.



The SessionContext Object

Use the `SessionContext` object to access EJB objects, obtain current transaction status, and obtain security information. `SessionContext` extends `EJBContext`.

```

1 import javax.ejb.*;
2 import javax.annotation.*;
3
4 @Stateful
5 public class BankingBean implements Bank {
6
7     @Resource private javax.ejb.SessionContext context;
8
9 }
```

Session Bean Packaging and Deployment

To package and deploy a session bean, you must perform the following tasks.

- Optionally create a DD file for the session bean component
- Create a deployable session bean component archive
- Deploy the session bean component archive

Introducing DDs

This section provides an introduction to DDs by using a FAQ list to highlight the information you need to know about DDs.

- What is a DD?

A DD is an XML file that provides configuration information to the application server to configure the enterprise bean component during deployment.

- Are there different types of DDs?

There are two main types:

- Enterprise bean component DD

Enterprise bean component DD file is named `ejb-jar.xml`. The `ejb-jar.xml` file contains deployment information for session, entity, and message-driven beans. Component DDs contain the following types of configuration information.

- Structural information

Structural information describes the structure of an enterprise bean and declares an enterprise bean's external dependencies.

- Application assembly information

Application assembly information describes how the enterprise bean (or beans) in the ejb-jar file is composed into a larger application deployment unit.

A single `ejb-jar.xml` file can contain deployment information for one or more enterprise beans. Refer to chapter 18 of JSR 220: Enterprise JavaBeans™, Version 3.0 EJB Core Contracts and Requirements for the XML schema for DDs associated with enterprise bean components.

Session Bean Packaging and Deployment

- Java EE application DD
The Java EE DD file is named `application.xml`. It contains deployment information for Java EE application. Refer to chapter 8 of Java 2™ Platform Enterprise Edition Specification, v5.0, Chapter 8 for the XML schema for DDs associated with Java EE applications.
- Is a DD required?
As of EJB 3.0, configuration information for enterprise components can be embedded using metadata annotations in the bean component's source code.
If all the configuration information needed for an enterprise bean is supplied using metadata annotations you are no longer required to supply a DD. You are required to supply a DD if any of the required configuration information was not supplied using metadata annotations. Optionally you can use a DD to override any application assembly information originally embedded in the components source code using metadata annotations. However, you must not override structural information.
- How does an enterprise bean DD relate to an enterprise bean component archive?
The deployment DD for an enterprise bean is part of the beans component archive. In addition to the DD, an enterprise bean's component archive also contains all the enterprise bean classes.

Example of a DD for an Enterprise Bean

Code 3-6 contains an example of a DD for a session bean.

Code 3-6 Session Bean Deployment Descriptor Example

```

1 <ejb-jar>
2   <enterprise-beans>
3     <session>
4       <ejb-name>ReportBean</ejb-name>
5       <business-remote>services.Report</business-remote>
6       <ejb-class>services.ReportBean</ejb-class>
7       <session-type>Stateful</session-type>
8       <transaction-type>Container</transaction-type>
9     </session>
10    <!-- Deployment information for additional beans goes here-->
11  </enterprise-beans>
```

```
12
13 <assembly-descriptor>
14   <!-- Add assembly information here-->
15 </assembly-descriptor>
16 </ejb-jar>
```

The DD shown in Code 3-6 contains deployment information for a single session bean. You can add deployment information for additional enterprise beans at the same level as the session element.

Creating a Session Bean Component Archive

A session bean component archive is an EJB module that is sometimes referred to as an EJB JAR file. You can create EJB JAR files using a GUI associated with an IDE or a Java EE application server. Alternatively, you can create the EJB JAR file using a command-line tool (utility), such as the Java technology archive tool `jar`. The following steps outline the process of creating an EJB JAR file using the `jar` utility.

1. Create a working directory structure.
 - a. Create a root directory.
 - b. Off the root directory, create the package directory structure corresponding to the EJB component classes and helper classes.
2. Copy the EJB component class files and helper classes into their corresponding subdirectories. For example, you should include the following classes:
 - The enterprise bean implementation class, business interfaces, and their super classes
 - All classes referenced by the enterprise bean's class, interfaces and their super classes

An EJB module's JAR file must *not* contain the following:

- Java EE platform API classes and Java SE platform API classes
For example, the JAR file must not contain classes, such as `java.lang.String` or `javax.ejb.EJBException`.

3. Create a `META-INF` subdirectory off the root directory.
4. If a DD is used to configure the enterprise bean, then copy the DD file (`ejb-jar.xml`) for the EJB components into the `META-INF` directory.

You need to examine the DD file to ensure that all required elements have been supplied for the EJB components included in the EJB JAR archive.

Most application servers also require you to include a vendor-specific DD in the META-INF directory. For example, the Java EE 1.5 Software Development Kit (SDK) application server names this file sun-ejb-jar.xml.

5. From a command line, execute the jar utility to create the EJB JAR archive.

If you examine the created EJB JAR archive, you will note that it includes a manifest file called MANIFEST.MF in the META-INF directory.

Figure 3-5 illustrates the directory structure and class files used to create the auction system EJB module's JAR file.

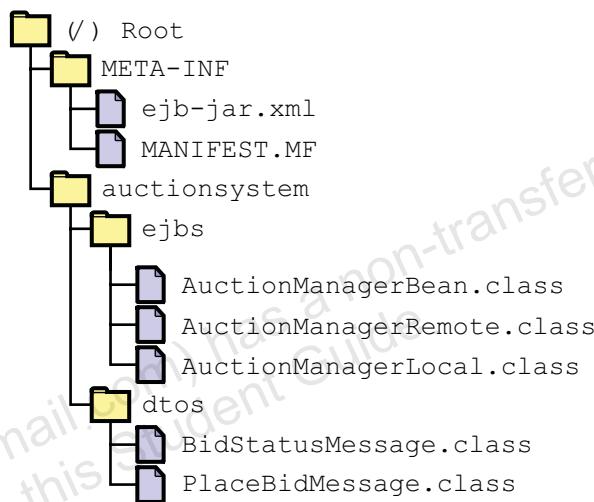


Figure 3-5 Example of the EJB Module JAR File

Deploying a Session Bean Component Archive

Deploying a session bean component archive or for that matter any EJB module is specific to the application. Most application servers provide the following two deployment options.

- IDE-based deployment support
- Command-line based deployment support

For more information on deployment, refer to the documentation associated with the application server you are using.

Creating a Session Bean Client

Before proceeding to the task of creating session bean clients it is important to examine the answers to the following questions.

- What are some examples of session bean clients?
 - Another enterprise bean, such as another session or entity bean.
 - A web component, such as a servlet
 - A plain Java class, such as an application client.
- What factors should be considered when writing a session bean client?

The most important factor is the availability of standard container services, in particular naming lookup and injection services. These services are supplied as standard by the application server container for Java EE components.

In the case of a plain Java class, such as an application client, you can use an application client container to provide these services.

Alternatively you can obtain the services an application client requires from Java SE class methods, such as methods from the `javax.naming.InitialContext`.

- What is an application client container?

An application client container is a set of classes that work together to provide a hosting environment for an application client. The hosting environment includes a JVM machine and a set of services. These services include security, naming, injection, and communication (with the application server container) services. A client using the application container executes in a different JVM to that of the application server.

The packaging of the classes that make up the application-client container is vendor specific. Some vendors provide an application-client container as a JAR file. Application-client containers are installed on the client machine.

The following sections examine creating a session bean client in the following contexts:

- Creating a client using container services
- Creating a client without using container services

Creating a Client Using Container Services

The following steps outline a process for creating a client that uses container services to access the services of a session bean.

1. Use the injection services of the container to obtain a reference to the session bean object.
2. Invoke the required services of the session bean object.

Code 3-7 contains an example of a client that uses container services to access a session bean.

Code 3-7 Container Based Session Bean Client

```
1 import javax.ejb.*;
2 import some.calculators;
3
4 public class InternalClient {
5     @EJB                         // step 1
6     private static Calculator calc;
7
8     public static void main (String args[]) {
9         int sum = calc.add(1, 2);    // step 2
10    }
11 }
```

Creating a Client Without Using Container Services

The following steps outline a process for creating a session bean client that is external to a container.

1. Use the JNDI naming services to obtain a reference to the session bean object.
2. Invoke the required services of the session bean object.

Code 3-8 contains an example of a client that uses JNDI naming services to access a session bean.

Code 3-8 Non-Container-Based Session Bean Client

```
1 import javax.naming.*;
2 import some.calculators;
3
4 public class InternalClient {
5     public static void main (String args[]) {
6         // step 1 begin
7         InitialContext ctx = new InitialContext();
8         Object obj = ctx.lookup("ejb/Calculator");
9         Calculator calc = (Calculator)
10            PortableRemoteObject.narrow(obj, Calculator.class);
11         // step 1 end
12         int sum = calc.add(1, 2);           // step 2
13     }
14 }
```

Reviewing Session Beans

This module describes how you can implement EJB 3.0 session beans. The key issues can be summarized as:

1. Session bean types: stateful and stateless
2. Declare the business interface for the session bean.
3. Create the session bean class that implements the business interface.
4. Annotate the session bean class:
 - a. Session bean type
 - b. Local or remote
 - c. Dependency injection
 - d. Transaction
 - e. Security
5. If required, add handling code for life-cycle callback events.
6. Package the session bean.
7. Deploy the session bean.
8. Create the client.

Module 4

Implementing Entity Classes: The Basics

Objectives

Upon completion of this module, you should be able to:

- Examine Java persistence
- Define entity classes: Essential tasks
- Manage entity instance life-cycle states
- Deploy entity classes

Additional Resources



Additional resources – The following reference provides additional information on the topics described in this module:

- Sun Microsystems, “JSR 220: Enterprise JavaBeans™, Version 3.0 Java Persistence API.” [<https://sdlc3e.sun.com/ECom/EComActionServlet;jsessionid=CEAAE57A3BAB8A76D4555E3C5A1F4031>], accessed July 25, 2006.

Examining Java Persistence

This section uses a frequently asked questions (FAQ) list to examine the features of Java persistence, as defined in the Java Persistence Specification (JSR 220).

- What is data persistence?

Data persistence is the mechanism used by applications to preserve application data (that would be lost by application shutdown or computer power down) in a persistence store, such as a database.

- What is the Java persistence specification?

The Java persistence specification is the specification of the Java API for the management of persistence and object/relational mapping with Java EE and Java SE. The technical objective of the specification work is to provide an object/relational mapping facility for the Java application developer using a Java domain model to manage a relational database.

- How does the Java Persistence API relate to Java EE application servers?

All Java EE application servers are required to provide an implementation of the Java Persistence API.

- What are the key features of the persistence model specified in the Java Persistence API?

The Java persistence model is best examined in two stages. The first stage presents a static view (see Table 4-1) of persistence and the second stage presents the dynamic view of persistence.

- The Java Persistence API establishes the static relationships of the persistence model by defining the entity component. The API defines the entity class as the object tier equivalent of a table in the database tier. An entity instance is defined as the object tier equivalent of a row in a database table.

Table 4-1 Mapping Object Tier Elements to Database Tier Elements

Object Tier Element	Database Tier Element	Comment
Entity class	Database table	For an illustration see Figure 4-1 on page 4-5 and Figure 4-2 on page 4-6.
Field of entity class	Database table column	Applies to a non-transient field or property only.

Examining Java Persistence

Table 4-1 Mapping Object Tier Elements to Database Tier Elements

Object Tier Element	Database Tier Element	Comment
Entity instance	Database table row	For an illustration see Figure 4-3 on page 4-7.

- The Java Persistence API establishes the dynamic relationship of the persistence model by defining an entity manager object. The entity manager object is tasked with synchronizing data contained in an entity instance with the data contained in the equivalent data row in the database.
For example, during the course of application execution whenever a field in a (managed) entity instance is updated, the entity manager updates the equivalent row in the database.
- What tasks need to be performed by an application component developer to use the Java Persistence API?
 - Define entity classes
 - Package and deploy the entity classes
 - Provide application code (for example in session beans, message-driven beans, or helper classes) to create and manage (the life-cycle state of) entity instances as required by the use cases of the application.

Object Tier / Data Tier Static and Dynamic Mapping Example

This section shows an example of the static and dynamic relationships between elements in the object and data tiers.

Figure 4-1 and Figure 4-2 on page 4-6 illustrate the static mapping that exists between the data tier and the object tier. Figure 4-1 shows three tables in the database tier.

Auction Table

Auction ID	Seller	Item	StartAmount	Increment	...

Bid Table

BidID	Auction	Bidder	Amount	BidTime	Authorization

Item Table

ItemID	Description	Image

Figure 4-1 Static Relationship Mapping - Data Tier Elements

Examining Java Persistence

Figure 4-2 shows three entity classes in the object tier that correspond to the tables from the data tier shown in Figure 4-1. You should observe the correspondence between the fields of the entity classes shown in Figure 4-2 and the columns of the tables shown in Figure 4-1 on page 4-5.

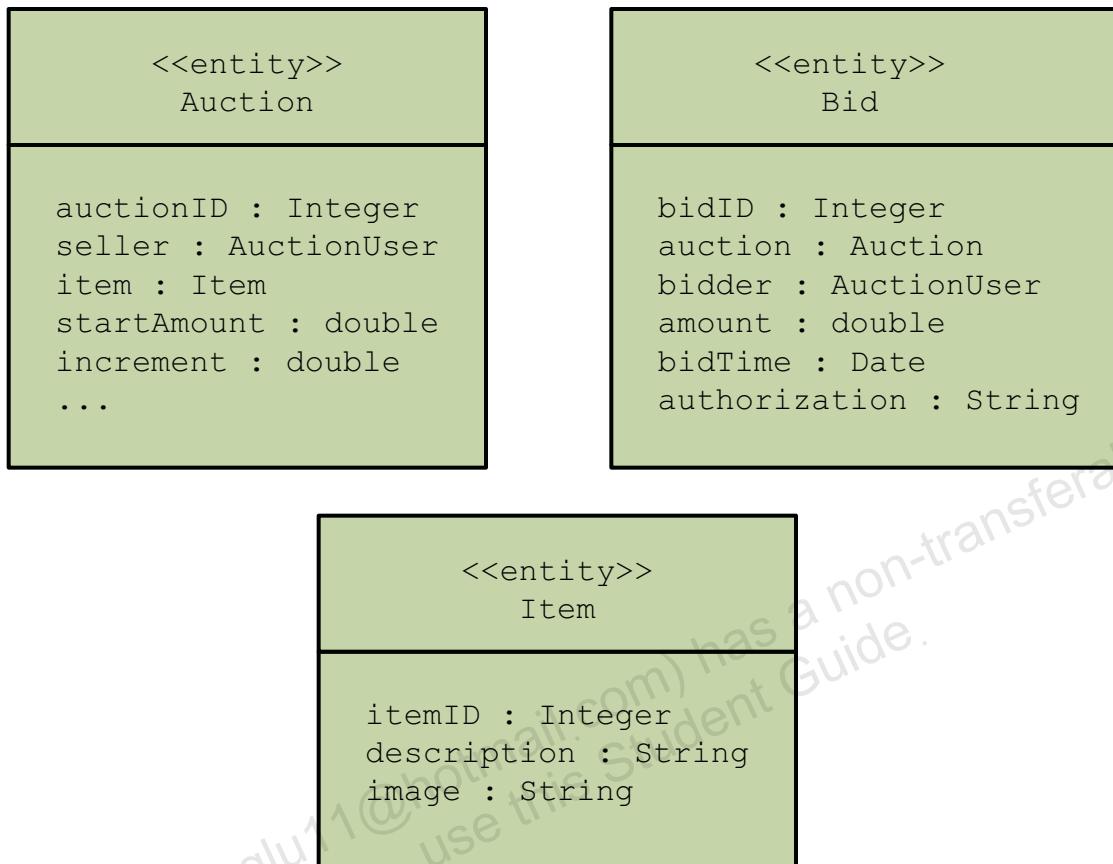
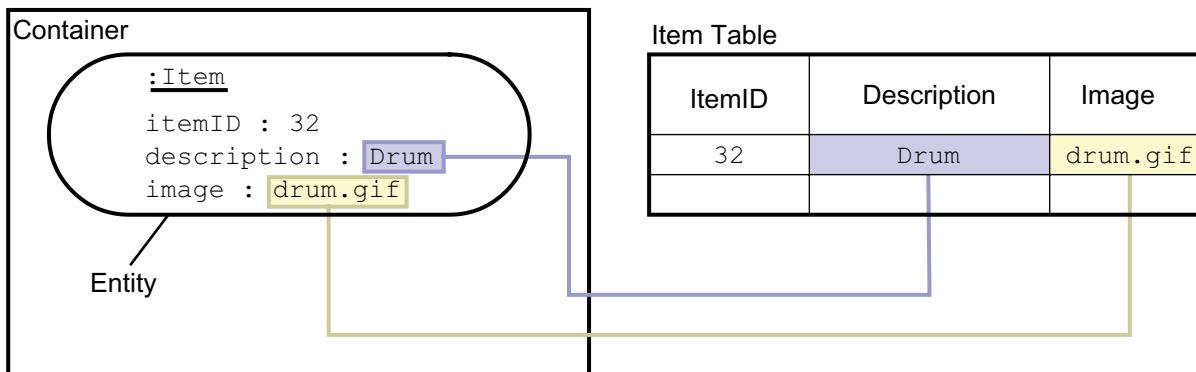


Figure 4-2 Static Relationship Mapping - Object Tier Elements

Figure 4-3 illustrates the dynamic relationship consisting of data synchronization between the object and data tiers. The data synchronization service is provided by the persistence provider through one or more entity manager objects.



With entities, the data synchronization is maintained by the persistence provider.

Figure 4-3 Dynamic Relationship - Object/Data Tier Data Synchronization

Code 4-1 shows a possible implementation of the `Item` entity class. This code is provided for the purpose of showing what an entity class looks like. The section “Defining Entity Classes: Essential Tasks” on page 4-9 covers the topic of creating entity classes in great detail. It also includes an explanation for the annotations (`Entity`, `Id`, and `GeneratedValue`) used in this code example.

Code 4-1 Example Source Code for an Entity Class

```

1  @Entity
2  public class Item {
3
4      @Id
5      @GeneratedValue
6      private Integer itemID;
7      private String description;
8      private String image;
9
10
11     /** Creates a new instance of Item */
12     public Item() {
13     }
14
15     public Item(String description, String image){
16         setDescription(description);

```

Examining Java Persistence

```
17     setImage(image);
18 }
19
20 public Integer getItemID() {
21     return itemID;
22 }
23
24 public String getDescription() {
25     return description;
26 }
27
28 public void setDescription(String description) {
29     this.description = description;
30 }
31
32 public String getImage(){
33     return image;
34 }
35
36 public void setImage(String image) {
37     this.image = image;
38 }
39 }
```

Defining Entity Classes: Essential Tasks

This section provides an overview of the tasks required to define a Java Persistence entity class. To define an entity class, you are required to perform the following tasks:

- Declare the entity class.

When declaring the entity class, you can use fields (attributes of the entity class) or properties (get and set methods of the entity class) to model the persistence state of the entity class.

- Verify and override the default mapping.

The following subsections describe these tasks in more detail.

Declare the Entity Class

Use the information collected in the persistence data definition activity to declare the entity class. The following steps outline a process you can use to declare the entity class.

1. Collect information required to declare the entity class

This step involves identifying the application domain object (identified in the object-oriented analysis and design phase of application development) that you want to persist.

- Use the domain object name as the class name.
- Use the domain object field names and data types as the field names and types of entity class.

2. Declare a public Java technology class.

The class must not be final, and no methods of the entity class can be final. The class can be concrete or abstract.

3. If an entity instance is to be passed by value as a detached object through a remote interface, then ensure the entity class implements the `Serializable` interface.

4. Ensure the class does not define the `finalize` method.

5. Annotate the class with the `Entity` annotation.

6. Declare the attributes of the entity class.

Attributes should be declared private. Attributes must not have public visibility. They can have private, protected, or package visibility.

Defining Entity Classes: Essential Tasks

7. You can optionally declare a set of public getter and setter methods for every attribute declared in the previous step.
8. Annotate the primary key field or the getter method that corresponds to the primary key column with the `Id` annotation.
9. Declare a public or protected no-arg constructor that takes no parameters.

The container uses this constructor to create instances of the entity class. The class can have additional constructors.

Code 4-2 shows an example of an entity class.

Code 4-2 Entity Class Code Example

```
1  @Entity
2  public class Customer implements Serializable {
3      private Long custId;
4      private String name;
5
6      // No-arg constructor
7      public Customer() {}
8
9      @Id
10     public Long getCustId() {
11         return custId;
12     }
13
14     public void setCustId(Long id) {
15         custId = id;
16     }
17
18     public String getName() {
19         return name;
20     }
21
22     public void setName(String name) {
23         this.name = name;
24     }
25 }
```

Table 4-2 shows the table that corresponds to the entity class in Code 4-2.

Table 4-2 CUSTOMER Table

CUSTID	NAME

Verifying and Overriding the Default Mapping



Note – This discussion examines the optional task of overriding the default settings. For the vast majority of situations, using the defaults should be sufficient. The content of this section is **optional** and could be considered advanced. It requires a thorough understanding of the previous section.

In this step, you examine the default settings and if required, add additional annotations to the entity class to override the defaults.

- Overriding the default setting of the database table name (Table annotation)

The Java Persistence API assigns the name of the entity class as the default name to the database table. Use the `Table` annotation to override the default setting. For example:

```
@Entity  
@Table(name="Cust") //Cust is the name of the database table  
public class Client {  
    //...  
}
```

- Overriding the default setting of the database table column name (Column annotation)

The Java Persistence assigns the name of the entity class property (or field) name as the default name to the database table column. Use the `Column` annotation to override the default setting. For example.

```
@Entity  
@Table(name="Cust")  
public class Client {  
    @Column(name="cname")  
    private String clientName;  
    //...  
}
```

- Automatically generating the primary key (the default behavior of the `GeneratedValue` annotation)

You have several options available for generating the values of primary keys. The simplest of these is to use the auto generation feature of the container, as shown in the following example.

```
@Entity
@Table(name="Cust")
public class Client {
    @Id
    @GeneratedValue
    private int clientReference;
    //...
}
```

- Excluding a field or property from being persisted by the container (`Transient` annotation)

You can use the `Transient` annotation to inform the container to not persist a field or property of the entity class.

```
@Entity
@Table(name="Cust")
public class Client {
    @Transient
    private int tempValue1;
    //...
}
```

- Mapping the data types of non-relationship fields or properties to column data types (`Basic` and `LOB` annotations)

If the type of the field or property is one of the following, the container uses the default `Basic` annotation: Java primitive types, wrappers of the primitive types, `String`, `BigInteger`, `BigDecimal`, `java.util.Date`, `Calendar`, `java.sql.Date`, `java.sql.Time`, `java.sql.Timestamp`, `byte[]`, `Byte[]`, `char[]`, `Character[]`, enums, any other type that implements the `Serializable` interface.

Use the `Lob` annotation to persist a field or property as a large object to a database-supported large object type. A `Lob` can be either a binary or character type, as defined by the `type` attribute of the `Lob` annotation. For example,

```
@Lob
@Column(name="PHOTO" columnDefinition="BLOB NOT NULL")
protected JPEGImage picture;
```

```
@Lob @Basic(fetch=EAGER) @Column(name="REPORT")
protected String report;
```

Examining Managing Entity Instance Life-Cycle States

This section examines the following topics and their relationship to entity life-cycle state management.

- Entity life-cycle states

There are four (new; managed; detached; removed) life-cycle states of an entity instance. More details are provided in Table 4-3.

- Entity manager

An entity manager is the service object that manages entity life-cycle instances.

- Persistence context

Every entity manager is associated with a persistence context. A persistence context is a set of entity instances in which, for any persistent entity identity, there is a unique entity instance.

- Persistent identity

The persistent identity is a unique value used by the persistence provider to map the entity instance to the corresponding table row in the database. Persistent identity should not be confused with Java object identity.

Table 4-3 provides a summary of the significance of each entity state shown in Figure 4-4.

Table 4-3 Entity Life-Cycle States

Entity State	Significance
New	A new entity instance is a new instance of the entity created using the new keyword. There is no corresponding database table row in the persistence tier associated with the entity's primary key.
Managed	There is a corresponding database row and data in the row is kept synchronized (by the persistence provider) with data in the entity
Detached	There is a corresponding database row but data in the row and data in the entity are not kept synchronized.
Removed	This state represents a pending removal of the corresponding data row in the database.

Figure 4-4 shows the possible states of an entity instance. The persist, merge and remove methods are part of the entity manager API.

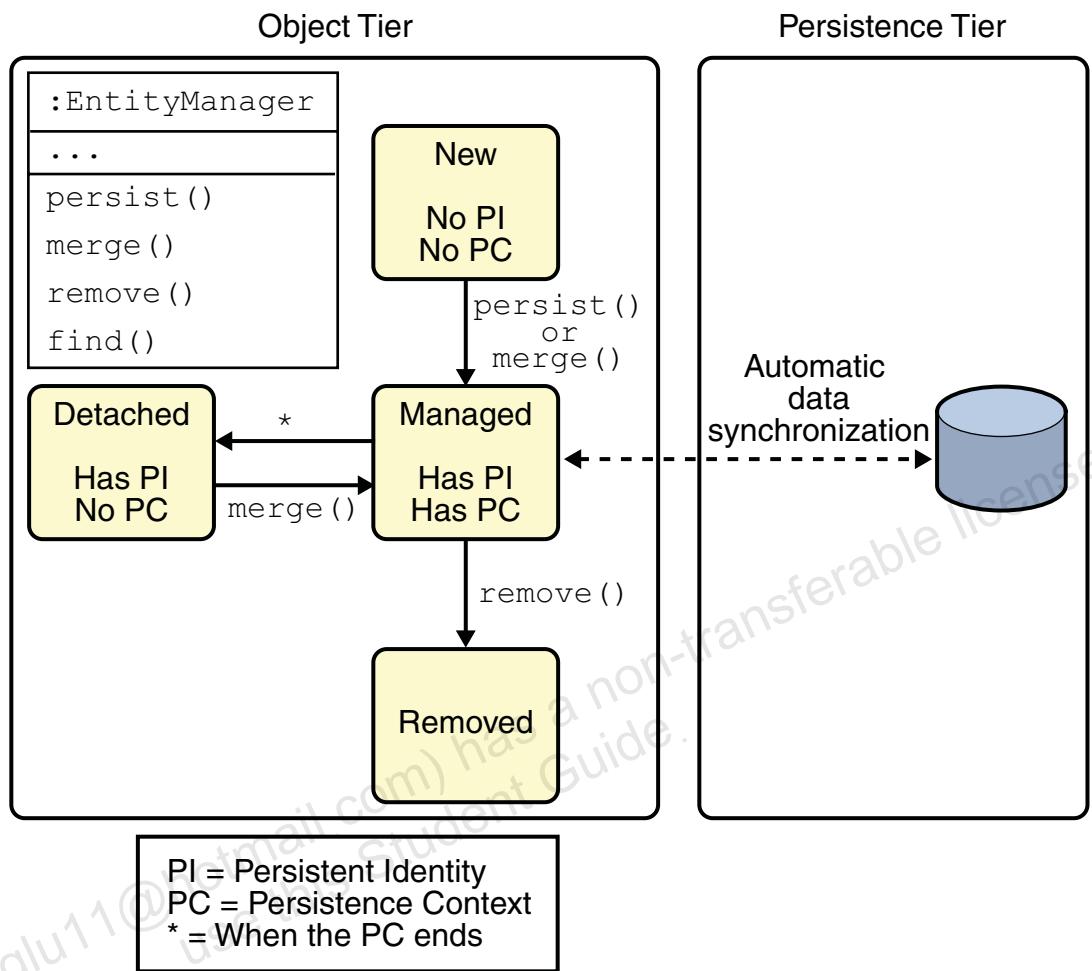


Figure 4-4 Entity Instance State Diagram



Note – Under some circumstances, persistence identity might not be available (for entity instances that changed state from new to managed) until the transaction commits. This applies,⁷ for example to entities that use generated primary keys.

Examining Managing Entity Instance Life-Cycle States

Table 4-4 shows the entity manager methods or events required to effect the change of an entity life-cycle state.

Table 4-4 Changing Entity Life-Cycle States

Old Entity State	Required Operation	New Entity State
Does not exist	Use new operator to create a new instance	New
New	Use entity manager's persist operation	Managed
Managed	Use entity manager's remove operation	Removed
Managed	Happens as the consequence of the persistence context ending. Instances generated by serialization or cloning (of managed instances) are always detached.	Detached
Detached	Use entity manager's merge operation	Managed
Removed	Use entity manager's persist operation	Managed
Removed	Lose reference to removed instance	Non existent

Table 4-5 shows several other useful entity manager methods.

Table 4-5 Other Entity Manager Methods

Entity Manager Method	Comment
flush	Forces the synchronization of the database with entities in the persistence context.
refresh	Refreshes the entity instances in the persistence context from the database.
find	Finds an entity instance by executing a query by primary key on the database.
contains	Returns true if the entity instance is in the persistence context. This signifies that the entity instance is managed.

The following questions provide additional information regarding persistence contexts and entity managers.

- What causes a persistence context to end?

A persistence context can have the following (lifetime) scopes.

- Transaction scope
A transaction scoped persistence context exists for the duration of the transaction.
- Extended scope
An extended scoped persistence exists for the duration of multiple transactions.
- How is a persistence context created?
You do not create a persistence context directly, instead you use the persistence context that the container allocates to the entity manager when it is created.
To use a transaction scoped persistence context, you obtain a transaction scoped entity manager.
To use an extended scoped persistence context, you obtain or create an extended scoped entity manager
- How is an extended scoped entity manager created in a Java EE container?
Use injection, as shown in the following code lines.

```
@PersistenceContext (type=EXTENDED)  
private EntityManager entityManager;
```
- How is a transaction scoped entity manager created in a Java EE container?
Use injection, as shown in the following code lines.

```
@PersistenceContext  
private EntityManager entityManager;
```

Note – Both extended scoped and transaction scoped entity managers can also be obtained in Java EE containers by using JNDI lookup.



- What tasks are relevant to managing the life-cycle state of an entity instance?
 - Decide between using a transaction scoped entity manager or extended scoped entity manager.
 - Obtain or create an entity manager.
 - Write code that uses the entity manager methods to manage the entity life-cycle state to meet the requirements of the application use cases.

Using Entities to Interact With the Database

Figure 4-5 illustrates the database tasks you can perform using entity instances.

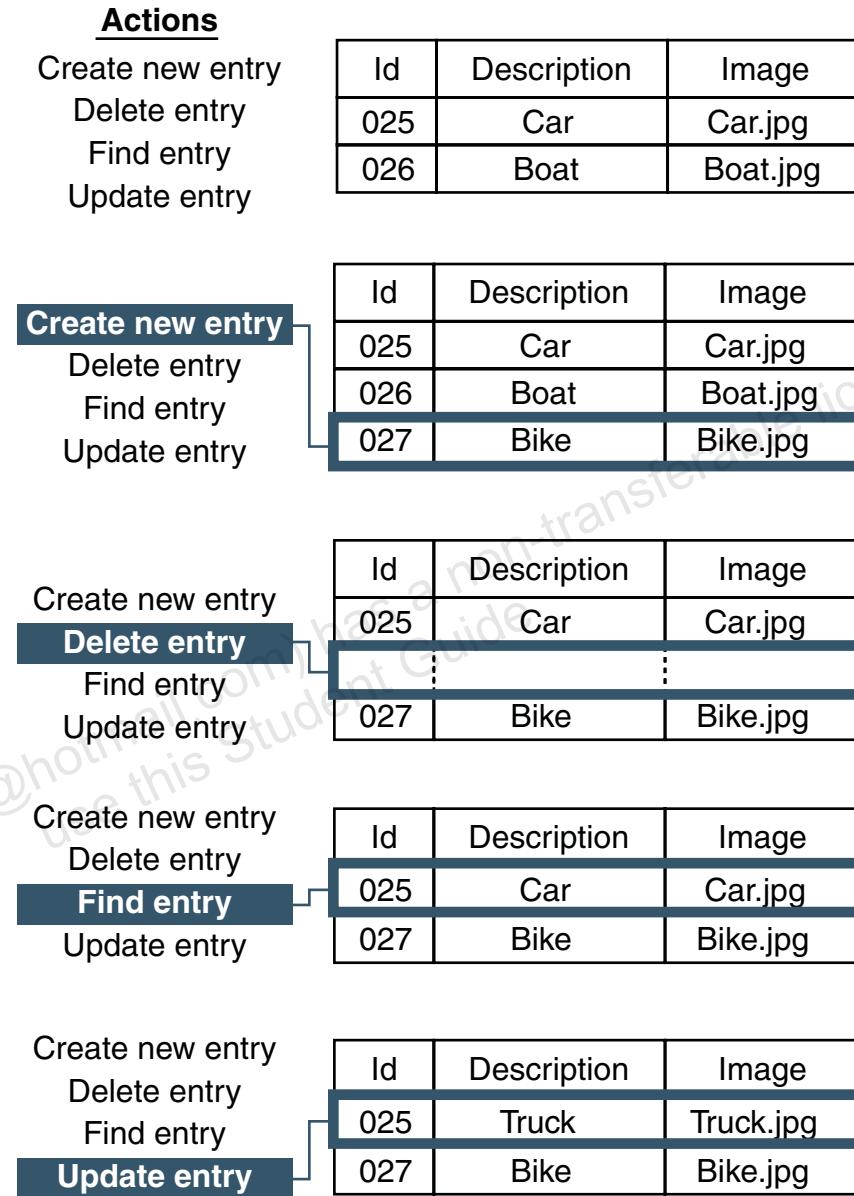


Figure 4-5 Database Tasks

This section shows how you can use entity instances and the EntityManager API to perform the tasks illustrated in Figure 4-5.

- Create a new entry in the database

The `createItemEntry` method in Code 4-3 on page 4-21 is an example of the code you require to perform the task of creating a new entry in the database. The steps involved are:

- a. Create a new instance of an entity class that corresponds to the database table.
- b. Use the `persist` method of the entity manager object to create an entry in the database table.



Note – The code listing for the `Item` entity class is listed in Code 4-1 on page 4-7. The `Item` entity class uses the persistence providers automatic primary key generation facility to generate primary keys.

The entity manager object shown in Code 4-1 on page 4-7 is a transaction scoped entity manager. You can assume all methods of the `AppManagerBean` class run in a transaction.

- Locate and fetch an entry from the database

The `findItem` method in Code 4-3 on page 4-21 is an example of the code you require to perform the task of finding and returning (as an entity instance) an existing entry in the database. The following step is all that is required.

- a. Use the `find` method of the entity manager object to locate an entry in the database table. This method returns a managed object.
- Update an entry in the database

The overloaded `updateItemImage` method in Code 4-3 on page 4-21 are two examples of the code you require to perform the task of updating an entry in the database. The principles involved are:

- a. You must use the property method (`setImage`) to change the value of the image attribute of the entity instance.
- b. You must either use the property method on a managed instance as shown in the `updateItemImage(Item item, String newImage)` method, or you must merge a detached instance after updating it as shown in the `updateItemImage(String newImage, Item item)` method.

Using Entities to Interact With the Database

- Delete an existing entry in the database

The overloaded `deleteItem` method in Code 4-3 on page 4-21 are two examples of the code you require to perform the task of deleting an entry from the database. The principles involved are:

- a. Use the `remove` method of the entity manager object.
- b. The entity instance passed to the `remove` method must be a managed object.

Code 4-3 Entity Instance Management Using Transaction Scoped Persistence Context

```
1  @Stateful
2  public AppManagerBean implements AppManager {
3      @PersistenceContext
4      private EntityManager entityManager;
5
6      public Item createItemEntry(String description, String image) {
7          Item item = new Item(description, image);
8          entityManager.persist(item);
9          return item;
10     }
11
12    public Item findItem(Integer key) {
13        Item item = entityManager.find(Item.class, key);
14        return item;
15    }
16
17    public Item updateItemImage(Item item) {
18        // assume item has been updated prior to method invocation
19        Item item1 = entityManager.merge(item);
20        return item1;
21    }
22
23    public Item updateItemImage(Integer key, String newImage) {
24        Item item = entityManager.find(Item.class, key);
25        item.setImage(newImage);
26        return item;
27    }
28
29    public Item updateItemImage(Item item, String newImage) {
30        // item is detached intance; item1 is managed instance
31        Item item1 = entityManager.merge(item);
32        item1.setImage(newImage);
33        return item1;
34    }
35
36    public Item updateItemImage(String newImage, Item item) {
37        item.setImage(newImage); // item is detatched instance
38        Item item1 = entityManager.merge(item); // item1 is managed
39        return item1;
40    }
41
42    public Item deleteItem(Integer key) {
43        Item item = findItem(key);
```

Using Entities to Interact With the Database

```

44     entityManager.remove(item);
45     return item;
46 }
47
48 public Item deleteItem(Item item) {
49     item = entityManager.find(Item.class, item.getId());
50     entityManager.remove(item); // managed instance
51     return item;
52 }
53 }
```

Using an Extended Persistence Context Entity Manager

Code 4-4 on page 4-23 shows a version of AppManagerBean that uses an extended persistence context entity manager. The main difference between using a transaction scoped entity manager and an extended scoped entity manager are listed as follows:

- With a transaction scoped entity manager, all managed entity instances (in the entity manager's persistence scope) become detached when the transaction completes (commits or rolls back).
- With an extended scoped entity manager, all managed entity instances (in the entity manager's persistence scope) stay managed for the duration of the lifetime of the entity manager.

The entity manager in Code 4-4 on page 4-23 is obtained using the `PersistenceContext(type=EXTENDED)` injection. The persistence context type associated with entity manager is extended scope. It exists for the duration of the `AppManagerBean` instance.

For this example, you should assume that each method executes in its own transaction.

Note – The struck-through lines of code contained in Code 4-4 on page 4-23, indicate that these lines of code that were required when using a transaction scoped entity manager are not required when using an extended scoped transaction manager.



Code 4-4 Entity Instance Management Using Extended Scoped Persistence Context

```
1  @Stateful
2  public AppManagerBean implements AppManager {
3      @PersistenceContext(type=Extended)
4      private EntityManager entityManager;
5
6      public Item createItemEntry(String description, String image) {
7          Item item = new Item(description, image);
8          entityManager.persist(item);
9          return item;
10     }
11
12     public Item findItem(Integer key) {
13         Item item = entityManager.find(Item.class, key);
14         return item;
15     }
16
17     // The following method is no longer needed
18     public Item updateItemImage(Item item) {
19         // assume item has been updated prior to method invocation
20         Item item1 = entityManager.merge(item);
21         return item1;
22     }
23
24     public Item updateItemImage(Integer key, String newImage) {
25         Item item = entityManager.find(Item.class, key);
26         item.setImage(newImage);
27         return item;
28     }
29
30     public Item updateItemImage(Item item, String newImage) {
31         // item is managed instance
32     Item item1 = entityManager.merge(item);
33         item.setImage(newImage);
34         return item;
35     }
36
37     // The following method is identical to the previous method
38     public Item updateItemImage(String newImage, Item item) {
39         item.setImage(newImage); // item is managed instance
40     Item item1 = entityManager.merge(item); // item1 is managed
41         return item;
42     }
43 
```

Using Entities to Interact With the Database

```
44     public Item deleteItem(Integer key) {  
45         Item item = findItem(key);  
46         entityManager.remove(item);  
47         return item;  
48     }  
49  
50     public Item deleteItem(Item item) {  
51         item = entityManager.merge(item);  
52         entityManager.remove(item); // managed instance  
53         return item;  
54     }  
55 }
```

Note – Due to the nature of session bean transaction scoping, only a stateful session bean can use extended scoped persistence context.



Deploying Entity Classes

This section examines the issues and tasks associated with deploying entity classes. It uses a FAQ list to present the information you require.

- What is required to deploy entity classes?
To deploy entity classes, you are required to create a persistence unit.
- What is a persistence unit?

A persistence unit is a logical grouping of all the elements required by the container to support the persistence management of an application.

Table 4-6 shows the list of required components of a persistence unit.

Table 4-6 Components of the Persistent Unit

Persistence Unit Component	Comment
All managed (entity) classes.	These classes define what is persisted.
Object relational mapping metadata	This mapping defines the mapping between the managed classes and database tables.
Entity manager factory and entity managers	This information specifies which entity manager factory and which entity managers are to be used.
Configuration information for the entity manager factory and entity managers	The configuration information defines how the entity manager factory and entity managers are to be configured.
A <code>persistence.xml</code> file	This file defines the persistence unit.

- Is a persistence unit a standalone deployable module?

The persistent unit is not a standalone deployable unit. Instead, the components that make up the persistent unit are placed in selected deployable Java EE modules.

Deploying Entity Classes

- Where should the components of the persistence unit be placed?
For deployment to a Java EE application server, the components of the persistence unit must be placed in one of the following locations:
 - In a EAR file
 - In a EJB-JAR file
 - In a WAR file
 - In a application client JAR file
- Within an EJB-JAR file, at which locations should the components of the persistence unit be placed?
To minimize your work, you should use the default locations for each of the required components of the persistence unit.

Table 4-7 Default Locations for Persistence Unit Components

Persistence Unit Component	Default Location
All Managed (entity) classes.	In directories that map the package structure of the managed classes. The directory path root is the root of the EJB-JAR file.
Object relational mapping metadata	These are included as annotations in the entity classes. An alternative default location is the XML deployment descriptor file.
Entity manager factory and entity managers	The default entity manager factory and default entity managers are placed in a persistence provider specific location. These are known to the Java EE application server.
Configuration information for the entity manager factory and entity managers	The default configuration information is known to the Java EE application server.
The <code>persistence.xml</code> file	In the <code>META-INF</code> directory off the root of the EJB-JAR file.

Creating a Persistence Unit Using Default Settings

Figure 4-6 shows an example of persistence unit components in default settings within an EJB-JAR. It also shows other components of an EJB-JAR archive.

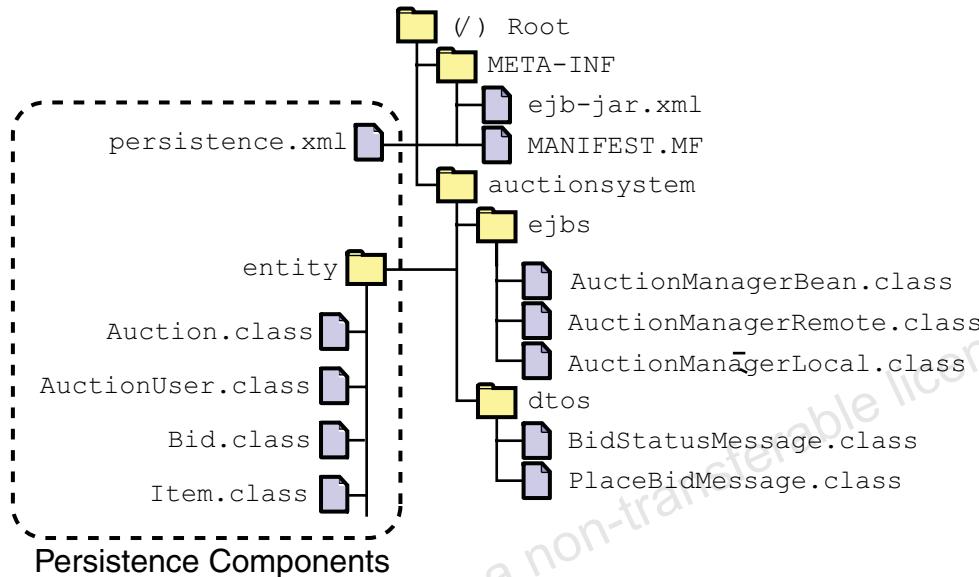


Figure 4-6 Example of Persistence Unit Components in Default Settings

The following steps outline a process you can follow to create a persistence unit within in an EJB-JAR file using the default settings for various components of the persistence unit. This process assumes that the entity classes are fully annotated and all classes will be placed in the root directory.

1. If not already expanded, expand the EJB-JAR archive.
2. Create subdirectories that reflect the full qualified names of the managed classes you need to include in the persistence unit.
3. Copy all the classes you need into the directories created in Step 2.
4. Create a minimal `persistence.xml` file.

At minimum, you must provide an XML file with the following elements.

- The `persistence` element
- The `persistence-unit` element with a value for the `name` attribute.

Deploying Entity Classes

Code 4-5 shows the listing of the minimum content of the persistence.xml file.

Code 4-5 Example of the Minimal Persistence XML file

```
1 <persistence>
2   <persistence-unit name="OrderManagement" />
3 <persistence>
```

5. Copy the persistence.xml file into the META-INF directory off the EJB-JAR root directory.
6. Create (archive) the EJB-JAR module.

Examining a Persistence Unit Using Non Default Settings



Note – This discussion examines the **optional** task of using non-default settings to create a persistence unit. For the vast majority of situations, using the defaults should be sufficient. The content of this section is fairly advanced and requires a thorough understanding of the previous section.

Table 4-8 contains the alternative (non-default) locations for the persistence archive information.

Table 4-8 Alternative (Non Default) Locations for Persistence Unit Components

Persistence Unit Component	Non Default Location	Required Locating Information
Managed (entity) classes and object relational mapping metadata	Explicitly specified JAR files Explicitly specified classes The orm.xml file One or more explicitly specified XML mapping files	The jar-file element of the persistence.xml file. The class element of the persistence.xml file. In META-INF directory The mapping-file element of the persistence.xml file.
Entity manager factory and entity managers	Anywhere in the classpath of the class loader.	The provider element of the persistence.xml file.
Configuration information for the entity manager factory and entity managers	The persistence.xml file	The properties element of the persistence.xml file.
The persistence.xml file	A non-default location does not apply. This file must be located in the META-INF directory off the root of the EJB-JAR file.	

Table 4-8 introduces three new files (explicitly specified jar files, orm.xml file and mapping files). These files are defined in the following sections.

Deploying Entity Classes

- The JAR files off root

The JAR files off the root are optional. They are an alternative method for you to provide the managed set of entity classes. Every JAR file provided requires a corresponding `jar-file` element in the `persistence.xml` file. The presence of the `jar-file` elements in the `persistence.xml` file prompts the application server to load the classes archived in the JAR file.

- The mapping XML files off root

The mapping XML files off the root are an alternative location for object/relational mapping information. These mapping files require a corresponding `mapping-file` element in the `persistence.xml` file.

The mapping XML files are optional. All mapping information can be supplied through the appropriate annotations in the entity class.

- The `orm.xml` file

The `orm.xml` file is an alternate location for object/relational mapping. The `orm.xml` file is optional. All mapping information can be supplied through the appropriate annotations in the entity class or through the optional mapping XML files.

Code 4-6 shows an example a persistence.xml file that defines persistence unit components in non default locations.

Code 4-6 Example of a Persistence XML file Denoting the Use of Non-Default Settings

```
1 <persistence>
2   <persistence-unit name="OrderManagement5" transaction-type=JTA>
3     <provider>com.acme.persistence</provider>
4     <jta-data-source>jdbc/MyPartDB</jta-data-source>
5     <mapping-file>product.xml</mapping-file>
6     <mapping-file>order.xml</mapping-file>
7     <exclude-unlisted-classes/>
8     <jar-file>product.jar</jar-file>
9     <jar-file>product-supplemental.jar</jar-file>
10    <class>com.acme.Order</class>
11    <class>com.acme.Customer</class>
12    <class>com.acme.Item</class>
13    <properties>
14      <property name="com.acme.persistence.sql-logging" value="on" />
15    </properties>
16  </persistence-unit>
```

Deploying Entity Classes

Table 4-9 interprets the persistence archive information specified in the `persistence.xml` file listed in Code 4-6 on page 4-31.

Table 4-9 Locations for Persistence Unit Components That Correspond to Code Example Code 4-6

Persistence Unit Component	Non Default Location	Required Locating Information
Managed (entity) classes and object relational mapping metadata	In directories that map the package structure of the managed classes. The <code>product.jar</code> file The <code>product-supplemental.jar</code> <code>com.acme.Order.class</code> <code>com.acme.Customer.class</code> <code>com.acme.Item</code> The <code>orm.xml</code> file The <code>product.xml</code> file The <code>order.xml</code> file	Implied The <code>jar-file</code> element The <code>jar-file</code> element The <code>class</code> element The <code>class</code> element The <code>class</code> element If one exists in the <code>META-INF</code> directory The <code>mapping</code> element The <code>mapping</code> element
Entity manager factory and entity managers	Anywhere in the classpath of the class loader.	The <code>provider</code> element
Configuration information for the entity manager factory and entity managers	<code>com.acme.persistence.sql-logging" = "on"</code>	The <code>properties</code> element
The <code>persistence.xml</code> file	This non-default file is located in the <code>META-INF</code> directory off the root of the EJB-JAR file.	

Module 5

Implementing Entity Classes: Modelling Data Association Relationships

Objectives

Upon completion of this module, you should be able to:

- Examine association relationships in the data and object models
- Use relationship properties to define associations
- Implement one-to-one unidirectional associations
- Implement one-to-one bidirectional associations
- Implement many-to-one/one-to-many bidirectional associations
- Implement many-to-many bidirectional associations
- Implement many-to-many unidirectional associations
- Examine fetch and cascade mode settings

Additional Resources



Additional resources – The following reference provides additional information on the topics described in this module:

- Sun Microsystems, “JSR 220: Enterprise JavaBeans™, Version 3.0 Java Persistence API.” [<https://sdlc3e.sun.com/ECom/EComActionServlet;jsessionid=CEAAE57A3BAB8A76D4555E3C5A1F4031>], accessed July 25, 2006.

Examining Association Relationships in Data and Object Models

Most EISs use relational databases to persist data. Relational databases store data in tables. Tables consist of rows and columns. The columns represent attributes and the rows store attribute values that are semantically related. You use a primary key to uniquely identify a row of data.

In many cases, a row of data in one table can be semantically related to one or more rows in one or more tables. In relational databases, you use foreign keys to maintain such relationships.

Figure 5-1 shows some of the relationships within the data model that represents the auction application in the EIS tier.

Table 5-2 Bid

BidID	Auction	Bidder	Amount	BidTime	Authorization

Table 5-1 Auction

Auction ID	Seller	Item	StartAmount	Increment	...

Table 5-3 Item

ItemID	Description	Image

Figure 5-1 Partial View of Relationships Within the Auction System's Data Model

Examining Association Relationships in Data and Object Models

Figure 5-2 shows the part of the object model that is used to represent the auction application in the middle tier. It shows the object tier equivalent auction-bid relationship and the auction-item relationship shown in Figure 5-1 on page 5-3.

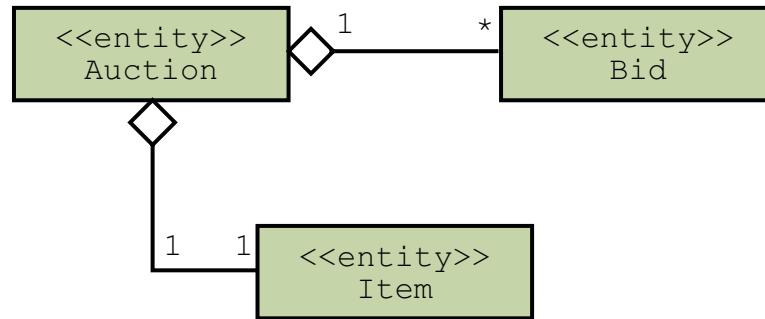


Figure 5-2 Object Model Showing Object Association Relationships

The first step in modelling the association relationships in the data tier is to define the relationship in terms of a set of relationship properties. These relationship properties can then be used to implement the same relationships in the object model using entity classes.

Using Relationship Properties to Define Associations

The following four properties are required to describe association relationships between objects in the object model (to reflect the association between data held in two related database tables):

- Cardinality

Cardinality specifies the quantity relationship between two entities in a relationship. In the Java Persistence API, cardinality is expressed using one of the following annotations:

- OneToOne
- ManyToOne
- OneToMany
- ManyToMany

- Direction

Direction implies navigation or visibility. Direction is expressed using one of the following terms:

- Bidirectional

In a bidirectional relationship, each entity can see the other entity in the relationship. You can include code in either entity that navigates to the other entity to obtain information or services from the other entity.

In the Java Persistence API, the bidirectional nature of the relationship is specified by the use of one of the cardinality annotations in both the entity classes on either side of the relationship.

- Unidirectional

In a unidirectional relationship, only one of the entities can see the other entity in the relationship.

In the Java Persistence API, the unidirectional nature of the relationship is specified by the use of one of the cardinality annotations in the entity class that owns the relationship.

Using Relationship Properties to Define Associations

- Ownership

Ownership specifies the owning side of the relationship. In a unidirectional relationship, ownership is implied. Following is a list of terminology used when discussing ownership:

- Owning side

In the Java Persistence API, the owning side of the relationship specifies the physical mapping.

- Inverse side

The non-owning side or the inverse side of the relationship.

- Cascade type (operation propagation)

The cascade property specifies the propagation of the effect of an operation to associated entities. The cascade functionality is most typically used in parent-child (composition) relationships. The cascade property is expressed using one of the following terms:

- All
- Persist
- Merge
- Remove
- Refresh
- None

Examples of Association Relationships

Table 5-1 shows the seven possible cardinality-direction combinations of CMR relationships.

Table 5-1 Cardinality-Direction Combinations

	Cardinality	Direction
1	One-to-one	Bidirectional
2	One-to-one	Unidirectional
3	Many-to-One / One-to-many	Bidirectional
4	Many-to-One	Unidirectional
5	One-to-Many	Unidirectional
6	Many-to-many	Bidirectional
7	Many-to-many	Unidirectional

Example of One-to-One Bidirectional Relationships

Figure 5-3 shows an example of a one-to-one bidirectional relationship. In this example (taken from outside the auction application), a Car entity instance is restricted to a one-to-one relationship with an Engine entity instance. The bidirectional nature of the relationship enables either entity instance to navigate to the other entity.

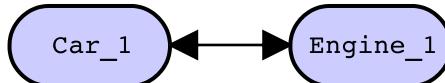


Figure 5-3 One-to-One Bidirectional Relationship Example

Example of One-to-One Unidirectional Relationship

Figure 5-4 shows an example of a one-to-one unidirectional relationship. In this example, an Auction entity instance is restricted to a one-to-one relationship with an Item entity instance. The unidirectional nature of the relationship enables the Auction entity instance to navigate to the Item entity, but the Item entity cannot navigate to the Auction entity.

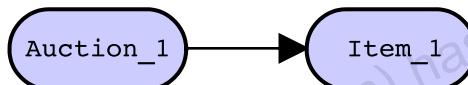


Figure 5-4 One-to-One Unidirectional Relationship Example

Examples of Many-to-One / One-to-Many Bidirectional Relationship

Figure 5-5 shows an example of a many-to-one/one-to-many bidirectional relationship taken from the auction application. In this example, an Auction entity instance is related to many Bid entity instances. From the Auction entity instance, you can locate all its associated Bid classes. From any Bid entity instance, you can locate the Auction entity instance to which it relates.

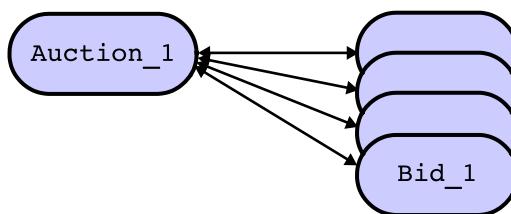


Figure 5-5 Many-to-One / One-to-Many Bidirectional Relationship Example

Using Relationship Properties to Define Associations

Example of Many-to-One Unidirectional Relationship

Figure 5-6 shows an example of a many-to-one unidirectional relationship taken from a banking application. In this example, an AccountType entity instance is related to many Account entity instances. The unidirectional nature of the relationship restricts navigation from an Account entity instance to its related AccountType entity instance.

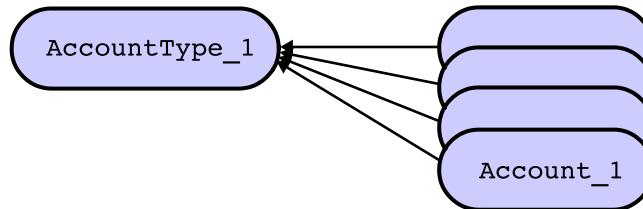


Figure 5-6 Many-to-One Unidirectional Relationship Example

Example of One-to-Many Unidirectional Relationship

Figure 5-7 shows an example of a one-to-many unidirectional relationship. In this example, an Order entity instance is related to many OrderItem entity instances. From the Order entity instance, you can locate all of its associated OrderItem classes. However, the designers of the relationship have chosen to hide the Order entity instance from the OrderItem entity instances.

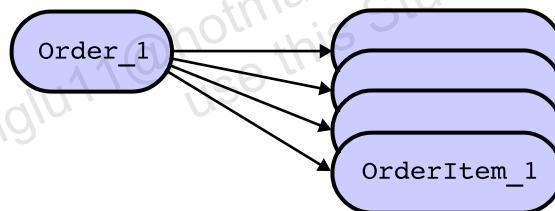


Figure 5-7 One-to-Many Unidirectional Relationship Example

Example of Many-to-Many Bidirectional Relationship

Figure 5-8 shows an example of a many-to-many bidirectional relationship. This example models a relationship that shows that a company has many employees and an employee can work for many companies. Navigation is possible in either direction. That is, given a company, you can navigate to all its employees, and given an employee you can navigate to all the companies for which the employee works.

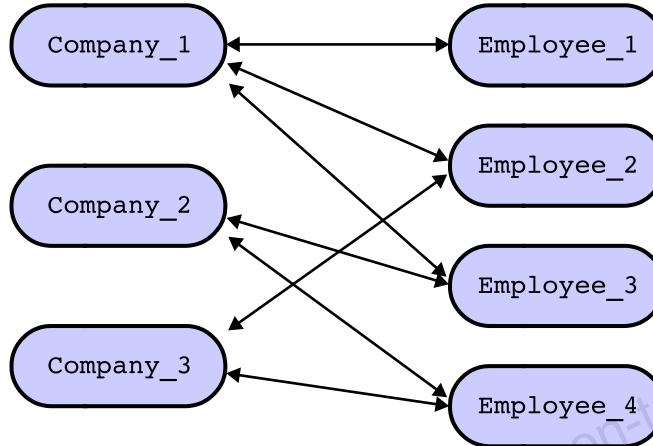


Figure 5-8 Many-to-Many Bidirectional Relationship Example

Example of Many-to-Many Unidirectional Relationship

Figure 5-9 shows an example of a many-to-many unidirectional relationship. This example models a unidirectional relationship between Order instances and InventoryItems instances.

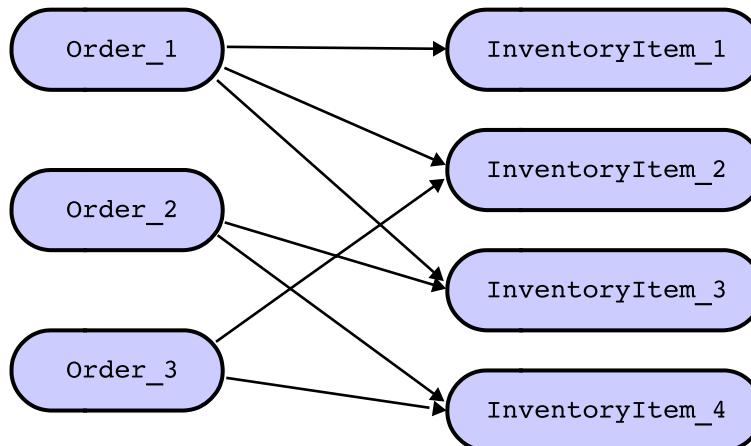


Figure 5-9 Many-to-Many Unidirectional Relationship Example

Implementing One-to-One Unidirectional Association

Figure 5-10 shows a one to one unidirectional association between two entities.

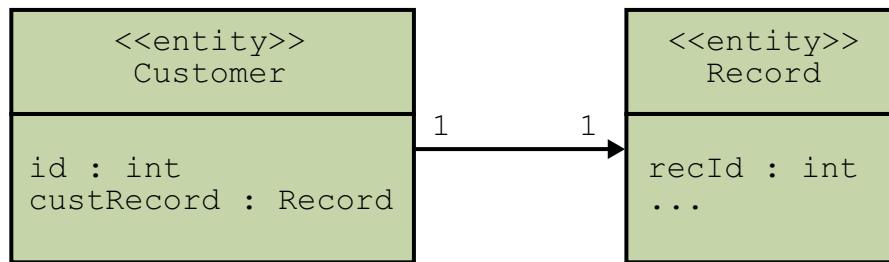


Figure 5-10 One-to-One Unidirectional Association

The following steps outline a process you can follow to define a one-to-one unidirectional association between two entity classes.

1. Define the two entity classes.
2. Identify the entity class that is the owner of the relationship.
3. In the owning entity, create a (relationship) property or field to represent the relationship this entity has with the other (target) entity.
4. In the owning entity, annotate the relationship field or property with the `@OneToOne` annotation.
5. Because this is a unidirectional one-to-one association, no code changes are required on the target entity.

Code 5-1 shows an example of the owning side of a unidirectional one-to-one relationship.

Code 5-1 Unidirectional One-to-one Association Owning Entity Example

```

@Entity
public class Customer {
    @Id
    private int id;
    @OneToOne
    private Record custRecord;
    ...
}
  
```

Code 5-2 shows the inverse side of the relationship.

Code 5-2 Unidirectional One-to-one Association Inverse Entity Example

```
@Entity
public class Record {
    @Id
    private int recId;
    ...
}
```

Figure 5-11 illustrates the table with join column matching the corresponding entities.

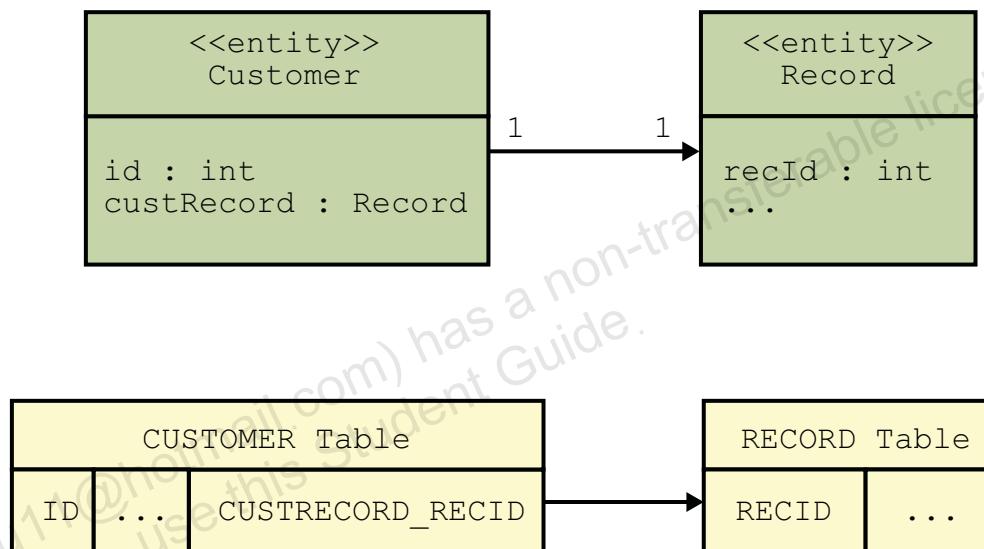


Figure 5-11 Join Column for One-to-One Unidirectional Association

The previously discussed procedure is a fairly generic way of specifying a unidirectional one-to-one annotation. The following notes highlight variations to the generic procedure.

- You can override the default name of the join column by following the `OneToOne` annotation with a `JoinColumn` annotation in the owning entity.
- If the two tables share the same primary key values you should use the `PrimaryKeyJoinColumn` annotation instead of the `JoinColumn` annotation in the owning entity.
- If the two tables are joined using multiple columns you must use the `JoinColumns` annotation instead of the default or a `JoinColumn` annotation in the owning entity.

Implementing One-to-One Bidirectional Association

Figure 5-12 shows a one-to-one bidirectional association between two entities.

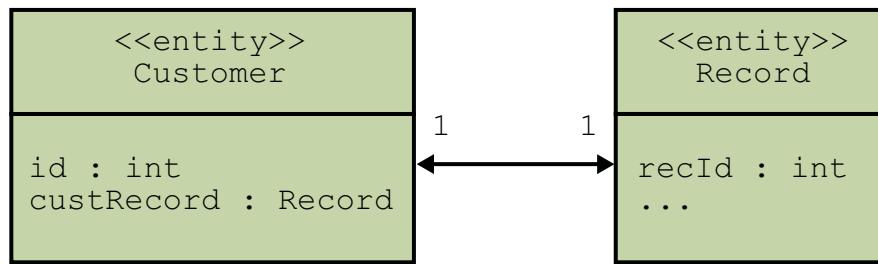


Figure 5-12 One-to-One Bidirectional Association

The following steps outline a process you can follow to define a one-to-one bidirectional association between two entity classes.

1. Define the two entity classes.
2. Identify the entity that is the owner of the relationship.
3. In the owning entity, create a (relationship) property or field to represent the relationship this entity has with the other entity.
4. In the owning entity, annotate the relationship field or property with the `OnetoOne` annotations.
5. In the inverse entity, create a relationship property or field to represent the relationship this entity has with the owning entity.
6. In the inverse entity, annotate the relationship field or property with the `OnetoOne` annotation. Set the value of the `mappedBy` attribute of the `OnetoOne` annotation to the value of the relationship field or property in the owning entity.

Code 5-3 shows an example of the owning side of a bidirectional one-to-one relationship.

Code 5-3 Bidirectional One-to-One Association Owning Entity Example

```

@Entity
public class Customer {
    @Id
    private int id;
    @OneToOne
    private Record custRecord;
    ...
}
  
```

Code 5-4 shows the inverse side of the relationship.

Code 5-4 Bidirectional One-to-One Association Inverse Entity Example

```
@Entity
public class Record {
    @Id
    private int recId;
    @OneToOne(mappedBy="custRecord")
    private Customer customer;
    ...
}
```

Figure 5-13 illustrates the table with join columns matching the corresponding entities..

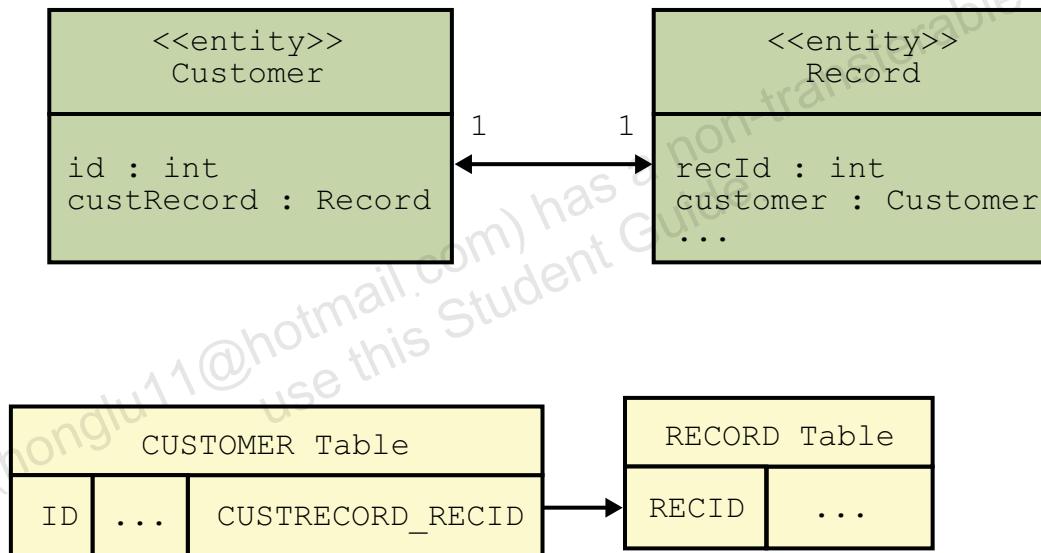


Figure 5-13 Join Columns for One-to-One Bidirectional Association

The previously discussed procedure uses the persistence API's object relational mapping defaults to specify a one-to-one relationship. The following notes highlight variations to using the defaults.

- If the two tables are joined using multiple columns you should use the `JoinColumns` annotation instead of the `JoinColumn` annotation in the owning entity.
- If the two tables share the same primary key values you should use the `PrimaryKeyJoinColumn` annotation instead of the `JoinColumn` annotation in the owning entity.

Implementing One-to-Many/Many-to-One Bidirectional Association

Figure 5-14 shows a one-to-many/many to one bidirectional association between two entities.

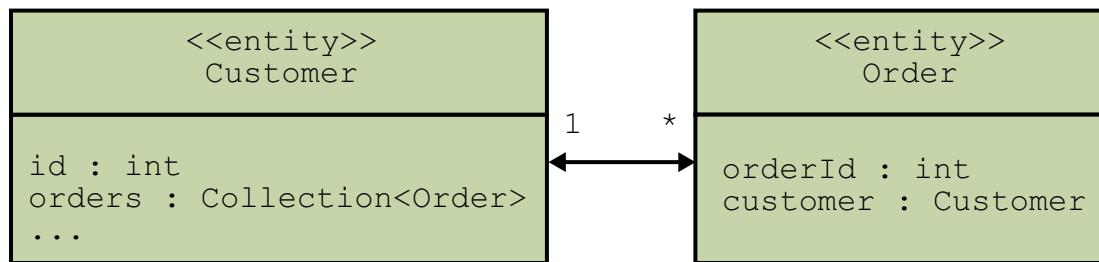


Figure 5-14 One-to-Many/Many-to-One Bidirectional Association

The following steps outline a process you can follow to define a many-to-one/one-to-many bidirectional association between two entity classes.



Note – In many-to-one/one-to-many associations, the owning entity is always the entity on the many side of the relationship.

1. Define the two entity classes.
2. Identify the entity that is the many-side of the relationship.
3. In the owning entity, create a (relationship) property or field to represent the relationship this entity has with the other entity.
4. In the many-side entity, annotate the relationship field or property with the `ManyToOne` annotations.
5. In the one-to-many entity, create a relationship property or field to represent the relationship this entity has with the other entity.
6. In the one-to-many entity, annotate the relationship field or property with the `OneToOne` annotation.
7. Set the value of the `mappedBy` attribute to the value of the relationship or property in the owning entity. The `mappedBy` attribute is always set on the inverse entity. For a many-to-one/one-to-many association, the inverse entity is the class that contains the `OneToOne` annotation.

Code 5-5 shows an example of a bidirectional one-to-many relationship.

Code 5-5 Bidirectional One-to-Many Association Example

```
@Entity
public class Customer {
    @Id
    private int id;
    @OneToMany(mappedBy="customer")
    private Collection <Order> orders;
    ...
}
```

Code 5-6 shows an example of a bidirectional many-to-one relationship.

Code 5-6 Bidirectional Many-to-One Association Example

```
@Entity
public class Order {
    @Id
    private int orderId;
    @ManyToOne
    private Customer customer;
    ...
}
```

Implementing One-to-Many/Many-to-One Bidirectional Association

Figure 5-15 illustrates the table with join columns matching the corresponding entities.

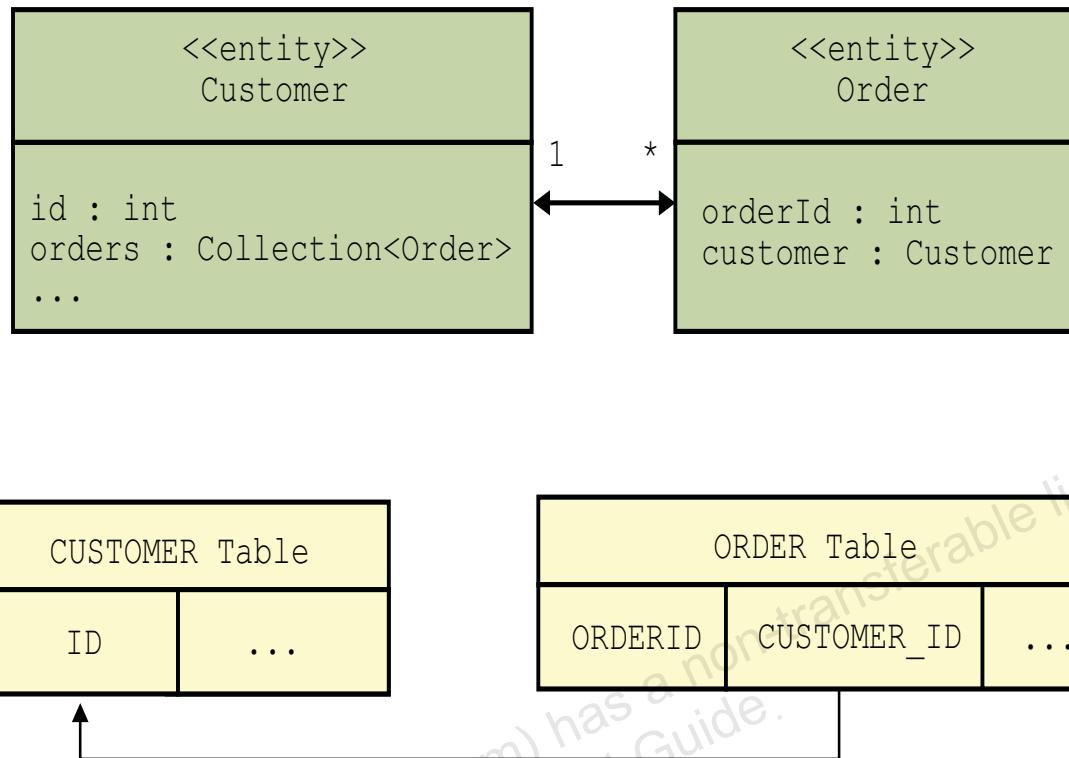


Figure 5-15 Join Columns for One-to-Many/Many-to-One Bidirectional Association

Implementing Many-to-Many Bidirectional Association

Figure 5-16 shows a many-to-many bidirectional association between two entities.

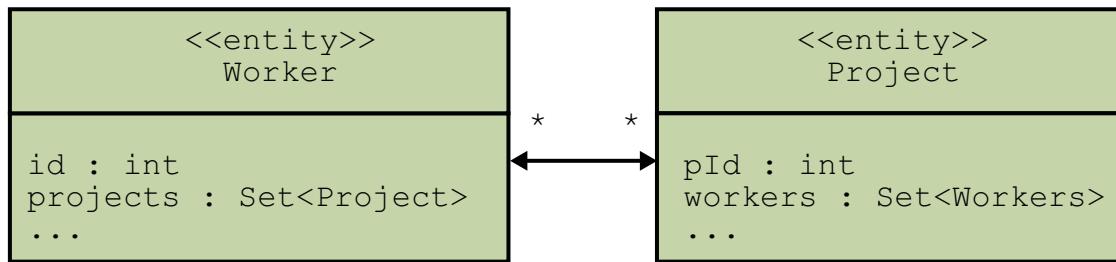


Figure 5-16 Many-to-many Bidirectional Association

The following steps outline a process you can follow to define a many-to-many bidirectional association between two entity classes.

1. Define the two entity classes.
2. Identify the entity that is the owner of the relationship.
3. In the owning entity, create a (relationship) property or field to represent the relationship this entity has with the other entity.
4. In the owning entity, annotate the relationship field or property with the `ManyToMany` annotations.
5. In the inverse entity, annotate the relationship field or property with the `ManyToMany` annotation. Set the value of the `mappedBy` attribute of the `ManyToMany` annotation to the value of the relationship field or property in the owning entity.

Code 5-7 shows an example of the owning side of a bidirectional many-to-many relationship.

Code 5-7 Bidirectional Many-to-Many Association Owning Entity Example

```

@Entity
public class Worker {
    @Id
    private int id;
    @ManyToMany
    private Set<Project> projects;
    ...
}
  
```

Code 5-8 shows the inverse side of the relationship.

Code 5-8 Bidirectional Many-to-Many Association Inverse Entity Example

```
@Entity
public class Project {
    @Id
    private int pId
    @ManyToMany(mappedBy="projects")
    private Set <Worker> workers;
    ...
}
```

Figure 5-17 illustrates the table with join columns matching the corresponding entities.

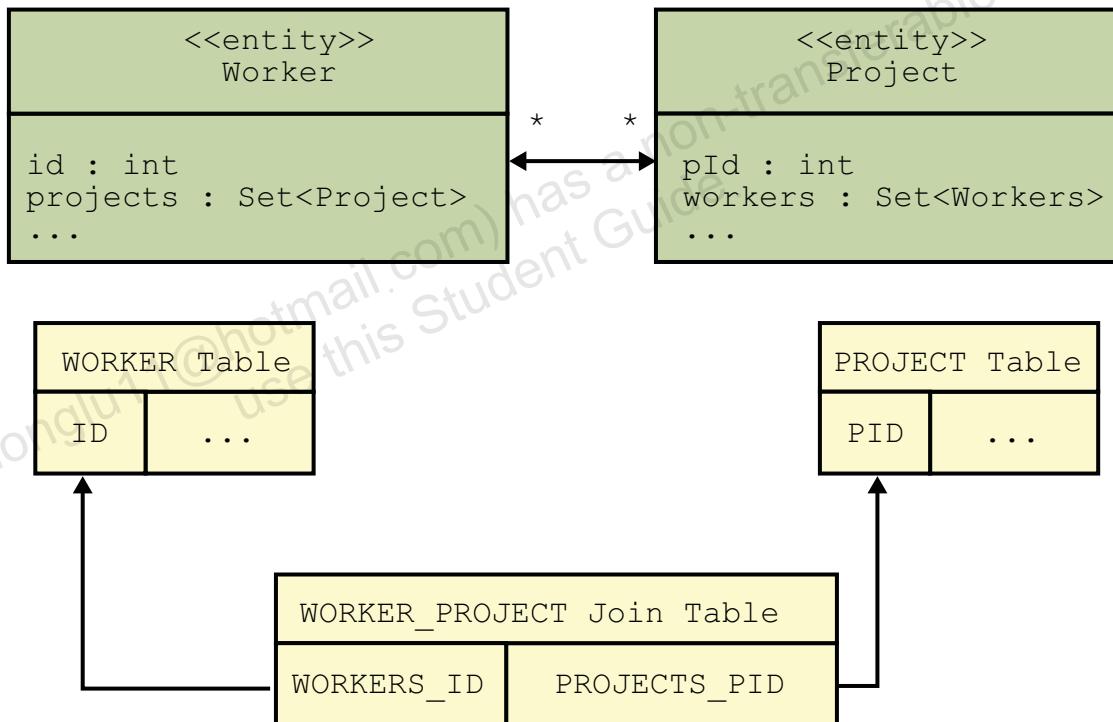


Figure 5-17 JoinTable for One-to-Many/Many-to-One Bidirectional Association

Examining Fetch and Cascade Mode Settings

Association mapping annotations have a number of optional attributes you can set. This section discusses the following two attributes:

- Fetch mode attribute
- Cascade mode attribute

Fetch Mode Attribute

All annotations that specify association mappings have a fetch mode attribute. For example,

```
@ManyToOne(fetch=EAGER)  
private Customer customer;
```

Fetch mode settings are instructions to the persistence provider. You can specify two possible values (EAGER or LAZY) for the fetch mode attribute. Fetch mode settings are hints to the persistence provider with regard to the timing of the loading into memory and the targets of the association. The container interprets the fetch mode setting as follows.

- EAGER

A fetch mode setting of EAGER (either explicit or by default) instructs the persistence provider to load the association target at the time the entity is loaded from the database. The EAGER strategy is a requirement on the persistence provider runtime that data must be eagerly fetched.

- LAZY

A fetch mode setting of LAZY (either explicit or by default) is a *hint* to the persistence provider to defer the loading of the association target to the time it (association target) is accessed. The implementation is permitted to eagerly fetch data for which the LAZY strategy hint has been specified. In particular, lazy fetching might only be available for mappings for which property-based access is used.

Examining Fetch and Cascade Mode Settings

In addition to association annotations, you can also specify the fetch mode for the `Basic` and `LOB` annotation. Table 5-2 contains the default values for annotations.

Table 5-2 Fetch Mode Default values

Annotation	Default
<code>Basic</code>	<code>EAGER</code>
<code>OneToOne</code>	<code>EAGER</code>
<code>ManyToOne</code>	<code>EAGER</code>
<code>OneToMany</code>	<code>LAZY</code>
<code>ManyToMany</code>	<code>LAZY</code>

The `Lob` annotation does not have a fetch mode attribute. To override the default, follow the `LOB` annotation with a `Basic` annotation. For example,

```
@Lob @Basic
private Image photo;
```

The default fetch mode for the `Basic` annotation is eager.

Cascade Mode Attribute

In addition to fetch mode settings, you can also specify cascade mode settings for all association mapping annotations. For example,

```
@OneToOne(cascade=PERSIST)
private Customer customer;
```

Cascade mode settings can cause specific entity instance life-cycle events to cascade across relationships. Cascade mode settings are used when the EntityManager API is invoked. The default is for no cascading to occur. Table 5-3 contains the possible values for the cascade attribute.

Table 5-3 Cascade Mode Attribute Values

Value	Comment
PERSIST	Causes the persist operation to cascade to the target entity of the association when the entity manager's persist operation is invoked on the source entity.
MERGE	Causes the merge operation to cascade to the target entity of the association when the entity manager's merge operation is invoked on the source entity.
REMOVE	Causes the remove operation to cascade to the target entity of the association when the entity manager's remove operation is invoked on the source entity.
REFRESH	Causes the refresh operation to cascade to the target entity of the association when the entity manager's refresh operation is invoked on the source entity.
ALL	Causes all the entity state change operations (persist, merge, remove, and refresh) to cascade to the target entity of the association.

Examining Fetch and Cascade Mode Settings

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Hong Lu (honglu11@hotmail.com) has a non-transferable license to
use this Student Guide.

Module 6

Implementing Entity Classes: Modelling Inheritance and Embedded Relationships

Objectives

Upon completion of this module, you should be able to:

- Examine entity inheritance
- Examining Object/Relational Inheritance Hierarchy Mapping Strategies
- Inherit from an entity class
- Inherit using a mapped superclass
- Inherit from a non-entity class
- Examine inheritance mapping strategies
- Use an embeddable class
- Use a composite primary key

Additional Resources



Additional resources – The following reference provides additional information on the topics described in this module:

- Sun Microsystems, “JSR 220: Enterprise JavaBeans™, Version 3.0 Java Persistence API.” [<https://sdlc3e.sun.com/ECom/EComActionServlet;jsessionid=CEAAE57A3BAB8A76D4555E3C5A1F4031>], accessed July 25, 2006.

Examining Entity Inheritance

This section examines entity inheritance. It uses a FAQ list to present the context information you need to use entity inheritance.

- What is meant by entity inheritance?
Entity inheritance is the provision of standard Java technology inheritance capability to entity classes. The end result is that you have the capability to create a set of entity classes that have a hierarchical relationship.
- From a developers perspective, what are the key decisions associated with using inheritance in entity classes?

The two key decisions you would need to make are:

- The choice of class for the inheriting classes' superclass
- The choice of object/relational mapping strategy
- What choices exist for selecting the entity classes superclass?
An entity instance's superclass can be any of the following:
 - An entity class (abstract or concrete)
 - A mapped superclass (abstract or concrete)
 - A non-entity class (abstract or concrete)
- What object/relational mapping strategies are available to map entity classes that use inheritance?

The Java Persistence API proposes the following object/relational mapping strategies for entity inheritance:

- Single table per class hierarchy strategy
- Table per class strategy
- Joined subclass strategy
- How is the object/relational mapping for entity inheritance specified?

The object relational mapping strategy for entity inheritance is specified using the `strategy` attribute of the `Inheritance` annotation or the `inheritance` element in the XML descriptor.

Examining Object/Relational Inheritance Hierarchy Mapping Strategies

This section introduces you to the available object/relational inheritance mapping strategies. Consider the following hierarchy of entity classes shown in Code 6-1.

```
1 // hierarchy root
2 @Entity public class InsurancePolicy{
3     @Id int policyId;
4     String client;
5     double premium;
6     //
7 }
8
9 // Direct child of hierarchy root
10 @Entity public class CarInsurance extends InsurancePolicy {
11     int vin;          // vehicle identification number
12     boolean comprehensive;
13     //
14 }
15
16 // Another direct child of hierarchy root
17 @Entity public class HouseInsurance extends InsurancePolicy {
18     String dwellingType;
19     String address;
20     boolean flood;
21     //
22 }
```

Code 6-1 Entity Class Hierarchy

To map the inheritance hierarchy shown in Code 6-1 to a relational database, you must use one of the following strategies.

- Single table per class hierarchy strategy
- Table per class strategy
- Joined subclass strategy

Object relational mapping strategy for entity inheritance is specified using the `strategy` attribute of the `Inheritance` annotation or the `inheritance` element in the deployment descriptor.



Note – The mapping strategies discussed in this section apply to entity inheritance hierarchy that has an entity class as the root.

The following sections provide details of each of these mapping strategies.

Mapping Inheritance Using the Single Table per Class Hierarchy Strategy

The following FAQ list highlights the context information you need to use the single table per class hierarchy object/relational mapping strategy.

- What is the single table per class hierarchy object/relational mapping strategy?
 - All the classes in a hierarchy map to a single table in the relational database.
 - This single table has a column for each unique persistence field in the hierarchy.
 - In addition, the single table contains an extra column called the discriminator column. The value in the discriminator column identifies the specific subclass to which the instance that is represented by the row belongs.

Figure 6-1 shows the table structure for the single table per class hierarchy strategy for the entity hierarchy shown in Code 6-1 on page 6-4.

INSURANCE_POLICY							
POLICYID	DISC	CLIENT	PREMIUM	VIN	COMPREHENSIVE	ADDRESS	FLOOD

Figure 6-1 Single Table per Hierarchy Example

- What are the benefits of using single table per class hierarchy object/relational mapping strategy?

This mapping strategy provides good support for polymorphic relationships between entities and for queries that range over the class hierarchy.

Examining Object/Relational Inheritance Hierarchy Mapping Strategies

- What are the drawbacks of using single table per class hierarchy object/relational mapping strategy?
The drawback, however, is that it requires that the columns that correspond to state specific to the subclasses are nullable.
- What steps are required to implement the single table per class hierarchy object/relational mapping strategy?

Annotate the root of the inheritance hierarchy as follows:

```
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
```

Note – The single table strategy is the default. If you use the single table strategy, you do not need to annotate the root class with the Inheritance annotation.



Mapping Inheritance Using a Table per Class Strategy

The following FAQ list highlights the context information you need to use the table per class object/relational mapping strategy.

- What is the table per class object/relational mapping strategy?
 - This mapping strategy requires a table for each concrete class in the hierarchy.
 - Each table has a column for each persistence field (including inherited fields) in the entity to which it maps.

Figure 6-2 shows the table structure for the table per class strategy for the entity hierarchy shown in Code 6-1 on page 6-4

INSURANCE_POLICY				
POLICYID	CLIENT	PREMIUM		
CAR_INSURANCE				
POLICYID	CLIENT	PREMIUM	VIN	COMPREHENSIVE
HOUSE_INSURANCE				
POLICYID	CLIENT	PREMIUM	ADDRESS	FLOOD

Figure 6-2 Table per Entity Class Example

- What are the drawbacks of using a table per class object/relational mapping strategy?
 - It provides poor support for polymorphic relationships.
 - It typically requires that SQL UNION queries (or a separate SQL query per subclass) be issued for queries that are intended to range over the class hierarchy.

Note – The Java Persistence API does not require application servers to support the table per class mapping strategy



Mapping Inheritance Using the Joined Subclass Strategy

The following FAQ list highlights the context information you need to use the joined subclass object/relational mapping strategy.

- What is the joined subclass object/relational mapping strategy?
 - This mapping strategy requires a common table to represent the root entity class in the hierarchy.
 - Each entity subclass is represented by a separate table that contains columns specific to the subclass as well as the columns that represent its primary key.
 - The primary key column(s) of the subclass table serves as a foreign key to the primary key of the superclass table.

Figure 6-3 shows the table structure for the joined subclass strategy for the entity hierarchy shown in Code 6-1 on page 6-4.

INSURANCE_POLICY		
POLICYID	CLIENT	PREMIUM
CAR_INSURANCE		
POLICYID	VIN	COMPREHENSIVE
HOUSE_INSURANCE		
POLICYID	ADDRESS	FLOOD

Figure 6-3 Example: Tables for Joined Subclass Mapping Strategy

- What are the benefits of using a joined subclass object/relational mapping strategy?
This strategy provides support for polymorphic relationships between entities.
- What are the drawbacks of using a joined subclass object/relational mapping strategy?
It requires that one or more join operations be performed to instantiate instances of a subclass. In deep class hierarchies, this can lead to unacceptable performance. Queries that range over the class hierarchy likewise require joins.

Inheriting From an Entity Class

You can create an entity class that inherits from either a concrete or abstract entity class.

The following steps outline a process you can follow to create an inheritance hierarchy of entity classes.

1. Declare the root entity class of the inheritance hierarchy.
The root class can be an abstract entity class.
2. Choose the object/relational strategy you plan to use for the inheritance hierarchy you are about to create.
3. Use the `strategy` attribute of the `Inheritance` annotation to specify your selected object/relational mapping strategy on the root entity class.

Some application servers might allow you to override the strategy specified in the root entity by permitting you to specify an overriding strategy in the child class.

4. Declare the remaining entity classes of the hierarchy.
Each child entity class uses the `extend` keyword to extend the parent entity class.

Code 6-2 shows an example of an entity class used as the root of an inheritance hierarchy.

Code 6-2 Root Entity Class Example

```
1  @Entity @Table(name= "EMP" )
2  @Inheritance(strategy=InheritanceType.JOINED)
3  public abstract class Employee {
4      @Id protected Integer empId;
5      @Version protected Integer version;
6      @ManyToOne protected Address address;
7      // ...
8  }
```

Code 6-2 uses the joined object/relational mapping strategy.

Inheriting From an Entity Class

Code 6-3 shows an example of a child entity class extending the root of the inheritance hierarchy.

Code 6-3 Child Entity Extending Entity Class Example 1

```

1  @Entity @Table(name="FT_EMP")
2  @DiscriminatorValue("FT")
3  @PrimaryKeyJoinColumn(name="FT_EMPID")
4  public class FullTimeEmployee extends Employee {
5      // Inherit empId, but mapped in this class to FT_EMP.FT_EMPID
6      // Inherit version mapped to EMP.VERSION
7      // Inherit address mapped to EMP.ADDRESS fk
8      protected Integer salary;
9      // Defaults to FT_EMP.SALARY
10     public Integer getSalary() {
11         return salary;
12     }
13     // ...
14 }
```

The use of the `DiscriminatorValue` annotation is optional and applies to only concrete entity classes. If it is not supplied, a persistence provider specific discriminator value is used. Note that Code 6-2 on page 6-9 does not contain a `DiscriminatorColumn` annotation declaration. In this case, the default discriminator column having the column name `DTYPE` and column data type `STRING` is used. The joined subclass strategy can use a discriminator column to implement object/relational mapping, but it is not required to do so.

Code 6-4 shows a second example of a child entity class extending the root of the inheritance hierarchy.

Code 6-4 Child Entity Extending Entity Class Example 2

```

1  @Entity @Table(name="PT_EMP")
2  @DiscriminatorValue("PT")
3  // PK field is PT_EMP.EMPID due to PrimaryKeyJoinColumn default
4  public class PartTimeEmployee extends Employee {
5      protected Float hourlyWage;
6      // ...
7 }
```

Inheriting Using a Mapped Superclass

You can create an entity class that inherits from either a concrete or abstract non entity class known as a mapped superclass.

The following FAQ list highlights the context information you need to create entity classes that inherit from a mapped superclass.

- What is a mapped superclass?
A mapped superclass is a plain Java technology class that is designated with the `MappedSuperclass` annotation or `mapped-superclass` XML descriptor element.
- What is the purpose of inheriting from a mapped superclass class?
A mapped superclass is primarily created to provide state and the corresponding object/relational mapping information to child classes through the inheritance mechanism.
- What is inherited?
All behavior, state, and state mapping information.
- What annotations in the superclass are processed?
All object/relational mapping annotations declared in the mapped superclass.
- What features are common between a mapped superclass and an entity?
The object/relational mapping metadata of the state variables is common to both mapped superclasses and entities.
- How does a mapped superclass differ from an entity?
State variables declared in an entity map directly to a table in the database. In contrast, a mapped superclass does not have a table that corresponds to it directly. Instead, the state variables declared in a mapped superclass map to a table that corresponds to the inheriting entity subclass.
Mapped superclasses cannot be passed as arguments to methods of the `EntityManager` or `Query` interfaces. A mapped superclass cannot be the target of a persistent relationship.
Using a mapped superclass loses the benefit of polymorphism.

Inheriting Using a Mapped Superclass

Code 6-5 is an example of a mapped superclass that is extended by the entity class shown in Code 6-6 and Code 6-7.

Code 6-5 Mapped Superclass Example

```

1  @MappedSuperclass
2  public class Employee {
3      @Id protected Integer empId;
4      @Version protected Integer version;
5      @ManyToOne @JoinColumn(name="ADDR")
6      protected Address address;
7      public Integer getEmpId() { ... }
8      public void setEmpId(Integer id) { ... }
9      public Address getAddress() { ... }
10     public void setAddress(Address addr) { ... }
11 }
```

Code 6-6 shows an example of a child entity class extending the mapped superclass shown in Code 6-5.

Code 6-6 Example of Entity Extending Mapped Superclass

```

1 // Default table is FTEMPLOYEE table
2 @Entity
3 public class FTEmployee extends Employee {
4     // Inherited empId field mapped to FTEMPLOYEE.EMPID
5     // Inherited version field mapped to FTEMPLOYEE.VERSION
6     // Inherited address field mapped to FTEMPLOYEE.ADDR fk
7     protected Integer salary; // Defaults to FTEMPLOYEE.SALARY
8
9     public FTEmployee() {}
10    public Integer getSalary() { ... }
11    public void setSalary(Integer salary) { ... }
12 }
```

Code 6-7 shows a second example of a child entity class extending the mapped superclass shown in Code 6-5.

Code 6-7 Second Example of Entity Extending Mapped Superclass

```
1  @Entity @Table(name= "PT_EMP" )
2  @AssociationOverride(name= "address",
3      joinColumns=@JoinColumn(name= "ADDR_ID" ))
4  public class PartTimeEmployee extends Employee {
5      // Inherited empId field mapped to PT_EMP.EMPID
6      // Inherited version field mapped to PT_EMP.VERSION
7      // address field mapping overridden to PT_EMP.ADDR_ID fk
8      @Column(name= "WAGE" ) protected Float hourlyWage;
9
10     public PartTimeEmployee() {}
11     public Float getHourlyWage() { ... }
12     public void setHourlyWage(Float wage) { ... }
13 }
```

Inheriting From a Non-Entity Class

Inheriting From a Non-Entity Class

You can create an entity class that inherits from either a concrete or abstract non-entity class.

The following FAQ list highlights the context information you need to create entity classes that inherit from a non-entity class.

- What is the purpose of inheriting from a non-entity class?
To inherit the behavior of the non-entity parent and not the state. All inherited attributes are transient.
- What is persistent and what is not persistent?
State declared in the entity class is persistent. State declared in the non-entity superclass is not persistent.
This non-persistent state is not managed by the `EntityManager`, nor is it required to be retained across transactions.
- What annotations are processed?
Annotations declared on the entity class are processed. Any annotations on non entity superclasses are ignored.
- What other restrictions apply to non entity superclasses?
Non entity classes cannot be passed as arguments to methods of the `EntityManager` or `Query` interfaces and cannot carry mapping information.

Code 6-8 is an example of a non entity class that is extended by the entity class shown in Code 6-9.

Code 6-8 Example of a Non-Entity Class for Use as a Superclass

```

1  public class Cart {
2      // This state is transient
3      Integer operationCount;
4
5      public Cart() {
6          operationCount = 0;
7      }
8
9      public Integer getOperationCount() {
10         return operationCount;
11     }
12
13     public void incrementOperationCount() {
14         operationCount++;
15     }
16 }
```

Code 6-9 shows an example of a child entity class extending a non-entity class.

Code 6-9 Example of an Entity Inheriting From a Non-Entity Class

```
1  @Entity public class
2  ShoppingCart extends Cart {
3      Collection<Item> items = new Vector<Item>();
4      public ShoppingCart() {
5          super();
6      }
7      //...
8      @OneToMany
9      public Collection<Item> getItems() {
10         return items;
11     }
12     public void addItem(Item item) {
13         items.add(item);
14         incrementOperationCount();
15     }
16 }
```

Entity Classes: Using an Embeddable Class

You can create an entity class that embeds another class defined by the Java persistence API as an embeddable class. The following FAQ list highlights the context information you need to create entities that use embeddable classes.

- What is an embeddable class?

An embeddable class is a fine grained class that encapsulates common state used by many domain classes. An entity class then embeds the embeddable class.

- Why use an embeddable class?

Embeddable classes are appropriate for the expression of composition relationships. For example, an Address class would make a good candidate for being marked as an embeddable class. The states represented by an Address class (street, city, postal code, country) are commonly required in many domain classes that represent customer, employee, and supplier. It would be inappropriate for these domain classes to inherit from the Address class.

- How is the persistence identity affected?

Embeddable class instances do not have persistent identity. Instead, they exist only as embedded objects of the entity to which they belong.

- What are the requirements for an embeddable class?

Embeddable classes must meet the requirement for an entity class. Embeddable classes are annotated with the `Embeddable` annotation instead of the `Entity` annotation.

- How many levels of embedding is permitted?

Support for only one level of embedding is required by the Java persistence specification.

Defining Entity Classes: Using an Embeddable Class

The rules for defining an embeddable class are the same as that for defining an entity class. Annotate the embeddable class with the `Embeddable` annotation instead of the `Entity` annotation. Code 6-10 shows an example of an embeddable class.

Code 6-10 Example of an Embeddable Class

```
1  @Embeddable
2  public class Address implements Serializable {
3      String addressLine1;
4      String addressLine2;
5      String city;
6      String postalCode;
7      String country
8
9      public Address() {}
10 }
```

Code 6-11 shows an example of using an embeddable class.

Code 6-11 Example of Using an Embedded Class

```
1  @Entity
2  public class Customer implements Serializable {
3      @Id
4      private int custId;
5      private String name;
6      @Embedded
7      private Address address;
8
9      public Customer() {}
10
11     //...
12 }
```

Entity Classes: Using a Composite Primary key

Entity Classes: Using a Composite Primary key

Under some circumstances, you might need to use a composite primary key. This section uses a FAQ list to present the context information you require about embeddable classes.

- What is composite primary key?

A composite primary key (as opposed to a simple primary key) uses the value of multiple fields or properties as a composite to uniquely identify an entity.

- What options are available to define a composite primary key?

You have two options:

- Use the `EmbeddedId` annotation
- Use the `IdClass` annotation

Defining a Composite Primary Key Using the EmbeddedId Annotation

Using the `EmbeddedId` annotation is a two step process.

1. Define an embeddable class that contains only the fields or properties that make up the composite primary key.
2. Embed the composite primary key class in the entity class using the `EmbeddedId` annotation.

Code 6-12 shows an example of an embeddable composite key class.

Code 6-12 Example of an Embeddable Class

```
1  @Embeddable
2  public class EmployeePK{
3      public String name;
4      public Date birthDate;
5  }
```

Code 6-13 shows an example of using an embeddable composite key class.

Code 6-13 Example of Using an Embeddable Class as a Composite Key

```
1  @Entity
2  public class Employee{
3      @EmbeddedId
4      protected EmployeePK empPK;
5      //...
6  }
```

Defining a Composite Primary Key Using the `IdClass` Annotation

The following procedure summarizes the steps you require to use the `IdClass` annotation to define a composite primary key.

1. Define a (primary key) class that contains only the fields or properties that make up the composite primary key.

The class you define must conform to the following rules:

- The primary key class must be public and must have a public no-arg constructor.
- If property-based access is used, the properties of the primary key class must be public or protected.
- The primary key class must be serializable.
- The primary key class must define `equals` and `hashCode` methods.

2. Annotate the entity class using the `IdClass` annotation.

3. Declare, in the entity class, a duplicate of every field or property declared in the primary key class.

The names of primary key fields or properties in the primary key class and those of the entity class must correspond and their types must be the same.

4. Annotate every field or property of the primary key class declared in the entity class with the `Id` annotation.

Code 6-14 shows an example of a class that can be used as a composite primary key class.

Code 6-14 Example of a Composite Primary Key Class

```
1  public class EmployeePK implements Serializable{  
2      String name;  
3      Date birthDate;  
4  }
```

Code 6-15 shows an example of using the `IdClass` annotation to define a composite key for an entity class.

Code 6-15 Example of Using an Embeddable Class as a Composite Key

```
1  @IdClass(EmployeePK.class).  
2  @Entity  
3  public class Employee{  
4      @Id String name;  
5      @Id Date birthDate;  
6      String department  
7      //...  
8  }
```


Module 7

Using the Java Persistence Query Language

Objectives

Upon completion of this module, you should be able to:

- Describe querying over entities and their persistence state
- Examine query objects
- Create and use query objects
- Declare Java persistence query language statements:
 - Declare the `SELECT` statement
 - Declare the `UPDATE` statement
 - Declare the `DELETE` statement

Additional Resources

Additional Resources

Additional resources – The following references provide additional information on the topics described in this module:

- Sun Microsystems, “JSR 220: Enterprise JavaBeans™, Version 3.0 Java Persistence API.” [<https://sdlc3e.sun.com/ECom/EComActionServlet;jsessionid=CEAAE57A3BAB8A76D4555E3C5A1F4031>], accessed July 25, 2006.
- Sun Microsystems, “The Java EE 5 Tutorial, Chapter 29.” [<http://java.sun.com/javaee/5/docs/tutorial/doc/>], accessed July 25, 2006.
- Sun Microsystems, “Enterprise JavaBeansQuery Language,” [http://java.sun.com/j2ee/tutorial/1_3-fcs/doc/EJBQL.html], accessed July 3, 2006.

Introducing Querying Over Entities and Their Persistence State

The Java Persistence API provides a mechanism to query over entities and their persistence state. This mechanism consists of:

- Query statement strings written in the Java Persistence Query Language (QL)
QL is further described in “Introducing the Java Persistence Query Language” on page 7-8.
- Objects that implement the `javax.persistence.Query` interface
Query objects are further described in “Examining Query Objects” on page 7-4.

The following steps outline the process required to use QL statements and query objects in a Java EE application. In particular you should note the requirement to wrap QL statement strings using query objects of type `javax.persistence.Query`, shown in Step 2.

1. Create QL statement string.

```
String queryString = "SELECT o FROM Order o "+  
" WHERE o.shippingAddress.state = 'CA'";
```

This select statement returns `Order` objects with a shipping address in the state of California.

2. Create a query object wrapper for the QL string, of type `javax.persistence.Query` interface.

```
// Assume entityManager references an EntityManager object  
Query query = entityManager.createQuery(queryString); // create Query
```

3. Execute the query object.

```
Collection <Order> orders = query.getResultList(); // execute Query
```

Examining Query Objects

A query object is an instance of the `javax.persistence.Query` interface. A query object encapsulates either a QL query string or a native (SQL) query string. The following FAQs provide a brief introduction to query objects.

- How is a query created?

A query object is created by invoking the `createQuery` method of an entity manager instance. For example:

```
Query query = entityManager.createQuery
("select o from Order o where customer.id = :custId " +
"order by o.createdDatetime");
```

- What operations are commonly invoked on a query object?

Code 7-1 shows several operations that are commonly invoked on a query object.

Code 7-1 Query Code Example

```
1 public List findOrdersByCustomer(Customer cust, int max){
2     return entityManager.createQuery(
3         "select o from Order o where customer.id = :custId " +
4         "order by o.createdDatetime")
5         .setParameter("custId", cust.getId())
6         .setMaxResults(max)
7         .getResultList();
8 }
```

The `setParameter` method assigns a value to the named parameter (of the QL string) `custID`. For positional parameters, use the overloaded `setParameter(int position, Object value)` method.

The `setMaxResults` method sets the maximum number of results to retrieve. To control the start position of the result set, you can use the `setFirstResult(int startPosition)` method.

The `getResultList` method executes the select query and returns the result as a list object. To return a single object, use the `getSingleResult` method. For update or delete queries, use the `executeUpdate` method.

- What is a named query object?

Named queries are static queries expressed in metadata. Named queries can be defined in QL or in SQL. Query names are scoped to the persistence unit.

- How is a named query created?

A named query is first specified using the `NamedQuery` metadata annotation. It is then created using the `createNamedQuery` method of the entity manager. The `NamedQuery` annotation can be applied to an entity or mapped superclass. Code 7-2 contains an example of the creation and use of a named query.

Code 7-2 Named Query Declaration Example

```

1  @Entity
2  @NamedQuery(name="FindAllClosedAuctions",
3      query=" SELECT OBJECT(a) FROM Auction AS a " +
4      "WHERE a.status= 'CLOSED'" )
5  public class Auction implements Serializable {
6      // ...
7 }
```

Code 7-3 shows a code snippet that uses the named query declared in Code 7-2.

Code 7-3 Using Named Query Example

```

1 // This is a code snippet
2 // Assume entityManager references an instance of an EntityManager
3 Query query =
4     entityManager.createNamedQuery("FindAllClosedAuctions");
5 Collection<Auction> auctions = query.getResultList();
6 // ...
```

- What is a native query?

Queries can be expressed in native SQL. The result of a native SQL query might consist of entities, scalar values, or a combination of the two. The entities returned by a query might be of different entity types. The SQL query facility is intended to provide support for those cases where it is necessary to use the native SQL of the target database in use (and/or where QL cannot be used). Native SQL queries are not expected to be portable across databases.

- How is a native query created?

A native query is created using one of the `createNativeQuery` methods of the entity manager. For example:

```
Query q = em.createNativeQuery("SELECT o.id, o.quantity, o.item " +
    "FROM Order o, Item i " +
    "WHERE (o.item = i.id) AND (i.name = 'widget')", com.acme.Order.class);
```

Creating and Using Query Objects

The following steps provide a summary of the actions needed to create and use a query object to select, or update, or delete from the database.

1. Identify the task you need to perform: select, update, or delete information from the database.
2. Determine if the task can be performed using an QL query string. If not, you need to create a SQL query string.
3. Create the appropriate query string.

For example,

```
"select o from Order o where customer.id = :custId " +  
    "order by o.createdDatetime"
```

4. Use the query string to create an instance of the Query interface.

For example,

```
Query query = entityManager.createQuery  
( "select o from Order o where customer.id = :custId " +  
    "order by o.createdDatetime" );
```

5. Configure the query object for execution

For example,

```
int id = 24;  
int max = 20;  
query.setParameter( "custId", id )  
query.setMaxResults( max )  
Collection orders = query.getResultList();
```

6. Execute the query using one of the Query interface's query execute methods.

For example,

```
Collection orders = query.getResultList();
```

Declaring and Using Positional Parameters

The code example shown in Step 4 on page 7-6 uses a named parameter (:custId). You could rewrite the same code to use a positional parameter, as shown in the following code snippet:

```
Query query = entityManager.createQuery  
("select o from Order o where customer.id = ?1 " +  
"order by o.createdDatetime");
```

If this was the case, you use the alternative positional format of the setParameter method to set the parameter value for the query, as shown in the following code snippet:

```
int id = 24;  
int max = 20;  
query.setParameter(1, id)  
query.setMaxResults(max)  
Collection orders = query.getResultList();
```

Introducing the Java Persistence Query Language

QL is a query specification language for dynamic queries and for static queries expressed through metadata. QL can be compiled to a target language, such as SQL, of a database or other persistent store. This allows the execution of queries to be shifted to the native language facilities provided by the database, instead of requiring queries to be executed on the runtime representation of the entity state. As a result, query methods can be optimized as well as portable.

QL supports the following statement types.

- **SELECT statement**

The QL SELECT statement enables the querying of entity states and relationships. Behind the scenes, the query operates on the underlying persistence store (database).

A SELECT statement can return row data stored in the database as entity objects or column data (one or more fields of entity instances) as objects.

- **UPDATE statement**

The QL UPDATE statement updates one or more entity instances. Behind the scenes, the update to the entity instances are reflected in updates to rows of a specific table in the database.

- **DELETE statement**

The QL DELETE statement deletes one or more entity instances. Behind the scenes, the delete of entity instances are reflected in the deletion of one or more rows from a specific table in the database.

QL statements consist of QL keywords, entity class names, and entity field identifiers. The following QL select statement example capitalizes, for clarity, the QL keywords it uses. QL keywords are not case sensitive.

```
SELECT o FROM Order AS o
SELECT c.id, c.status FROM Customer c JOIN c.orders o WHERE o.count > 100
```

Note – For easy identification, all QL keywords used in this module are capitalized.



Declaring Query Strings: The SELECT Statement

The QL SELECT statement enables the querying of entities. A SELECT statement can return entity objects (row data stored in the database) or objects that contain one or more fields of entity instances (column data). The following FAQs provide a brief introduction to the constituent clauses of the SELECT statement.

- What is the purpose of a SELECT statement?

The QL SELECT statement is used to select data from entity instances. Depending on what is selected, the query is executed on the persistence tier and data is returned to the object tier either as entity instances or as objects.

- What is the structure of a SELECT statement?

A SELECT statement contains the following structure:

```
SELECT_clause FROM_clause [WHERE_clause] [GROUPBY_clause] [HAVING_clause]  
[ORDERBY_clause]
```



Note – The use of [] in QL expression usage templates used in this module indicates an optional QL keyword or clause.

For example,

```
SELECT o FROM Order o  
SELECT c.id, c.status FROM Customer c JOIN c.orders o WHERE o.count > 100
```

The FROM clause specifies the search domain. The search domain in the FROM clause is entity types.

The SELECT clause specifies what is returned. You can specify the return of entity instances that correspond to entities specified in the FROM clause. You can also further selectively specify one or more fields of entity instances.

The WHERE clause is optional. It specifies a set of restrictive conditions that must be met by the entity instances that are selected for return.

The GROUP BY construct enables the aggregation of values according to the properties of an entity class. The HAVING construct enables conditions to be specified that further restrict the query result as restrictions upon the groups. The GROUP BY and HAVING clauses are often used in combination.

The ORDER BY clause allows the objects or values that are returned by the query to be ordered.

The following sections provide a more detailed examination of the FROM, WHERE, SELECT, GROUP BY and HAVING, and ORDER BY clauses.

Examining the FROM Clause

The FROM clause specifies the search domain in the persistence tier. The following FAQs provide a more in-depth examination of the FROM clause.

- What does a FROM clause specify?

A FROM clause specifies which persistence entities (that is the *from* part) should be queried by the SELECT query.

- How is the FROM clause specified?

A FROM clause is specified using one or more path expressions. For example, the FROM clause in the following SELECT statement contains a single path expression. For clarity, the path expression is underlined.

```
SELECT o.quantity FROM Order o
```

The FROM clause in the following SELECT statement contains two path expressions.

```
SELECT DISTINCT o1 FROM Order o1, Order o2 WHERE o1.quantity > o2.quantity AND o2.customer.lastname = 'Smith' AND o2.customer.firstname = 'John'
```

You should note that path expressions are separated from each using a comma.

- What does a path expression define?

A path expression associates a variable with a persistence entity or a persisted field (or property) of a persistence entity. For example,

```
SELECT DISTINCT o FROM Order o, IN(o.lineItems) l WHERE l.product.productType = 'officeSupplies'
```

In this example, the range variable *o* is associated with the persistence entity Order and the collection member variable *l* is associated with the collection of line-items navigated to from order objects.

- How are path expressions classified?
 - Range variable path expressions
 - Collection member path expressions
 - Association traversing path expressions

Range Variable Path Expressions

A range variable path expression evaluates to a single entity instance (single row of a table). The following statements are two examples of range variable path expressions.

```
SELECT o FROM Order o  
SELECT o.quantity FROM Order o
```

Collection Member Path Expressions

A collection member path expression is a path expression that evaluates to a member of a collection. Collection value path expressions are used when a path expression involves evaluating an entity field that returns a collection. For example, in the following path expression, the state field `lineItems` evaluates to a collection of `LineItem` instances. The variable `l` evaluates to a specific `LineItem` instance within that collection.

```
SELECT DISTINCT o FROM Order o, IN(o.lineItems) l WHERE  
l.product.productType = 'officeSupplies'
```

You should note the declaration of the range variable `o` and the use of the keyword `IN`, in the declaration of the collection member variable.

Association Traversing Path Expressions

An association traversing path expression is used in queries that involve two or more entities that have an association relationship. An association traversing path expression provides SQL type JOIN query capability. An association traversing path expression begins with a root entity and then traverses to a second entity using the association field that links the second entity to the first.

A join is a rule that defines a relationship by determining which records are retrieved. A join is a conditional expression that is applied to two (or more) tables to retrieve a set of records that meet the condition.

The Java persistence specification supports the following association traversing query capabilities.

- Inner joins also known as relationship joins

An inner join returns all records that meet the join condition.

Declaring Query Strings: The SELECT Statement

- Left outer joins
A left outer join returns all records that meet the join condition as well as those that have no matching values.
- Fetch joins
Fetch joins are variations of the inner and outer joins. A fetch join enables the pre-fetching of related data items as a side effect of a query.
- What is the syntax for an inner join path expression?

The syntax for the inner join operation is:

```
[INNER]JOIN join_association_path_expression [AS] identification_variable
```

For example, the following query joins over the relationship between customers and orders. This type of join typically equates to a join over a foreign key relationship in the database.

```
SELECT c FROM Customer c JOIN c.orders o WHERE c.status = '1'
```

The keyword INNER is optional. For example,

```
SELECT c FROM Customer c INNER JOIN c.orders o WHERE c.status = '1'
```

The following equivalent but less clear form of query omits the JOIN keyword but keeps the IN keyword. The query selects those customers of status 1 for which at least one order exists:

```
SELECT OBJECT(c) FROM Customer c, IN(c.orders) o WHERE c.status = '1'
```

- What is the syntax for an outer join path expression?

The syntax for a left outer join is

```
LEFT [OUTER]JOIN join_association_path_expression [AS]
identification_variable
```

For example:

```
SELECT c FROM Customer c LEFT JOIN c.orders o WHERE c.status = '1'
```

The keyword OUTER is optional. For example:

```
SELECT c FROM Customer c LEFT OUTER JOIN c.orders o WHERE c.status = '1'
```

- What is the syntax for an inner or outer fetch join path expression?

The syntax for a fetch join is:

```
[LEFT [OUTER] | INNER]JOIN FETCH join_association_path_expression
```

For example, the following query results in the fetching of all employees in the department that satisfies the join condition.

```
SELECT d FROM Department d LEFT JOIN FETCH d.employees WHERE d.deptno =
'1'
```

The association referenced by the right side of the FETCH JOIN clause must be an association that belongs to an entity that is returned as a result of the query. It is not permitted to specify an identification variable for the entities referenced by the right side of the FETCH JOIN clause, and hence references to the implicitly fetched entities cannot appear elsewhere in the query.

Note – Entities that are pre-fetched as a result of a fetch join are not part of the explicit query result.



Examining the WHERE Clause

The WHERE clause is optional. It specifies a set of restrictive conditions that must be met by the entities (which in turn map to database rows) that are selected for return. The answers to the following questions provide a more in-depth examination of the WHERE clause.

- What is the structure of a WHERE clause?

The WHERE clause consists of the WHERE keyword followed by either a simple conditional expression or a complex conditional expression.

```
WHERE Simple_Conditional_Expression
WHERE Complex_Conditional_Expression
```

- What is the structure of a simple conditional expression?

The syntax for a simple conditional expression is:

```
Operand_1 Conditional_Expression_Operator Operand_2
```

Where Operand_1 and Operand_2 can be any of the following:

- QL identifiers declared in the FROM clause
- QL literals
- QL functions
- Query method parameters

Table 7-1 contains a list of conditional expression types and the conditional expression operators.

Table 7-1 Where Clause Conditional Expression Types

Conditional Expression Type	Conditional Expression Operators	Operand Type
Comparison expression	=, >, >=, <, <=, <>	Arithmetic
Between expression	[NOT] BETWEEN	Arithmetic, string, date_time
Like expression	[NOT] LIKE	String
In expression	[NOT] IN	All types
Null comparison expression	IS [NOT] NULL	
Empty collection comparison expression	IS [NOT] EMPTY	Collection

Table 7-1 Where Clause Conditional Expression Types

Conditional Expression Type	Conditional Expression Operators	Operand Type
Collection member expression	[NOT] MEMBER OF	Collection
Exist expression	[NOT] LIKE	String

The following select statements demonstrate the use of some of the conditional operators listed in Table 7-1.

```
SELECT o1 FROM Order o1, Order o2 WHERE o1.quantity > o2.quantity
SELECT p FROM Person p WHERE p.age BETWEEN 15 and 19
SELECT a FROM Address a WHERE a.country IN ( 'UK', 'US' , 'France' )
SELECT o FROM Order o WHERE o.lineItems IS EMPTY
SELECT a FROM Address a WHERE a.phone LIKE '12%3'
```



Note – The LIKE operator operates on a string expression. The string expression must have a string value. The pattern value is a string literal or a string-valued input parameter in which an underscore (_) stands for any single character, a percent (%) character stands for any sequence of characters (including the empty sequence), and all other characters stand for themselves.

- What is the structure of a complex conditional expression?

The syntax for a complex conditional expression is:

Simple_Cond_Expr_1 Binary_Relational_Operator Simple_Cond_Expr_2

Where the binary relational operator can be one of the following:

- AND
- OR

For example,

```
SELECT DISTINCT o1 FROM Order o1, Order o2 WHERE o1.quantity >
o2.quantity AND o2.customer.lastname = 'Smith' AND o2.customer.firstname=
'John'
```

- What QL literals are available for use in conditional expressions?

You can use the following types of literals in conditional expressions.

- String literals: For example, 'Stephen'
- Numeric Literals: For example, 52, 3.55
- Boolean Literals: TRUE, FALSE

Declaring Query Strings: The SELECT Statement



Note – For more information refer to section 4.6.1 of JSR 220: Enterprise JavaBeans, Version 3.0 Java Persistence API.

- What QL functions are available for use in conditional expressions?

QL defines several functions for use in expressions in the WHERE clause. Table 7-2 lists these functions together with the input parameters and the types returned. Functions are classified as either string functions or arithmetic functions.

Table 7-2 QL Functions

Function Name and Parameters	Return Type
CONCAT(String, String)	String
SUBSTRING(String, start, length)	String
TRIM(char, String)	String
LOWER(String)	String
UPPER(String)	String
LOCATE(String, String [,simple arithmetic expression])	int
LENGTH(String)	int
ABS(number)	int, float or double
SQRT(double)	double
MOD(int, int)	int
SIZE(Collection_valued_path_expr}	int

String functions operate on strings and return either a string or an integer property of the string operated on. Arithmetic functions operate on numeric types and return a number. The argument to an arithmetic function can be a numeric Java technology object type or primitive type.

- What conventions are available for specifying query method parameters in conditional expressions?

You can use one of the following two conventions to specify query method parameters:

- Positional specification of parameters

Positional specification is based on using the position of the parameter in the query method. For example, suppose you have a query method with the following signature,

```
public Collection findAuctionsWithBidsGreaterThan(Double amount);
```

To reference parameters from QL queries, you use the ?1 syntax for the first parameter as shown in the following query.

```
SELECT DISTINCT auc FROM Auction auc, IN(auc.bids) AS b  
WHERE b.amount > ?1
```

For methods with multiple parameters use ?2 for the second parameter and ?3 for the third parameter.

- Named specification of parameters

A named parameter is an identifier that is prefixed by the ":" symbol. For example,

```
SELECT DISTINCT auc FROM Auction auc, IN(auc.bids) AS b  
WHERE b.amount > :amount
```

Note – You cannot mix the use of the positional parameter convention with the named parameter convention in the same QL query string.



Examining the SELECT Clause

The SELECT clause denotes the query result. More than one value can be returned from the SELECT clause of a query. In addition to the DISTINCT keyword, the SELECT clause can contain one or more of the following elements:

1. A single identification variable that ranges over an entity abstract schema type, for example:

```
SELECT o FROM Order o WHERE o.quantity > 500
```

2. A single-valued path expression, for example:

```
SELECT c.id, c.status FROM Customer c JOIN c.orders o WHERE o.count > 100
```

3. An aggregate select expression, for example:

```
SELECT AVG(o.quantity) FROM Order o
```

4. A constructor expression, for example:

```
SELECT NEW com.acme.example.CustomerDetails(c.id, c.status, o.count) FROM Customer c JOIN c.orders o WHERE o.count > 100
```

The optional DISTINCT keyword is used to specify that duplicate values must be eliminated from the query result. For example,

```
SELECT DISTINCT l.product FROM Order AS o, IN(o.lineItems) l
```

Examining the GROUP BY and HAVING Clauses

The GROUP BY construct enables the aggregation of values according to the properties of an entity class. The HAVING construct enables conditions to be specified that further restrict the query result as restrictions upon the groups. The GROUP BY and HAVING clauses are often used in combination. The following SELECT statements contain examples of the use of the GROUP BY and HAVING clauses.

```
SELECT c.country, COUNT(c) FROM Customer c GROUP BY c.country HAVING COUNT(c) > 3
```

```
SELECT c.status, avg(c.filledOrderCount), COUNT(c) FROM Customer c GROUP BY c.status HAVING c.status IN (1, 2)
```

The following FAQs provide a more in depth examination of the GROUP BY and HAVING clauses.

- What is the effect of having both a WHERE clause and a GROUP BY clause?

If a query contains both a WHERE clause and a GROUP BY clause, the effect is that of first applying the where clause, and then forming the groups and filtering them according to the HAVING clause.

- What requirement does the GROUP BY clause impose on the SELECT clause?

Any item that appears in the SELECT clause (other than as an argument to an aggregate function) must also appear in the GROUP BY clause.

- Is it possible to use the GROUP BY clause without the HAVING clause?

If there is no GROUP BY clause and the HAVING clause is used, the result is treated as a single group, and the select list can only consist of aggregate functions. The use of HAVING in the absence of GROUP BY is not required to be supported by an implementation of Java Persistence specification.

Examining the ORDER BY Clause

The ORDER BY clause allows the objects or values that are returned by the query to be ordered. The following FAQs provide a more in depth examination of the ORDER BY clause.

- What is the syntax of the ORDER BY clause?

```
ORDER BY orderby_item [ASC | DESC]{, orderby_item [ASC | DESC]}*
```

- What is the relationship between the returned item in the SELECT clause and the orderby item in the ORDER BY clause?

If the SELECT clause returns an identification variable, then each order by item must be an orderable state-field of the entity type returned by the SELECT clause. For example,

```
SELECT o FROM Customer c JOIN c.orders o JOIN c.address a WHERE a.state = 'CA' ORDER BY o.quantity, o.totalcost
```

If the SELECT clause returns a single valued path expression, then each orderby item must be an orderable state-field of the entity type returned by the SELECT clause. For example,

```
SELECT e.address FROM Employee e ORDER BY e.address.zipCode
```

If the SELECT clause returns a state field path expression then the orderby item must evaluate to the same state-field returned by the SELECT clause. For example,

```
SELECT o.quantity, a.zipcode FROM Customer c JOIN c.orders o JOIN c.address a WHERE a.state = 'CA' ORDER BY o.quantity, a.zipcode
```

- How is ascending or descending order specified?

Each orderby item can optionally be followed by the ASC keyword denoting ascending order or the DESC keyword denoting descending order. For example,

```
SELECT o FROM Customer c JOIN c.orders o JOIN c.address a WHERE a.state = 'CA' ORDER BY o.quantity DESC, o.totalcost ASC
```

The ascending order is the default.

- How are null values treated by the ORDER BY clause?

SQL rules for the ordering of null values apply: That is, all null values must appear before all non-null values in the ordering or all null values must appear after all non-null values in the ordering, but it is not specified which.

Declaring Query Strings: The BULK UPDATE Statement

The QL UPDATE statement updates one or more rows of a specific table in the database. The UPDATE statement consists of an UPDATE clause and an optional WHERE clause.

```
UPDATE_clause [WHERE_clause]
```

For example,

```
UPDATE customer c SET c.status = 'outstanding' WHERE c.balance < 10000
```

You can update multiple fields, for example,

```
UPDATE order o SET o.paid = TRUE, o.delivery = 'pending'
```

Update statements are restricted to apply on entity instances of a single entity class type. For example, you cannot create a single update statement that applies to both the Customer and Order entity class types.

An update statement is used when you need to update multiple entity instances. Hence the term bulk update statement.



Note – You can update a single entity instance programmatically, without using the update statement. You would need to either update the fields of a managed entity instance or update a detached entity instance and then pass it as an argument to the merge method of the entity manager.

Declaring Query Strings: The **DELETE** Statement

The QL **DELETE** statement deletes one or more rows from a specific table in the database. The **DELETE** statement consists of an **DELETE** clause and an optional **WHERE** clause.

DELETE_clause [WHERE_clause]

For example,

```
DELETE FROM Customer c WHERE c.status = 'inactive'
```

Delete statements are restricted to apply on entities of a single entity class type. For example, you cannot create a single delete statement that deletes rows from both the Customer and Order tables.

A delete statement is used when you need to delete multiple entity instances. Hence the term bulk delete statement.

Note – You can delete a single entity instance programmatically, without using the delete statement. You would need to pass the entity instance as an argument to the `remove` method of the entity manager.



Module 8

Developing Java EE Applications Using Messaging

Objectives

Upon completion of this module, you should be able to:

- Review JMS technology
- Describe the roles of the participants in the JMS API messaging system
- Create a queue message producer
- Create a synchronous message consumer
- Create an asynchronous message consumer
- List the capabilities and limitations of EJB components as messaging clients

Additional Resources



Additional resources – The following references provide additional information on the topics described in this module:

- Sun Microsystems, “The Java EE 5 Tutorial, Chapter 32.” [<http://java.sun.com/javaee/5/docs/tutorial/doc/>], accessed July 25, 2006.

Reviewing JMS Technology

This section provides an overview of JMS API technology. Figure 8-1 shows the relationship among the main participants of the JMS API messaging system.

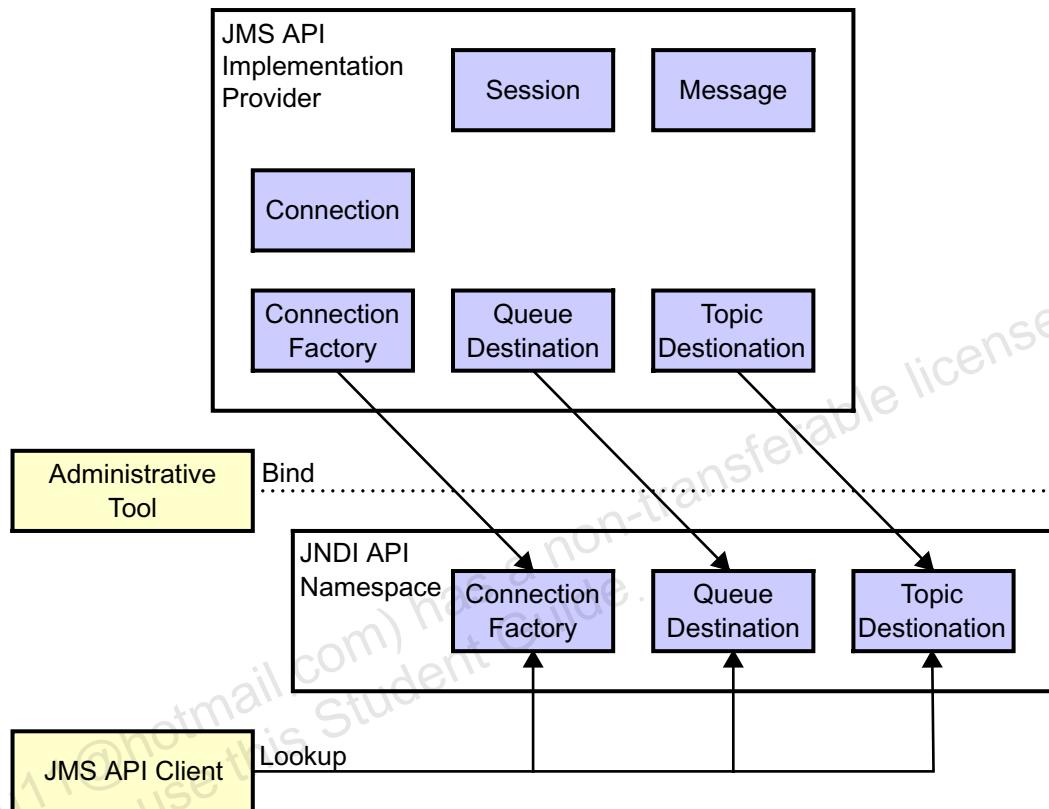


Figure 8-1 Messaging System Participants

The JMS API consists of a set of interfaces and classes. A JMS technology provider (JMS provider) is a messaging system that provides an implementation of the JMS API. For an application server to support JMS technology, you need to place the administered objects (connection factories, queue destinations, and topic destinations) in the JNDI technology namespace of the application server. You use the administrative tool supplied by the application server to perform this task.

Administered Objects

Administered objects are objects that are configured administratively (as opposed to programmatically) for each messaging application. A JMS provider supplies the administered objects. An application server administrator or application deployer configures these objects and places them in the JNDI technology namespace.

The two types of administered objects are destinations and connection factories.

Destinations are message distribution points. Destinations receive, hold, and distribute messages. Destinations fall into two categories: queue and topic destinations:

- Queue destinations implement the point-to-point messaging protocol.
- Topic destinations implement the publish/subscribe messaging protocol.

Connection factories are used by a JMS API client (JMS client) to create a connection to a JMS API destination (JMS destination). Connection factories fall into two categories: queue and topic factories.

Figure 8-2 shows both types of administered objects supplied as part of a JMS provider.

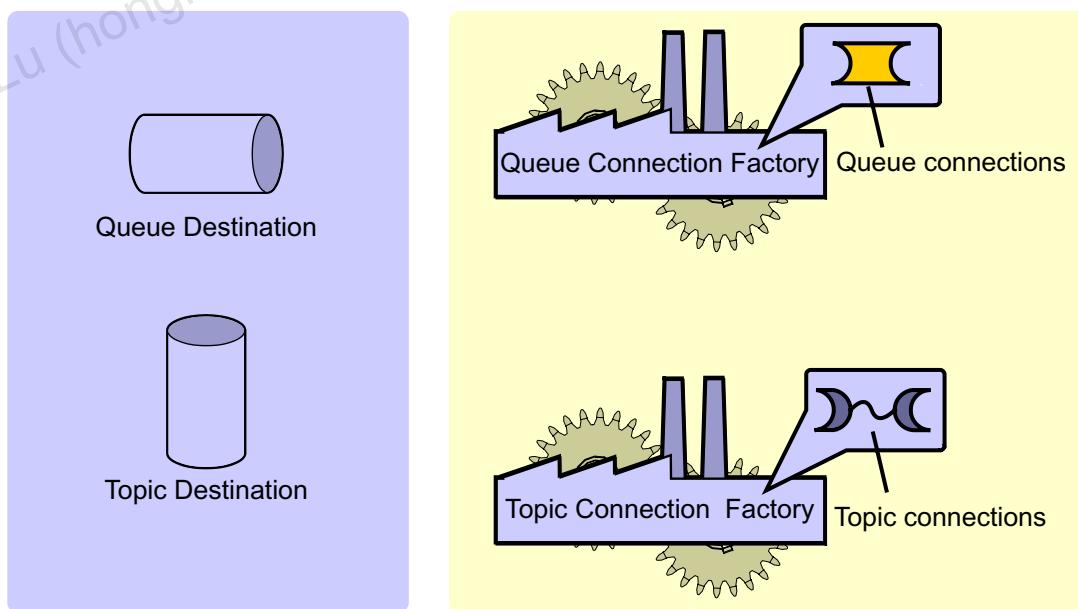


Figure 8-2 Administered Objects

Messaging Clients

Clients are applications that produce or consume messages. Message producer clients create and send messages to destinations.

Message consumer clients consume messages from destinations. Message consumers can be either asynchronous message consumers or synchronous message consumers.

Asynchronous consumer clients register with the message destination. When a destination receives a message, it notifies the asynchronous consumer, which then collects the message.

Synchronous message consumer clients collect messages from the destination. If the destination is empty, the client blocks until a message arrives.

Figure 8-3 shows the three types of messaging clients.

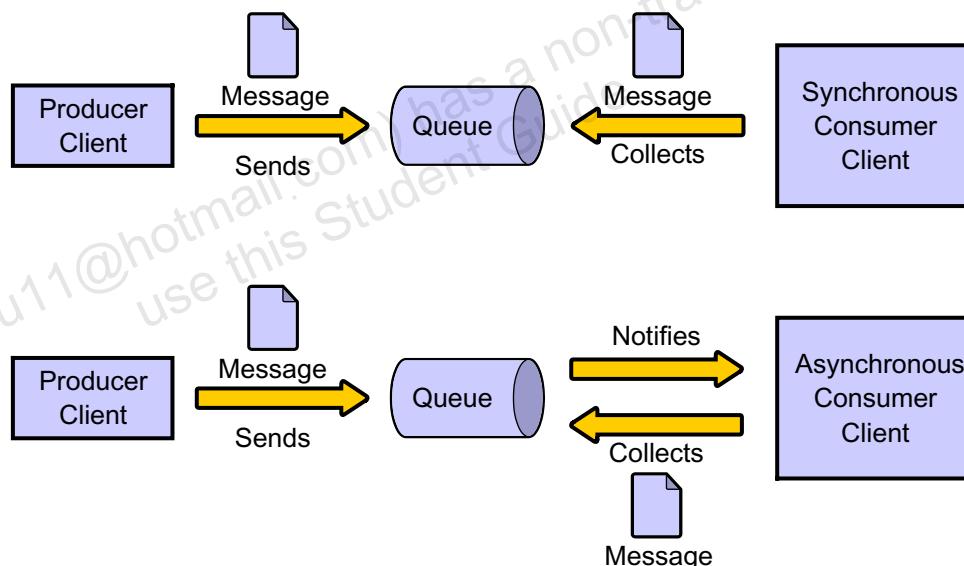


Figure 8-3 Messaging Clients

Messages

Messages are objects that encapsulate application information created by a message producer. Messages are sent to a destination that distributes them to message consumers.

Figure 8-4 shows the three parts of a JMS API message.

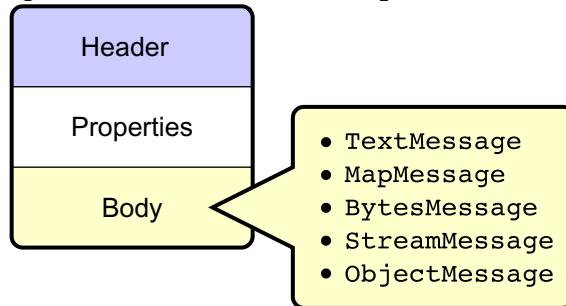


Figure 8-4 Message Construction and Types

The following three characteristics apply to messages:

- The header contains standard information for identifying and routing the message.
- Properties provide a mechanism for the classification of messages. Properties are name/value pairs. You can set or read the value of any property. You can customize the property list by adding a name/value pair. Properties allow destinations to filter messages based on property values. Properties also provide interoperability with some message service providers.
- The body contains the actual message in one of several standard formats.

Table 8-1 lists the JMS API message types.

Table 8-1 JMS Technology Message Types

Message Type	Contents of the Message Body
TextMessage	A <code>java.lang.String</code> object
MapMessage	A set of name-value pairs, for example, a hash table
BytesMessage	A stream of uninterpreted bytes or binary data
StreamMessage	A stream of Java technology primitive values filled and read sequentially
ObjectMessage	A serializable Java technology object

Point-to-Point Messaging Architecture

Figure 8-5 on page 8-8 illustrates the point-to-point messaging architecture, which is based on the concept of a message queue. A message queue retains all messages until they are consumed. The distinguishing factor between the point-to-point and publish/subscribe architectures is that in most cases, each point-to-point message queue has only one message consumer, and there are no timing dependencies between the sender and receiver. That is, a receiver gets all the messages that were sent to the queue, even those sent before the creation of the receiver. The queue then deletes messages on the acknowledgement of successful processing from the message consumer.



Note – A message queue can have multiple consumers. When one consumer consumes a message from the queue, the message is marked as consumed. Other consumers cannot also consume the same message.

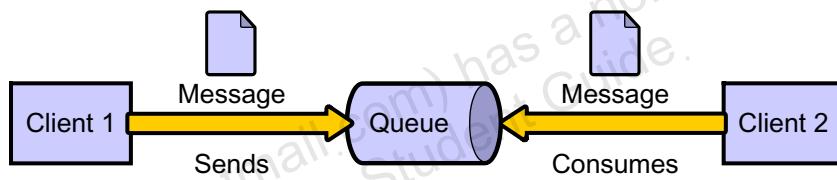


Figure 8-5 Point-to-Point Architecture



Publish/Subscribe Messaging Architecture

Figure 8-6 illustrates the publish/subscribe architecture, which is based on the concept of a message topic. A topic can have multiple consumers. Message producers publish messages to a topic. The topic retains messages until the messages are distributed to all consumers. There is a timing dependency between publishers and subscribers. That is, subscribers do not receive messages that were sent prior to their subscription to the topic or while the subscriber is inactive.

Note – The JMS API supports the concept of a durable subscriber. A topic retains all messages for subsequent delivery to inactive durable subscribers.

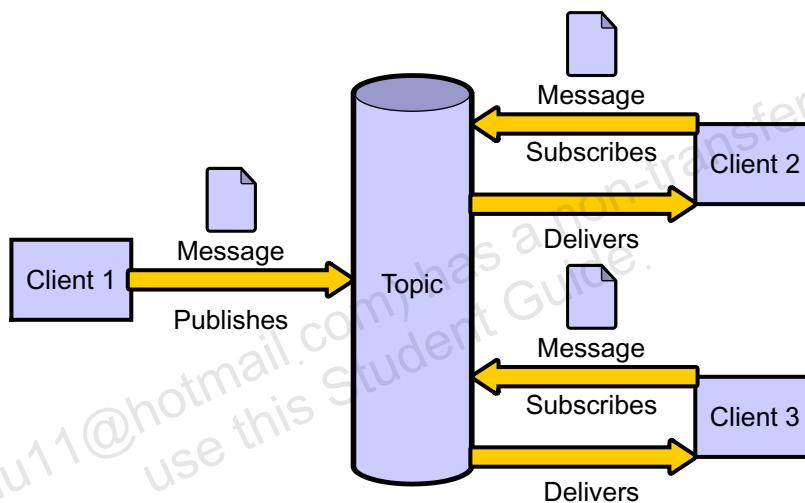


Figure 8-6 Publish/Subscribe Architecture

Creating a Queue Message Producer

Creating a Queue Message Producer

Figure 8-7 illustrates the steps required to create a queue message producer.

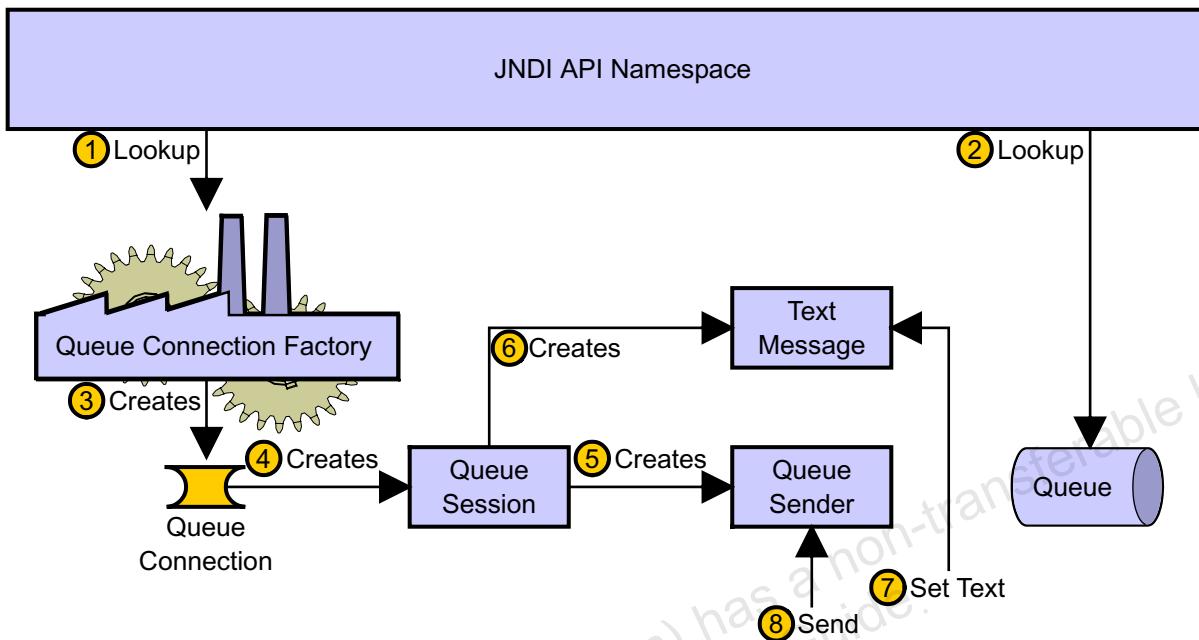


Figure 8-7 Creating a Queue Message Producer

The following steps explain how you can create a message queue producer.

1. Obtain a connection factory using injection or the JNDI API.
A connection factory is an administered object. An instance of the `QueueConnectionFactory` class is required. You use it to create the `Connection` object in Step 3.
2. Obtain the message queue using injection or the JNDI API.
A message queue is an administered object. A message queue implements the point-to-point message protocol destination.

3. Create a Connection object using the connection factory.
A connection encapsulates a virtual connection with a JMS API provider enabling messaging clients to send messages and to receive messages from destinations.
4. Create a Session object using the connection.
A Session object provides a single-threaded context for producing and consuming messages.
5. Create a QueueSender object using the Session object.
A QueueSender object provides the API to send messages to a queue destination.
6. Create one or more Message objects using the Session object.
Use the session object to create the required type of message object. After creation, populate the message with the required data.
7. Populate the message with text using the setText method of the TextMessage object.
8. Send one or more Message objects using the QueueSender object.
Use the send method of the QueueSender to send the messages to the queue destinations.

Queue Message Producer Code Example

Code 8-1 shows the queue message producer code.

Code 8-1 Example Code for Queue Message Producer

```
1 import javax.jms.*;
2 import javax.naming.*;
3
4 public class SimpleQSender {
5     public static void main(String[] args) {
6         String queueName = null;
7         Context jndiContext = null;
8         QueueConnectionFactory queueConnectionFactory = null;
9         QueueConnection queueConnection = null;
10        QueueSession queueSession = null;
11        Queue queue = null;
12        QueueSender queueSender = null;
13        TextMessage message = null;
14        int NUM_MSGS = 5;
```

Creating a Queue Message Producer

```

15
16     // Read queue name from command line and display it.
17     if (args.length != 1) {
18         System.out.println("Usage: java SimpleQSender " +
19             "<queue-name> ");
20         System.exit(1);
21     }
22     queueName = new String(args[0]);
23     System.out.println("Queue name is " + queueName);
24
25     // Create a JNDI API InitialContext object if none exists yet
26     try {
27         jndiContext = new InitialContext();
28     } catch (NamingException e) {
29         System.out.println("Could not create JNDI API " +
30             "context: " + e.getMessage());
31         System.exit(1);
32     }
33
34     // Look up connection factory and queue.  If either does
35     try {
36         queueConnectionFactory = (QueueConnectionFactory)
37             jndiContext.lookup("QueueConnectionFactory");
38         queue = (Queue) jndiContext.lookup(queueName);
39     } catch (NamingException e) {
40         System.out.println("JNDI API lookup failed: " +
41             e.getMessage());
42         System.exit(1);
43     }
44
45     /*
46      * Create connection.
47      * Create session from connection; false means session is
48      * not transacted.
49      * Create sender and text message.
50      * Send messages, varying text slightly.
51      * Send end-of-messages message.
52      * Finally, close connection.
53      */
54     try {
55         queueConnection =
56             queueConnectionFactory.createQueueConnection();
57         queueSession =
58             queueConnection.createQueueSession(false,
59                 Session.AUTO_ACKNOWLEDGE);
60         queueSender = queueSession.createSender(queue);

```

```
61     message = queueSession.createTextMessage();
62     for (int i = 0; i < NUM_MSGS; i++) {
63         message.setText("This is message " + (i + 1));
64         System.out.println("Sending message: " +
65             message.getText());
66         queueSender.send(message);
67     }
68
69     // Send a non-text control message indicating end of
70     // messages.
71     queueSender.send(queueSession.createMessage());
72 } catch (JMSEException e) {
73     System.out.println("Exception occurred: " + e.getMessage());
74 } finally {
75     if (queueConnection != null) {
76         try {
77             Thread.sleep(60000); // delay before closing
78             queueConnection.close();
79         } catch (JMSEException e) {}
80     }
81 }
82 }
83 }
84 }
```

A session bean or message-driven bean can use resource injection services of the container to obtain references to administered objects such as a QueueConnectionFactory or a Queue or a Topic. For example,

```
@Resource(name="jms/QueueConnectionFactory")
javax.jms.QueueConnectionFactory queueConnectionFactory;
@Resource(name="jms/someQueue")
javax.jms.Queue queue;
```

Creating a Synchronous Message Consumer

Creating a Synchronous Message Consumer

Figure 8-8 illustrates the steps required to create a synchronous queue consumer.

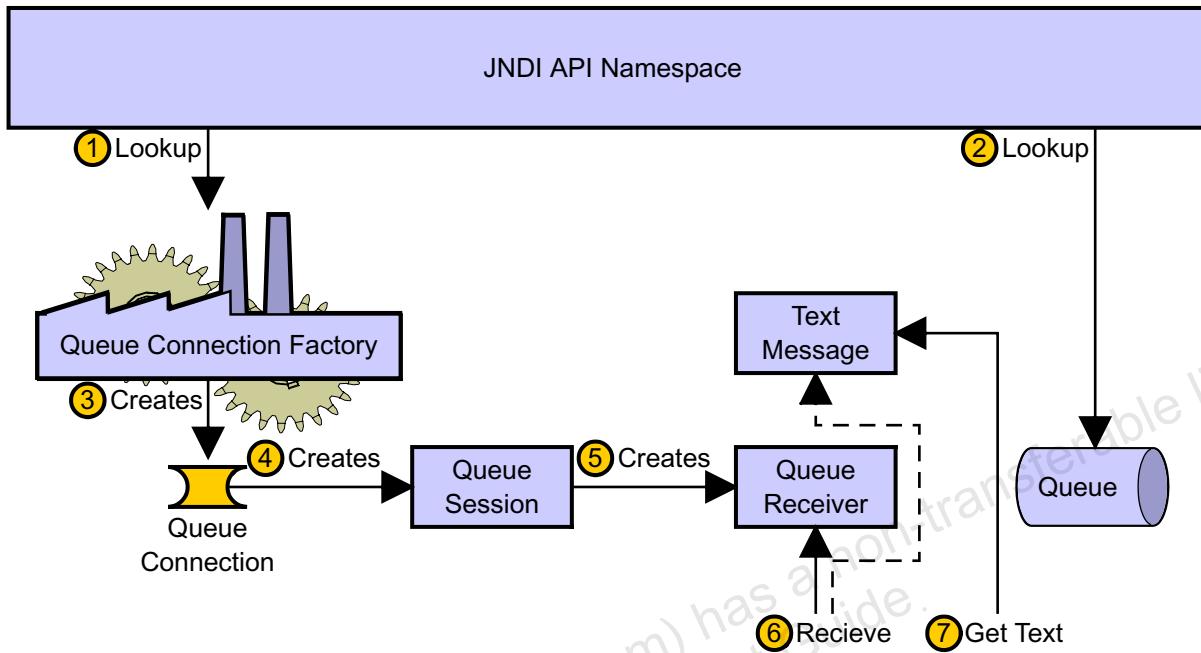


Figure 8-8 Synchronous Queue Consumer

To create a synchronous queue consumer:

1. Obtain a connection factory using injection or the JNDI API.
A connection factory is an administered object. You need an instance of the `QueueConnectionFactory` class, which you use to create the `Connection` object in Step 3.
2. Obtain the message queue using injection or the JNDI API.
A message queue is an administered object. It implements the point-to-point message protocol destination.

3. Create a Connection object using the factory.

A connection encapsulates a virtual connection with a JMS API provider enabling messaging clients to send messages and to receive messages from destinations.

4. Create a Session object.

A Session object provides a single-threaded context for producing and consuming messages.

5. Create a QueueReceiver object.

A QueueReceiver object provides the API to receive messages from a queue destination.

6. Call the receive method of the QueueReceiver object.

The receive method is blocked if the queue is empty.

7. Process the message returned by the receive method.

Use the getText method of the TextMessage object.

Synchronous Queue Consumer Code Example

Code 8-2 shows the code for a synchronous queue consumer.

Code 8-2 Example Code for Synchronous Queue Consumer

```
1 import javax.jms.*;
2 import javax.naming.*;
3
4 public class SimpleQReceiver {
5     public static void main(String[] args) {
6         String queueName = null;
7         Context jndiContext = null;
8         QueueConnectionFactory queueConnectionFactory = null;
9         QueueConnection queueConnection = null;
10        QueueSession queueSession = null;
11        Queue queue = null;
12        QueueReceiver queueReceiver = null;
13        TextMessage message = null;
14
15        // Read queue name from command line and display it.
16        if (args.length != 1) {
17            System.out.println("Usage: java " +
18                "SimpleQueueReceiver <queue-name>");
```

Creating a Synchronous Message Consumer

```

19         System.exit(1);
20     }
21     queueName = new String(args[0]);
22     System.out.println("Queue name is " + queueName);
23
24     // Create a JNDI API InitialContext object if none exists yet
25     try {
26         jndiContext = new InitialContext();
27     } catch (NamingException e) {
28         System.out.println("Could not create JNDI API " +
29             "context: " + e.getMessage());
30         System.exit(1);
31     }
32
33     // Look up connection factory and queue.  If either does
34     try {
35         queueConnectionFactory = (QueueConnectionFactory)
36             jndiContext.lookup("QueueConnectionFactory");
37         queue = (Queue) jndiContext.lookup(queueName);
38     } catch (NamingException e) {
39         System.out.println("JNDI API lookup failed: " +
40             e.getMessage());
41         System.exit(1);
42     }
43
44     /*
45      * Create connection.
46      * Create session from connection; false means session is
47      * not transacted.
48      * Create receiver, then start message delivery.
49      * Receive all text messages from queue until
50      * a non-text message is received indicating end of
51      * message stream.
52      * Close connection.
53      */
54     try {
55         queueConnection =
56             queueConnectionFactory.createQueueConnection();
57         queueSession =
58             queueConnection.createQueueSession(false,
59             Session.AUTO_ACKNOWLEDGE);
60         queueReceiver = queueSession.createReceiver(queue);
61         queueConnection.start();
62         while (true) {
63             Message m = queueReceiver.receive(1);
64             if (m != null) {

```

```
65         if (m instanceof TextMessage) {
66             message = (TextMessage) m;
67             System.out.println("Reading message: " +
68                 message.getText());
69         } else {
70             break;
71         }
72     }
73 }
74 } catch (JMSEException e) {
75     System.out.println("Exception occurred: " + e.getMessage());
76 } finally {
77     if (queueConnection != null) {
78         try {
79             queueConnection.close();
80         } catch (JMSEException e) {}
81     }
82 }
83 }
84 }
85 }
```

Creating an Asynchronous Queue Consumer

The following steps describe the process that you use to create an asynchronous queue consumer.

1. Look up a connection factory using the JNDI API.

A connection factory is an administered object. You need an instance of the `QueueConnectionFactory` class, which you use to create the `Connection` object in Step 3.

2. Look up the message queue using the JNDI API.

A message queue is an administered object. The message queue implements the point-to-point message protocol destination.

3. Create a `Connection` object using the factory.

A connection encapsulates a virtual connection with a JMS API provider enabling messaging clients to send and receive messages to and from destinations.

4. Create a `Session` object.

A `Session` object provides a single-threaded context for producing and consuming messages.

5. Create a `QueueReceiver` object.

A `QueueReceiver` object enables the API to register a `MessageListener` object with the queue destination.

6. Call the `setMessageListener` method of the `QueueReceiver` object passing it an instance of a `MessageListener` interface.

Create a class that implements the `MessageListener` interface. This interface has a single method called `onMessage`. Create an instance of your class that implements the `MessageListener` interface. Use this instance as the input parameter to the `setMessageListener` method.

7. Process the received message in the `onMessage` method of the `MessageListener` instance.

The queue destination calls the `onMessage` method whenever it receives a message.

Asynchronous Queue Consumer Code Example

Code 8-3 shows the code for an asynchronous queue consumer.

Code 8-3 Example Code for Asynchronous Queue Consumer

```
1 import javax.jms.*;
2 import javax.naming.*;
3
4 public class SimpleMessageListener implements MessageListener {
5     public void onMessage(Message message) {
6         TextMessage msg = null;
7         try {
8             if (message instanceof TextMessage) {
9                 msg = (TextMessage) message;
10                System.out.println("Reading message: " +
11                    msg.getText());
12            } else {
13                System.out.println("Message of wrong type: " +
14                    message.getClass().getName());
15            }
16        } catch (Exception e) {
17            System.out.println("Exception in onMessage(): " +
18                e.getMessage());
19        }
20    }
21
22    public static void main(String[] args) {
23        String queueName = null;
24        Context jndiContext = null;
25        QueueConnectionFactory queueConnectionFactory = null;
26        QueueConnection queueConnection = null;
27        QueueSession queueSession = null;
28        Queue queue = null;
29        QueueReceiver queueReceiver = null;
30        MessageListener listener = null;
31
32        listener = new SimpleMessageListener();
33        // Read queue name from command line and display it.
34        if (args.length != 1) {
35            System.out.println("Usage: java " +
36                "SimpleQueueReceiver <queue-name>");
37            System.exit(1);
38        }
39        queueName = new String(args[0]);
```

Creating an Asynchronous Queue Consumer

```

40     System.out.println("Queue name is " + queueName);
41
42     // Create a JNDI API InitialContext object
43     try {
44         jndiContext = new InitialContext();
45     } catch (NamingException e) {
46         System.out.println("Could not create JNDI API " +
47             "context: " + e.getMessage());
48         System.exit(1);
49     }
50
51     // Look up connection factory and queue
52     try {
53         queueConnectionFactory = (QueueConnectionFactory)
54             jndiContext.lookup("QueueConnectionFactory");
55         queue = (Queue) jndiContext.lookup(queueName);
56     } catch (NamingException e) {
57         System.out.println("JNDI API lookup failed: " +
58             e.getMessage());
59         System.exit(1);
60     }
61
62     /*
63      * Create connection.
64      * Create session from connection; false means session is
65      * not transacted.
66      * Create receiver, then start message delivery.
67      * Receive all text messages from queue until
68      * a non-text message is received indicating end of
69      * message stream.
70      * Close connection.
71      */
72     try {
73         queueConnection =
74             queueConnectionFactory.createQueueConnection();
75         queueSession =
76             queueConnection.createQueueSession(false,
77             Session.AUTO_ACKNOWLEDGE);
78         queueReceiver = queueSession.createReceiver(queue);
79         queueReceiver.setMessageListener(listener);
80         queueConnection.start();
81     } catch (JMSEException e) {
82         System.out.println("Exception occurred: " + e.getMessage());
83     } finally {
84         if (queueConnection != null) {
85             try {

```

```
86         queueConnection.close();
87     } catch (JMSEException e) {}
88 }
89 }
90 }
91 }
92 }
```

Evaluating the Capabilities and Limitations of EJB Components as Messaging Clients

Table 8-2 shows the messaging capabilities of enterprise beans.

Table 8-2 Messaging Capabilities of Enterprise Beans

Enterprise Beans	Producer	Synchronous Consumer	Asynchronous Consumer
Session bean	Yes	Not recommended	Not possible
Message-driven bean	Yes	Not recommended	Yes

Using EJB Components as Message Producers

Entity, session, and message-driven beans are capable of becoming message producers. To add this capability, you include code to invoke the required methods of the JMS API.

Using EJB Components as Message Consumers

Synchronous message consumers block and tie up server resources. For this reason, you should not use entity, session, and message-driven beans as synchronous message consumers. To add this capability, you include code to invoke the required methods of the JMS API.

Message-driven beans implement the `javax.jms.MessageListener` interface. Consequently, message-driven beans consume messages asynchronously.

Module 9

Developing Message-Driven Beans

Objectives

Upon completion of this module, you should be able to:

- Describe the properties and life cycle of message-driven beans
- Create a JMS message-driven bean
- Create life-cycle event handlers for a JMS message-driven bean
- Create a non-JMS message-driven bean

Additional Resources



Additional resources – The following references provide additional information on the topics described in this module:

- Sun Microsystems, “JSR 220: Enterprise JavaBeans™, Version 3.0 EJB 3.0 Simplified API.” [<https://sdlc3e.sun.com/ECom/EComActionServlet;jsessionid=CEAAE57A3BAB8A76D4555E3C5A1F4031>], accessed July 25, 2006.
- Sun Microsystems, “The Java EE 5 Tutorial, Chapter 28.” [<http://java.sun.com/javaee/5/docs/tutorial/doc/>], accessed July 25, 2006.

Introducing Message-Driven Beans

Message-driven beans are designed to function as an asynchronous message consumers. Message-driven beans implement a message listener interface. For example, JMS message-driven beans implement the JMS API `MessageListener` interface. The bean designer is responsible for the message listener method code (the `onMessage` method for JMS message-driven beans).

The deployer is required to supply the information for the container to register the message-driven bean as a message listener with the destination.

Java EE Technology Client View of Message-Driven Beans

Figure 9-1 shows the Java EE technology client view of a message-driven bean.

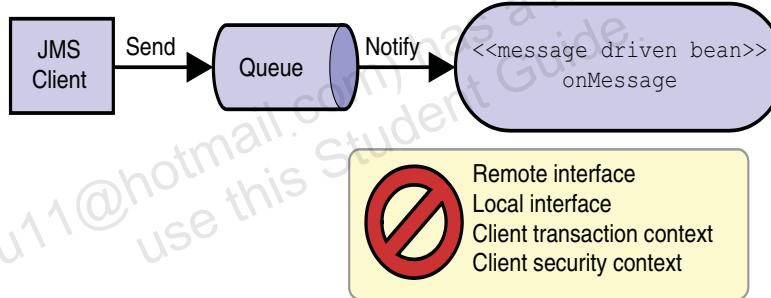


Figure 9-1 Java EE Technology Client View of a Message-Driven Bean

Message-driven beans do not have remote component, or local component interfaces.

Message-driven beans do not provide Java EE components with a direct interface. Java EE technology clients communicate with a bean by sending a message to the destination (queue or topic) for which the bean is the `MessageListener` object.

Message-driven beans have no client-visible identity. They hold no client conversational state. They are anonymous to clients.

Introducing Message-Driven Beans

Message-driven beans are transaction aware. The client's transaction and security contexts are unavailable to a message-driven bean. Consequently, message-driven beans cannot execute in the client's transaction context or security context.

Containers can create and pool multiple instances of message-driven beans to service the same message destination. The operation of these instances are independent of each other. That is, when a message arrives at a message destination, the container picks the next available instance associated with that destination to service the message.

Life Cycle of a Message-Driven Bean

Figure 9-2 shows the life cycle of a message-driven bean.

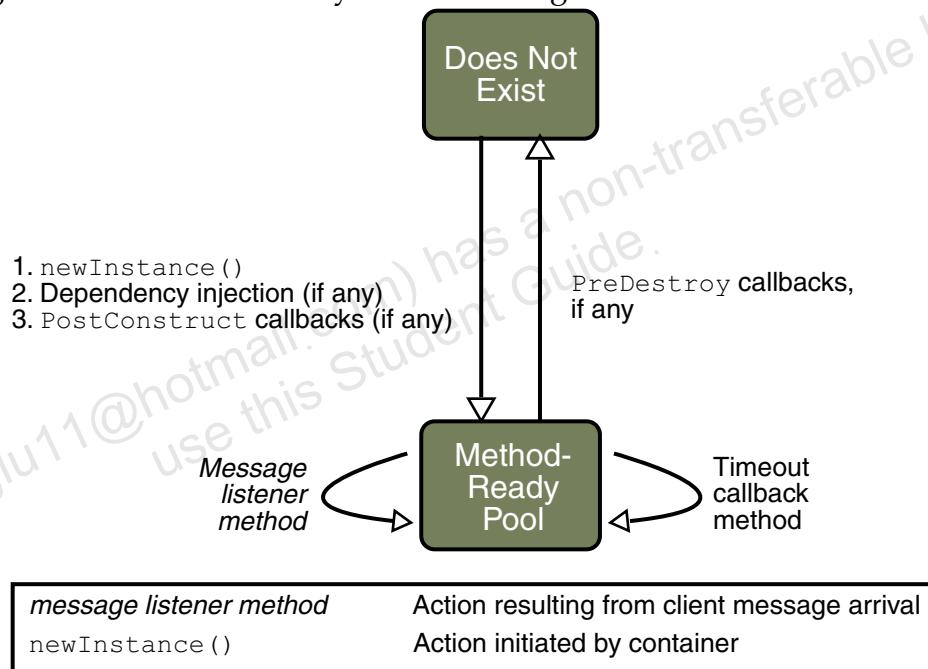


Figure 9-2 Message-Driven Bean Life Cycle

The container creates a message-driven bean by invoking the `newInstance` method. After the bean instance is created, the container, if applicable injects the bean's `MessageDrivenContext` instance, and performs any other dependency injection as specified by metadata annotations on the bean class or by the deployment descriptor. The container then calls the bean's `PostConstruct` callback method, if any. After which the container adds the bean to the method-ready pool. The bean is then ready to service the message notifications presented to it through the message listener method.

Types of Message-Driven Beans

As of the EJB 2.1 specification, EJB containers support the following two types of message-driven beans. The two types are distinguished by the message listener interface type they implement:

- JMS message-driven beans

JMS message-driven beans directly or indirectly implement the `javax.jms.MessageListener` interface. In addition JMS message-driven beans can optionally directly or indirectly implement the `javax.ejb.MessageDrivenBean` interface.

An application server can integrate the support for JMS message-driven beans directly into the container or can use a supporting resource adapter based on the Java EE connector architecture.

- Non JMS message-driven beans

Non JMS message-driven beans implement a message listener interface specific to the messaging service type that they support. In addition non JMS message-driven beans can optionally directly or indirectly implement the `javax.ejb.MessageDriven` interface.

Non JMS message-driven beans depend on a Java EE connector based resource adapter based on the specific messaging service type.

Creating a JMS Message-Driven Bean: Essential Tasks

To create a message-driven bean class, you perform the following steps:

1. Declare the message-driven bean class.
The class must be public, but not final or abstract.
2. Annotate the message-driven bean class using the `MessageDriven` metadata annotation.

The `MessageDriven` metadata annotation contain a number of attributes. The following list discusses the most important of these attributes.

- Use the `messageListenerInterface` attribute to specify the indirectly implemented message listener interface to the string "`javax.jms.MessageListener`".
As an alternative, you can include the `MessageListener` interface in the message-driven beans `implements` clause.
 - Use the `mappedName` attribute to specify the name of the message destination associated with the message-driven bean.
3. Optionally, include in the message-driven bean class the use of resource injection to obtain the `MessageDrivenContext` instance associated with the bean.
 4. Include, in the message-driven bean class, a public no-argument constructor.
Create or acquire any resources the `onMessage` method might require.
 5. Write the `onMessage` method of the `MessageListener` interface.
Supply the code to process the received message. This code might include:
 - Accessing entity instances and session beans executing in the same JVM machine using their local interfaces
 - Accessing session beans executing in another JVM machine using their remote interfaces
 - Sending messages to a message destination
 6. Do not define a `finalize` method.

Code 9-1 shows a simple example of a message-driven bean class.

Code 9-1 Message-Driven Bean Class Simple Example

```

1 import javax.ejb.*;
2 import javax.jms.*;
3
4 @MessageDriven(mappedName="MDBQueue")
5 public class MDB implements MessageListener {
6
7     public void onMessage(Message msg) {
8         System.out.println("Got message! ");
9     }
10 }
```

Code 9-2 shows how you can create a more complex message-driven bean class that uses information in the message header to selectively process messages.

Code 9-2 Message-Driven Bean Class Complex Example

```

1 // Copyright 2002 Sun Microsystems, Inc. All Rights Reserved.
2 import javax.ejb.*;
3 import javax.naming.*;
4 import javax.jms.*;
5
6 /**
7 * The MessageBean class is a JMS message-driven bean. It
8 * implements javax.jms.MessageListener interface.
9 * It is defined as public (but not final or abstract).
10 * It defines a constructor and the onMessage method.
11 */
12 @MessageDriven {
13     mappedName="destinationType",
14     messageListenerInterface = javax.jms.MessageListener,
15     activationConfig = {
16         @ActivationConfigProperty(propertyName="messageSelector",
17             propertyValue="RECIPIENT = 'MDB' ")
18     }
19 }
20 public class MessageBean {
21     @Resource private MessageDrivenContext mdc = null;
22
23     // Constructor, which is public and takes no arguments.
24     public MessageBean() {
```

Creating a JMS Message-Driven Bean: Essential Tasks

```
25     System.out.println("In MessageBean.MessageBean()");  
26 }  
27  
28 // onMessage method, public not final or static  
29 // Casts the incoming Message to a TextMessage and displays  
30 // the text.  
31  
32 public void onMessage(Message inMessage) {  
33     TextMessage msg = null;  
34     try {  
35         if (inMessage instanceof TextMessage) {  
36             msg = (TextMessage) inMessage;  
37             System.out.println("MESSAGE BEAN: Message " +  
38                 "received: " + msg.getText());  
39         } else {  
40             System.out.println("Message of wrong type: " +  
41                 inMessage.getClass().getName());  
42         }  
43     } catch (JMSEException e) {  
44         System.err.println("MessageBean.onMessage: " +  
45             "JMSEException: " + e);  
46         mdc.setRollbackOnly();  
47     } catch (Throwable te) {  
48         System.err.println("MessageBean.onMessage: " +  
49             "Exception: " + te);  
50     }  
51 }  
52 }  
53 }
```

The following notes provide additional information with regard to creating message-driven beans to handle specific situations.

- Using message header message selectors

You can declare the JMS message selector to be used in determining which messages a JMS message-driven bean is to receive. The message selector to be used is specified using the activation configuration property `messageSelector`. For example:

```
@MessageDriven(
    activationConfig={
        @ActivationConfigProperty(
            propertyName="messageSelector",
            propertyValue="JMSType = 'car' AND color = 'blue' and weight >2500"
        )
    }
)
```

- Specifying message acknowledgement

Specifying message acknowledgement depends on the transaction demarcation mode specified for the message-driven bean. For CMT, the container is responsible for message acknowledgement. For BMT, you can specify message acknowledgement using the activation configuration property `acknowledgeMode`. The value of the `acknowledgeMode` property must be either `Auto-acknowledge` or `Dups-ok-acknowledge` for a JMS message-driven bean. For example:

```
@MessageDriven(
    activationConfig={
        @ActivationConfigProperty(
            propertyName="acknowledgeMode",
            propertyValue="Dups-ok-acknowledge"
        )
    }
)
```

If the `acknowledgeMode` property is not specified, JMS `AUTO_ACKNOWLEDGE` semantics are assumed.

Creating a JMS Message-Driven Bean: Adding Life-Cycle Event Handlers

With EJB 3.0, you are not required to provide life-cycle event handlers for message-driven beans. However message-driven beans do generate life-cycle events and you can optionally provide event handlers for these methods.

Message-driven beans generate the following life-cycle events:

- PostConstruct callback

The PostConstruct callback occurs after the construction of the bean instance but before the first message listener method invocation on the bean. This is at a point after which any dependency injection has been performed by the container.

- PreDestroy callback

The PreDestroy callback occurs at the time the bean instance is removed from the pool or destroyed.

You can define the event handler in the bean class.

Defining a Callback Handler in the Bean Class

To define a callback handler in a message-driven bean class, you perform the following steps:

1. Declare, in the bean class, a callback method with the following signature:
`anyAccessModifier void methodName()`
2. The callback method must conform to the following rules:
 - It can have any type of access modifier (public, default, protected or private).
 - It must not throw an application exception.
 - It can throw a runtime exception.
 - Must not use dependency injections.
 - Must not rely on a specific transaction or security context.
3. Annotate the method with the appropriate life-cycle event handler metadata annotation. For a post -construct method use the PostCosntruct metadata annotation. For a pre-destroy method use the PreDestroy metadata annotation.

Code 9-3 shows an example of life-cycle event handler methods `obtainResources` and `releaseResources` defined within a bean class.

Code 9-3 Example of a Callback Method in a Bean Class

```
1  @MessageDriven { // ... }
2  public class MessageBean {
3      @Resource private MessageDrivenContext mdc;
4
5      public MessageBean() { } // constructor
6
7      @PostConstruct void obtainResources() { }
8
9      @PreDestory void releaseResources() { }
10
11     public void onMessage(Message inMessage) { }
12 }
```

Creating a Non-JMS Message-Driven Bean

To create a non-JMS message-driven bean class, you perform the following steps:

1. Declare the message-driven bean class:

The class must be public, but not final or abstract.

2. Annotate the message-driven bean class using the `MessageDriven` metadata annotation.

The `MessageDriven` metadata annotation contain a number of attributes. The following list discusses the most important of these attributes.

- Use the `messageListenerInterface` attribute to specify the indirectly implemented message listener interface. This step defines the message-driven bean type.
 - Use the activation configuration properties to provide deployment information. The actual property names and values used would depend on the (non-JMS) messaging service and the associated connector. The connector is a resource adapter (plugin software installed in the application server) that the container uses to support the non-JMS messaging service.
3. Optionally, include in the message-driven bean class the use of resource injection to obtain the `MessageDrivenContext` instance associated with the bean.
 4. Include, in the message-driven bean class, a public no-argument constructor.
Create or acquire any resources the message listener method might require.
 5. Implement all the methods of the non-JMS message listener interface.
 6. Do not define a `finalize` method.

Module 10

Implementing Interceptor Classes and Methods

Objectives

Upon completion of this module, you should be able to:

- Describe interceptors and interceptor classes
- Create a business interceptor method in the enterprise bean class
- Create an interceptor class
- Associate multiple business interceptor methods with an enterprise bean
- Include life-cycle callback interceptor methods in an interceptor class
- Create entity life-cycle callback methods

Additional Resources



Additional resources – The following reference provides additional information on the topics described in this module:

- Sun Microsystems, “JSR 220: Enterprise JavaBeans™, Version 3.0 EJB Core Contracts and Requirements, Chapter 4 and 12.” [<https://sdlc3e.sun.com/ECom/EComActionServlet;jsessionid=CEAAE57A3BAB8A76D4555E3C5A1F4031>], accessed July 25, 2006.
- Sun Microsystems, “JSR 220: Enterprise JavaBeans™, Version 3.0 EJB 3.0 Simplified API.” [<https://sdlc3e.sun.com/ECom/EComActionServlet;jsessionid=CEAAE57A3BAB8A76D4555E3C5A1F4031>], accessed July 25, 2006.

Introducing Interceptors and Interceptor Classes

The following FAQ list highlights the information you need to be aware of regarding interceptors and interceptor classes.

- What is an interceptor?

An interceptor is a class that works in conjunction with a bean class to interpose on business method invocations and receive life-cycle callback notifications. Interceptors are classified as:

- Business interceptor methods

A business interceptor method intercepts the invocation of a business method of a session bean or listener method of a message-driven bean.

Business interceptor methods are sometimes referred to as `AroundInvoke` methods.

- Life-cycle callback interceptor methods

A life-cycle callback interceptor method intercepts life-cycle events generated by session bean and message-driven bean instances. For example, the `PostConstruct` event generated by the creation of a session bean instance.

- What types of enterprise beans support interceptors?

Interceptors and interceptor classes are supported by the following enterprise bean types:

- Stateful and stateless session beans
- Message-driven beans

- What is the signature of an interceptor method?

The following is the method signature of a business interceptor method:

```
@AroundInvoke  
public Object anyMethodName(InvocationContext ic) throws Exception
```

The following is the method signature of a life-cycle callback interceptor method:

```
@anyCallbackAnnotation  
anyAccessModifier void anyMethodName(InvocationContext ic)
```

Both types of interceptor methods are passed an `InvocationContext` object.

Introducing Interceptors and Interceptor Classes

The API for the `InvocationContext` object is:

```

1  public interface InvocationContext {
2      public Object getTarget();
3      public Method getMethod();
4      public Object[] getParameters();
5      public void setParameters(Object[] params);
6      public Map<String, Object> getContextData();
7      public Object proceed() throws Exception;
8  }
```

- Can a business interceptor method be associated with a specific business method?

An interceptor method can be associated with:

- A bean class

If associated with a bean class, the same interceptor method is invoked, regardless of which business method of the bean is invoked. However, you can include code in an interceptor method to identify (from the `InvocationContext` argument passed to it) the business method it is intercepting and thus change its functionality accordingly.

- A specific business method of the bean class

This functionality allows you to create custom interceptor methods targeted at specific business methods.

- Where is an interceptor method defined?

An interceptor method (business or life-cycle) can be defined in either of the following locations:

- The bean class
- An interceptor class associated with the bean
- Is it possible to define multiple interceptor methods for an enterprise bean?

You can define multiple interceptor methods for an enterprise bean. This is a multi-step process.

- Each interceptor method must be defined in its own interceptor class
- Optionally, define an interceptor method in the bean class
- Use the `Interceptors` metadata annotation to associate the interceptor classes with bean class.

- When multiple interceptors are defined, what is the order of their invocation?

The interceptors are invoked in the order in which they are defined in the `Interceptors` annotation after which the interceptor method (if any) defined in the bean class is invoked.

Creating a Business Interceptor Method in the Enterprise Bean Class

The rules for business interceptor methods included in a bean class are identical to business interceptor methods included in an interceptor class. To add an interceptor method to an enterprise bean, you are required to perform the following tasks:

1. Declare an interceptor method in the bean class that conforms to the following signature.

```
@AroundInvoke
public Object anyMethodName(InvocationContext ic) throws Exception
```

2. Ensure the return from the interceptor method invokes the InvocationContext objects proceed method. For example,

```
return ic.proceed();
```

This ensures the invoking of the original target business method (or the next interceptor).

Code 10-1 shows a stateless session bean AccountManagementBean, which contains an interceptor method called profile.

Code 10-1 Example of Interceptor in the Bean Class

```
1  @Stateless
2  public class AccountManagementBean implements AccountManagement {
3      public void createAccount(int accountNumber,
4          AccountDetails details) { ... }
5      public void deleteAccount(int accountNumber) { ... }
6      public void activateAccount(int accountNumber) { ... }
7      public void deactivateAccount(int accountNumber) { ... }
8
9      @AroundInvoke public Object profile(InvocationContext inv)
10     throws Exception {
11         long time = System.currentTimeMillis();
12         try {
13             return inv.proceed(); }
14         finally {
15             long diffTime = time - System.currentTimeMillis();
16             System.out.println(inv.getMethod() + " took " + diffTime + "
17                 milliseconds.");
18         }
19     }
```

When creating interceptor methods, you should be aware of the following:

- A compilation unit (enterprise bean class or interceptor class) can contain only one interceptor method.
- Business interceptor methods can throw runtime exceptions or application exceptions that are allowed in the throws clause of the business method.
- Business interceptor method invocations occur within the same transaction and security context as the business method for which they are invoked.
- A business interceptor method cannot be static or final.
- A business interceptor method is required to invoke the proceed method of the InvocationContext object that it receives as an input parameter.

Creating an Interceptor Class

Creating an Interceptor Class

To create an Interceptor class and associate it with an enterprise bean, you are required to perform the following tasks:

1. Declare a Java technology class
2. Include a public no-arg constructor.
3. Declare an interceptor method in the Java technology class that conforms to the following signature.

```
@AroundInvoke
public Object anyMethodName(InvocationContext ic) throws Exception
```

4. Ensure that the interceptor method invokes the InvocationContext objects proceed method. For example,

```
return ic.proceed();
```

This ensures the invoking of the original target business method (or the next interceptor method in the chain).

5. Annotate the enterprise bean with the Interceptors annotation. Include the name of the interceptor class as an attribute of the Interceptors annotation.

Code 10-2 shows a stateless session bean AccountManagementBean, which is associated with one interceptor class (AccountAudit,).

Code 10-2 Example of Using an Interceptor Class

```
1  @Stateless
2  @Interceptors({
3      com.acme.AccountAudit.class,
4  })
5  public class AccountManagementBean implements AccountManagement {
6      public void createAccount(int accountNumber,
7          AccountDetails details) { ... }
8      public void deleteAccount(int accountNumber) { ... }
9      public void activateAccount(int accountNumber) { ... }
10     public void deactivateAccount(int accountNumber) { ... }
11 }
12
13 public class AccountAudit {
14     @AroundInvoke public Object auditAccountOperation
15         (InvocationContext inv) throws Exception {
16     try {
17         Object result = inv.proceed();
18         Auditor.audit(inv.getMethod().getName(),
```

```
19     inv.getParameters()[0]);
20     return result;
21 } catch (Exception ex) {
22     Auditor.auditFailure(ex);
23     throw ex;
24 }
25 }
26 }
```

An interceptor class can contain at most only one business interceptor method.

Associating Multiple Business Interceptor Methods With an Enterprise Bean

You can define multiple business interceptor methods for an enterprise bean. This is a multi-step process.

1. Define each business interceptor method in its own interceptor class.
2. Optionally, define a business interceptor method in the bean class.
3. Use the `Interceptors` metadata annotation to associate the method interceptors with a bean class. For example, assuming you have declared two interceptor classes called `AccountAudit`, and `CustomSecurity`, you would then associate them as interceptors with the bean class, as follows.

```
@Stateless @Interceptors({
    com.acme.AccountAudit.class,
    com.acme.CustomSecurity.class
})
public class AccountManagementBean implements AccountManagement { }
```

Note – The `Interceptors` annotation can also be applied to a business method of a bean.

Code 10-3 shows a stateless session bean `AccountManagementBean` that is associated with two interceptor classes (`AccountAudit` and `CustomSecurity`).

Code 10-3 Example of Using Multiple Business Interceptor Methods

```
1  @Stateless
2  @Interceptors({
3      com.acme.AccountAudit.class,
4      com.acme.CustomSecurity.class
5  })
6  public class AccountManagementBean implements AccountManagement {
7      public void createAccount(int accountNumber,
8          AccountDetails details) { ... }
9      public void deleteAccount(int accountNumber) { ... }
10     public void activateAccount(int accountNumber) { ... }
11     public void deactivateAccount(int accountNumber) { ... }
12
13     @AroundInvoke public Object profile(InvocationContext inv)
14         throws Exception {
15         long time = System.currentTimeMillis();
```

```

16     try {
17         return inv.proceed();
18     } finally {
19         long diffTime = time - System.currentTimeMillis();
20         System.out.println(inv.getMethod() + " took " + diffTime + " "
21                         + "milliseconds.");
22     }
23 }
24 }
25
26 public class AccountAudit {
27     @AroundInvoke public Object auditAccountOperation
28         (InvocationContext inv) throws Exception {
29     try {
30         Object result = inv.proceed();
31         Auditor.audit(inv.getMethod().getName(),
32             inv.getParameters()[0]);
33         return result;
34     } catch (Exception ex) {
35         Auditor.auditFailure(ex);
36         throw ex;
37     }
38 }
39 }
40
41 public class CustomSecurity {
42     @Resource EJBContext ctx;
43     @AroundInvoke public Object customSecurity(InvocationContext inv)
44         throws Exception {
45         doCustomSecurityCheck(ctx.getCallerPrincipal());
46         return inv.proceed();
47     }
48     private void doCustomSecurityCheck(Principal caller)
49         throws SecurityException {
50         //...
51     }
52 }
53

```

Code 10-3 on page 10-10 shows three interceptors:

- `profile(InvocationContext inv)` contained in the session bean `AccountManagementBean`.
- `auditAccountOperation(InvocationContext inv)` contained in the interceptor class `AccountAudit`.

Associating Multiple Business Interceptor Methods With an Enterprise Bean

- `customSecurity(InvocationContext inv)` contained in the interceptor class `CustomSecurity`.

The interceptors are invoked in the order in which they are defined in the `@Interceptors` annotation, after which the interceptor method defined in the bean class is invoked.

Including Life-Cycle Callback Interceptor Methods in an Interceptor Class

You can create a life-cycle callback interceptor method in an interceptor class to handle the following life-cycle events generated by session and message-driven beans:

- PostConstruct event

This event is generated by stateful and stateless session beans and message-driven beans.

PostConstruct callbacks occur after any dependency injection has been performed by the container and before the first business method (for session beans) or message listener method (for message-driven beans) invocation on the bean.

- PreDestroy callback

This event is generated by stateful and stateless session beans and message-driven beans.

PreDestroy callbacks occur at the time the bean instance is destroyed.

- PostActivate callback

This event is generated by stateful session beans.

A PostActivate event occurs when the container restores to primary storage, the state of a session bean instance that it had previously cached in secondary storage.

- PrePassivate callback

This event is generated by stateful session beans.

A PrePassivate event occurs when the container decides to remove the stateful session bean instance from primary storage and cache it in secondary storage.

Note – Life-cycle callback methods can also be included in the bean class. For information, refer to “Creating Session Beans: Adding Life-Cycle Event Handlers” on page 3-12 of the “Implementing EJB 3.0 Session Beans” module.

Creating a Life-Cycle Callback Interceptor Method

To create a life-cycle callback interceptor method, you are required to perform the following tasks:

1. Declare a Java technology class.
2. Include a public no-arg constructor.
3. Declare a method in the Java technology class that conforms to the following signature.

```
anyAccessModifier void anyMethodName(InvocationContext ic)
```

The access modifier can be public, private, protected, or package-level access.

4. Annotate the method using one of the following life-cycle callback event annotations: PostConstruct; PreDestroy; PostActivate; PreActivate. For example:

```
@PreDestroy  
private void endShoppingCart(InvocationContext ic) {...};
```

5. Ensure that the interceptor method invokes the `InvocationContext` objects `proceed` method. For example:

```
return ic.proceed();
```

This ensures the invoking of the next callback interceptor method, if any in the chain.

6. Annotate the enterprise bean with the `Interceptors` annotation. Include the name of the interceptor class as an attribute of the `Interceptors` annotation.

Code 10-4 demonstrates the definition of a life-cycle callback interceptor method in an interceptor class.

Code 10-4 Interceptor Class Example With Callback Interceptor Method

```
1  public class CartCallbackListener {  
2      public CartCallbackListener() {} // public no-arg constructor  
3      // callback interceptor method  
4      @PreDestroy  
5      private void endShoppingCart(InvocationContext ic) {...}  
6  }
```

Code 10-5 demonstrates the use of the `Interceptors` annotation that associates the interceptor class shown in Code 10-4 with the bean class contained in Code 10-5.

Code 10-5 Bean Class Associated With Interceptor Class

```
1  @Interceptors({CartCallbackListener.class})
2  @Stateful public class ShoppingCartBean implements ShoppingCart {
3      private float total;
4      private Vector productCodes;
5      public int someShoppingMethod(){...};
6      //...
7  }
```

The following rules apply to life-cycle callback interceptor methods.

- A life-cycle callback interceptor method defined in the interceptor class is required to have the following signature:

```
anyAccessModifier void anyMethodName(InvocationContext ic)
```

- The life-cycle callback interceptor method cannot be static or final.
- The life-cycle callback interceptor methods can throw runtime exceptions.
- A life-cycle interceptor method can be annotated to handle more than one life-cycle event. For example:

```
@PostConstruct
@PreDestroy
private void statisticsCollector(InvocationContext ic) {...}
```

Defining Entity Classes: Adding Life-Cycle Event Handlers

With EJB 3.0 you are not required to provide life-cycle event handlers for entity classes. However, entity classes do generate life-cycle events and you can optionally provide event handlers for these methods.

An entity class generates the following life-cycle events:

- PrePersist callback

The entity manager invokes the PrePersist callback method before executing the persist operation on the entity instance.

- PostPersist callback

The entity manager invokes the PostPersist callback method after the execution of the persist operation on the entity instance.

- PreRemove callback

The entity manager invokes the PreRemove callback method before executing the remove operation on the entity instance.

- PostRemove callback

The entity manager invokes the PostRemove callback method after the execution of the remove operation on the entity instance.

- PreUpdate callback

The entity manager invokes the PreUpdate callback method before the database update operations to entity data. This might be at the time the entity state is updated or it can be at the time state is flushed to the database or at the end of the transaction.

- PostUpdate callback

The entity manager invokes the PostUpdate callback method after the database update operations to entity data. This might be at the time the entity state is updated or it can be at the time state is flushed to the database or at the end of the transaction.

- PostLoad callback

The PostLoad method for an entity is invoked after the entity has been loaded into the current persistence context from the database or after the refresh operation has been applied to it. The PostLoad method is invoked before a query result is returned or accessed or before an association is traversed.

Defining Life-Cycle Event Handlers

The rules to define life-cycle event handlers for entity classes are the same as those for session classes. You can define the event handler in either of the following locations:

- In the entity class
- In a callback listener class

The following rules are common to callback methods irrespective of where they are defined.

- A callback method is designated using the appropriate callback annotation (`PrePersist`, `PostPersist`, `PreRemove`, `PostRemove`, `PreUpdate`, `PostUpdate`, `PostLoad`)
- The same method can be designated to handle multiple callback events.
- Within an entity class each callback event can only have one callback method designated to handle it. Each callback event can have multiple listener classes specified to handle it.
- A callback method can access entries in the entity class' environment.
- Callback methods can throw runtime exceptions. A runtime exception thrown by a callback method that executes within a transaction causes that transaction to be rolled back.
- Callback methods must not throw checked exceptions. They can throw unchecked exceptions.
- Dependency injection is not defined for callback listener classes.

Defining a Callback Handler in the Entity Class

The following steps outline a process you can follow to add callback handlers to a entity class.

1. Define a callback method in the entity class that conforms to the following signature:

```
public void methodName()
```
2. Annotate the method with the appropriate callback annotation.
3. Repeat steps 1 and 2 as required for additional callback methods.

Defining Entity Classes: Adding Life-Cycle Event Handlers

Code 10-6 shows an example of life-cycle event handler methods (`validateCreate` and `adjustPreferredStatus`) defined within an entity class.

Code 10-6 Example of Callback Method in an Entity Class

```

1  @Entity
2  public class Account {
3      @Id
4      Long accountId;
5      Integer balance;
6      boolean preferred;
7      public Long getAccountId() { ... }
8      public Integer getBalance() { ... }
9      @Transient // because status depends upon non-persistent context
10     public boolean isPreferred() { ... }
11     public void deposit(Integer amount) { ... }
12     public Integer withdraw(Integer amount) throws NSFException {...}
13     @PrePersist
14     public void validateCreate() {
15         if (getBalance() < MIN_REQUIRED_BALANCE) throw new
16             AccountException("Insufficient balance to open an account");
17     }
18     @PostLoad
19     public void adjustPreferredStatus() {
20         preferred =
21             (getBalance() >= AccountManager.getPreferredStatusLevel());
22     }
23 }
```

Defining a Callback Handler in a Callback Listener Class

The following steps outline a process you can follow to define callback handlers in a callback listener class.

1. Declare a class for use as the callback listener class.
2. Declare a public no-arg constructor in the callback listener class.
3. Define a callback method in the callback listener class that conforms to the following signature:
`public void methodName(EntityClassType b)`
4. Annotate the method with the appropriate callback annotation.
5. Repeat steps 3 and 4 as required for additional callback methods.

Code 10-7 demonstrates the definition of a callback handler method in a callback listener class.

Code 10-7 Callback Listener Class Example

```

1  public class AlertMonitor {
2      @PostPersist
3      public void newAccountAlert(Account acct) {
4          Alerts.sendMarketingInfo(acct.getAccountId(), acct.getBalance());
5      }
6  }
```

Code 10-5 demonstrates the use of the callback listener annotation that associates the callback listener class shown in Code 10-7 with the entity class contained in Code 10-5.

Code 10-8 Entity Class Associated With Callback Listener Class

```

1  @Entity
2  @EntityListeners({com.acme.AlertMonitor.class})
3  public class Account {
4      @Id
5      Long accountId;
6      Integer balance;
7      boolean preferred;
8      public Long getAccountId() { ... }
9      public Integer getBalance() { ... }
10     @Transient // because status depends upon non-persistent context
11     public boolean isPreferred() { ... }
12     public void deposit(Integer amount) { ... }
13     public Integer withdraw(Integer amount) throws NSFException { ... }
14     @PrePersist public void validateCreate() {
15         if (getBalance() < MIN_REQUIRED_BALANCE) throw new
16             AccountException("Insufficient balance to open an account");
17     }
18
19     @PostLoad
20     public void adjustPreferredStatus() {
21         preferred =
22             (getBalance() >= AccountManager.getPreferredStatusLevel());
23     }
24 }
```

Reviewing Interceptors

The following rules apply to interceptor classes.

- The interceptor class must have a public no-arg constructor.
- An interceptor class can have:
 - At most one business interceptor method.
 - At most one life-cycle callback interceptor method per life-cycle event.

The following rules are common to all interceptor methods

- An interceptor method cannot be static or final.
- An interceptor method is required to invoke the proceed method of the `InvocationContext` object that it receives as an input parameter.

The following rules apply to business interceptor methods

- A business interceptor method defined in the interceptor class is required to have the following signature:

```
public Object anyMethodName(InvocationContext ic) throws Exception
  ● Business interceptor methods are allowed to catch and suppress business method exceptions.
  ● Business interceptor methods are allowed to throw runtime exceptions or any checked exceptions that the business method allows within its throws clause.
```

The following rules apply to life-cycle callback interceptor methods

- A life-cycle callback interceptor method defined in the interceptor class is required to have the following signature:

```
anyAccessModifier void anyMethodName(InvocationContext ic)
  ● Life-cycle callback interceptor methods can throw runtime exceptions.
  ● A life-cycle interceptor method can be annotated to handle more than one life-cycle event. For example,
```

```
@PostConstruct
@PreDestroy
private void statisticsCollector(InvocationContext ic) {...}
```

Module 11

Implementing Transactions

Objectives

Upon completion of this module, you should be able to:

- Describe the transaction demarcation management
- Implement CMT
- Interact programmatically with an ongoing CMT transaction
- Implement BMT
- Apply transactions to messaging

Additional Resources



Additional resources – The following reference provides additional information on the topics described in this module:

- Sun Microsystems, “JSR 220: Enterprise JavaBeans™, Version 3.0 EJB Core Contracts and Requirements, Chapter13.” [<https://sdlc3e.sun.com/ECom/EComActionServlet;jsessionid=CEAAE57A3BAB8A76D4555E3C5A1F4031>], accessed July 25, 2006.
- Sun Microsystems, “JSR 220: Enterprise JavaBeans™, Version 3.0 EJB 3.0 Simplified API.” [<https://sdlc3e.sun.com/ECom/EComActionServlet;jsessionid=CEAAE57A3BAB8A76D4555E3C5A1F4031>], accessed July 25, 2006.

Introducing Transaction Demarcation Management

This section provides an overview of transaction demarcation as it applies to enterprise beans. This overview describes:

- The transaction demarcation task
- Transaction demarcation implementation options
- Guidelines for selecting a transaction demarcation policy

Transaction Demarcation Task

As an enterprise bean developer, your main task is to determine the transaction demarcation policy for each method of the enterprise bean. The transaction demarcation policy determines where transactions begin and end; that is, which groups of operations form a transaction. The application server is responsible for coordinating the various back-end resources that make up your transaction. The developer's task is unchanged regardless of whether the transaction operates on a single database, multiple databases, or a mixture of databases and other systems.

Introducing Transaction Demarcation Management

Figure 11-1 shows two calls made by a client on the same method of an enterprise bean. Each method call is made with a different transactional context.

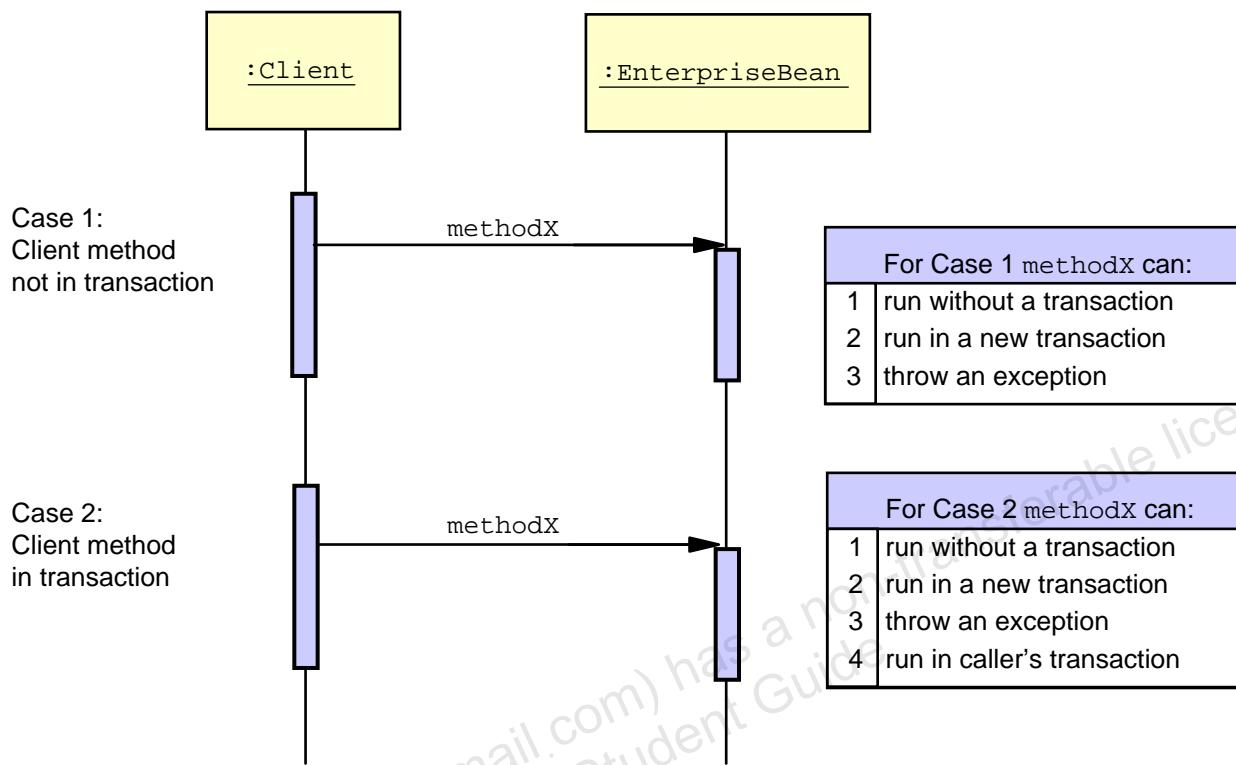


Figure 11-1 Transaction Policy Options

In the first case, the method invocation does not have an accompanying transaction context. When a method is invoked without a transaction, you have the following three options:

- Run the enterprise bean method without a transaction
- Run the enterprise bean method in a new transaction
- Throw an exception

In the second case, the method invocation has an accompanying transaction context. When a method is invoked with an accompanying transaction context, you have the following four options:

- Suspend the callers transaction context and run the enterprise bean method without a transaction
- Suspend the callers transaction context and run the enterprise bean method in a new transaction
- Throw an exception
- Run the enterprise bean method in the caller's transaction context

To specify the transaction demarcation policy for an enterprise bean method, you select one option from the first list (called without transaction case) and one option from the second list (called with transaction case).

Theoretically, you can form twelve unique transaction policies by combining the two lists. In practice, however, some combinations are not practical. For example, you will never combine option 3 (throw an exception) from each of the two lists.

Guidelines for Selecting a Transaction Demarcation Policy

The application assembler must select a transaction demarcation policy. To effectively select a transaction demarcation policy for a method or a group of methods, the application assembler needs detailed knowledge about the application domain, the use cases, the invocation order of the methods that service the use cases, and the transactional resources used by the methods. The application assembler then analyzes this knowledge using the following guidelines to derive the transaction demarcation policies for each method of the application:

- Minimize the duration of a transaction

This guideline recommends you isolate each method in its own transaction. This isolation minimizes the duration of the locking of shared resources.

Note – The efficiency of entity instances increases with the increasing scope of the transaction because the entity instances do not need to be synchronized to their database while the data is locked in a transaction. Thus, it is unwise to have many, short transactions involving entity instances unless the application logic demands this.



Introducing Transaction Demarcation Management

- Group all methods required for a use case into a single transaction

This guideline is based on the assumption that each coarse-grained interaction with the client (each use case) is a transaction. This guideline contradicts the first guideline. This guideline ensures the integrity of the data (state) maintained by the application. It does so by ensuring that the data changed by methods servicing a use case are either always committed or always rolled back.

When considering this guideline, you must examine the methods that are called during a use-case, and ask which of the following is true:

- If this method fails, should the whole transaction roll back?
- If this method fails, should only the work in this method roll back?
- If this method fails, should nothing roll back?

The answers to these questions identify candidate methods for inclusion or isolation from a transaction.

- If possible, isolate access to shared resources in a separate transaction.

This guideline is a compromise between the first two guidelines. The use of this guideline requires good design. What happens if the isolated transaction rolls back? Will it affect the use case? If the answer is yes, you have two options to ensure state integrity:

- Option 1 – Ensure that the transaction associated with the use case does not commit.
- Option 2 – If the transaction associated with the use has committed, then use a compensating transaction to undo the actions performed by the use case commit.

Implementation Options for Transaction Demarcation

Figure 11-2 shows the two implementation options: CMT and BMT. You use one of these options to implement the transactional policy associated with enterprise bean methods.

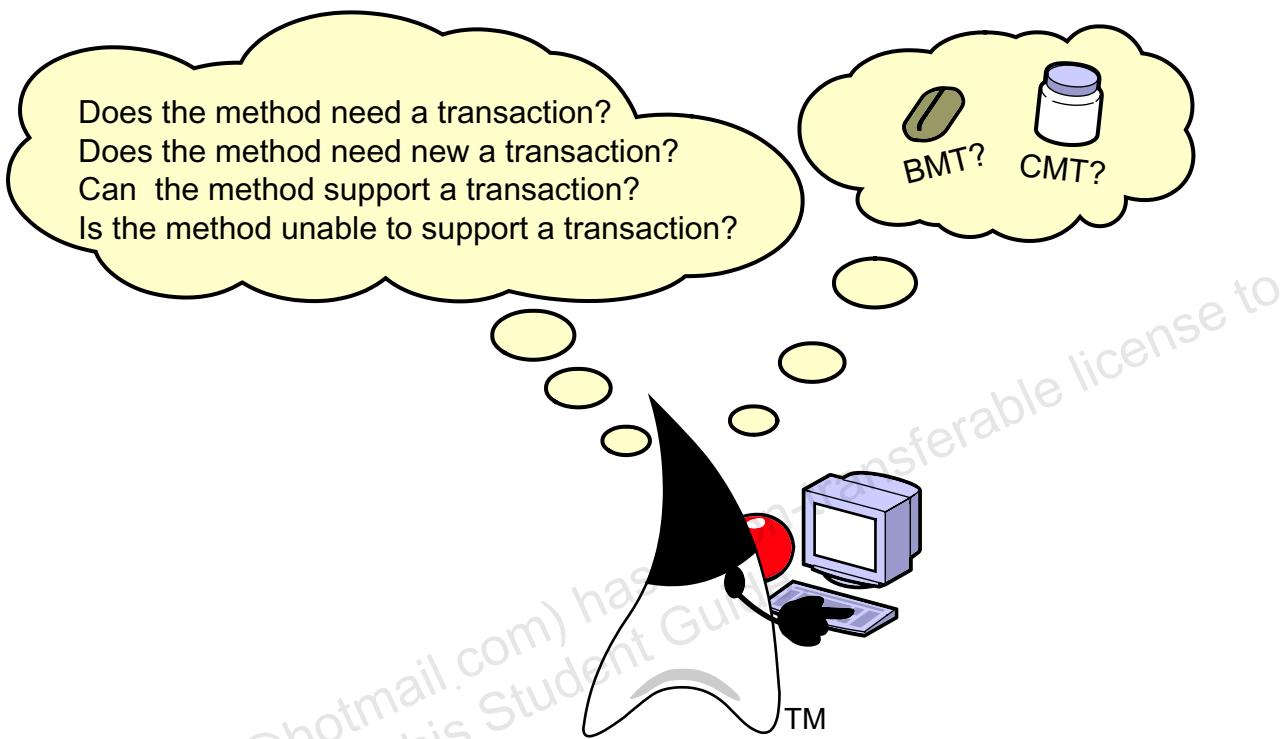


Figure 11-2 Transaction Demarcation Management Options

Overview of CMT

When using CMT, you delegate the management of the required transaction demarcation policy to the container.

CMT provides the application assembler with six predefined transaction demarcation policies: Required, RequiresNew, Supports, Mandatory, NotSupported, and Never.

Figure 11-3 on page 11-10 shows the transaction demarcation behavior of each of these six policies.

To apply CMT to the methods of an enterprise bean, the application assembler should, on a method-by-method basis, consider the suitability of accepting the EJB 3.0 default transaction policy (Required). If this is the case, then no further action is required. If not then for each method perform the following tasks:

Introducing Transaction Demarcation Management

- Select the most suitable policy from the list of available transaction demarcation policies
- Specify the selected policy using either the transaction attribute metedata annotations (`TransactionAttribute`) or the equivalent DD elements
- Mark the enterprise bean as using CMT by using either the transaction management metadata annotation (`TransactionManagement`) or include the equivalent entry in the DD.

At runtime the container manages the transaction demarcation of the method in accordance with the policy specified either by annotation or in the DD.

Overview of BMT

BMT is restricted to session and message-driven beans. To implement BMT, you must:

- Code the required transaction policy in the enterprise bean method
This code issues the required begin and commit or roll back requests to the transaction manager of the data resource.
- Mark the enterprise bean as using BMT by using either the transaction management annotation (`TransactionManagement`) or include the equivalent entry in the DD.

Using CMT

When using CMT, you delegate the transaction demarcation management to the container. You use the transactional attribute tag of the DD to specify the transactional behavior required for the method. During runtime, the container uses the value of this attribute to provide the required transaction demarcation.

CMT Transaction Attributes

With CMT, you specify the transactional behavior of the enterprise bean by applying a particular transaction attribute to each method. The container uses this attribute, in combination with the current transaction state (if any), to determine how to adjust the transaction context on entry to each method. On exit from the method, the container completes whatever transactional work it began on entry. For example, if the transaction attribute specifies that the method should run in a transaction, and there is no transaction in effect, then the container creates a new transaction. On exit from that method, because the transaction was begun on entry, the container must attempt to commit it. Similarly, if the attribute specified that there should be no transaction in that method, and a transaction was in effect when the method was called, the container suspends the current transaction before entering the method. On exit from the method, the container resumes the transaction it suspended.

Using CMT

The EJB specification refers to the six CMT transactional demarcation policies (REQUIRED, REQUIRES_NEW, SUPPORTS, MANDATORY, NOT_SUPPORTED, NEVER) as attributes. Figure 11-3 shows the transaction demarcation policies implemented by each CMT transaction attribute.

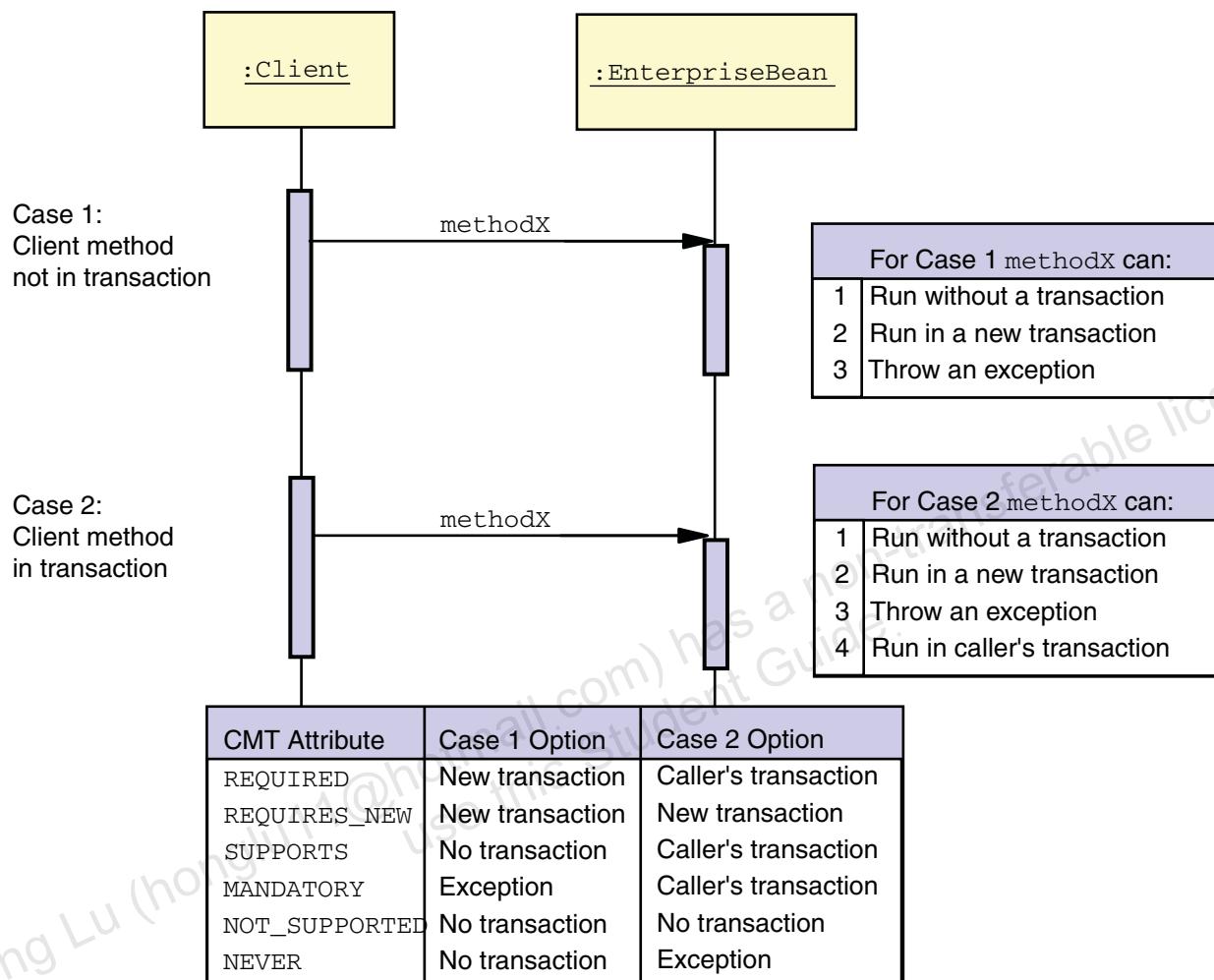


Figure 11-3 CMT Transaction Attributes

The transaction demarcation characteristics of these CMT transaction attributes are described in the following sections.

- The REQUIRED attribute

This transactional attribute is applicable for methods that are *required* to execute in a transaction but not necessarily a new transaction. This attribute has the following behavior:

- In case 1, a client method not executing in a transaction invokes an enterprise bean method with the CMT attribute REQUIRED. Because the client method is not executing in a transaction, the container starts a new transaction before it invokes the enterprise bean method. At the completion of the method, the container either commits or rolls back the new transaction.
- In case 2, a client method executing in a transaction invokes an enterprise bean method with CMT attribute REQUIRED. Because the client method is executing in a transaction, the container is able to invoke the enterprise bean method in the client's transaction.

- The REQUIRES_NEW transactional attribute

This transactional attribute is applicable for methods that must execute in a new transaction. Because the Java EE technology transaction model is flat, only one transaction can be in effect at a given time for a given thread of execution. This attribute has the following characteristics:

- In case 1, a client method not executing in a transaction invokes an enterprise bean method with the CMT attribute REQUIRES_NEW. Because the client method is not executing in a transaction, the container starts a new transaction before it invokes the enterprise bean method. At the completion of the method, the container either commits or rolls back the new transaction.
- In case 2, a client method executing in a transaction invokes an enterprise bean method with the CMT attribute REQUIRES_NEW. The container suspends the client's transaction before starting a new transaction for the enterprise bean method to execute in. At the completion of the enterprise bean method, the container either commits or rolls back the new transaction. It then resumes the client transaction before returning to the client method.

The effect of all this is to ensure that, if the method fails, its own transactional work is rolled back, but the caller's transaction is unaffected. The REQUIRES_NEW transactional attribute is useful for situations where an operation involves multiple steps and those steps form a transaction, but if that transaction fails, the caller should be given the chance to try again.

Using CMT

- The SUPPORTS transactional attribute

This transactional attribute is applicable for methods that are permitted to *support* (that is, they are not required, but can support) a transaction. This attribute has the following characteristics:

- If the method is called with the calling thread currently in a transaction, then the method executes in the same transaction.
- If not, the method executes outside a transaction context.
- The MANDATORY transactional attribute

This transactional attribute is applicable for methods that must execute in the callers transaction. This attribute has the following characteristics:

- In case 1, a client method not executing in a transaction invokes an enterprise bean method with the CMT attribute MANDATORY. Because the client method is not executing in a transaction, the container throws an exception.
- In case 2, a client method executing in a transaction invokes an enterprise bean method with the CMT attribute MANDATORY. Because the client method is executing in a transaction, the container can invoke the enterprise bean method in the client's transaction.
- The NOT_SUPPORTED transactional attribute

This transactional attribute is applicable for methods that must *not support* a transaction, for example methods that access a legacy resource that cannot support a transaction. This behavior has the following characteristics:

- In case 1, a client method not executing in a transaction invokes an enterprise bean method with the CMT attribute NOT_SUPPORTED. Because the client method is not executing in a transaction, the container can invoke the enterprise bean method without any additional transaction-related interventions.
- In case 2, a client method executing in a transaction invokes an enterprise bean method with the CMT attribute NOT_SUPPORTED. The container suspends the clients transaction before it invokes the enterprise bean method. The enterprise bean method does not execute in a transaction.

- The NEVER transactional attribute

This transactional attribute is applicable for methods that must *never* be invoked in a transaction context. This behavior has the following characteristics:

- In case 1, a client method not executing in a transaction invokes an enterprise bean method with the CMT attribute NEVER. Because the client method is not executing in a transaction, the container can invoke the enterprise bean method.
- In case 2, a client method executing in a transaction invokes an enterprise bean method with the CMT attribute NEVER. Because the client method is executing in a transaction, the container throws an exception.

CMT Attribute Usage Restrictions

Table 11-1 shows the restrictions and recommendations placed on enterprise bean types relating to the six available CMT attributes. Some containers might support the use of the not recommended attributes for stateful session beans. However, the use of the not recommended attributes impacts the portability of these enterprise beans. For message-driven beans, only the NOT_SUPPORTED and REQUIRED attributes are meaningful.

Table 11-1 CMT Attributes and Enterprise Bean Types

Transaction Attribute	Stateless Session Beans	Stateful Session Beans	Stateful Session Bean implementing SessionSynchronization interface	Message-Driven Bean
NOT_SUPPORTED	Yes	Yes	No	Yes
REQUIRED	Yes	Yes	Yes	Yes
REQUIRES_NEW	Yes	Yes	Yes	No
SUPPORTS	Yes	Yes	No	No
MANDATORY	Yes	Yes	Yes	No
NEVER	Yes	Yes	No	No

Implementing CMT

To implement CMT for an enterprise bean:

1. Select the CMT transaction attribute for each method.

To select a transaction attribute use the guidelines described in “Guidelines for Selecting a Transaction Demarcation Policy” on page 11-5 and the characteristics and restrictions associated with the attributes described in “CMT Transaction Attributes” on page 11-9.

2. Apply the transactional attribute to the method using the TransactionAttribute annotation.

For example, Code 11-1 shows a container-transaction element that specifies the transaction attribute for the onMessage method of the PlaceBidMDB message-driven bean.

Code 11-1 Example of container-transaction Element

```

1 import javax.ejb.TransactionAttribute;
2 import javax.ejb.TransactionAttributeType;
3
4 @Stateless public PayrollBean implements Payroll {
5     @TransactionAttribute(TransactionAttributeType.MANDATORY)
6     public void setBenefitsDeduction(int empId, double deduction) {...}
7
8     @TransactionAttribute(TransactionAttributeType.REQUIRED)
9     public double getBenefitsDeduction(int empId) {...}
10
11    public double getSalary(int empid) {...}
12
13    @TransactionAttribute(TransactionAttributeType.MANDATORY)
14    public void setSalary(int empId, double salary) {...}
15 }
```

Note – By default the REQUIRED transaction attribute is assigned to the getSalary method.



3. The default transaction demarcation management setting for enterprise beans is CMT. However, you can optionally specify the use of CMT by annotating the bean code with the TransactionManagement (CONTAINER) annotation, as shown in Code 11-2.

Code 11-2 Explicitly Specifying CMT

```
1 @Stateless
2 @TransactionManagement(CONTAINER)
3 public PayrollBean implements Payroll {
4     /**
5 }
```

Interacting Programmatically With an Ongoing CMT Transaction

The EJB specification provides APIs for *programmatically* interacting with ongoing CMT transactions. You can use these APIs to perform the following tasks.

- Get or set the roll back status of a CMT transaction
- Enable a stateful session bean to monitor a CMT transaction

Getting or Setting the Rollback Status During CMT

The container uses a rollback flag to monitor the state of a transaction. You can set this flag programmatically using the `setRollbackOnly` method of the `EJBContext` object. Alternatively, the container sets this flag in response to a `SystemException` thrown in the method.

An enterprise bean method with `Required`, `RequiresNew`, or `Mandatory` attribute values can call the following methods on the `EJBContext` object:

- `setRollbackOnly` – Dooms the current transaction by setting the rollback flag. A doomed transaction is a transaction that is eventually rolled back.
- `getRollbackOnly` – Determines whether the current transaction is doomed.

If the rollback flag is set, the container rolls back the transaction at the completion of the method.

Figure 11-4 shows an example of an enterprise bean using the EJBContext methods to get the status of the current transaction, and subsequently doom it.

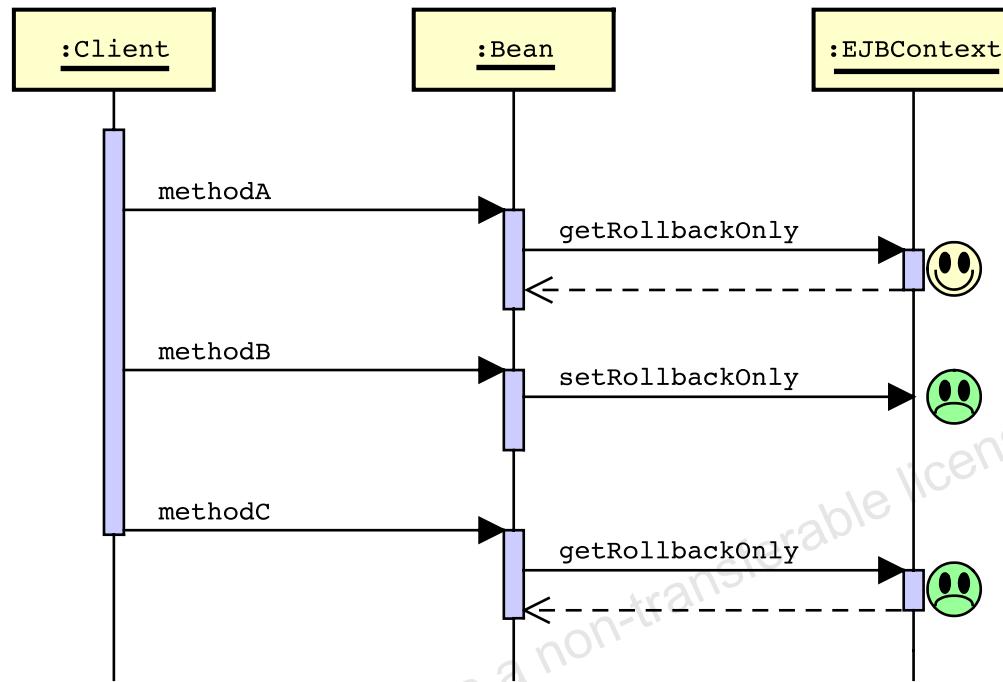


Figure 11-4 Controlling Transaction Outcome in CMT

Enabling a Stateful Session Bean Instance to Monitor a CMT Transaction

This section contains the following subsections.

- The problem: Surviving transaction rollback

This subsection presents a problem that is associated with state maintenance in stateful session beans participating in CMT transactions.

- The solution: Use the `SessionSynchronization` interface

This subsection presents a solution to the problem associated with state maintenance in stateful session beans participating in CMT transactions.

The Problem: Surviving Transaction Rollback

Consider the following facts relating to stateful session beans:

- If a stateful session bean takes part in a transaction, then any changes to its instance variables are transactional changes.
- If the transaction fails, the instance variables must revert to their pre-transaction states.
- Unlike entity instances, the container cannot help with reverting the instance variables to their pre-transaction states. The container cannot correct instance variables unless they derive directly from the database.

For example, consider the following sequence of operations in which all methods have the Required transaction attribute.

1. A client (with no transaction) calls a method on a stateless session bean. The container begins the transaction.
2. The stateless session bean calls the method on the stateful session bean. The container joins the transaction. The stateful session bean changes its state (instance variables).
3. The stateless session bean calls the method on the entity instance. The container joins the transaction.
4. The stateless session bean's method returns.
5. The container attempts to merge the values in the entity instance because the transaction is about to commit.
6. The merge method fails. The transaction must be rolled back.

The significant point here is that the stateful session bean cannot manage its own transaction commit and roll back as part of its business methods. In the previous example, the stateful session bean's business method was called, and it completed successfully before the whole transaction had to roll back. What is needed is a mechanism for the stateful session bean to be notified when a transaction of which it is a member is being committed or rolled back. This is the job of the SessionSynchronization interface.

The Solution: Use the SessionSynchronization Interface

Stateful session beans using CMT can optionally implement the SessionSynchronization interface. This interface provides the session bean with important feedback concerning transaction demarcation events.

Figure 11-5 shows the relationship among the three methods of the SessionSynchronization callbacks and the associated transaction demarcation events.

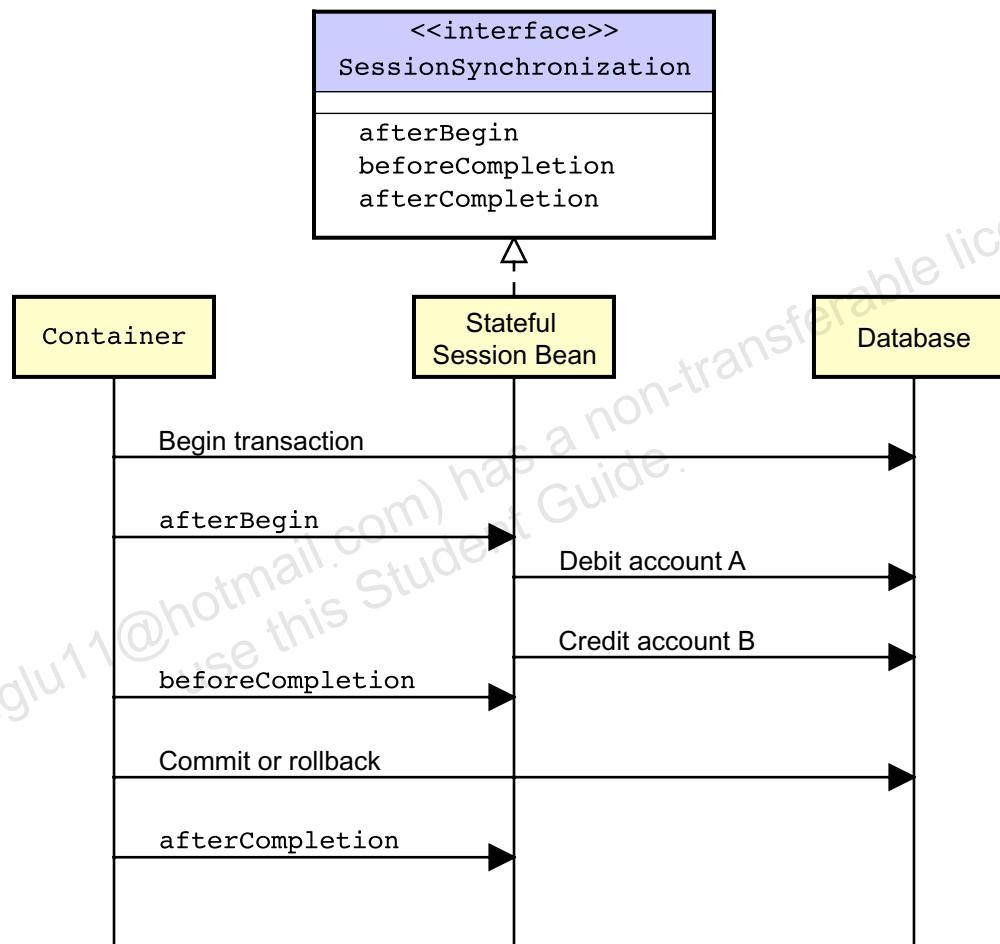


Figure 11-5 The SessionSynchronization Callbacks

The following list describes the events that trigger the calling of the SessionSynchronization methods.

- The container invokes the afterBegin callback method immediately after it begins a new transaction.
- The container invokes the beforeCompletion callback method immediately before it commits or rolls back a transaction.
- The container invokes the afterCompletion callback method immediately after it commits or rolls back a transaction.

In Figure 11-5 on page 11-20, the stateful session bean can transfer money between bank accounts on behalf of its clients. It is also caching, in its instance variables, the account balances to reduce the need to read the database. The problem is that if the instance variables change during a transaction, and the transaction fails, the database can roll back automatically, but the instance variables cannot.

Changes in the transaction state trigger calls to the SessionContext methods, as described in the following example:

- The client calls the method to transfer funds; the container begins a new transaction, and thus calls the afterBegin method. This tells the stateful session bean that a transaction is starting and it must do whatever is necessary to be able to roll back if it is told that. In many cases, the stateful session bean needs to make copies of the current values of the instance variables, so that they can be restored on roll back. In this example, the instance variables can be recovered from the database, so the implementation of the afterBegin method is probably empty.
- The stateful session bean now updates the database and its own instance variables.
- When the container is about to commit the transaction, it calls the beforeCompletion method. This is the stateful session bean's opportunity to abort the transaction if it has good reason to. It can do this by calling the setRollbackOnly method.
- After completing the transaction, the EJB container calls the afterCompletion method. The argument is true if the transaction was successful and false if it was not. If the transaction fails, the stateful session bean must restore the instance variables to the states they had when the afterBegin method was called. In this example, the stateful session bean can re-read the variables from the database.

Using BMT

BMT is restricted to session and message-driven beans only. To implement BMT, you must include the `TransactionManagement(BEAN)` annotation as shown in Code 11-2.

Code 11-3 Explicitly Specifying BMT

```
1  @Stateless  
2  @TransactionManagement(BEAN)  
3  public class StockBean implements Stock {  
4      //..  
5 }
```

- Code the enterprise bean's business methods to use the methods of the javax.transaction.UserTransaction interface to demarcate the transaction.

Enterprise bean methods obtain a UserTransaction object by invoking the `getUserTransaction` method on the context object associated with the enterprise bean.

Figure 11-6 illustrates the essential steps needed to implement BMT code.

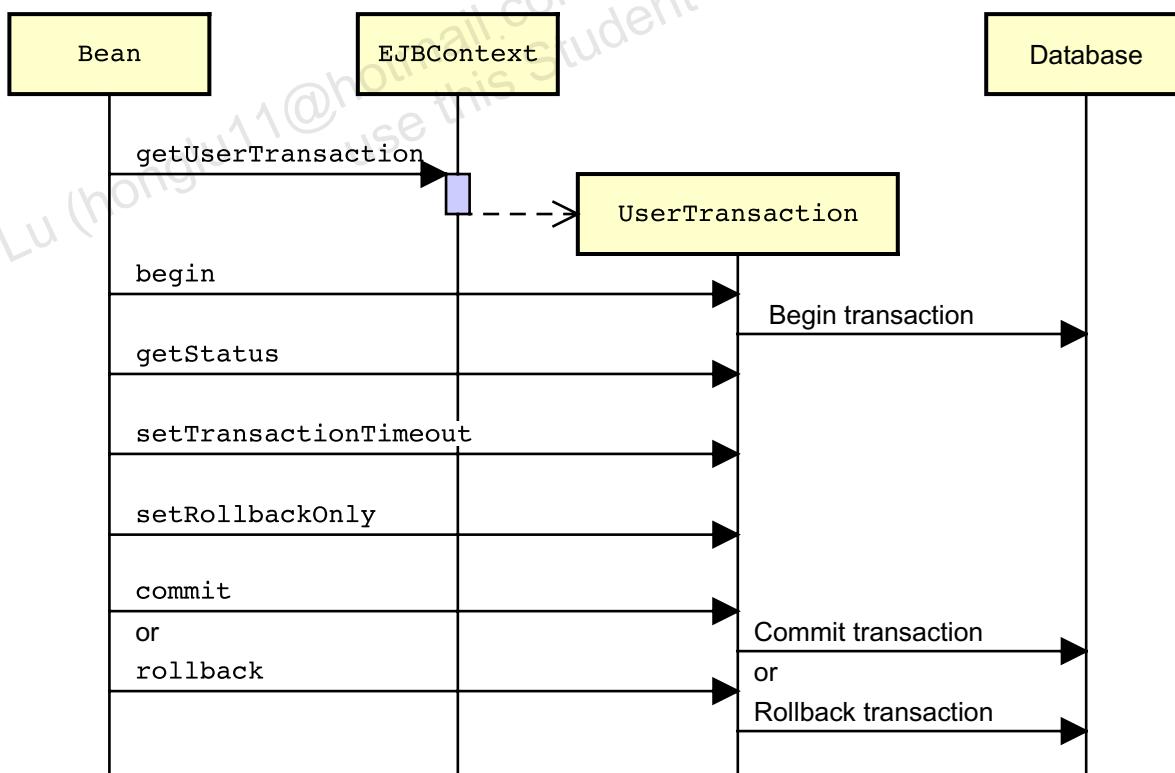


Figure 11-6 BMT Coding Sequence Diagram



Note – If the business case warrants, you can choose to not include any transaction management code in a enterprise bean that uses BMT.

Code 11-4 shows an example of BMT code.

Code 11-4 Example of BMT Code

```

1 import java.util.*;
2 import javax.ejb.*;
3 import java.sql.*;
4 import javax.sql.*;
5 import javax.naming.*;
6 import javax.transaction.*;
7
8 @Stateless
9 @TransactionManagement(BEAN)
10 public class StockBean implements Stock {
11
12     @Resource javax.Transaction.UserTransaction ut;
13     @Resource javax.sql.DataSource ds1;
14
15     public void updateStock(String symbol, double price) {
16         Connection con = null;
17         PreparedStatement prepStmt = null;
18         try {
19             con = ds1.getConnection();
20             ut.begin();
21             prepStmt = con.prepareStatement(
22                 "UPDATE Stock set price = ? + where symbol = ?");
23             prepStmt.setDouble(1, price);
24             prepStmt.setString(2, symbol);
25             int rowCount = prepStmt.executeUpdate();
26             ut.commit();
27         } catch (Exception ex) {
28             try {
29                 ut.rollback();
30             } catch (SystemException syex) {
31                 throw new EJBException
32                     ("Rollback failed: " + syex.getMessage());
33             }
34             throw new EJBException
35                 ("Transaction failed: " + ex.getMessage());
36         } finally {
37             try {
38                 prepStmt.close();

```

Using BMT

```

39         con.close();
40     } catch (SQLException e) {
41         throw new EJBException(
42             ("close failed: " + e.getMessage()));
43     }
44 }
45 }
46
47 public StockBean() {}
48
49 }
```

Restrictions on Transaction Demarcation Policies for Enterprise Beans Using BMT

This section presents the rules that restrict the transaction management policies for enterprise beans using BMT. It then presents the policies available to a bean developer coding stateless session beans, and message-driven beans, and stateful session beans.

Rules That Restrict BMT Transaction Demarcation Policies

The EJB specification specifies rules that restrict the transaction demarcation policies you can apply to enterprise beans.

The transaction demarcation policies that an enterprise bean developer can apply to an enterprise bean using BMT are governed by the following rules:

- The container suspends any client transaction before the method invokes an enterprise bean method, and resumes the client transaction after the enterprise bean method returns.
- An instance that starts a transaction must complete the transaction before it starts a new transaction.
- Any transaction started by a stateless session bean instance or message driven bean instance must commit or roll back before the method returns.

- A stateful session bean method is not required to close a transaction it is participating in before it returns:
 - The container permits multiple methods of stateful session beans to participate in the same transaction.
 - The container suspends the transaction and resumes it on the next call to a method on the same instance of the stateful session bean.

BMT Transaction Demarcation Policies for Stateless Session Beans and Message-Driven Beans

A stateless session bean or a message-driven bean using BMT must implement the following transaction demarcation policies:

- Policy 1 – No transaction
To implement this policy, you execute the enterprise bean method without a transaction.
- Policy 2 – Begin and end a BMT transaction
To implement this policy, you begin a transaction in the enterprise bean method and commit or rollback the transaction before the method returns.

Using BMT

Figure 11-7 illustrates the application of the BMT transaction demarcation policies to a stateless session bean. Figure 11-7 shows methodA and methodC implementing policy 2, and methodB implementing policy 1.

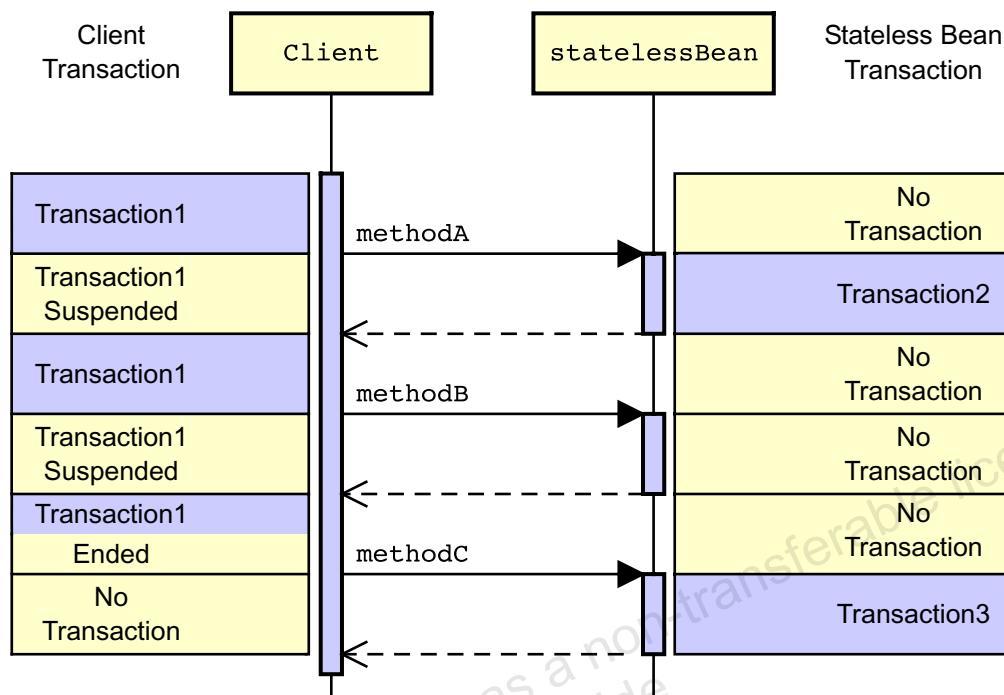


Figure 11-7 Example of Stateless Session Bean BMT Policies

BMT Transaction Demarcation Policies for Stateful Session Beans

Stateful session beans using BMT must implement the following transaction demarcation policies:

- Policy 1 – No transaction
To implement this policy, you execute the stateful session bean method without a transaction.
- Policy 2 – Begin and end a BMT transaction
To implement this policy, you begin a BMT transaction in the stateful session bean method and commit or roll back the transaction before the method returns.
- Policy 3 – Begin, but not end, a BMT transaction
To implement this policy, you begin a BMT transaction in the stateful session bean method and not commit or rollback the transaction before the method returns.
- Policy 4 – Continue an open BMT transaction but not end it
To implement this policy, you first verify that the stateful session bean instance has an ongoing BMT transaction (result of policy 3) associated with it. You allow the method to execute in that transaction context, and return without committing or rolling back the transaction.
- Policy 5 – Continue and end an open BMT transaction
To implement this policy, you first verify that the stateful session bean instance has an ongoing BMT transaction (result of policy 3) associated with it. You allow the method to execute in that transaction context, and commit or roll back the transaction before the method returns.

Using BMT

Figure 11-8 illustrates the application of the BMT transaction demarcation policies to a stateful session bean. Figure 11-8 shows methodA implementing policy 3, methodC implementing policy 5, and methodC implementing policy 2.

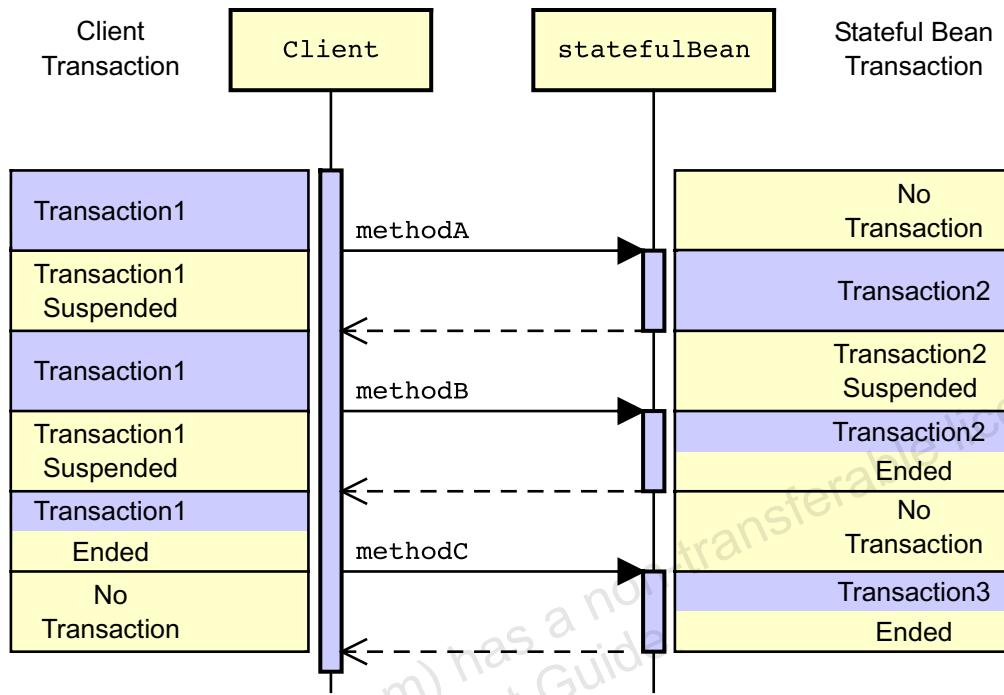


Figure 11-8 Example of Stateful Session Bean BMT Policies

Applying Transactions to Messaging

Although most developers understand that relational databases have transactional semantics, it is less well known that other back-end resources are also transactional. This is particularly true of messaging systems. If two or more operations have to succeed or fail together, then they form a transaction. This includes the case in which one (or more) of those operations is a messaging operation.

How does this work in practice? If the message producer puts two messages into two different queues, how are they rolled back if a transaction fails? The answer is that the messages are not released to consumers until the transaction commits. Consumers are not notified that messages are available until that point.

Similarly, if a consumer consumes a message as part of its transaction and that transaction fails, then the message is unconsumed. That is, the message is replaced in the queue. This is the state the message was in before the transaction, and although it sounds strange at first, it is semantically correct. If the consumer decides to re-try the transaction, the message is still there in the queue waiting to be consumed.



Note – Transactions are between the messaging clients and the messaging service. Messaging is asynchronous, so it does not provide a way to pass a transaction from one client to another.

Examining Transactions and JMS API Messages

Many message-oriented systems use asynchronous, bidirectional communication. For example, one client sends a message (asynchronously) to another. The recipient client processes the message, and then sends an acknowledgement (asynchronously) to the first client.

It is a common mistake to attempt to make one transaction span this entire operation. Under messaging technology, this is impossible because the producer's first message is not even released to the consumer until the transaction commits. Also, the transaction cannot commit until the consumer sends an acknowledgement, which never happens. This scenario is a classic deadlock problem.

Applying Transactions to Messaging

The solution is to accept that you cannot get an acknowledgement in the same transaction, as shown in Figure 11-9.

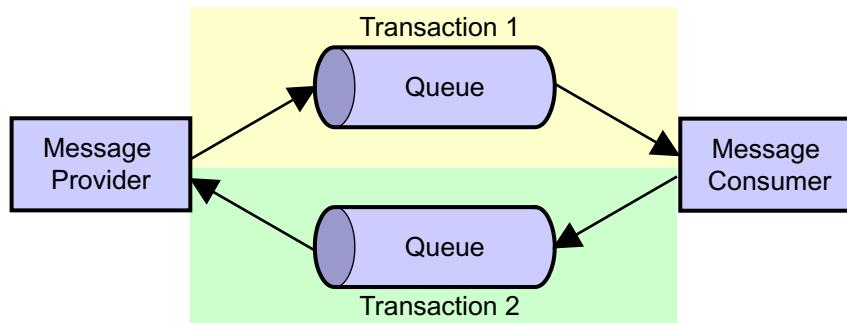


Figure 11-9 Separate Transactions Required for Sending and Receiving

Applying Transactions to Message-Driven Beans

Figure 11-10 illustrates the transaction settings applicable to message-driven beans.

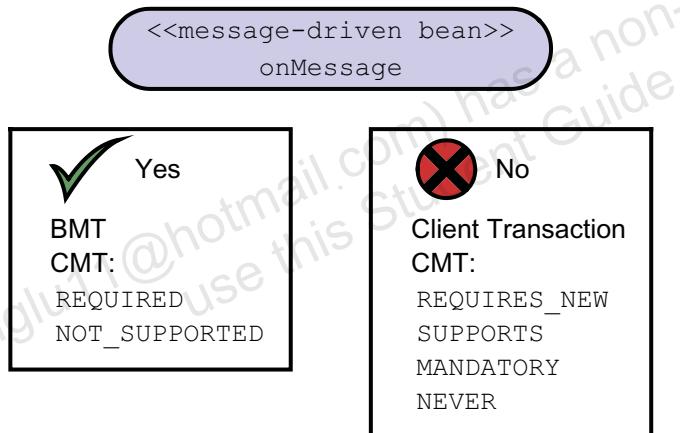


Figure 11-10 Transaction Settings Applicable to Message-Driven Beans

You cannot pass client transaction context to the `onMessage` method. Message-driven beans can use BMT or CMT. Message-driven beans using CMT must use the REQUIRED or NOT_SUPPORTED transaction attributes.

Message-driven beans using CMT with the REQUIRED transaction attribute always execute in a new transaction. This transaction context is propagated to the message origination resource manager and all other resource managers called by the `onMessage` method. If, for any reason, the transaction is rolled back, the message is resent to the message-driven bean. Such a scenario could cause an endless loop.

Figure 11-11 shows a message-driven bean executing under CMT REQUIRED transaction attribute.

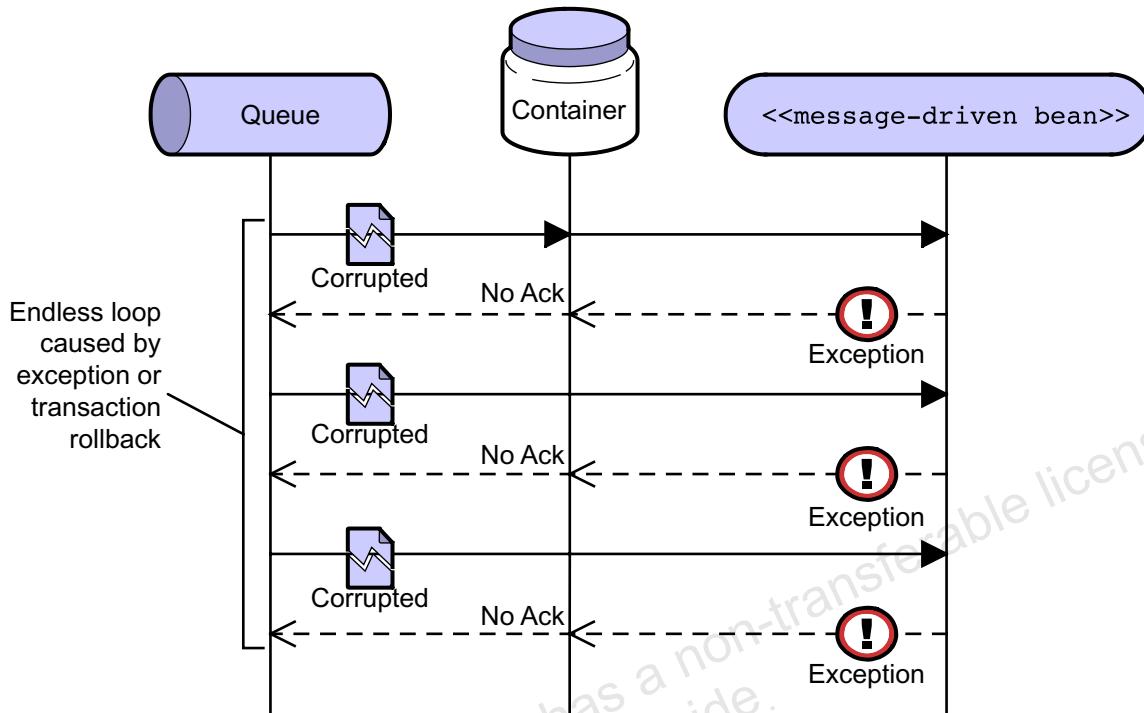


Figure 11-11 Endless Loop Caused by Transaction Roll Back

The message-driven bean receives a corrupted message, which results in the throwing of an exception and transaction roll back. This causes the message sender to resend the message, which results in an infinite loop. To summarize, a message-driven bean executing under CMT with the REQUIRED transaction attribute must never roll back the transaction.

The fact that a transaction roll back causes a re-delivery of the message is not an error or an inconvenience. This strategy is adopted because, if a transaction fails, the application server should give the application the opportunity to try again. This capability is important because there is no way for an asynchronous client to be notified that a message delivery has failed.

Applying Transactions to Messaging

In practice, application servers attempt to redeliver a fixed number of times before giving up, to avoid the endless loop shown in Figure 11-11 on page 11-31. The application designer should strive to ensure that the `onMessage` method does not call the `setRollbackOnly` method, or throw a system exception if the problem is one that it could deal with internally. If the message really is corrupt, as in Figure 11-11 on page 11-31, then throwing an exception is not really helpful. Trying again does not stop the message from being corrupt, it just gets the corrupt message back. On the other hand, if the `onMessage` method performs a database operation and that fails, then a retry might succeed. In such a case, rolling back the transaction and trying again is the right approach.

Module 12

Handling Exceptions

Objectives

Upon completion of this module, you should be able to:

- Introduce exceptions in Java EE applications
- Describe the exception path in a Java EE application environment
- Describe EJB container exception handling
- Handle exceptions in an enterprise bean's methods
- Handle exceptions in an enterprise bean's client code
- Review specific issues relating to exception handling in EJB technology applications

Additional Resources



Additional resources – The following reference provides additional information on the topics described in this module:

- Sun Microsystems, “JSR 220: Enterprise JavaBeans™, Version 3.0 EJB Core Contracts and Requirements, Chapter 14.” [<https://sdlc3e.sun.com/ECom/EComActionServlet;jsessionid=CEAAE57A3BAB8A76D4555E3C5A1F4031>], accessed July 25, 2006.
- Sun Microsystems, “JSR 220: Enterprise JavaBeans™, Version 3.0 EJB 3.0 Simplified API.” [<https://sdlc3e.sun.com/ECom/EComActionServlet;jsessionid=CEAAE57A3BAB8A76D4555E3C5A1F4031>], accessed July 25, 2006.

Introducing Exceptions in Java EE Applications

Java EE applications are distributed applications. Most Java EE technology clients access the application server either through the Internet or an intranet. Figure 12-1 shows some of the many locations that can throw exceptions in a Java EE application environment.

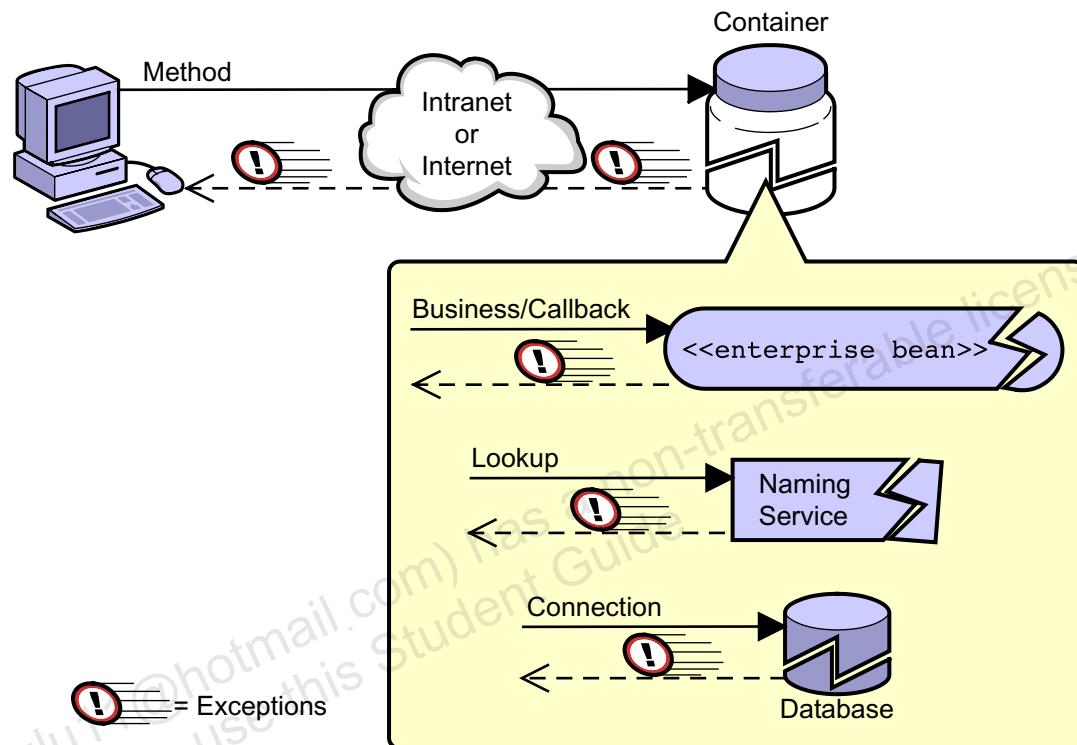


Figure 12-1 Examples of Exception Sources in Java EE Applications

Introducing Exceptions in Java EE Applications

For example, an exception can occur in the following areas:

- The client tier – The source could be the client software, the clients naming service, or the underlying local area network (LAN) infrastructure.
- The client-to-application server tier infrastructure – Exceptions can be caused by network failure. Such events promote uncertainty about the extent of method execution. That is, did the clients request get through to the application server? Did the exception occur on the return path from the application server?
- The application server tier – Exceptions in the application server tier can be classified as either *application* exceptions or *system* exceptions. Application exceptions are caused by logic errors in the application code. System exceptions are caused by errors in the services supplied to the enterprise beans by the Java EE technology container.
- The EIS tier – Exceptions in the EIS tier can be caused by an error in the underlying data resource, such as the inability to establish a database connection. In most cases, an exception is thrown from the method that failed. For example, failures in JDBC API operations mostly result in an `SQLException` being thrown. Exceptions of this type are mostly checked exceptions, and they must therefore be handled in the same way as application errors in the enterprise bean logic. If a method throws a runtime (system) exception, the enterprise bean can either handle it (if the exception is anticipated) or leave it to the container to handle. The way in which the container responds to uncaught system exceptions is described in “System Exception Handling by Containers” on page 12-15.

Java EE Platform Perspective of Exceptions

In Java technology, all exceptional conditions that the application code might be expected to handle are subclasses of the `Exception` class. The `Error` class is used to indicate conditions that an application can never be expected to handle, such as being out of memory.

Java technology classifies exceptions into the following categories:

- Checked exceptions
- Unchecked exceptions

Exceptions are classified as checked exceptions if they are required to be handled explicitly in Java technology code. Unchecked exceptions are those exceptions that the developer of a particular method does not need to handle, although it is often correct to handle these exceptions. Unchecked exceptions are all subclasses of the `RuntimeException` or `Error`.

The Java EE and EJB specifications classify exceptions differently:

- Application exceptions
- System exceptions

An approximate correspondence exists between these classifications. Application exceptions are generally checked exceptions. System exceptions are generally unchecked exceptions. There is not an exact match.

Introducing Exceptions in Java EE Applications

Application Exceptions

An application exception is an exception (but not including `java.rmi.RemoteException`) that is defined in the `throws` clause of the enterprise beans business interfaces or in a message listener interface.

Figure 12-2 illustrates the valid hierarchy for application exceptions.

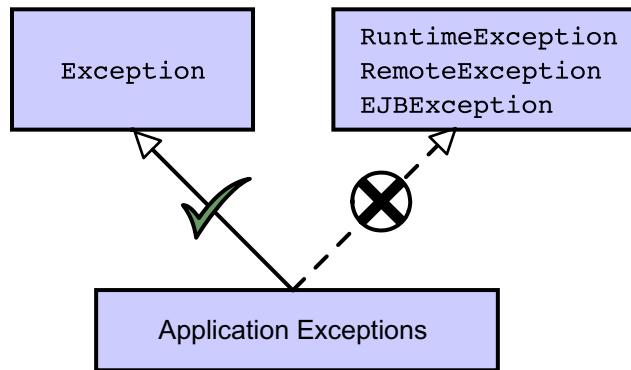


Figure 12-2 Application Exception Hierarchy

Application exceptions are restricted to subclasses of the `java.lang.Exception`, but they are not a subclass of `java.rmi.RemoteException`.

An application exception can also be a subclass of `java.lang.RuntimeException` provided it is annotated using the `ApplicationException` annotation. For example,

Code 12-1 Declaring an Application Exception

```

1  @ApplicationException(rollback=true)
2  public class BidFormatException extends IllegalArgumentException {
3      public BidFormatException(String s) {
4          super("Non numeric bid value: " + s);
5      }
6  }
  
```

Setting the `rollback` value of the `ApplicationException` annotation to `true` will result in a transaction rollback whenever the corresponding exception is thrown.

Application exceptions represent application-related error conditions. Application exceptions are used for notifying clients of abnormal application-level conditions, such as insufficient funds to perform a withdraw operation in a bank account.

Figure 12-3 lists the predefined application exceptions contained in the Java EE technology API.

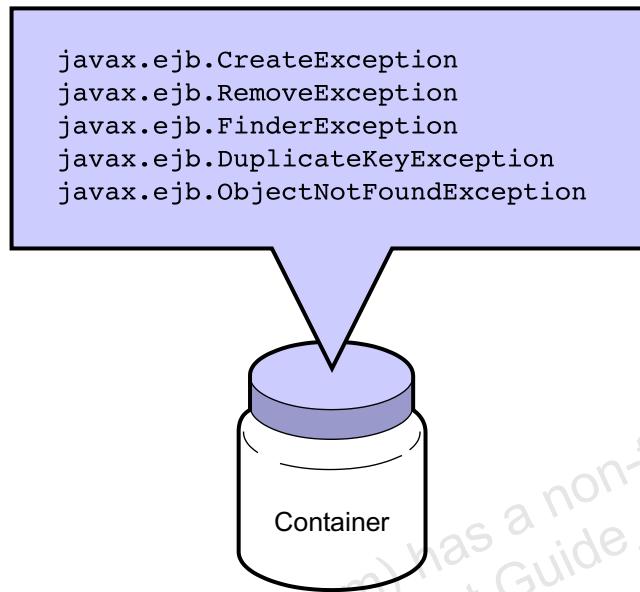


Figure 12-3 Predefined Application Exceptions

You can subclass these predefined exceptions to create application exception classes that better represent the context of your application.

Introducing Exceptions in Java EE Applications

System Exceptions

System exceptions are non-application exceptions. Figure 12-4 illustrates the valid hierarchy for system exceptions.

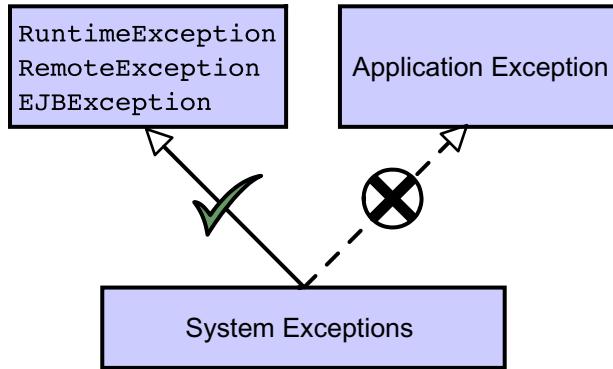


Figure 12-4 System Exception Hierarchy

System exceptions are caused by errors in the services supplied to the enterprise beans by the Java EE container. System exceptions can be thrown in business methods, container callback methods, or the message-driven bean's `onMessage` method.

Figure 12-5 shows some of the many sources that can throw system exceptions in a Java EE technology application server.

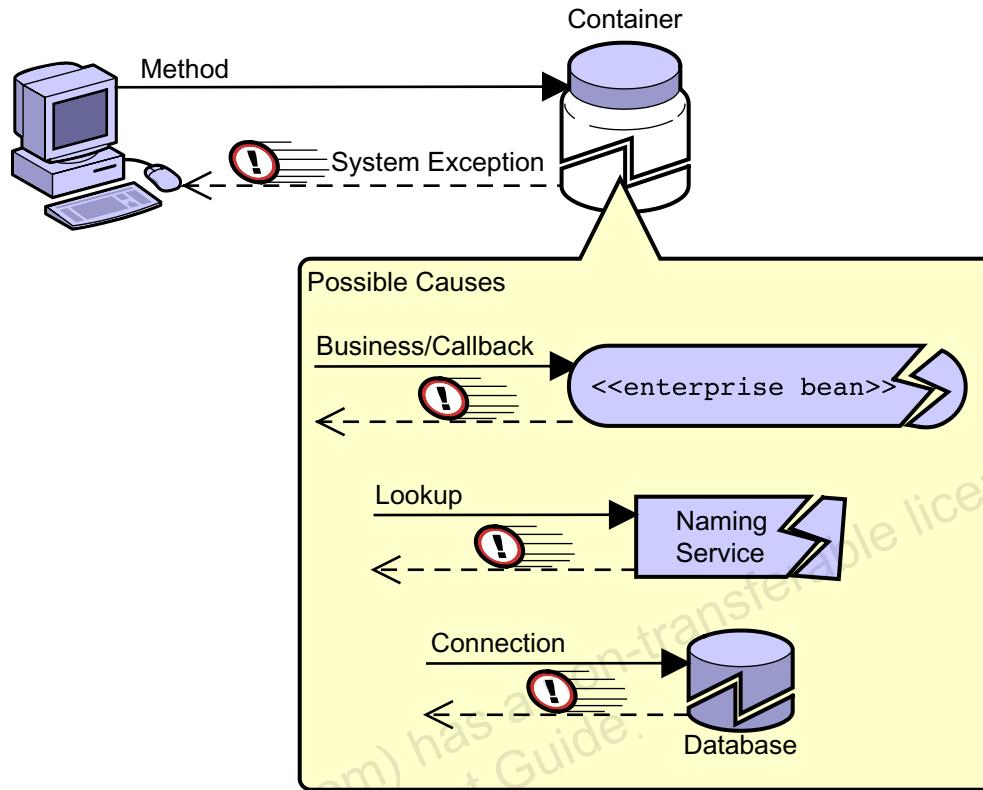


Figure 12-5 Examples of System Exception Sources

Failures in the underlying EJB container support services can throw exceptions. These failures include exceptions, such as a failure in JNDI technology service or a failure in the persistence support service, such as a failure to obtain a database connection. Although these can be considered low-level failures, they are not system exceptions in the sense that the EJB specification understands the term. The enterprise bean developer must write code that catches these exceptions and handles them.

In most cases, a failure in a resource is catastrophic and cannot be recovered. Thus, the enterprise bean developer must throw a system exception to the EJB container. This exception indicates that the state of the bean instance has been compromised beyond recovery.

Introducing Exceptions in Java EE Applications

The container can throw a system exception to the enterprise bean when it experiences a system error condition, such as insufficient memory, an unexpectedly closed network socket, or a threading problem. On the whole, the enterprise bean cannot handle or recover from such problems. Therefore, an enterprise bean should not normally intercept system exceptions on its way to the container. (See “Handling Exceptions in Enterprise Bean Methods” on page 12-17 for additional information.) However an enterprise bean that is a client of another enterprise bean should be prepared to handle system exceptions it receives from the EJB container as a result of a system exception condition in the service providing enterprise bean. (See “Handling System Exceptions in Bean Client Code” on page 12-22.)

Role of the RemoteException in EJB 1.0 – 2.0 Specifications

This section is supplied to provide a historic perspective of the role played by `java.rmi.RemoteException` in enterprise bean applications developed to EJB 1.0, 1.1 and 2.0 specifications.

In the EJB 1.0 specification, the `RemoteException` served two roles:

- Enterprise beans threw the `RemoteException` to indicate a catastrophic failure.

It was recommended practice to throw the `RemoteException` from the enterprise bean implementation to indicate a catastrophic failure

For example, to indicate an unrecoverable error from the `ejbLoad` method, the container would discard the instance, and possibly roll back a transaction of which that method was a part.

- Stubs threw the `RemoteException` to clients to indicate a failure in an RMI operation.

In the EJB 1.1 specification, this dual role was considered to be undesirable. The `RemoteException` was to retain its RMI failure significance only.

In the EJB 2.0 specification and later, an enterprise bean must never throw the `RemoteException`.

- For backward compatibility, this behavior is retained.
- Thus, the `RemoteException` is a system exception, and it is treated in the same way as runtime exceptions by the container.
- As a developer, do not throw a `RemoteException`, but be prepared to catch one in client code.

Examining the Exception Path in a Java EE Application Environment

Figure 12-6 shows the path that an exception takes in a Java EE application environment.

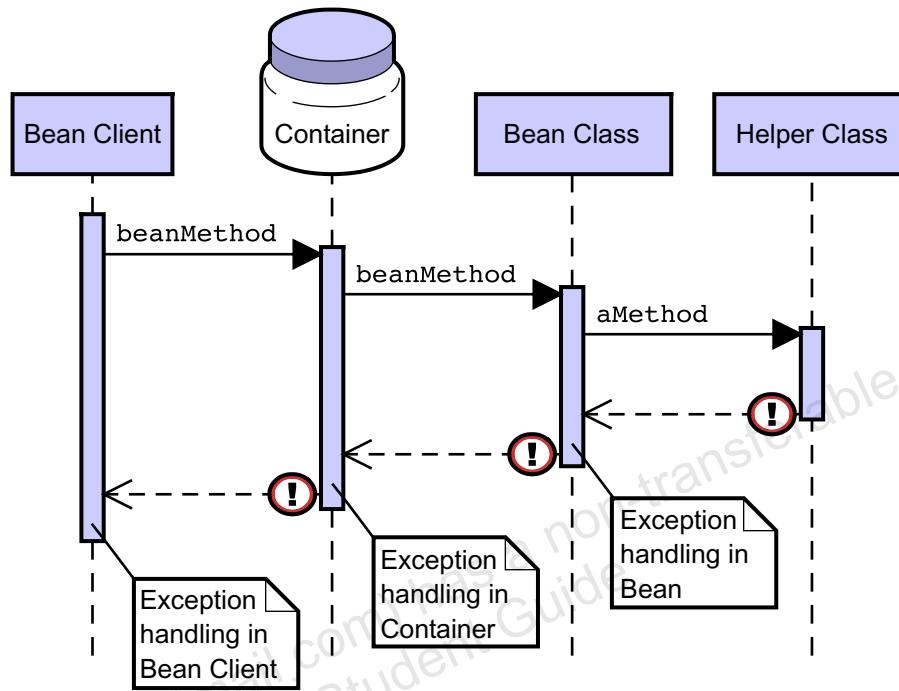


Figure 12-6 Exception Path in a Java EE Application Environment

An analysis of the path taken by an exception provides three strategic places for locating exception handling code:

- The enterprise bean method – The enterprise bean developer is responsible for including exception handling code in the enterprise bean methods.
- The container – The EJB specification defines the exception handling functionality of the container. The Java EE technology application server provider is responsible for implementation of the specification.
- The enterprise bean client – The enterprise bean client developer is responsible for including exception handling code in the enterprise bean client. The enterprise bean client can be another enterprise bean, an application client, an applet client, or a servlet.

Examining Exception Handling by Container

Figure 12-7 shows the factors that influence the handling of exceptions by the container.

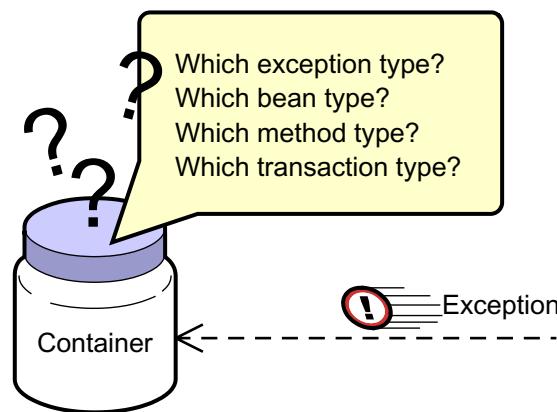


Figure 12-7 Factors Influencing Container Exception Handling

The two primary factors that influence the way the container handles exceptions are:

- The type of exception:
 - Application exception
 - System exception
- The transaction context in which the method (that threw the exception) is executing:
 - Caller's (another bean or client) transaction context
 - Container's transaction context started for this method
 - Unspecified transaction context
 - Bean-managed transaction context

The secondary factors that influence the way the container handles exceptions are:

- The type of bean:
 - Session bean
 - Message-driven bean
- The type of method:
 - Business method and business method interceptor methods
 - Container-invoked callback method

Application Exception Handling by Container

If an enterprise bean's business method throws an application exception, the container throws the application exception intact to the caller. This is exactly the behavior experienced in ordinary Java technology programming. However severe you might feel the exception to be, the container does not roll back a transaction merely because it caught an exception. If you want to cause a rollback, then you should arrange the enterprise bean code to call the `setRollbackOnly` method.

For example, suppose that an enterprise bean method performs a JDBC API operation and, if the operation fails, it throws the `SQLException` to its own caller. The `SQLException` generally indicates a serious failure but, in this case, the failure is not a system exception. Consequently, the container does not attempt to roll back a transaction.

However, if something has gone awry to the extent that an `SQLException` is thrown, then there is a strong likelihood that the container cannot commit the transaction when required. The likelihood is that the transaction rolls back anyway. But the important message is this: throwing an application exception itself, however severe the error, is not sufficient to cause a rollback. To ensure a rollback, you must roll back the transaction explicitly.

Examining Exception Handling by Container

Figure 12-8 shows the actions the container takes when it processes an application exception that occurred in the context of a container-started transaction.

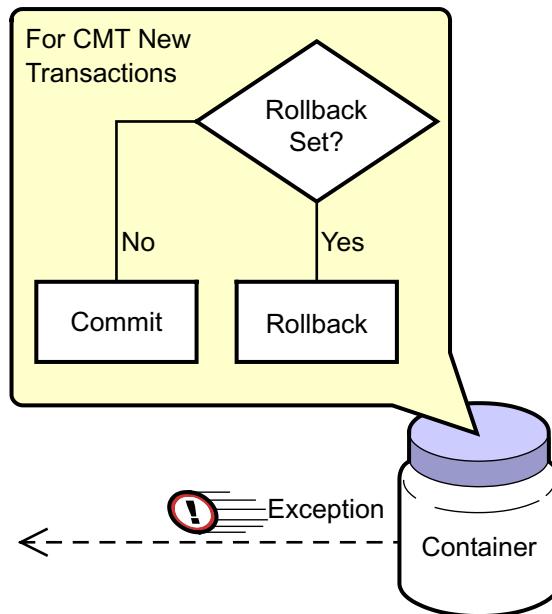


Figure 12-8 Application Exception Handling in a Container-Started Transaction Context

A transaction rollback occurs if either the `EJBContext.setRollbackOnly` method was invoked or the application exception was annotated to cause a transaction rollback.

System Exception Handling by Containers

Figure 12-9 shows the actions taken by the container when it handles a system exception.

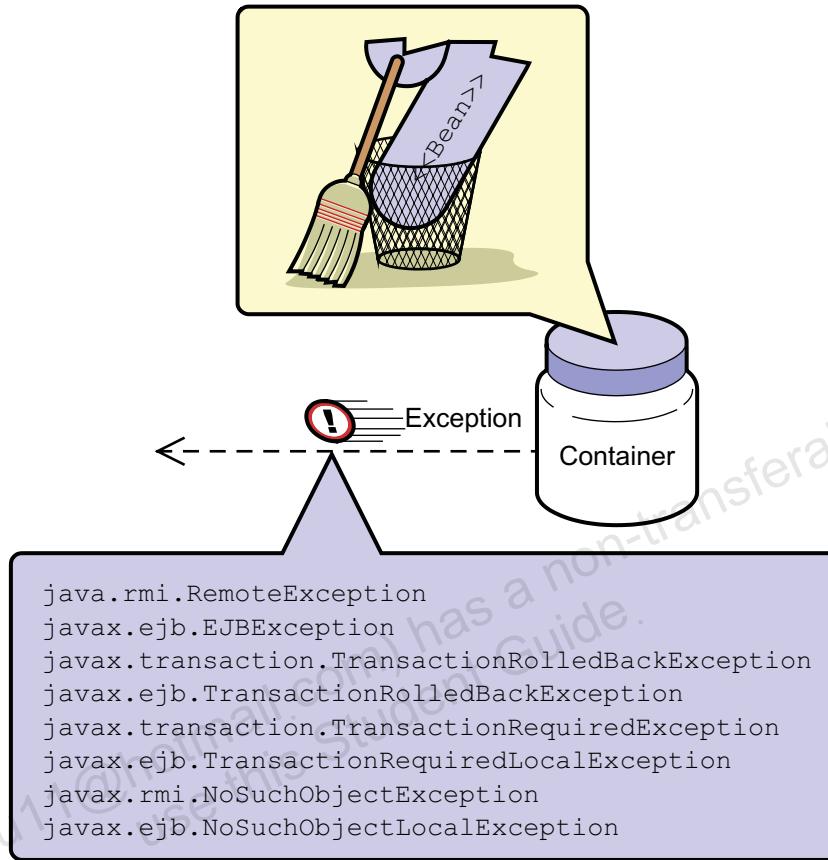


Figure 12-9 System Exception Handling by the Container

Examining Exception Handling by Container

When the container encounters a system exception it:

- Logs the error
- Performs all required clean up actions on the bean instance, including reclaiming resources, such as database connections allocated to the bean instance
- Discards the instance to ensure that no other method is invoked on the instance that threw the exception
- Informs the calling method of the condition causing the exception by throwing one of the following exceptions:
- `javax.ejb.EJBException`
`javax.transaction.TransactionRolledBackException`
`javax.ejb.TransactionRolledBackLocalException`
`javax.ejb.EJBTransactionRolledBackLocalException`
`javax.transaction.TransactionRequiredException`
`javax.ejb.TransactionRequiredLocalException`
`javax.ejb.EJBTransactionRequiredLocalException`
`javax.ejb.NoSuchEJBException`
`java.rmi.NoSuchObjectException`
`javax.ejb.NoSuchObjectLocalException`

Handling Exceptions in Enterprise Bean Methods

Figure 12-10 shows a flow chart that provides guidelines for designing exception handling code in bean methods.

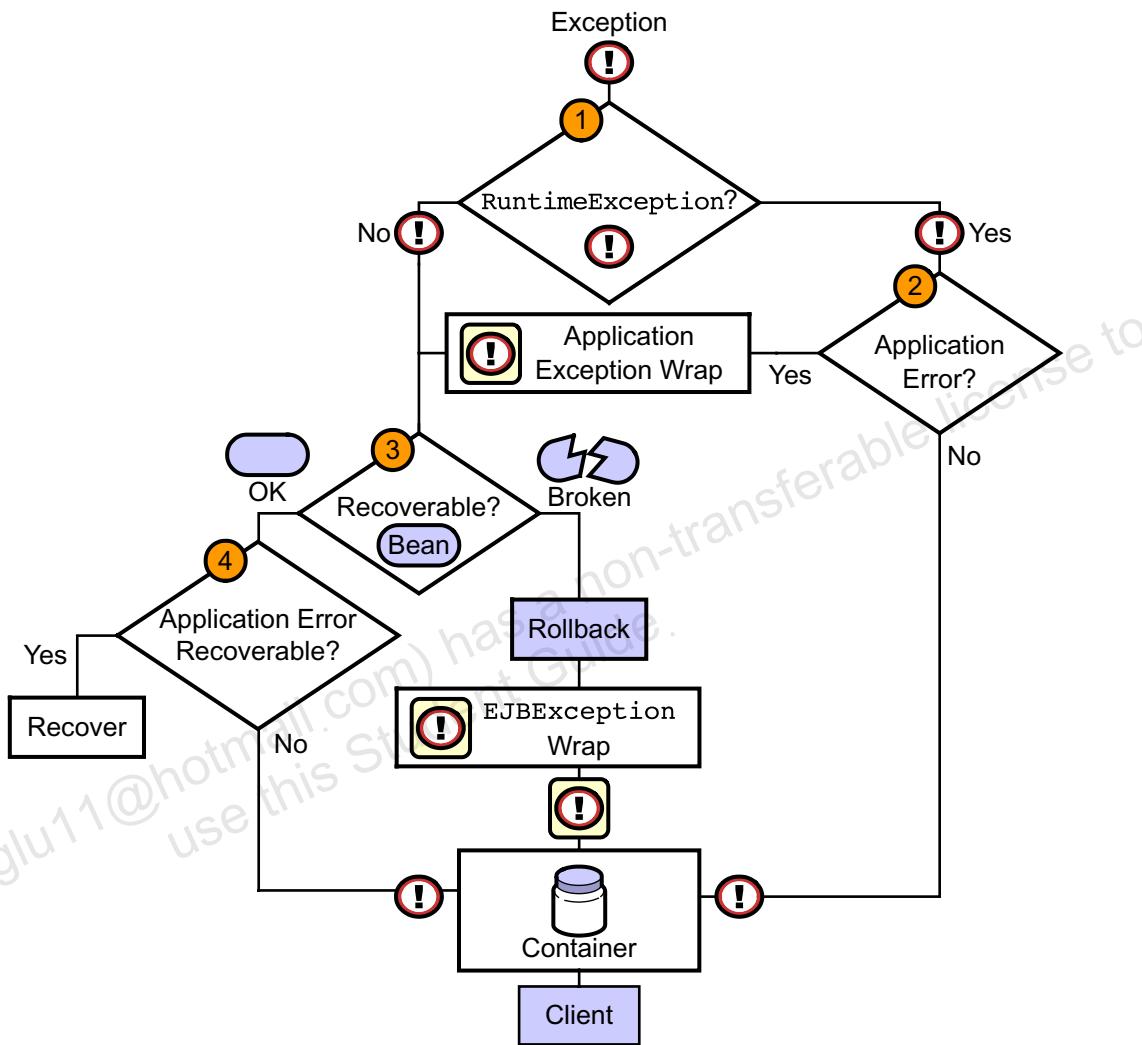


Figure 12-10 Guidelines for Handling Bean Method Exceptions

This flow chart applies to bean methods that are first recipients of newly thrown exceptions. For example, it applies to a bean method that receives an exception when it invokes a method in a helper class. Such an exception is sent directly from the helper class to the bean without passing through the container.

Handling Exceptions in Enterprise Bean Methods

The following steps explain the decision diamonds shown in Figure 12-10 on page 12-17.

1. Consider whether there are runtime exceptions that you need to handle. Remember that although the EJB specification treats runtime exceptions as system exceptions, not all runtime exceptions are true system exceptions, and some might need to be handled.
2. If the runtime exception is caused by an application error, then proceed as you would for other application errors. It might be helpful to re-throw the runtime exception as an application-defined exception to regulate the development of the rest of the method. For example the following line of code could potentially throw the runtime exception `NumberFormatException`,

```
// Assume bidString is the bid amount typed in as a string
double bidValue = bidString.parseDouble();
```

In this instance, the runtime exception should be caught and re-thrown as an application exception as follows:

```
// Assume bidString is the bid amount typed in as a string
try {
    double bidValue = bidString.parseDouble();
} catch (NumberFormatException e) {
    throw new BidFormatException(e.getMessage());
}
```

However, runtime exceptions that do not result from application logic should never be handled by the enterprise bean. The container should be allowed to deal with them. The risk with handling system exceptions in enterprise bean code is that you can prevent a rollback from occurring when it is required.

A good example of a runtime exception that you might have to handle is `NumberFormatException`. If your enterprise bean calls methods on helper classes, for example, to process currency values that have been supplied as `String` objects, then some conversion is required. The conversion methods typically throw the `NumberFormatException` if they fail. The Java programming language defines this as a runtime exception but, it is not a system exception in enterprise bean terms. The enterprise bean should probably handle this exception in some way.

3. If an application exception is caught, examine the state of the bean instance to determine whether there is any likelihood of being able to recover. If the cause of the failure is in the enterprise bean itself, you most likely cannot recover.

4. If the enterprise bean is intact, try to recover, but bear in mind that this requires providing exception handling code in the bean.

If the situation is unrecoverable, or the recovery fails, then you need to decide which of the following scenarios applies:

- The situation is a catastrophic failure that can never be recovered.

Throw a system exception. An `EJBException` suffices for most cases, but there are special exceptions defined for special cases.

- The situation is likely to have an effect on data integrity, and there must be a transaction rollback. However, the caller might be able to recover its own transaction.

This situation is relatively unusual, and mostly arises during the creation of entity instances. Here, you should call the `setRollbackOnly` method and throw an application exception (`CreateException` is an application exception).

- The caller might attempt to recover, and is given the opportunity to allow the transaction to proceed.

Throw an application exception. The caller can then proceed to examine the exception and make decisions about whether to abort the transaction.

Handling Exceptions in Enterprise Bean Client Code

In this section, client refers to any code that invokes a method on an enterprise bean. Using this definition, a client can be another enterprise bean, a Java technology servlet, an application, or an applet client.

Handling Application Exceptions in Bean Client Code

Figure 12-11 shows the issues you should consider when you add application exception handling code to the client code.

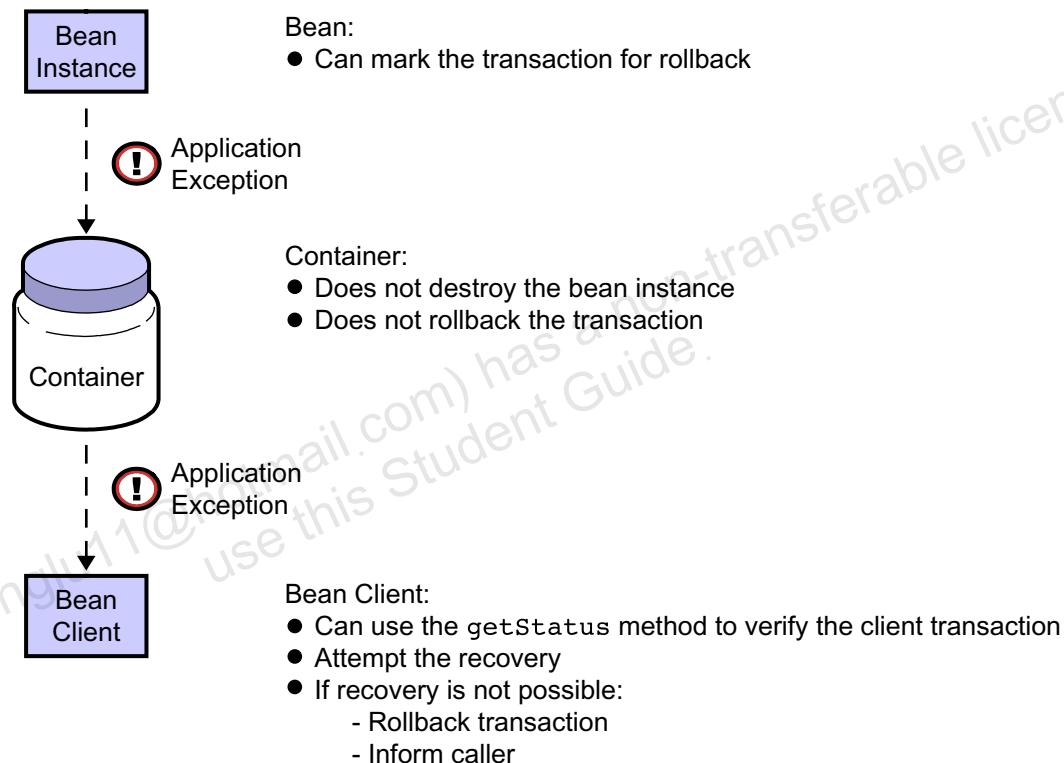


Figure 12-11 Handling Application Exceptions in Bean Client Code

When you write application exception recovery code, the container will *not*:

- Remove the `EJBObject`
- Roll back the transaction

You must include code to evaluate the information contained in the exception. If the business case permits, then attempt recovery. Recovery is not possible if the transaction has been rolled back.

If you decide to terminate the business process, then you must:

- Roll back any transaction that the business process might have started.
- Provide information about the failure to the application user or its calling client.

Note – If the method is executing in a client transaction, you can check the rollback status of the transaction by using the getStatus method of the javax.transaction.UserTransaction interface.



Handling System Exceptions in Bean Client Code

This section describes issues associated with system exception handling in the enterprise bean client code. These issues are described under sections that reflect the nature of the system error.

Method Invocation Failure

A remote client receives a `javax.ejb.EJBException` to indicate a method invocation failure. From a client's perspective, it is difficult to determine exactly where the invocation failed. For example, Figure 12-12 illustrates a method invocation that fails before reaching the remote object.

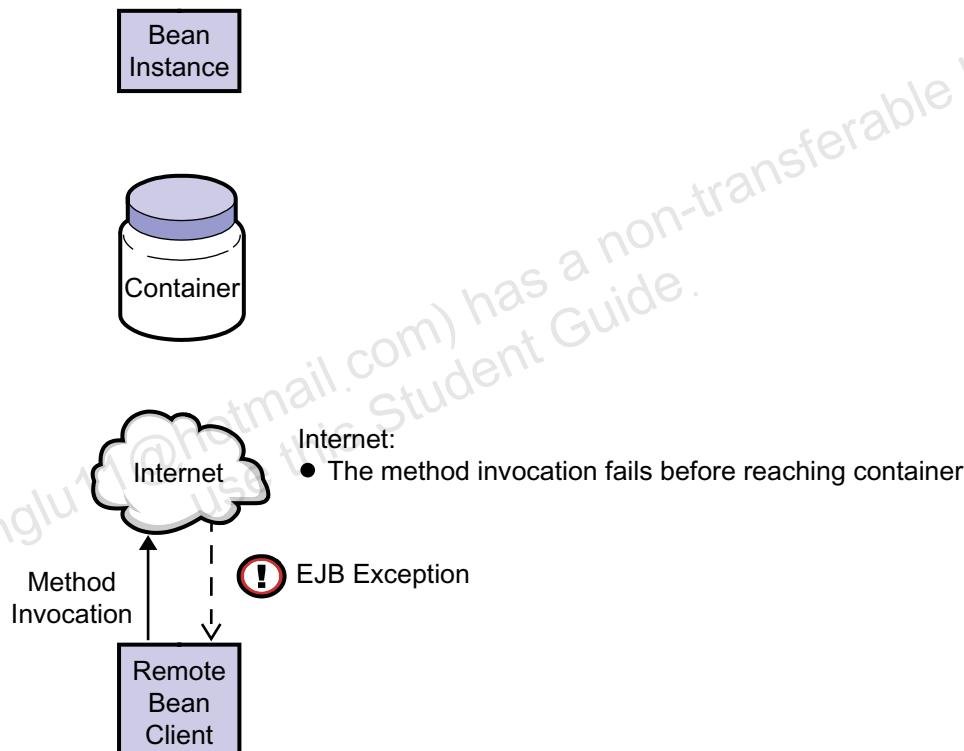


Figure 12-12 Method Invocation Failure Before Reaching the Remote Object

Figure 12-13 illustrates a method invocation failure caused by a system error in the bean instance.

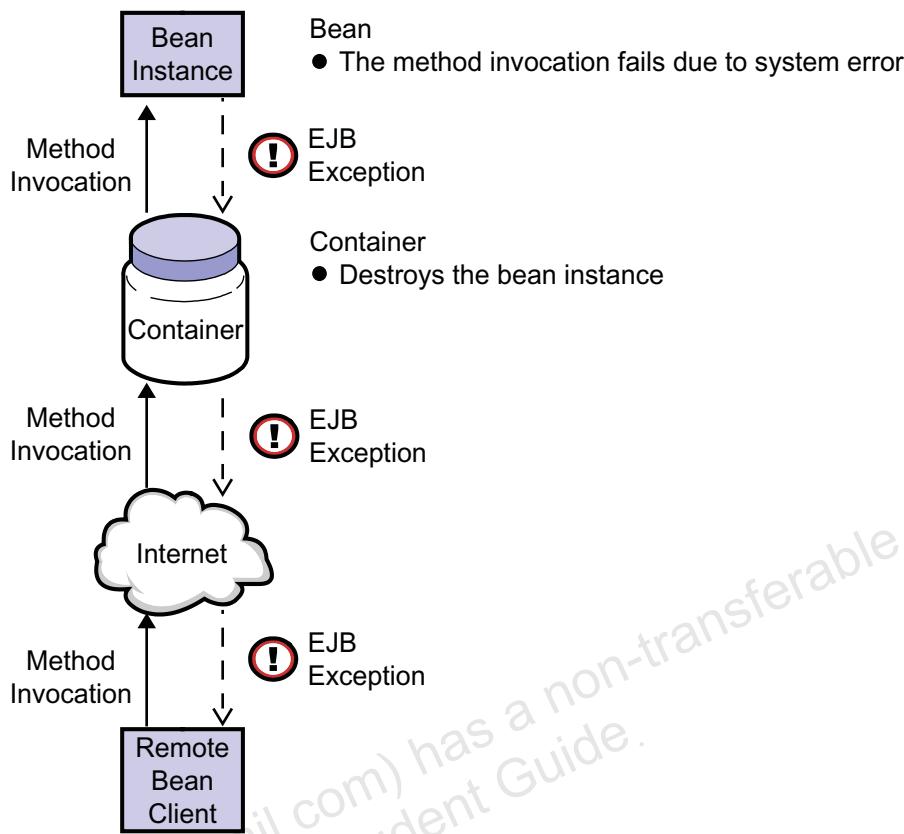


Figure 12-13 Method Invocation Failure in the Bean Instance

For example, a system error occurring in the beans methods cause an `EJBException`. In addition, a bean developer is expected to include code to throw an `EJBException` when a system related exception is encountered.

When a bean instance throws an `EJBException`, the container destroys the instance. The container then throws an `EJBException` to a remote bean client. This, however, is the same exception that a remote bean client receives in the scenario illustrated in Figure 12-12 on page 12-22. Consequently, you must consider the scenarios illustrated in both Figure 12-12 and Figure 12-13 when you write exception handling code to catch an `EJBException`.

Handling Exceptions in Enterprise Bean Client Code

Figure 12-14 illustrates a method invocation failure notification sent to a local enterprise bean client.

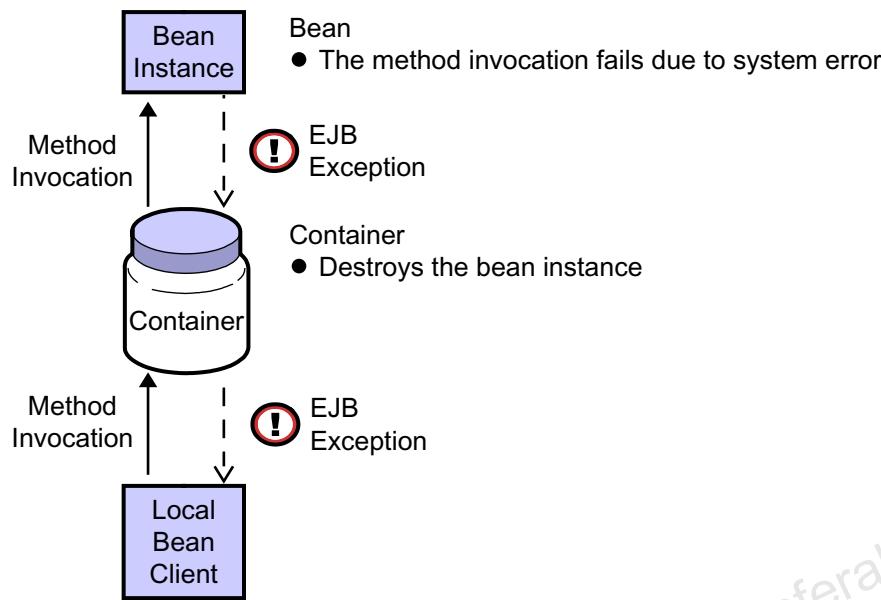


Figure 12-14 Method Invocation Failure Notification to a Local Bean Client

A local client receives a `javax.ejb.EJBException` to indicate a method invocation failure.

To summarize, a failed method invocation can be difficult to deal with. The primary reason for this is uncertainty about the extent of the invoked method's execution. The exception could have been thrown before the method invocation reaches the bean instance, during the method execution, or after the method execution.

If the exception was thrown by the target method, then you must deal with the possibility of the bean instance's destruction.

Finally, you must consider the possibility of a client transaction rollback. Use the `UserTransaction.getStatus` method to determine the state of the client transaction.

Transaction-Rollback Exception

Transaction rollback (in EJB 3.0) is indicated by the `javax.ejb.EJBTransactionRolledBackException` exception.

Figure 12-15 illustrates a transaction rollback exception scenario.

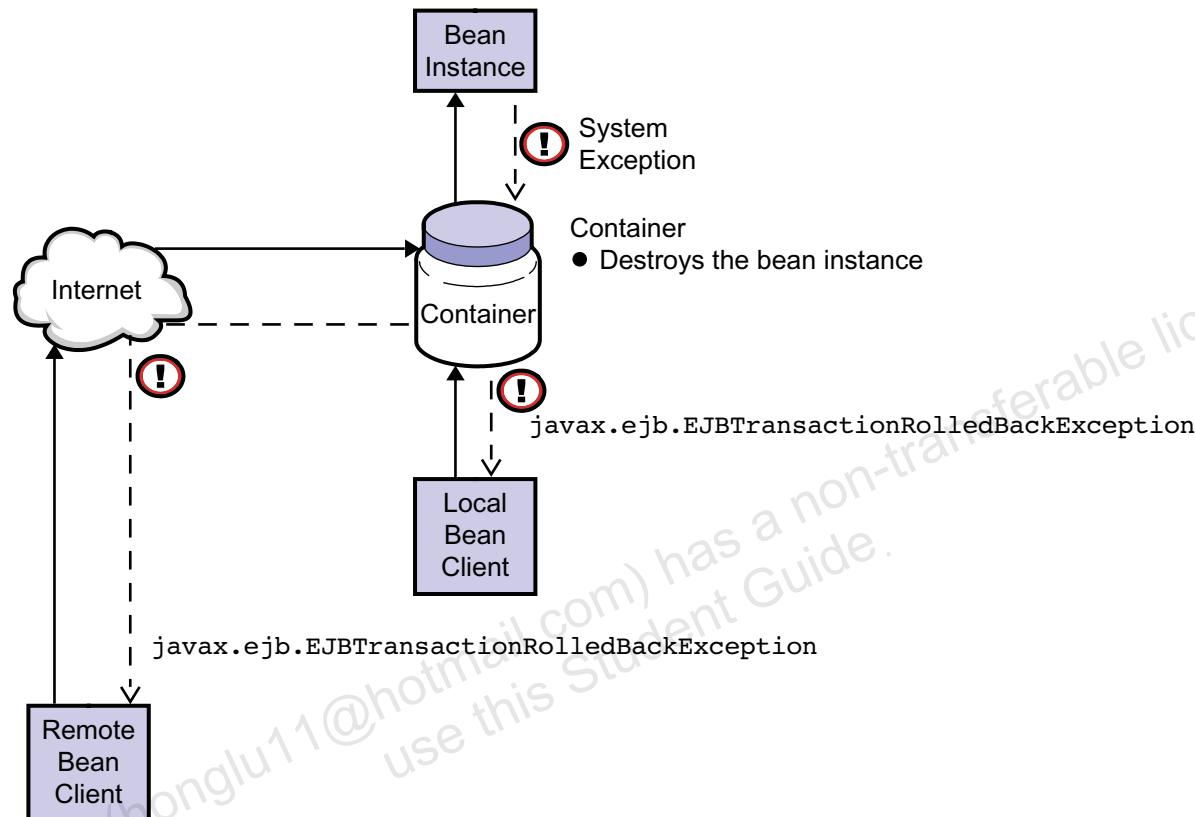


Figure 12-15 Handling Transaction Rollback

To handle this exception, you must include code to handle the following conditions:

- Client transaction rollback
- Bean instance destruction

Handling Exceptions in Enterprise Bean Client Code

Transaction-Required Exception

A transaction-required exception indicates that a method that requires a transaction context has been invoked without the required transaction context.

A transaction-required exception is indicated by the `javax.ejb.EJBTransactionRequiredException` exception.

Figure 12-16 illustrates a transaction-required exception scenario.

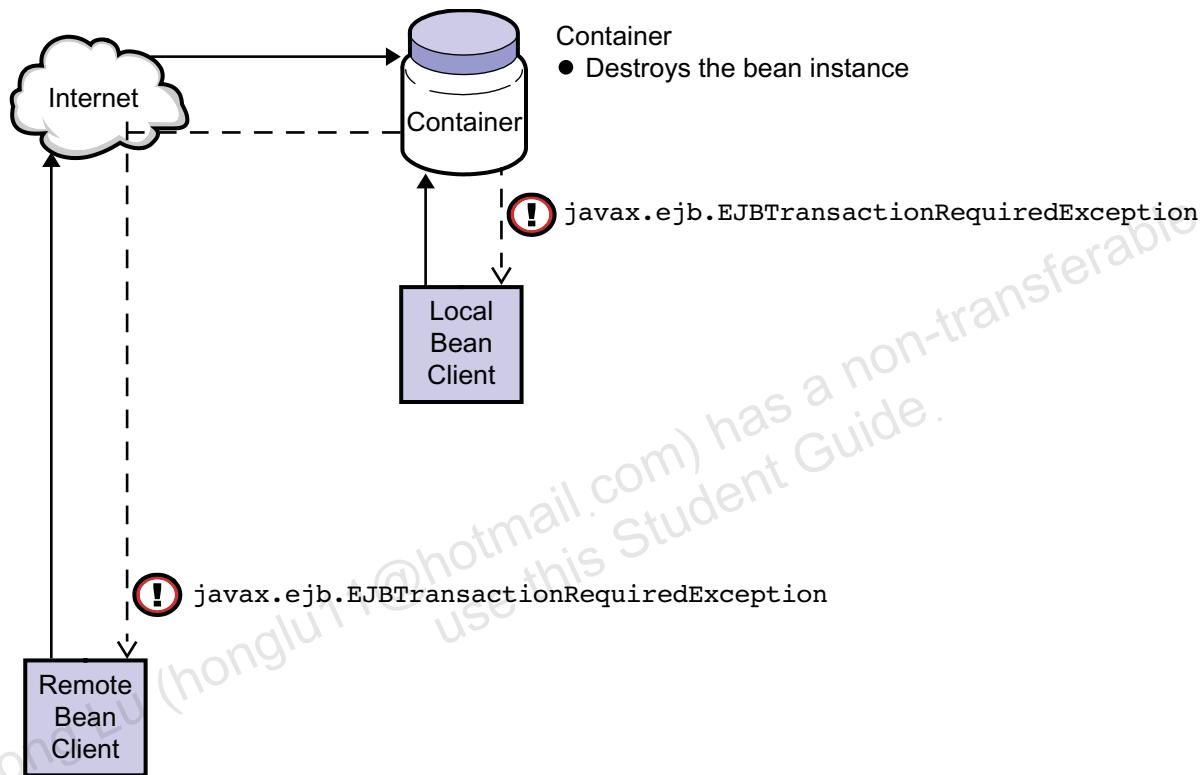


Figure 12-16 Handling a Transaction-Required Exception

Figure 12-16 shows an application assembly error. The application assembler should fix the problem by reassembling the application with the correct transaction settings.

Reviewing Exception Handling in EJB Applications

Exception handling is one of those areas in which EJB technology developers do not always exploit the capabilities of the EJB architecture to the fullest. The distinction between system exceptions and application exceptions is important because these exceptions have different effects on transaction management. In general, the enterprise bean should avoid becoming involved in the handling of system exceptions. The enterprise bean should, however, be able to throw a system exception to the container in appropriate circumstances. The container has a strategy for dealing with system exceptions, and the EJB technology should generally not attempt to subvert this.

The following notes summarize the issues you should consider when you write enterprise bean exception handling code:

- For the most part, exception handling in enterprise beans is the same as in ordinary Java technology programming.
- Enterprise beans divide exceptions into application and system exceptions.
- Complexities arise because of the use of RMI and the intervention of the container.
- The interaction between exceptions and transaction management is an important one, and developers should strive to become familiar with this interaction.
- There are differences between the local client view and the distributed client view. The local client view was intended to be used for tightly coupled entity beans.

Module 13

Using Timer Services

Objectives

Upon completion of this module, you should be able to:

- Describe timer services
- Create a timer callback notification
- Process a timer callback notification
- Manage timer objects

Additional Resources



Additional resources – The following reference provides additional information on the topics described in this module:

- Sun Microsystems, “JSR 220: Enterprise JavaBeans™, Version 3.0 EJB Core Contracts and Requirements, Chapter 18.” [<https://sdlc3e.sun.com/ECom/EComActionServlet;jsessionid=CEAAE57A3BAB8A76D4555E3C5A1F4031>], accessed July 25, 2006.
- Sun Microsystems, “JSR 220: Enterprise JavaBeans™, Version 3.0 EJB 3.0 Simplified API.” [<https://sdlc3e.sun.com/ECom/EComActionServlet;jsessionid=CEAAE57A3BAB8A76D4555E3C5A1F4031>], accessed July 25, 2006.

Introducing Timer Services

The EJB 2.1 specification was the first EJB specification to introduce timer service functionality to enterprise beans. A timer service is a container-provided service that enables the bean provider to register enterprise beans for timer callbacks. An enterprise bean can create a timer callback to occur at a specified time, after a specified elapsed time, or at specified intervals.

Using the timer functionality tasks, an enterprise bean can:

- Create a timer callback notification
- Process a timer callback notification
- Interrogate a timer callback notification object
- Obtain a list of outstanding timer notifications
- Cancel a timer notification

Timer functionality is applicable to the following types of enterprise beans:

- Stateless session beans

A timer created by a stateless session bean is associated with the stateless session bean type. When the timer expires, the container invokes the designated timer callback method on an instance of the specific stateless bean selected from the pool.

- Message-driven beans

A timer created by a message-driven bean is associated with the message-driven bean type. When the timer expires, the container invokes a designated timer callback method on an instance of the specific message-driven bean selected from the pool.

A timer callback method is designated using one of the following options.

- Annotate the method with the `Timeout` annotation.
- Implement the `javax.ejb.TimedObject` interface. This results in the timer invoking the `ejbTimeout` method.

Note – Timer functionality is available for EJB 2.1 entity beans but not for EJB 3.0 entity classes.



Creating a Timer Callback Notification

This section examines the types of timers available to enterprise beans and describes how to create a timer callback notification.

Examining Types of Timers

Java EE application servers support four types of timer notifications. Two of these timer notifications are single event timers. The other two timer notifications are interval (repeat event) timers.

Figure 13-1 shows an absolute time single event notification timer. You use this type of timer when an enterprise bean requires an absolute time notification.



Figure 13-1 Absolute Time Single Event Notification Timer

To create an absolute time notification timer, you use the `TimerService` interface method: `createTimer(Date expiration, Serializable info)`.

Note – You use the serializable `info` parameter to store information in the timer object for use when the timer event expires. You can use this information to assist with the identification of the timer or as data to process the timer event.



Figure 13-2 shows a relative time single event notification timer. You use this type of timer when an enterprise bean requires a relative time notification.

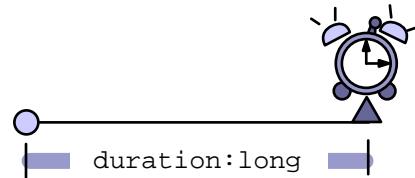


Figure 13-2 Relative Time Single Event Notification Timer

To create a relative time notification timer, you use the TimerService interface method: `createTimer(long duration, Serializable info)`. You specify the duration parameter in milliseconds.

Figure 13-3 shows an interval timer with an initial absolute time-based notification. Subsequent timer events are interval based, relative to the first timer event notification.

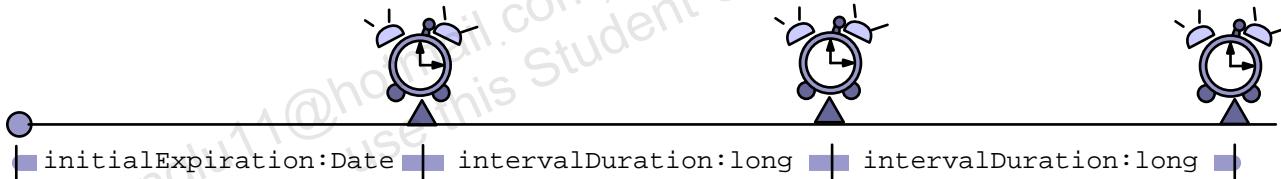


Figure 13-3 Interval Timer With Initial Absolute Time Trigger

To create an interval timer with an initial absolute time-based notification, you use the TimerService interface method:
`createTimer(Date initialExpiration, long intervalDuration, Serializable info)`.

Creating a Timer Callback Notification

Figure 13-4 shows an interval timer with an initial relative time-based notification. Subsequent timer events are interval based, relative to the initial timer event.

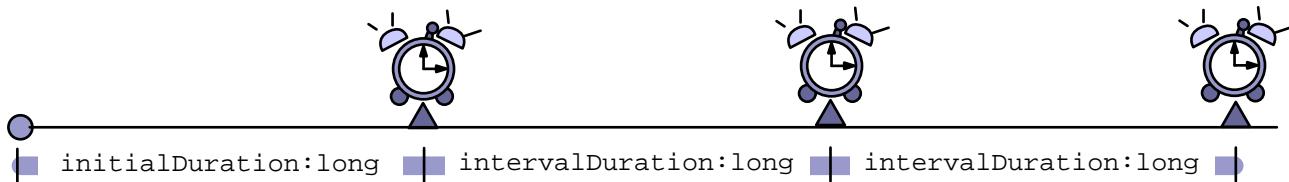


Figure 13-4 Interval Timer With Initial Relative Time Trigger

To create an interval timer with an initial relative time-based notification, you use the TimerService interface method:

```
createTimer(long initialDuration, long intervalDuration,  
Serializable info).
```

Creating a Timer Object

Figure 13-5 contains a sequence diagram illustrating the creation of a timer callback notification.

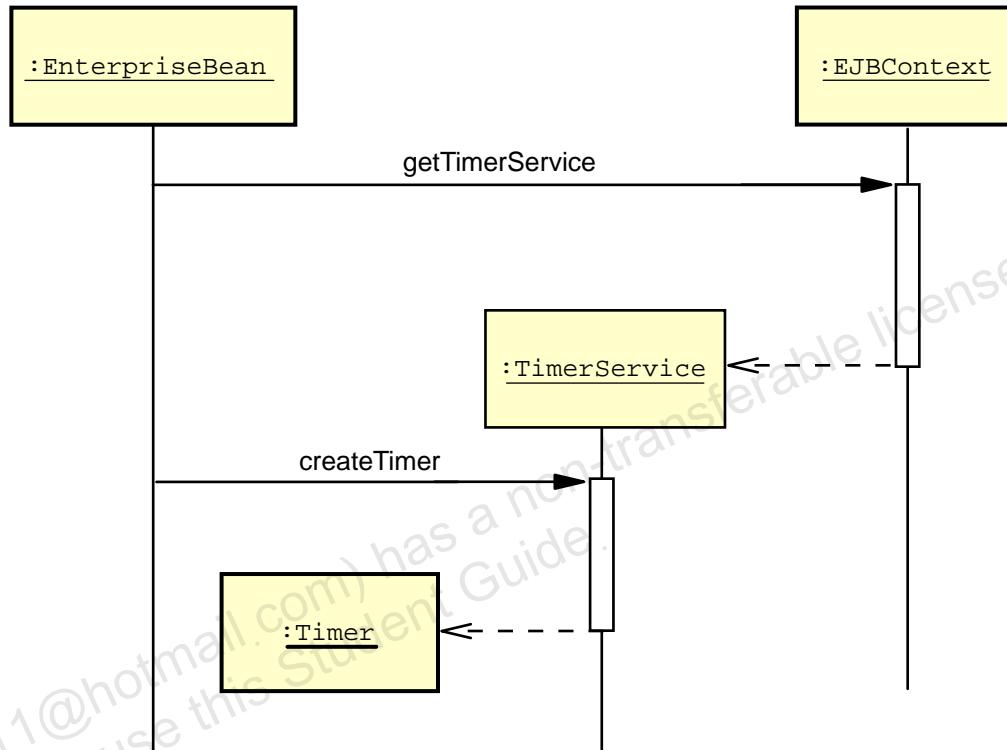


Figure 13-5 Creating a Timer Callback Notification

The following steps summarize the process shown in Figure 13-5:

1. Obtain a reference to a `TimerService` object.

You obtain a `TimerService` object by invoking the `getTimerService` method on the `EJBContext` object that is associated with the enterprise bean.

Alternatively, you can use resource injection to obtain a `TimerService` object. For example,

```
@Resource TimerService timerService;
```

2. Create a `Timer` object.

You create a `Timer` object by invoking one of the four `createTimer` methods on the `TimerService` object obtained in Step 1.

Creating a Timer Callback Notification

As preparation for this step, you need to determine the type of timer required and the information to be stored in the `info` parameter of the `createTimer` method.

Note – The `createTimer` method executes in the scope of the transaction of the invoking method. If the transaction rolls back, the container undoes the creation of the timer.



Processing a Timer Callback Notification Object

To process a timer callback notification, you are required to provide one of the following two alternatives.

- An enterprise bean class that implements the `TimedObject` interface
- A method (in the enterprise bean class) with the `@Timeout` annotation that conforms to the following method signature:

```
@Timeout  
public void method_name(Timer timer) { }
```

Creating the Timer Handler: Implementing the `TimedObject` Interface

Figure 13-6 illustrates the processing of a timer callback notification.

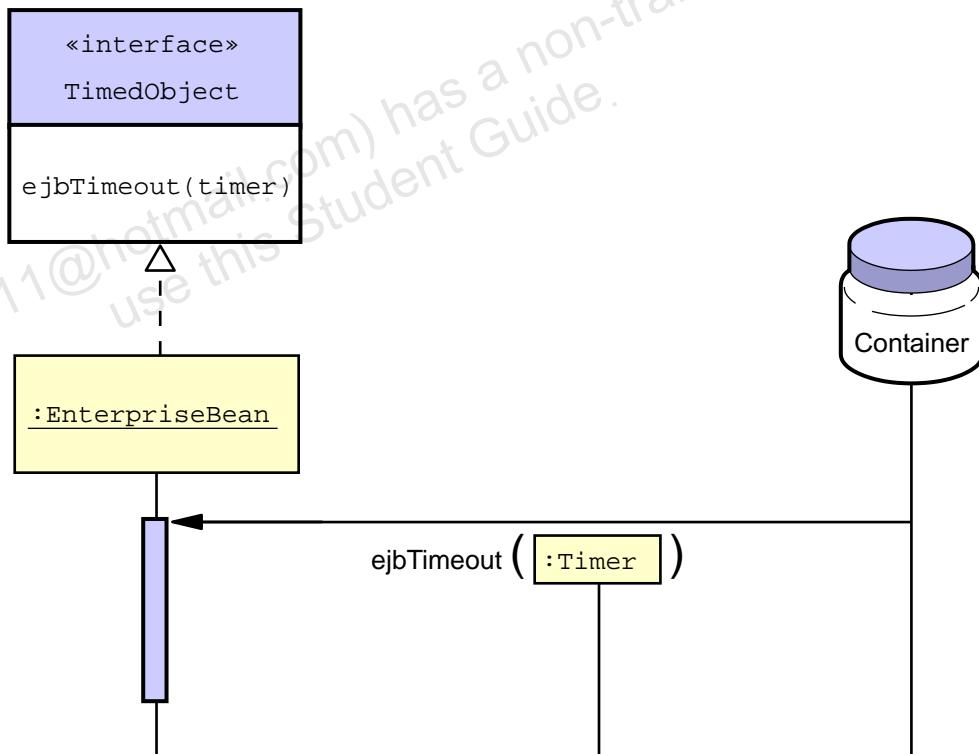


Figure 13-6 Processing a Timer Callback Notification

Processing a Timer Callback Notification Object

The TimedObject interface contains a single method called `ejbTimeout(Timer timer)`. When the timer expires, the container notifies the enterprise bean by invoking the `ejbTimeout` method. You include the code to process the timer notification in the body of the `ejbTimeout` method.

Code 13-1 shows an example of a stateless session bean implementing the `TimedObject` interface.

Code 13-1 Implementing TimedObject Interface Example

```

1  @Stateless public class ReportGeneratorBean implements
2      ReportGenerator, TimedObject {
3          private @Resource SessionContext ctx;
4
5          public void createReportTimer(Date firstDate, long interval,
6              String reportName) {
7              TimerService timerService = ctx.getTimerService();
8              timerService.createTimer(firstDate, interval, reportName);
9          }
10
11         public void ejbTimeout(Timer timer) {
12             String reportName = timer.getInfo();
13             // add code here to generate report with reportName
14         }
15     }

```

Creating the Timer Handler: Using the Timeout Annotation

Code 13-2 shows an example of an stateless session bean using the `Timeout` annotation.

Code 13-2 Using Timeout Annotation Example

```

1  @Stateless public class ReportGeneratorBean implements
2      ReportGenerator {
3          private @Resource SessionContext ctx;
4
5          public void createReportTimer(Date firstDate, long interval,
6              String reportName) {
7              TimerService timerService = ctx.getTimerService();
8              timerService.createTimer(firstDate, interval, reportName);
9          }

```

```
10
11     @Timeout
12     public void reportGenerator(Timer timer) {
13         String reportName = (String)timer.getInfo();
14         // add code here to generate report with reportName
15     }
16 }
```

Guidelines for Coding Timer Handler Method

When coding the timer handler method, you should consider the impact of the following factors:

- Timer identification

If you associate multiple timers with an enterprise bean, you must also apply a strategy to identify the timer that issues the notification. You can use the `info` parameter of the `createTimer` method to develop your identification and response strategy.

- Application server shutdown

A timer is guaranteed to survive application server crashes and shutdowns. Timers that expire during application server shutdowns are redelivered on server restart. In most cases, the application server, on power up, notifies the enterprise bean of all expired timers. For interval timers that have expired multiple times, an application server might issue a single notification.

- Transaction settings and rollback

The `ejbTimeout` method is invoked by the container. Use the `Required` or `RequiresNew` transaction attribute if you require the `ejbTimeout` method to execute in a transaction.

If the `ejbTimeout` method executes in a transaction context, you must code the `ejbTimeout` method to handle a transaction rollback. A transaction rollback causes the container to reinvoke the `ejbTimeout` method at least once.

- Security context

The `ejbTimeout` method is an internal method of the bean class. It has no client security context. The bean provider should use the `run-as` deployment descriptor element to specify a security identity to invoke methods from within the `ejbTimeout` method.

Managing Timer Objects

This section describes how you can perform the following tasks on timer objects.

- Interrogate a timer callback notification
You can perform this task in any method that contains a valid reference to a timer object.
- Obtain a list of outstanding timer notifications
You can perform this task in any method that contains a valid reference to a TimerService object.
- Cancel a timer notification
You can perform this task in any method that contains a valid reference to a timer object. Cancel can be performed only by the bean that created the timer.

Interrogating a Timer Callback Notification

Figure 13-7 illustrates how to obtain information from a timer object.

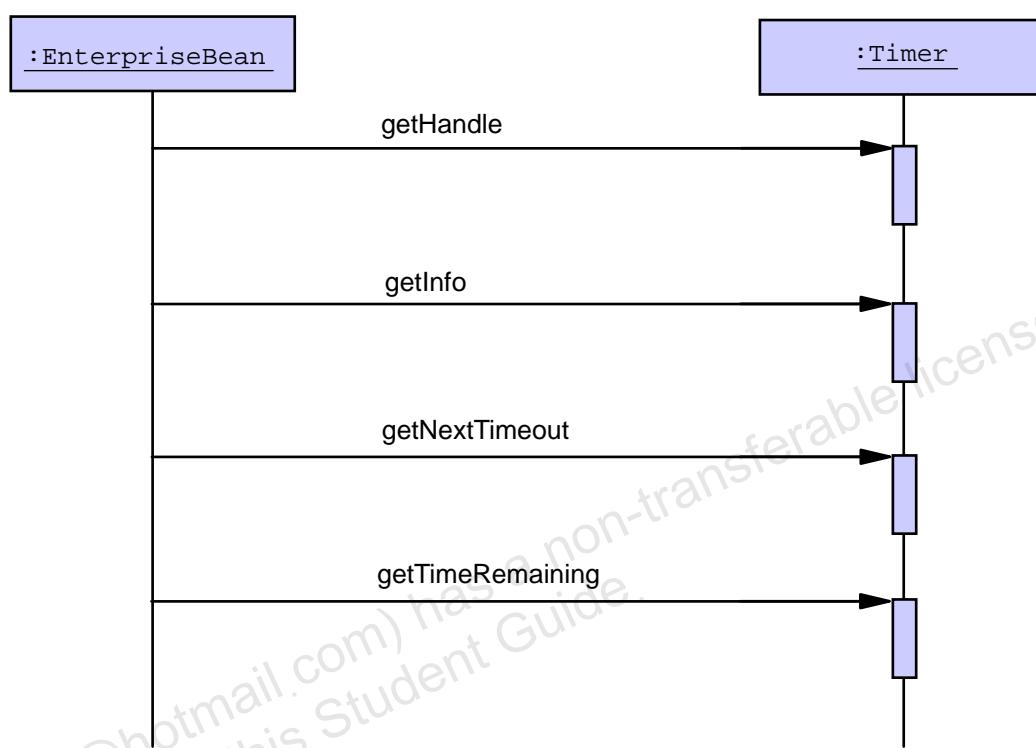


Figure 13-7 Obtaining Information From a Timer Object

You can use the following methods of the `Timer` class to extract information associated with the timer object.

- `getHandle`

Returns a serializable handle to the timer object. By persisting this handle, you can obtain a reference to the timer object at a future date.

- `getInfo`

Returns the `info` object associated with the timer during the timer creation. The `info` object can contain any kind of information as long as it is serializable.

Managing Timer Objects

- `getNextTimeout`
Returns a date object representing the absolute time to the next scheduled timer expiration.
- `getTimeRemaining`
Returns a long value representing the time in milliseconds to the next scheduled timer expiration.

Obtaining a List of Outstanding Timer Notifications

Figure 13-8 illustrates the `getTimers` method call that an enterprise bean issues on the `TimerService` object to obtain a collection of outstanding timers associated with the enterprise bean. The enterprise bean obtains a reference to the `TimerService` object by invoking the `getTimerService` method on its associated `EJBContext`.

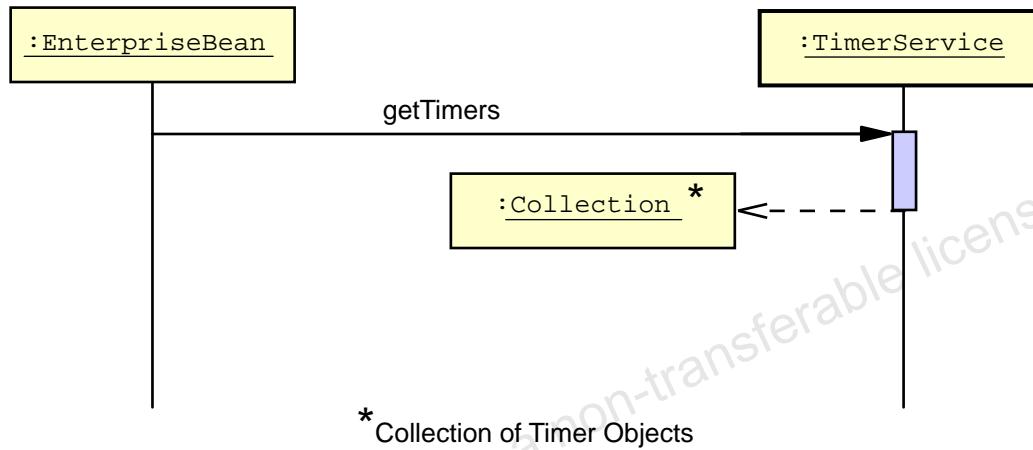


Figure 13-8 Obtaining Outstanding Timer Notifications

After obtaining the collection of outstanding timer objects you can:

- Interrogate one or more timers to perform the tasks described in “Interrogating a Timer Callback Notification” on page 13-13.
- Cancel one or more timers.

Module 14

Implementing Security

Objectives

Upon completion of this module, you should be able to:

- Understand the Java EE security architecture
- Authenticate the caller
- Examine Java EE authorization strategies
- Use declarative authorization
- Use programmatic authorization
- Examine the responsibilities of the deployer

Additional Resources



Additional resources – The following references provide additional information on the topics described in this module:

- Sun Microsystems, “JSR 220: Enterprise JavaBeans™, Version 3.0 EJB Core Contracts and Requirements, Chapter 17.” [<https://sdlc3e.sun.com/ECom/EComActionServlet;jsessionid=CEAAE57A3BAB8A76D4555E3C5A1F4031>], accessed July 25, 2006.
- Sun Microsystems, “JSR 220: Enterprise JavaBeans™, Version 3.0 EJB 3.0 Simplified API.” [<https://sdlc3e.sun.com/ECom/EComActionServlet;jsessionid=CEAAE57A3BAB8A76D4555E3C5A1F4031>], accessed July 25, 2006.

Understanding the Java EE Security Architecture

Most applications have security requirements. The application usually needs to be able to identify the user, decide what operations the user is allowed to perform, and maintain the confidentiality and the integrity of the data that is in transit.

Understanding the Java EE Security Architecture

Figure 14-1 illustrates the runtime security interventions performed by the Java EE infrastructure when responding to a method invocation by a client.

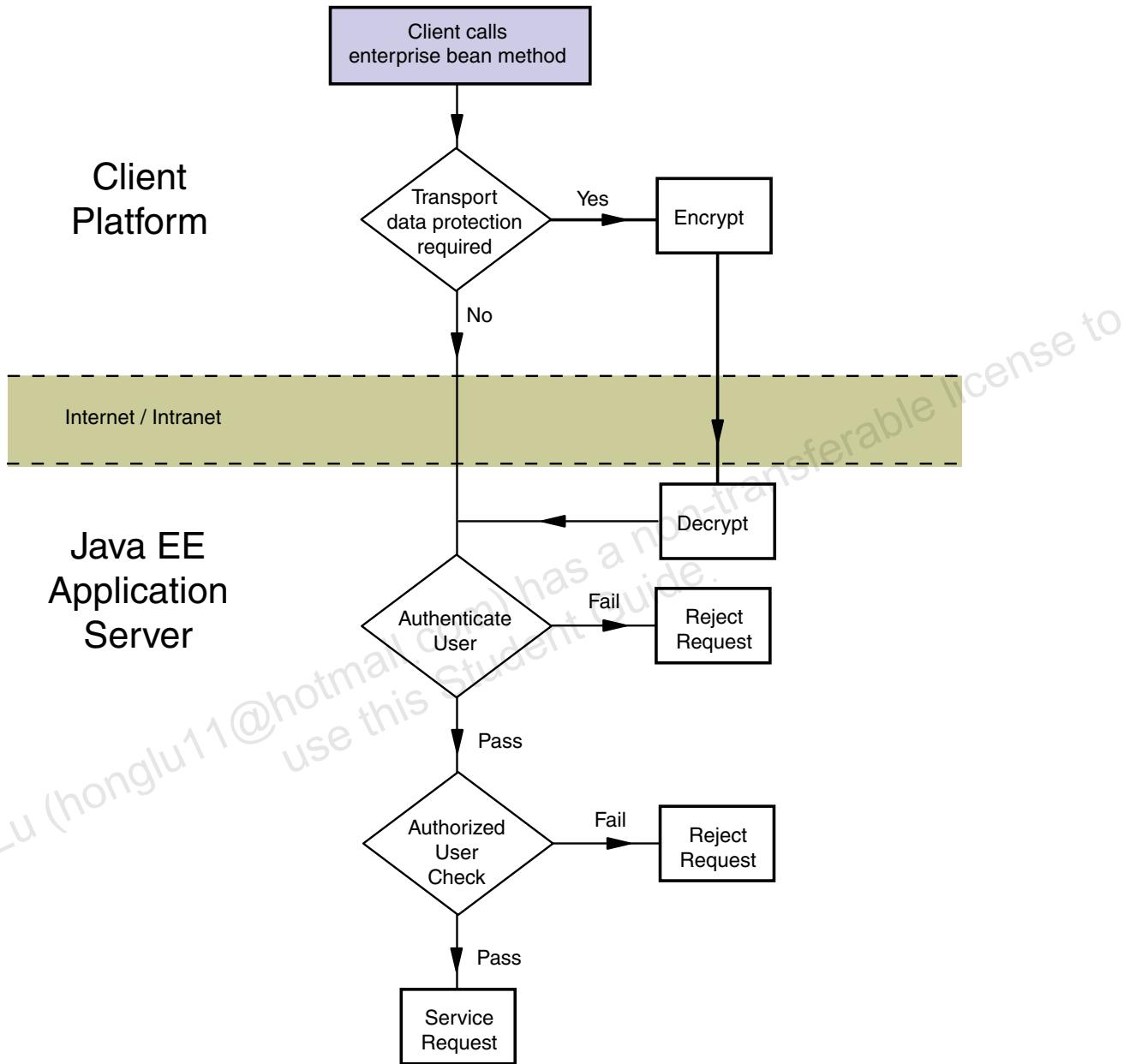


Figure 14-1 Security Interventions

These security interventions can be described as follows:

- Transport data protection

Transport security ensures the confidentiality and integrity of the information transported across the network. Systems use a combination of certificate exchange and encryption to achieve this security.

EJB, web-client and application-client containers are required to support both Secure Sockets Layer (SSL 3.0) protocol and the related Transport Layer Security (TLS 1.0) protocol for the Internet Inter-Orb Protocol (IIOP).

- Caller identification and authentication

Identification and authentication is the process of verifying what the user's identity is, and ensuring that they are who they say they are. Most systems use password challenges, client certificates, or digests to perform this task.

- Access control implementation for resources

Access control implementation for resources (authorization) determines who is allowed to do what. Authorization is the process of allocating permissions to authenticated users. These permissions give a user access to resources and services in the native domain.

A primary goal of the Java EE platform is to relieve the application developer from the details of security mechanisms and to facilitate the secure deployment of an application in diverse environments. When you use the Java EE security model, you increase development efficiency and portability.

Understanding the Java EE Security Architecture

The following list summarizes the characteristics of Java EE security. Java EE security:

- Contains no reference to the real security infrastructure
This makes the application portable.
- Can use declarative or programmatic authorization controls
- Maps the security policy to the target security domain
This is performed in an application server-specific way.
- Provides end-to-end security
End-to-end security, means that if security credentials are gathered in one part of the application, they become available to other parts. For example, if the user is using a web browser to interact with a servlet-based application, and the servlet application calls enterprise beans, then all the components (servlets and enterprise beans) that are invoked in a particular user interaction *see* the same user.

The rest of this module focuses on the authentication and authorization aspects of the Java EE technology security model.

Authenticating the Caller

Caller authentication is the process of verifying what the user's identity is, and ensuring that they are who they say they are. The authentication process consists of the following two phases.

- Establish user identities
- Authenticate caller against an established identity

Most systems use password challenges, client certificates, or digests to perform this task.

Establishing User Identities

The process of caller authentication requires that users of an application be known in advance to the security system. The Java EE specification recognizes the following two types of user identities:

- Principals
- Roles

Principal

A principal is an authenticated user in the Java EE security domain. That is, a principal is an entity that can be authenticated by an authentication protocol in a security service that is deployed in an enterprise. A principal is identified using a principal name and authenticated using authentication data. The content and format of the principal name and the authentication data can vary depending upon the authentication protocol.

Authenticating the Caller

A principal represents a user in the native security domain. A user from the native domain maps to a principal in the Java EE security domain. Figure 14-2 shows the mapping of users in the native security domain to principals in the Java EE security domain.

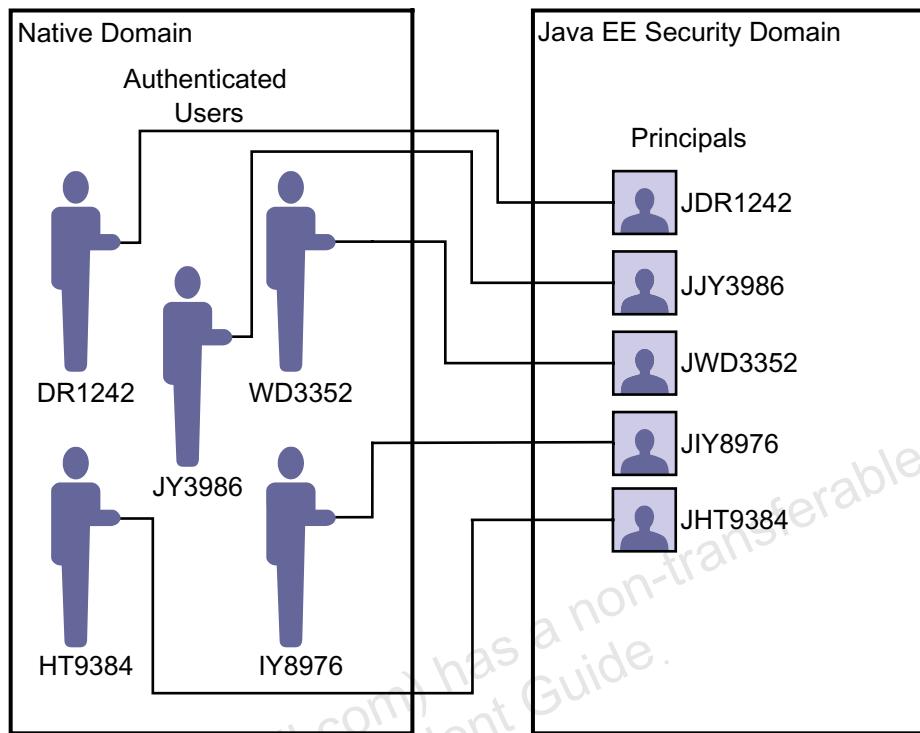


Figure 14-2 Mapping Native Domain Users to Principals

Mapping users in the native domain to principals is outside the scope of the Java EE technology specification. The specification delegates this task to the vendor-supplied tools of the application server. In addition, the specification does not define how the users' credentials are stored. In practice, most applications use either a directory server or a relational database. However, this is irrelevant to the developer. The Java EE security model abstracts away the implementation details.

The EJB specification designates the task of mapping native users to principals to the deployer.

Role

A role is a group of principals sharing a common set of permissions. A role has predefined credential requirements associated with it. The role membership is defined in the target environment, and roles are administered using application server vendor-supplied tools.

Figure 14-3 shows the mapping of principals to roles.

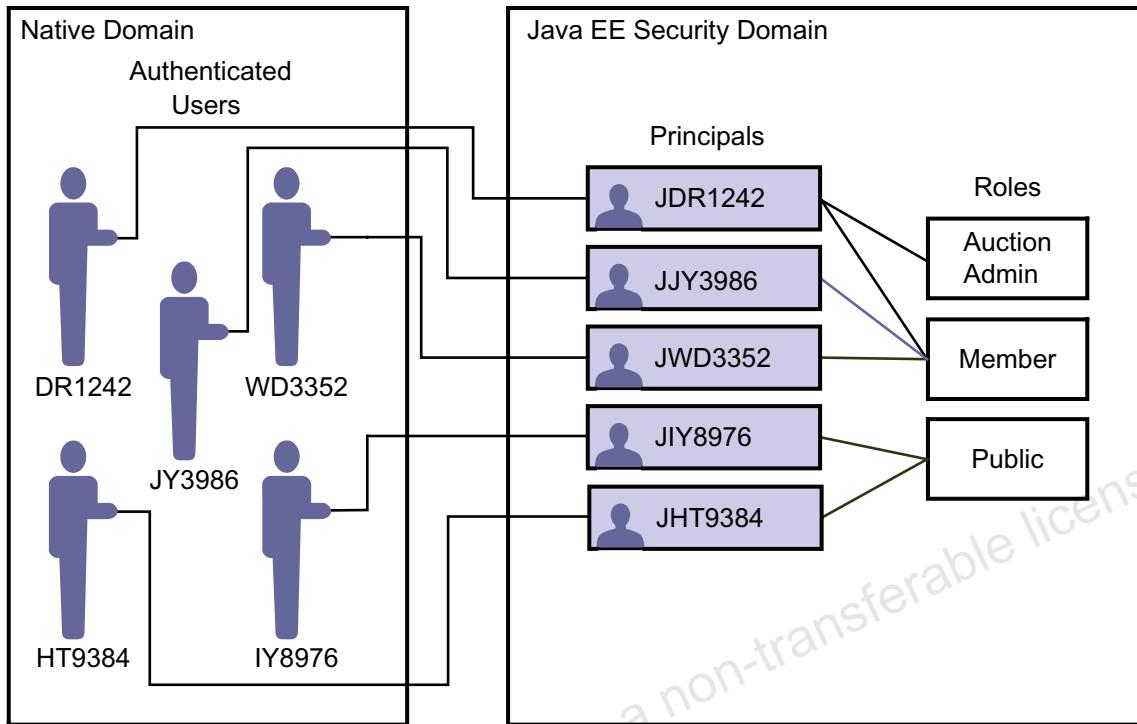


Figure 14-3 Mapping Principals to Roles

Each role has a set of method permissions associated with it. A method permission provides any member of the role with the required authorization to execute the method. The application assembler or the deployer can use either metadata annotations or the DD to associate method permissions with a role name. For more details on method permissions see “Declaring Method Permissions” on page 14-16.

Most real application servers work with security infrastructures that have their own concept of *group* or *role*. It is normally possible to make mappings between groups and Java EE technology roles, rather than working at the user or principal level. This ability is much more convenient.

The EJB specification designates the task of mapping principals to roles to the deployer.

Authenticating the Caller

Caller Authentication

The purpose of Java EE caller authentication is to authenticates the caller to a principal. The authentication is performed using input data supplied by the caller against credentials of the corresponding principal stored in the security infrastructure. The Java EE specification leaves the choice of the caller authentication mechanism to the deployer.

At minimum, the user requires some form of user interface to enter the data used to authenticate the user. To assist the deployer, the application server can provide authentication mechanisms of various levels of sophistication. Figure 14-4 illustrates a user name, password based authentication scheme.

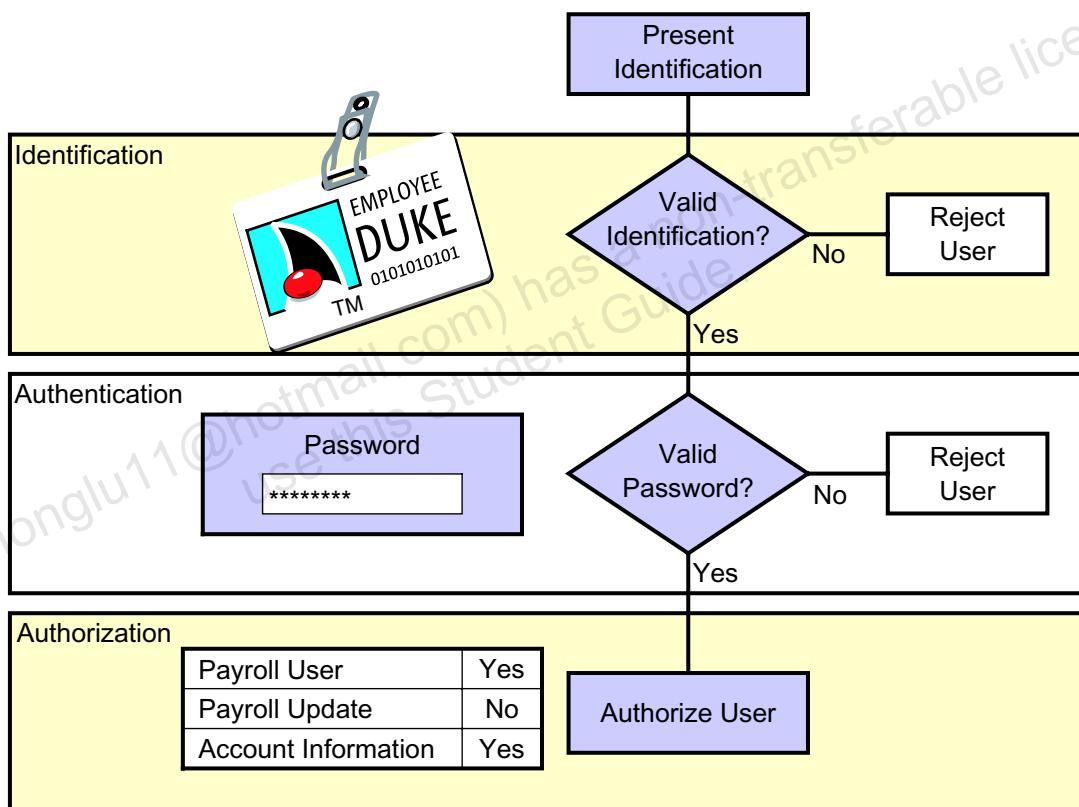


Figure 14-4 User Name Password Based Authentication

To allow for maximum flexibility, all Java EE application servers must implement the Java Authentication and Authorization Service (JAAS) API. JAAS enables services to authenticate and enforce access controls upon users. JAAS implements a Java technology version of the standard Pluggable Authentication Module (PAM) framework, and extends the access control architecture of the Java platform to support user-based authorization. The Java Authorization Service Provider Contract for Containers (JACC) defines a contract between a Java EE application server and an authorization service provider, allowing custom authorization service providers to be plugged into any Java EE product.

Examining the Java EE Authorization Strategies

The primary purpose of the Java EE security model is to control access to business services and resources in the Java EE technology application server. Business services are implemented as methods in enterprise beans. Resources are back-end services implemented by the Java EE technology server. For example, a Java EE server can offer persistence service using a database. The database is considered a resource.

The Java EE security model provides two complementary strategies for access control: programmatic access control and declarative access control. Both strategies assume that the user has been authenticated by the application server, and the roles of which the user is a member can therefore be determined by the EJB container.

Figure 14-5 provides a high-level comparison of the Java EE authorization strategies.

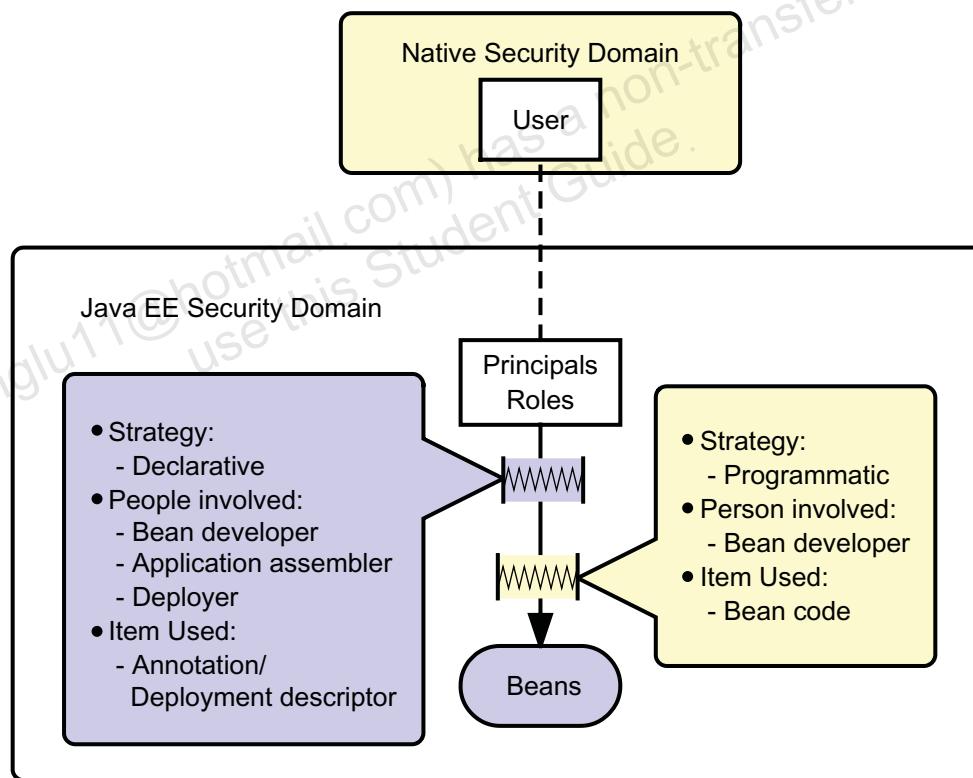


Figure 14-5 Overview of Java EE Authorization Strategies

With programmatic access control, access control decisions are made in the code using API calls. With declarative access control, access control decisions are made outside the enterprise bean execution code when the EJB container delegates control to the EJB technology at the start of a method call. The decisions are made on the basis of a security policy that is set up by the developer or the application assembler using either security-related annotations in the bean code or security-related elements in the DD. Before the EJB 3.0 specification, the setting declarative security policies was restricted to the DD.

Remember, using programmatic access control in the Java EE security model does not necessarily mean that you write code that interacts with the target (legacy) security infrastructure. Even with programmatic authorization, the application server remains responsible for authentication against some authentication database and the extraction of the roles assigned to particular principals. You have only limited programmatic control of authorization.

Figure 14-6 illustrates the differences between the programmatic authorization strategy and the declarative authorization strategy.

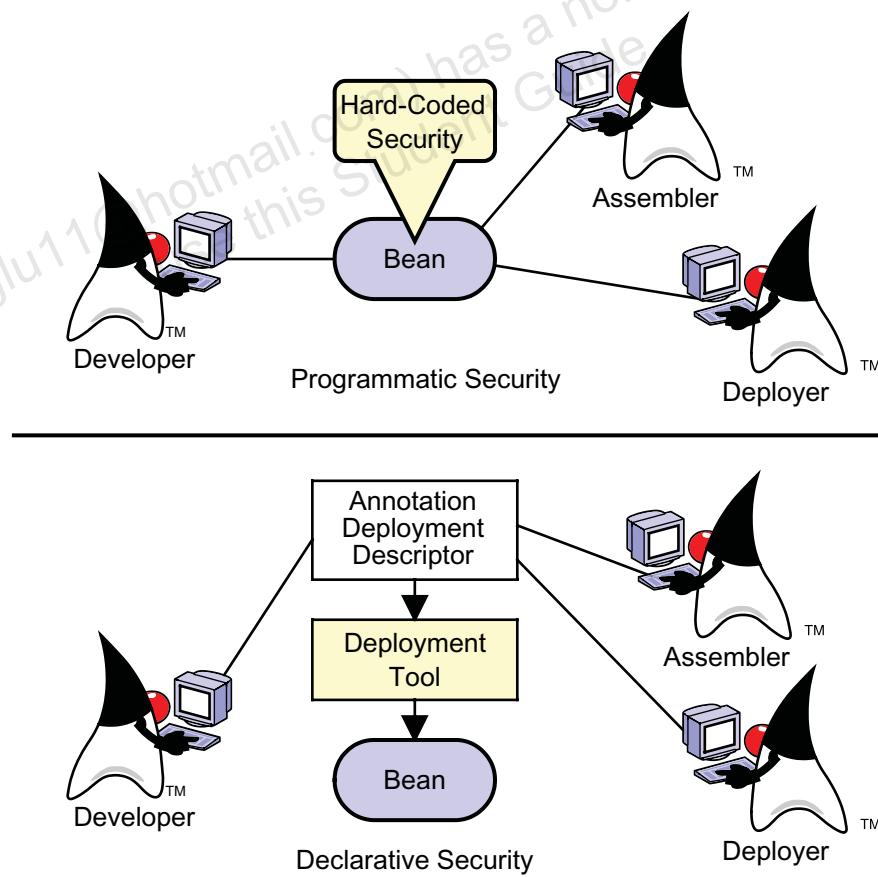


Figure 14-6 Differences Between the Authorization Strategies

Using Declarative Authorization

Declarative authorization can be configured by the assembler and deployer and, in many cases, by the bean provider. Usually the bean provider sets up some initial security properties when developing or testing the enterprise bean. The assembler and deployer can then modify these properties according to the needs of the application as a whole. The declarative security policy can be defined in the bean code using metadata annotations or in the DD. The use of vendor-supplied packaging and deployment tools can be helpful here, because they can provide a basic, graphical representation of the security policy.

Declarative authorization involves the following tasks:

- Security role declaration
- Method permission assignment
- Security identity assignment

Declaring Security Roles

You declare a security role using either or a combination of the following options.

- Use `DeclareRoles` annotation or the `RolesAllowed` annotation or both

Either or both annotations can be used to define security roles. If both annotations are used, the net effect is the aggregation of the role names declared in the `DeclareRoles` annotation and `RolesAllowed` annotation.

Code 14-1 demonstrates the use of the `DeclareRoles` annotation.

Code 14-1 The `DeclareRoles` Annotation

```
1  @DeclareRoles(value = {"auction-user", "auction-admin"})  
2  @Stateless()  
3  public class AuctionManagerBean
```

- Use the security-role DD element.

Code 14-2 is a snippet of a DD that contains the declaration of the *auction-admin* security role.

Code 14-2 The security-role Element

```
1 <assembly-descriptor>
2   <security-role>
3     <description>Needs to be able to create and delete arbitrary
4       auctions.
5     </description>
6     <role-name>auction-admin</role-name>
7   </security-role>
8
9   <security-role>
10    <description>...</description>
11    <role-name>...</role-name>
12  </security-role>
```

Security roles are used in method permission assignments and optionally in security identity assignments.

Declaring Method Permissions

A method permission authorizes one or more roles to execute an enterprise bean method. Figure 14-7 shows a graphical view of enterprise bean methods and the security roles authorized to execute them.

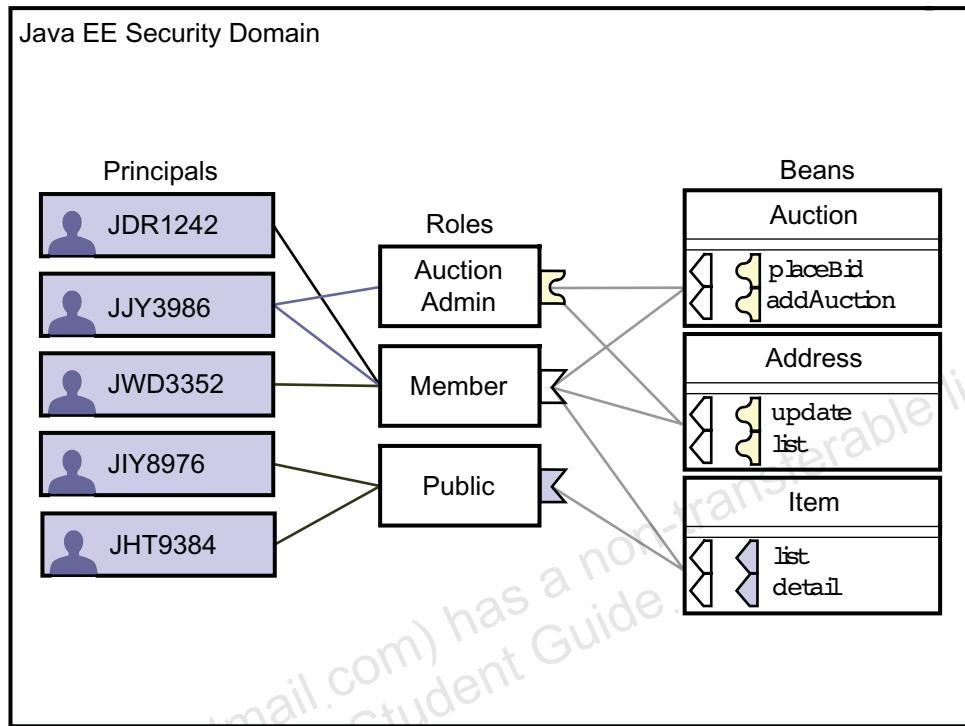


Figure 14-7 Method Permissions

You declare a method permission by using the `@RolesAllowed` metadata annotation or using the `method-permission` DD element.

Note – The failure to allocate a method permission to a method results in the container granting all roles access to that method.



Declaring Method Permissions Using Metadata Annotations

You can use the following annotations to specify method permissions. You can apply these annotations to a bean class or to a method. If applied to a bean class, they apply to all applicable business methods of the bean class.

- PermitAll

The PermitAll annotation enables all roles to execute the method(s).

- DenyAll

The DenyAll annotation specifies that no security roles are permitted to execute the specified method(s).

- RolesAllowed

The RolesAllowed annotation is a list of security role names to be mapped to the security roles that are permitted to execute the specified method(s). The RolesAllowed annotation can be applied to all methods of a class or to selected methods of a class.

Code 14-3 contains an example of the application of the RolesAllowed annotation to a class. The RolesAllowed(admin) annotation applies to all methods of the a Bean class.

Code 14-3 The RolesAllowed Annotation Applied to a Class

```

1 @Stateless
2 @RolesAllowed("admin")
3 public class aBean implements A{
4     public void aMethod () {...}
5     public void bMethod () {...}
6     ...
7 }
```

Code 14-4 contains an example of the application of the RolesAllowed annotation to a method. The RolesAllowed(HR) annotation applies to the cMethod only. The dMethod is not allocated an explicit method permission.

Code 14-4 The RolesAllowed Annotation Applied to a Method

```

1 @Stateless public class bBean implements B{
2     @RolesAllowed("HR")
3     public void cMethod () {...}
4     public void dMethod () {...}
5     ...
6 }
```

Using Declarative Authorization

Declaring Method Permissions Using the DD

The DD equivalent of the RolesAllowed annotation is the method-permission element. You can apply the method-permission element to all the methods in an enterprise bean or to specific methods in an enterprise bean.

Code 14-5 is a DD snippet that contains a method-permission element applied to all the methods in an enterprise bean class.

Code 14-5 The method-permission Element Applied to the Bean Class

```

1 <method-permission>
2   <role-name>auction-admin</role-name>
3   <method>
4     <ejb-name>AuctionManager</ejb-name>
5     <method-name>*</method-name>
6   </method>
7 </method-permission>
```

Code 14-6 is a DD snippet that contains method-permission elements applied to specific methods.

Code 14-6 The Method-permission Element Applied to Specific Methods

```

1 <method-permission>
2   <role-name>auction-admin</role-name>
3   <method>
4     <ejb-name>AuctionManager</ejb-name>
5     <method-name>closeAuction</method-name>
6   </method>
7 </method-permission>
8 <method-permission>
9   <role-name></role-name>
10  <unchecked/>
11  <method>
12    <ejb-name>AuctionManager</ejb-name>
13    <method-name>getAllAuctionsOfSeller</method-name>
14  </method>
15  <method>
16    <ejb-name>AuctionManager</ejb-name>
17    <method-name>getSeller</method-name>
18  </method>
19 </method-permission>
```

Declaring Security Identity

When an enterprise bean makes a call on another enterprise bean, it can either pass the identity of its own caller, or it can substitute a new caller identity. The identity that an enterprise bean uses to make a call on another enterprise bean is known as the security identity of the calling enterprise bean.

The application assembler declares the security identity for every enterprise bean in the application. In specifying the security identity, the application assembler can use one of the following identities:

- Use the caller identity
- Use an alternate identity

Specifying Use Caller Identity: The `use-caller-identity` Element

You can specify use caller identity in the DD by using the `use-caller-identity` element. The `use-caller-identity` element specifies that the caller's security identity be used as the security identity for the execution of the enterprise bean's methods. The `use-caller-identity` element cannot be specified for message-driven beans or for session beans that implement the `TimedObject` interface.

Code 14-7 is a DD snippet that demonstrates the use of the `use-caller-identity` element.

Code 14-7 Specifying the `use-caller-identity` Element

```
<security-identity>
    <use-caller-identity/>
</security-identity>
```

Note – The `use-caller-identity` element does not have an equivalent metadata annotation. An equivalent metadata annotation is not required unless the caller identity is used as the default.



Specifying an Alternate Identity: The `run-as` Element

When an enterprise bean makes a call on another enterprise bean, it can either pass the identity of its own caller, or it can substitute a new caller identity. This substitute caller identity is called the *run-as* identity. A substitute identity is required if a particular enterprise bean has privileged access to methods on other enterprise beans.

A run-as identity associates a role or principal with an enterprise bean. When a method is invoked on the enterprise bean, the container first checks the current role of the invoker against the list of roles that can execute the method. If the invoker is in the list, then the container switches the identity of the invoker to the run-as identity associated with the enterprise bean. From this point on, the new identity (run-as) is used only when the enterprise bean makes method calls on another enterprise bean.

Note – The caller identity is used throughout the execution of the method. The run-as identity is used only during the execution of all methods invoked by the method.

In practice, you usually specify the run-as identity in terms of a role, not a principal. That is, you want the enterprise bean to make subsequent method calls as a specific role, for example, the Member role, not a particular principal that has that role.



Figure 14-8 shows the use of the run-as identity.

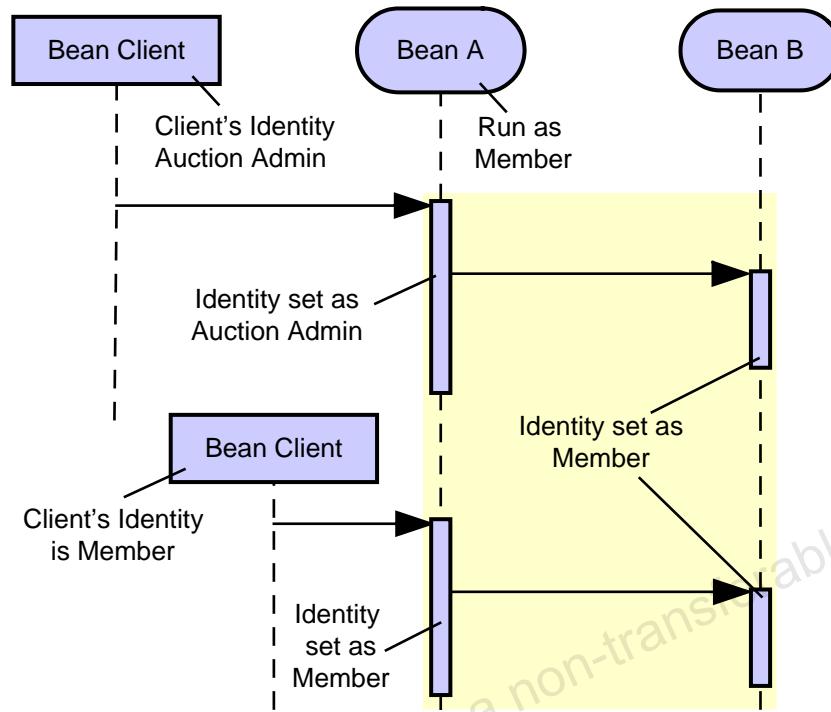


Figure 14-8 Run-as Identity

A bean can have only one run-as identity that affects all methods of the bean. During application assembly, the assembler specifies a role as the run-as identity. The deployer replaces the role with a principal.

Note – Although the run-as identity is given as a role, the IIOP, which enterprise bean servers must support for Remote Method Invocation (RMI) operations, does not have the concept of role. It does, however, know how to pass a principal in the protocol. Therefore, at deployment time, the run-as identity must be given in the form of a principal. Obviously, the deployer needs to ensure that the principal chosen is actually in the requested role. Some products can deal with this automatically, but regardless of whether they can, this is an issue that the enterprise bean developer need not be concerned about. It is a deployment or configuration issue.



Using Declarative Authorization

Code 14-8 shows a DD with the run-as element.

Code 14-8 A run-as Element Example

```
1 <enterprise-beans>
2   <session>
3     <ejb-name>AuctionManager</ejb-name>
4     ...
5     <security-identity>
6       <run-as>
7         <role-name>auction-admin</role-name>
8       </run-as>
9     </security-identity>
10    </session>
11  </enterprise-beans>
```

You can also specify the run-as identity using the RunAs metadata annotation. Code 14-9 demonstrates the use of the RunAs metadata annotation.

Code 14-9 A RunAs Annotation Example

```
1 @RunAs("auction-admin")
2 @Stateful public class AuctionManagerBean implements
3   AuctionManager {
4     // ...
5 }
```

Using Programmatic Authorization

Programmatic authorization is the responsibility of the bean developer. The following methods, in EJBContext objects passed to enterprise beans at initialization time, support programmatic authorization:

- The `isCallerInRole` method
- The `getCallerPrincipal` method

With regard to application maintenance, programmatic authorization is less flexible than declarative authorization. However, you can use programmatic authorization to provide additional security safeguards built on top of declarative authorization. Declarative authorization controls access at a role level. Programmatic authorization can selectively permit or block access to principals belonging to a role.

Using the `isCallerInRole` Method

A bean method can use the `isCallerInRole` method to verify the role of the caller. The `isCallerInRole` method allows recognition of roles without the bean trying to identify the principal. Code 14-10 contains an example of the use of the `isCallerInRole` method.

Code 14-10 Using the `isCallerInRole` Method

```
1 if(ctx.isCallerInRole("auction-admin"))
2     // allow auction to be deleted
3 else {
```

Declare auction-admin in the DD as reference.

Using Programmatic Authorization

Figure 14-9 shows the use of the `isCallerInRole` method.

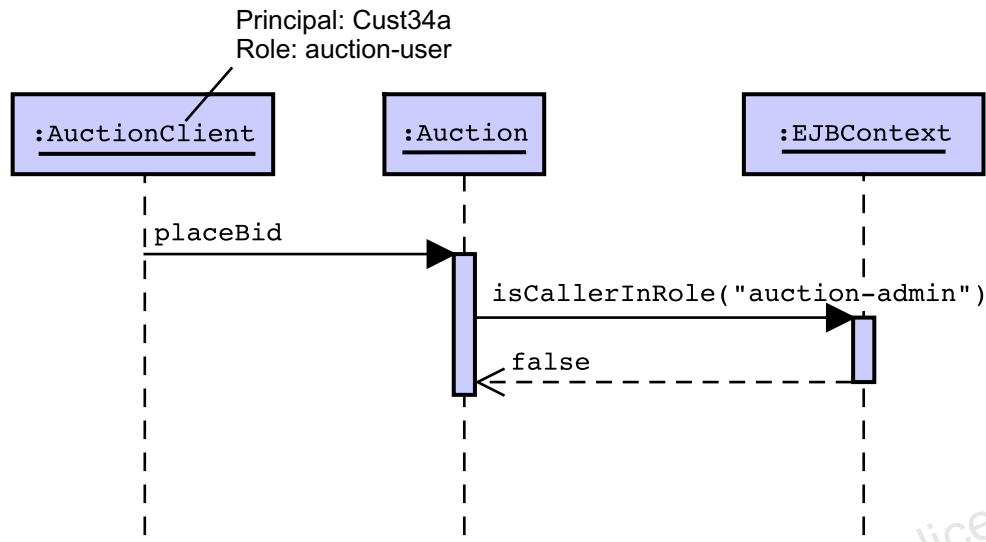


Figure 14-9 Using the `isCallerInRole` Method

Specifying Role References

Whenever you use the `isCallerInRole` method, you must also cite the role specified in the method call in a role-reference entry in the DD. The application assembler or deployer then associates the role used in the method call with a role defined in the Java EE security domain.

Although most enterprise bean developers strive to limit the use of programmatic access control in their enterprise beans, it is difficult to eliminate it completely. In any case, there are perfectly legitimate uses for the `isCallerInRole` method. For example, suppose you want to present a menu of choices to users, and the menu to display depends on the user's role. This task cannot easily be done declaratively.

Therefore, from time to time, you must use the `isCallerInRole` method. The problem is that this method takes a role name as an argument, and, if different enterprise beans are supplied by different development teams or different vendors, the teams of vendors might not have used consistent role names. Therefore, the EJB architecture provides a way for the coded role names to be reconciled with declared role names.

Whenever you use the `isCallerInRole` method, you must also cite the role specified in the method call in a role reference entry in the DD. The application assembler or deployer then associates the role used in the method call with a role defined in the Java EE security domain.

Code 14-11 shows that a security-role-ref element entry is created by the bean developer.

Code 14-11 The security-role-ref Element

```
1  <ejb-jar>
2    <enterprise-beans>
3      <entity>
4        <ejb-name>AuctionManager</ejb-name>
5        <security-role-ref>
6          <description>
7            Needs to be able to create and delete arbitrary auctions
8          </description>
9          <role-name>auction-admin</role-name>
10         </security-role-ref>
11       </entity></enterprise-beans>
12     ...
13   ...
14 </enterprise-beans>
15 </ejb-jar>
```

Code 14-11 contains a declaration of a security-role-ref element for the role name auction-admin, which is used in a bean method. During assembly, a corresponding role-link element is inserted to map this name to a declared security role, as shown in Code 14-12 on page 14-26.

Using Programmatic Authorization

Code 14-12 associates the auction-admin role used in a bean method with the Administrator role declared in the Java EE security domain.

Code 14-12 The role-link Element

```
1 <security-role-ref>
2   <description>
3     Needs to be able to create and delete arbitrary auctions.
4   </description>
5   <role-name>auction-admin</role-name>
6   <role-link>Administrator</role-link>
7
8 </security-role-ref>
```

Note – The creation of a role-link element is the task of the application assembler or deployer. Consequently, the security-role-ref and role-link elements are discussed again under the responsibilities of the application assembler in “Examining a Complete Example of the DD Security Section” on page 14-29.



Using the getCallerPrincipal Method

A bean method can use the `getCallerPrincipal` method to verify a principal. The method is used for programmatic security enforcement. Code 14-13, which contains a snippet from a DD, illustrates the use of the `getCallerPrincipal` method.

Code 14-13 The `getCallerPrincipal` Method

```
1 if(ctx.isCallerInRole("auction-admin")
2     // allow auction to be deleted
3 else {
4     Principal p = ctx.getCallerPrincipal();
5     // verify if principal is the seller
6     if (p.getName().equals(getSeller().getAuctionUserID()))
7         // allow auction to be deleted
8
9     else // principal is not the seller
10    throw new AuctionException(p.getName() + " not seller");
11 }
```

Using Programmatic Authorization

Figure 14-10 shows the use of the `getCallerPrincipal` method.

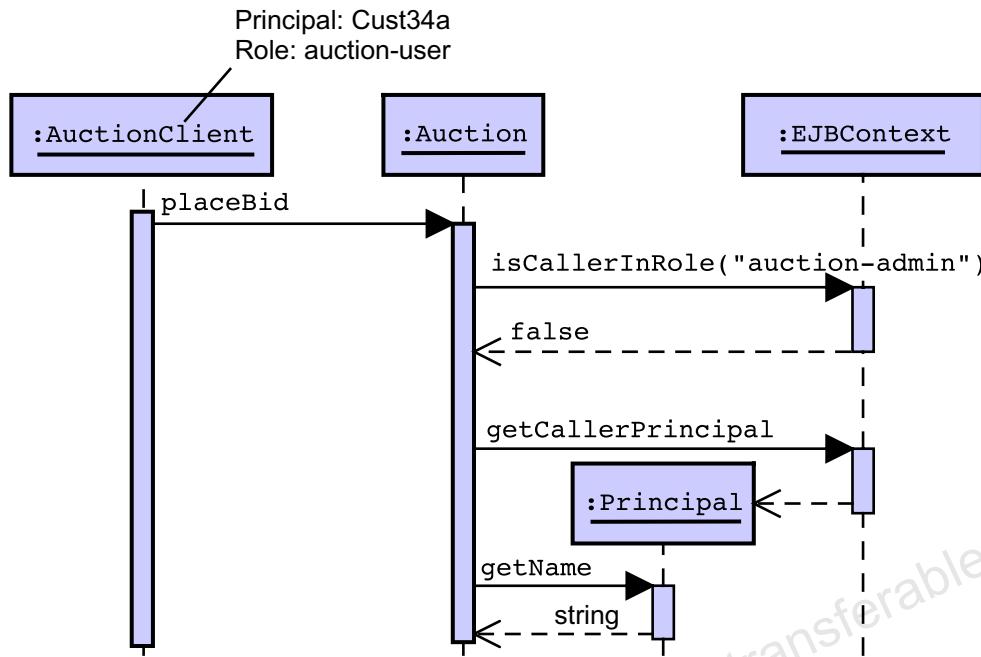


Figure 14-10 Using the `getCallerPrincipal` Method

Note – Role-based identification is preferable over principal-based identification.



Examining a Complete Example of the DD Security Section

Code 14-14 shows an example of a DD's security section.

Code 14-14 A Complete DD Security Section Code Example

```
1  <ejb-jar>
2    <enterprise-beans>
3      <entity>
4        <ejb-name>AuctionManager</ejb-name>
5        ...
6        <security-role-ref>
7          <description>
8            Needs to be able to create and delete
9              arbitrary auctions.
10             </description>
11
12             <role-name>auction-admin</role-name>
13             <role-link>Administrator</role-link>
14
15           </security-role-ref>
16         </entity>
17       </enterprise-beans>
18
19     <assembly-descriptor>
20       <security-role>
21         <description>Allows access to Auction
22           information (reads and updates)
23         </description>
24         <role-name>Administrator</role-name>
25       </security-role>
26
27       <security-role>
28         <description>...</description>
29         <role-name>...</role-name>
30       </security-role>
31
```

Examining a Complete Example of the DD Security Section

```
32     <method-permission>
33         <role-name>Administrator</role-name>
34         <method>
35             <ejb-name>AuctionManager</ejb-name>
36                 <method-name>closeAuction</method-name>
37             </method>
38     </method-permission>
39     <method-permission>
40         <role-name></role-name>
41         <unchecked/>
42         <method>
43             <ejb-name>AuctionManager</ejb-name>
44                 <method-name>getAllAuctionsOfSeller</method-name>
45             </method>
46         <method>
47             <ejb-name>AuctionManager</ejb-name>
48                 <method-name>getSeller</method-name>
49             </method>
50     </method-permission>
51
52     </assembly-descriptor>
53
54 </ejb-jar>
```

Examining the Responsibilities of the Deployer

Figure 14-11 shows the security tasks delegated to the deployer.

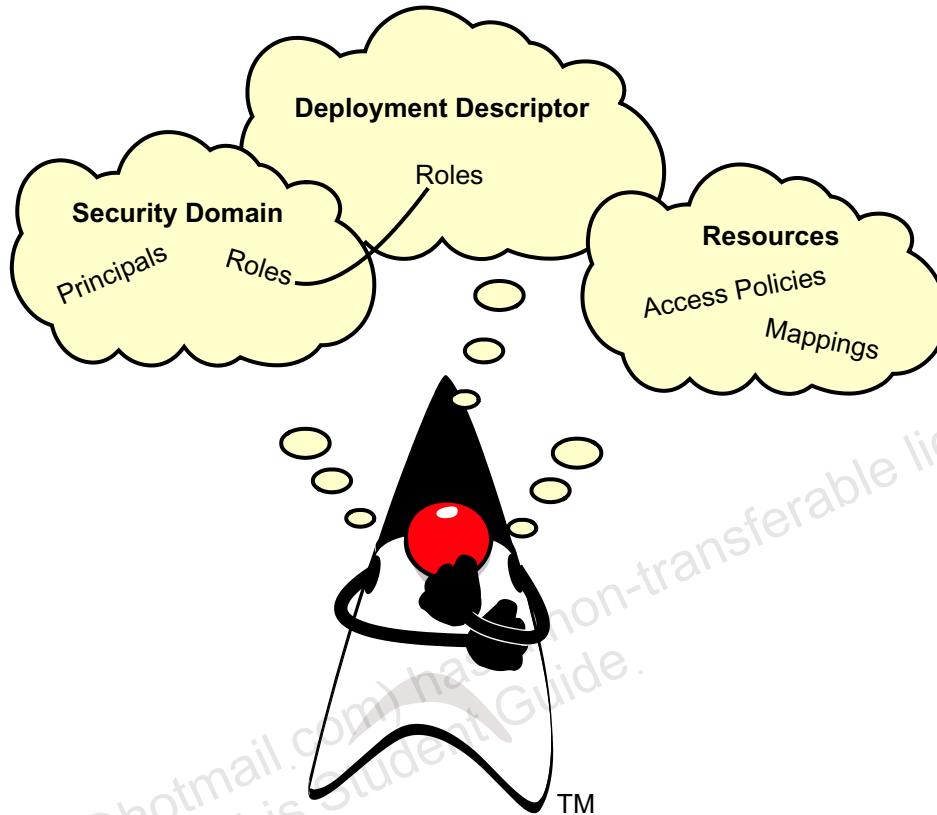


Figure 14-11 Security Tasks Delegated to the Deployer

The deployer:

- Reviews and overrides (if necessary) the application assembler's security
- Declares principals and roles in the security domain
- Ensures that all security role references in the DD are linked to roles declared in the security domain
- Configures principal delegation for inter-component calls
- Creates resource-access policies and mappings that involve using vendor-supplied tools to:
 - List roles that can access each resource
 - List principals that can access each resource
 - Specify a unique user name and password for each role/resource and principal/resource association

Reviewing the Java EE Application Security Tasks

This section summarizes the security-related tasks associated with developing a Java EE application into the following categories.

- Transport data protection

Transport security ensures the confidentiality and integrity of the information transported across the network. Use a combination of certificate exchange and encryption to achieve this security.

- Caller authentication

Identification and authentication is the process of verifying what the user's identity is, and ensuring that they are who they say they are. The authentication process consists of the following two phases.

- Establish user identities

Deployer creates principals and maps principals to users in the native domain. Deployer also maps principals to roles created by the application assembler.

- Authenticate caller against an established identity

Most systems use password challenges, client certificates, or digests to establish identity. You should examine the options provided by the application server. You should also consider using the JAAS API to create login modules.

- Caller authorization

Caller authorization is the process of imposing access control on resources. The Java EE security model provides the declarative access control and programmatic access control.

In declarative access control, the application assembler performs the following tasks:

- Security role declaration
- Method permission assignment
- Security identity assignment

Programmatic access control involves using the following methods:

- The `isCallerInRole` method

To support the use of this method, the bean developer declares the `security-role-ref` element in the DD.

- The `getCallerPrincipal` method

Module 15

Using EJB Technology Best Practices

Objectives

Upon completion of this module, you should be able to:

- Define best practices and state the benefits of using EJB technology best practices
- Select and apply known patterns to Java EE application design

Additional Resources



Additional resources – The following references provide additional information on the topics described in this module:

- Gamma Erich, Helm Richard, Johnson Ralph, Vlissides John. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.
- Marinescu Floyd. *EJB Design Patterns: Advanced Patterns, Processes, and Idioms*. Jon Wiley and Sons, 2002.
- Sun Microsystems, “Enterprise Java Technologies Tech Tips” [<http://developer.java.sun.com/developer/EJTechTips/>], accessed July 25, 2006.
- Sun Microsystems, “BluePrints Patterns,” [<http://java.sun.com/blueprints/patterns/index.html>], accessed July 15, 2006.

Defining Best Practices

The term *best practice* is generally defined as a technique, methodology or activity that has been proven through research and experience to offer significant and reliable benefits in a given field or activity.

Figure 15-1 shows the types of benefits of using EJB technology best practices.

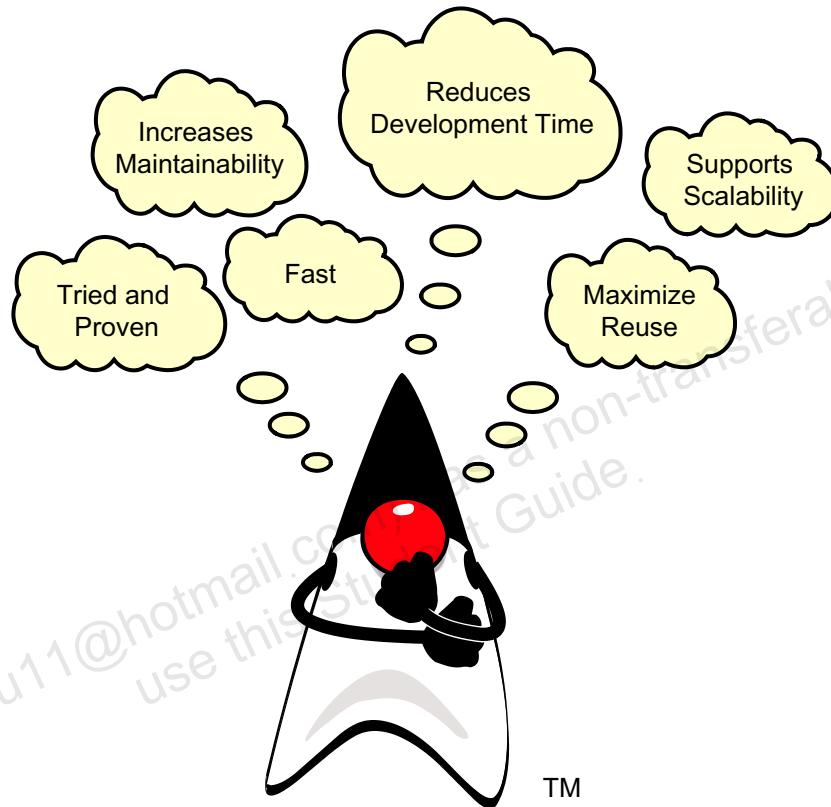


Figure 15-1 Characteristics of Best Practices

EJB technology best practices provide you with one or more of the following benefits:

- Minimizing application response times – Fast response time or perceived fast response time is one of the primary measurements used to determine user satisfaction for a given application.
- Supporting application scalability – The ability to grow an application to meet demand in a pain-free manner saves time and money, and provides increased customer responsiveness.
- Reducing application development time – Anything that reduces the development cycle saves money.

Defining Best Practices

- Increasing maintainability of applications – As much as 80 percent of engineering resources are dedicated to ongoing maintenance of business applications.
- Maximizing EJB component reuse – Reusable components can positively influence both development time and maintainability.

All of these benefits ultimately save money.

Choosing Between EJB and Other Technologies

EJB technology is both powerful and flexible, but its use can come with some development and runtime overhead. Designers of enterprise applications should consider the benefits and drawbacks of using enterprise beans early in the design process. Developers must concentrate on developing the business application logic rather than the enterprise-level support infrastructure when using enterprise beans.

Figure 15-2 shows common application infrastructure requirements that must be addressed during design and development.

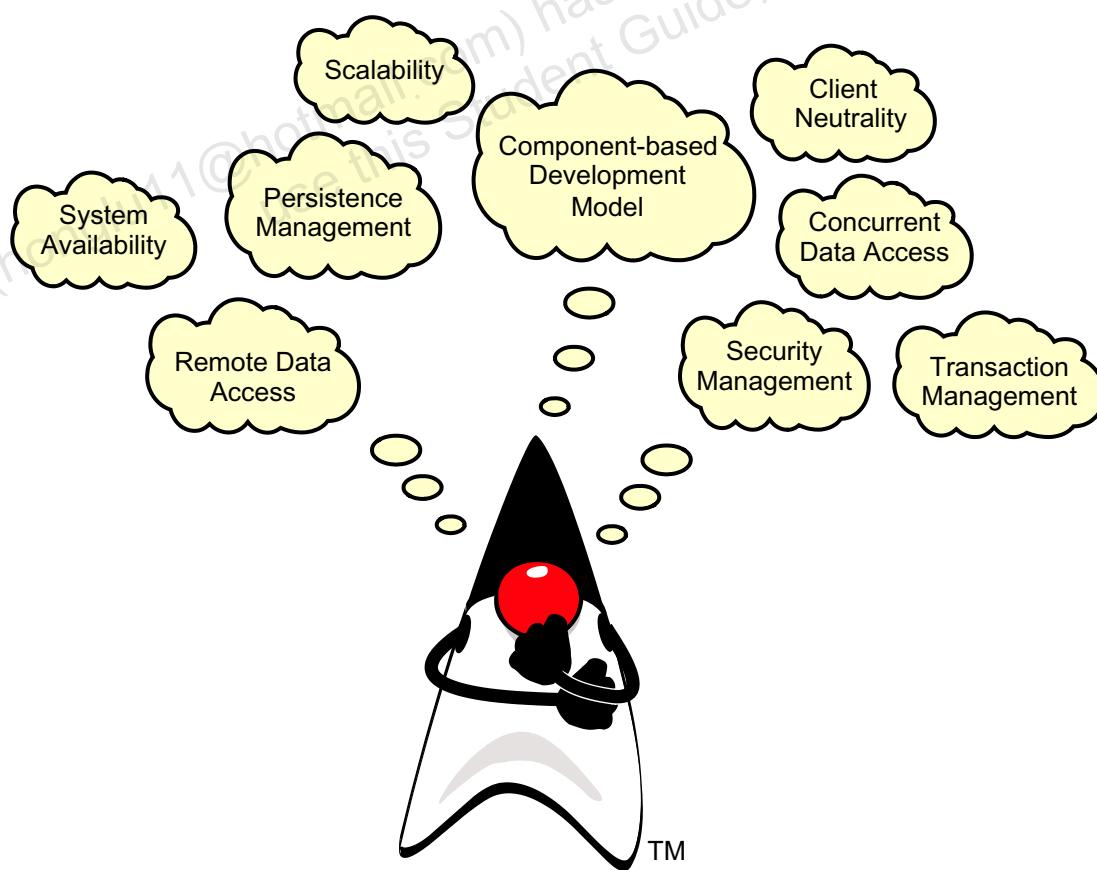


Figure 15-2 Choosing EJB Technology

If you are programming from scratch, you need to spend substantial development time addressing application infrastructure requirements. If your applications need many of the following requirements, your design could benefit from using enterprise beans.

- Persistence management – Entities are designed to provide automatic management of persistent enterprise data. You can use container services to provide declarative management of data manipulation. By using the JDBC API, you can have more flexibility and control with the possible cost of increasing the complexity and reducing the maintainability of the application.
- Transaction management – Enterprise data must be updated consistently. EJB technology provides you with CMT and BMT for both declarative and programmatic transaction management. These options offer both ease of use and flexibility.
- Security management – Enterprise data and applications must restrict access to only authorized users. EJB technology provides you with declarative and programmatic mechanisms for security management.
- Concurrent data access – Any enterprise application that allows multiple concurrent users must provide concurrent access to data without sacrificing consistency. Java EE technology application servers are designed to manage enterprise beans in a thread-safe manner.
- Remote data access – Remote interfaces provide access to location-independent enterprise data.
- Component-based development model – Software components can provide reusability, portability, and a clean separation of the interface from the implementation. By using enterprise beans, you can get these benefits because enterprise beans are true software components.
- Portability – An application that is portable across multiple hardware platforms and operating systems makes best use of existing information technology assets and provides flexibility in the future as requirements and budgets change. Portability also decreases vendor lock-in, decreasing the risk that an application becomes obsolete when operating systems are upgraded or hardware platforms are obsolete. The strict specification of Java EE technologies ensures that enterprise system services are consistently available across vendors and platforms.

Defining Best Practices

- System availability – Mission-critical systems need to be available all the time. Java EE technology application servers can provide automatic failover when a server crashes, when a server is taken down for maintenance, or when networks are unreliable. Enterprise beans also manage automatic data recovery when a failed system is restarted.
- Scalability – Applications often need to handle both increasing loads and new requirements. EJB containers can pool EJB components to maximize resource efficiency. Components can also be migrated to balance the load within a cluster of processors.
- Client neutrality – Some applications require access by multiple client types. Business objects modeled as enterprise beans provide access to an application model by any client type. Java technology clients can access enterprise beans through enterprise bean interfaces. Clients that are not based on Java technology can access enterprise beans using CORBA or web services interfaces.

There are several indications that EJB technology might be inappropriate, including the following:

- There are existing investments in a working technology other than enterprise beans. This can include existing developer skill sets as well as hardware and software.
- The limitations of enterprise beans are too restrictive for your application. If you have a true need for multi-threading, native code, or static variables, then enterprise beans might not be right for your application.
- The application focuses on querying and displaying data, with no real business logic. Other Java EE technologies, such as servlets, and JSP, might be suitable without incurring the overhead of EJB technology.
- The application is simple. If you are developing a system that will not evolve over time, is for one-time use, or is otherwise simplistic, enterprise beans might be more technology intensive than needed.

Using and Applying Known Patterns

Many best practices have been identified as common patterns. In the engineering field, a pattern is described as a recognized concept that captures solutions that have developed and evolved over time. The topic of design patterns is beyond the scope of the course. Many books have been written about both general design patterns as well as Java EE technology-specific design patterns.

An important best practice is to be aware of existing patterns and to use them as applicable. Figure 15-3 illustrates the need to have a personal library of knowledge about EJB technology patterns.

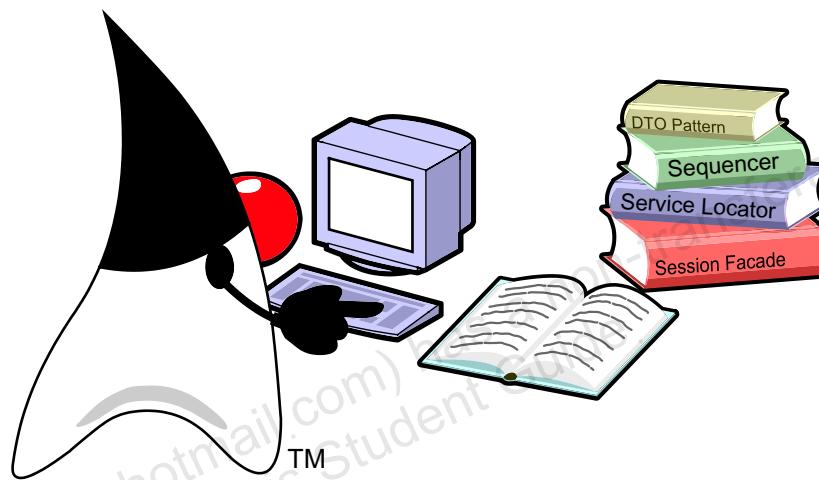


Figure 15-3 Applying Known Patterns

The auction application in this course implements several commonly used general patterns and Java EE technology patterns, including:

- The Session Façade pattern
- The Sequencer pattern
- The Value Object/Data Transfer Object (DTO) pattern
- The Message Façade pattern

Session Façade Pattern

The Session Façade pattern defines a higher-level business component that contains and centralizes complex interactions between lower-level business components. The Session Façade pattern is implemented as a session enterprise bean. In the auction application, the Session Façade pattern is implemented by the AuctionManagerBean session bean. The AuctionManagerBean gives clients coarse-grained methods to call. This, in turn, invokes multiple methods on individual entity instances. For example, the placeBid method makes calls to the AuctionBean, AuctionUserBean, and BidBean entities. Figure 15-4 illustrates the Session Façade pattern.

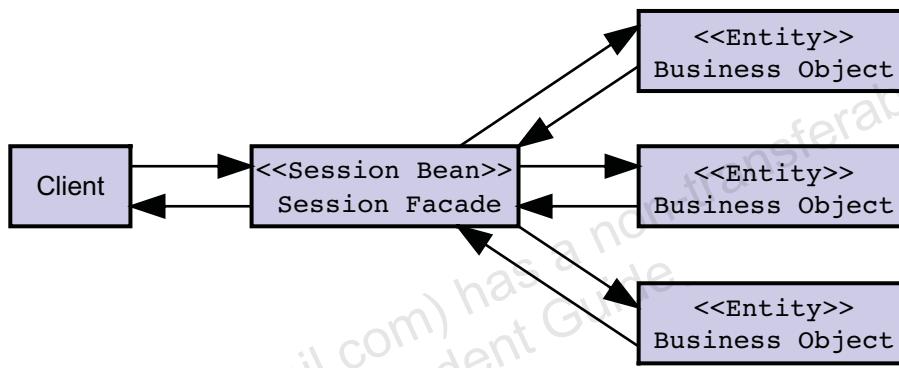


Figure 15-4 Session Façade Pattern

Sequencer Pattern

The Sequencer pattern defines one of many possible ways to generate unique primary keys. Figure 15-5 shows how you can implement the Sequencer pattern, which uses a session bean with a cache of unique primary keys for entity classes. The sequencer session bean distributes sequence numbers and accesses the individual entity instances only when its cache has been depleted.

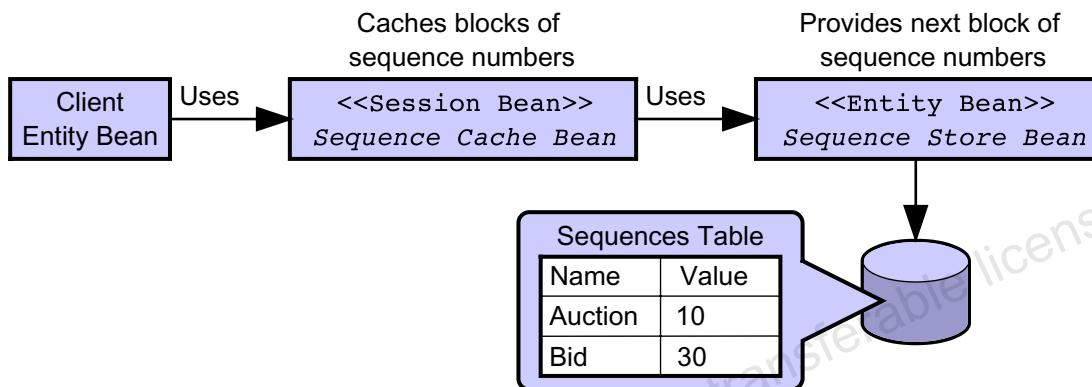


Figure 15-5 Sequencer Pattern as Used in pre EJB 3.0 Applications

The Sequencer pattern was developed for use in EJB applications that predate the EJB 3.0 specification. If you are developing to the EJB 3.0 specification, you can use the `GeneratedValue` annotation to generate primary key values for entity instances. For example,

```

@Id
@GeneratedValue(strategy=SEQUENCE, generator="AUTO")
@Column(name="CUST_ID")
Long id;
  
```

Value Object/Data Transfer Object Pattern

The Value Object/Data Transfer Object pattern suggests the use of serializable Java technology classes to encapsulate and transfer business data between the application server and its client. Figure 15-6 illustrates the Data Transfer Object pattern.

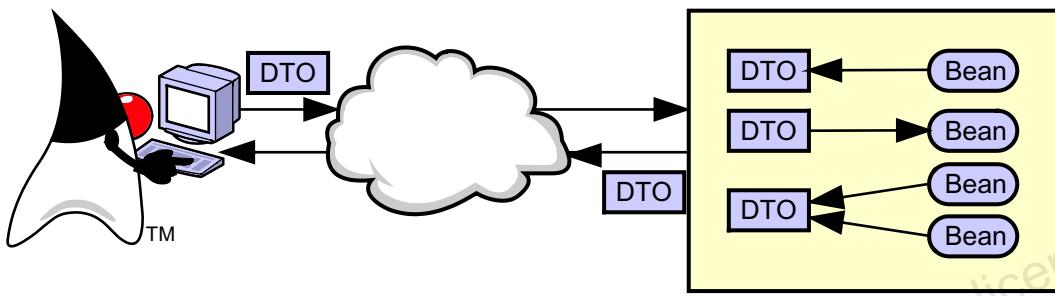


Figure 15-6 Data Transfer Object Pattern

Message Façade Pattern

The Message Façade pattern provides an asynchronous, fault-tolerant mechanism for invoking business use cases. You use this pattern when an immediate result of the process is not necessary. The auction application implements this pattern with the PlaceBidMDBBean message-driven bean, which has the ability to place bids asynchronously. Figure 15-7 illustrates the general design of the Message Façade pattern.

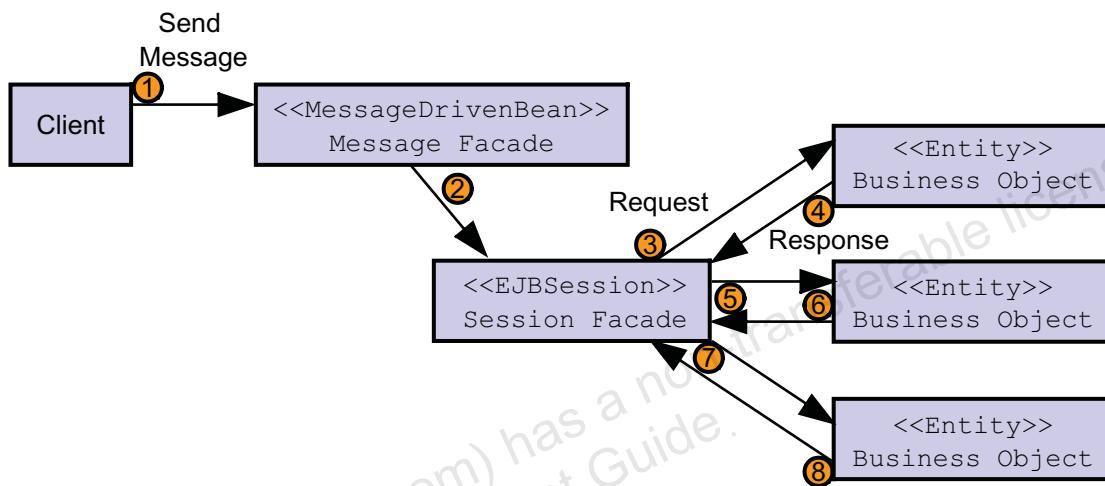


Figure 15-7 Message Façade Pattern

Selecting and Applying Java EE Application Design Decisions

This section outlines some critical Java EE application design decisions that relate to:

- Matching the EJB components to the tasks
- Maintaining session-specific data
- Minimizing network traffic

Matching the EJB Components to the Tasks

Entities, message-driven beans, and session beans all have distinct uses in a Java EE application. You should understand where each type of EJB component fits when you design an application. Figure 15-8 shows how each component should be used.

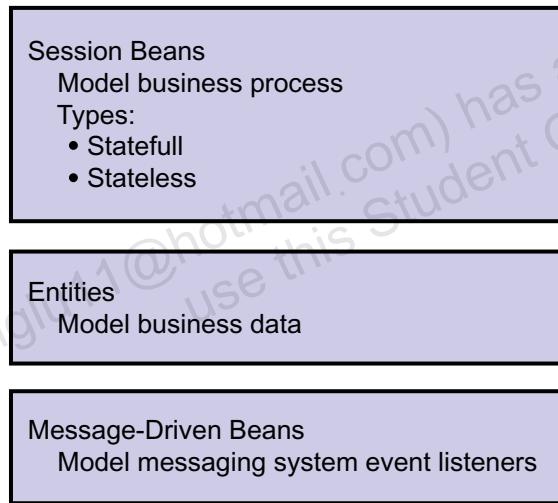


Figure 15-8 Matching EJB Components to the Tasks

Using Entities

When you use entities, consider the following:

- Entities are designed to model business data. Choose an entity when you need persistent storage for a business object. For example, in the auction application, entities are used to represent data from the AUCTION, BID, ITEM, and AUCTIONUSER tables.
- When considering whether to use container-managed persistence or bean-managed persistence, keep the following best practice information in mind. Entities have the potential for better performance than alternatives, such as direct use of the JDBC API, depending on your container vendor's implementation. Entities also provides database independence, allowing for future flexibility. Direct use of the JDBC API is a good choice if you need more control or flexibility than the current specification for Java persistence permits.

Using Message-Driven Beans

When you use message-driven beans, consider the following:

- JMS technology provides messaging services for Java EE applications. The primary mechanism for consuming JMS technology messages is to start a thread that blocks, waiting for a message. Message-driven beans are designed to provide asynchronous interactions with other components and clients. The application server controls blocked threads that wait on JMS technology queues and topics, and dispatches messages to message-driven beans.
- Message-driven beans are indicated when you do not need an immediate result from a call, you do not need fail-safe delivery of your call, or both. For example, in the auction application, you use a message-driven bean to provide the ability to place a bid asynchronously.
- Session beans should never be used as message consumers for the following reasons.
 - A session bean used as a synchronous message consumer will block while waiting for the next message. This interferes with the management responsibility for all instances of session beans allocated to the container.
 - A message delivered to a session bean used as an asynchronous consumer will bypass the container.

Using Session Beans

When you use session beans, consider the following:

- Session beans are designed to model business processes. Session beans can encapsulate code that executes on the server on behalf of, or as an extension of, the client. Session beans also are used to wrap fine-grained entity processing with coarse-grained business methods. For example, in the auction application, the `AuctionManagerBean` session bean provides the `placeBid` method to clients, encapsulating many finer-grained methods in several entity classes.
- Stateless session beans can be pooled and do not need to have the overhead of saving session-specific state data because they do not get passivated. Use a stateless session bean:
 - When a client makes a single method invocation to complete a business task
 - When all the data to complete a business task can be passed efficiently as parameters in a method call
 - To take advantage of the pooling, scalability, and clustering provided by the stateless session bean design
- Choose a stateful session bean:
 - If it is more efficient from a response-time and network perspective to store client data in the session bean rather than pass it as parameters in method calls
 - If it reduces multiple method invocations to complete a business task called by a client

Maintaining Session-Specific Data

A Java EE application provides several options for maintaining session-specific data. Figure 15-9 illustrates the two best practices options for maintaining session-specific data.

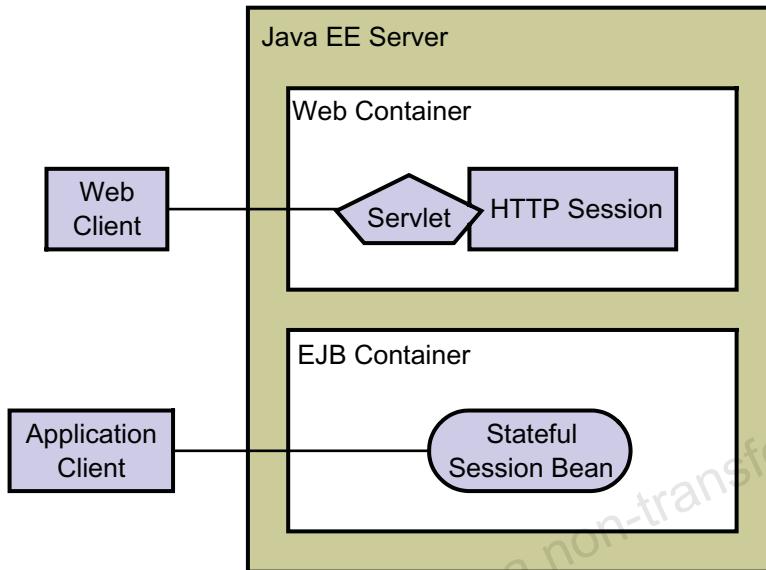


Figure 15-9 Maintaining Session Specific Data

When you need to maintain session-specific data, consider the following:

- Use a stateful session bean for business functionality and to store session data in the bean. This option is recommended when the application can be accessed by an application client that does not use Hypertext Transfer Protocol (HTTP). Your application server's features might also influence the decision to store state information in stateful session beans.
- Use a servlet controller or interface for your client, and store session data in an `HttpSession` object. Use this option if the client talks to the EJB application using HTTP. For example, in the auction application, the `AuctionManagerEJB` bean is a stateless session bean and session information is maintained in the servlet controller.
- Use an entity to store state information. For example, sites that save shopping basket information over long periods of time often use entities with their corresponding persistent storage to support them.
- Use an option, such as cookies, URL rewriting, or hidden form data to store session-specific option. These options are not recommended Java EE technology best practices.

Minimizing Network Traffic

Many design decisions are made for performance reasons. Performance is measured in many ways, including response time. One way to potentially improve response time is to minimize network traffic. You can minimize network traffic in two ways: by minimizing remote method calls and by optimizing method parameters. Figure 15-10 illustrates the latency involved in excess network traffic.

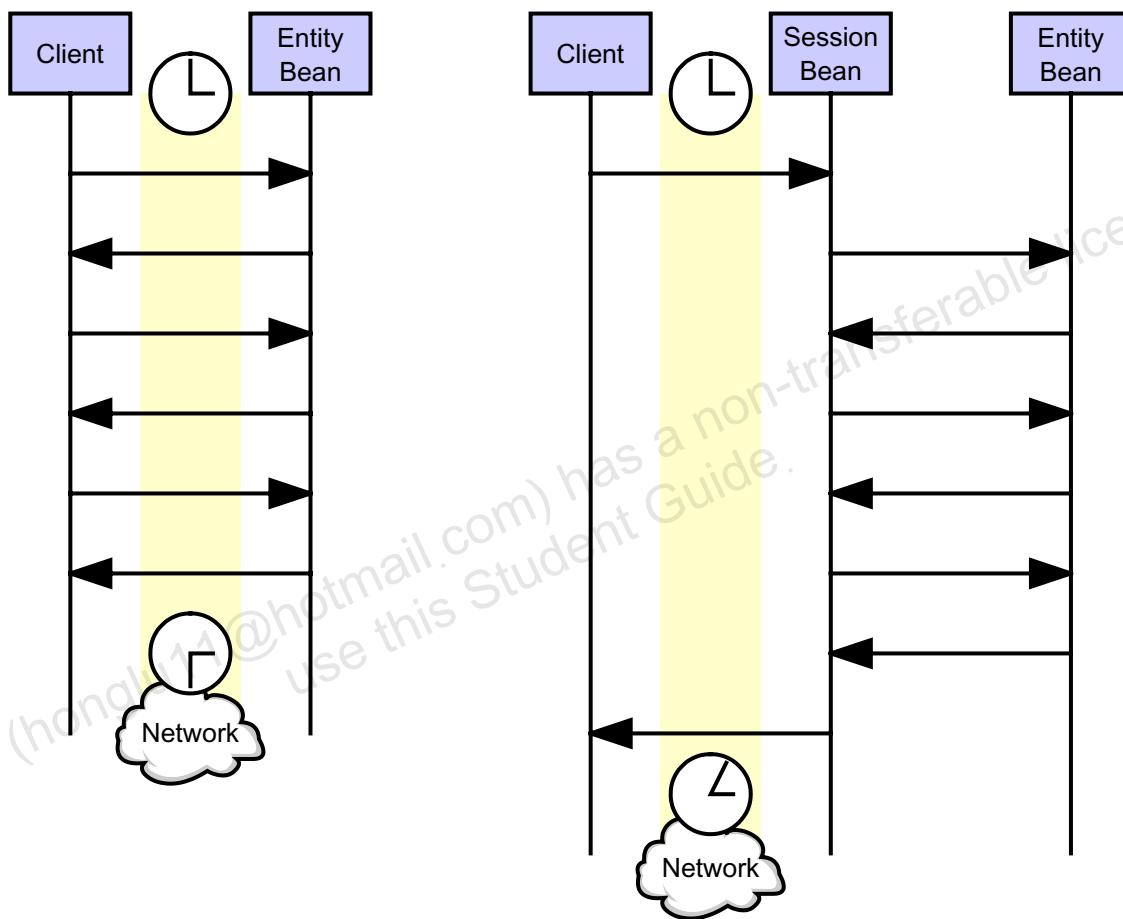


Figure 15-10 Minimizing Network Traffic

Minimizing the Use of Remote Method Calls

The primary reason to minimize remote method calls is to reduce the amount of data that must be serialized and deserialized and sent over the network. Parameters passed over the network are passed by value, and incur processing time during serialization and deserialization. Serialized objects passed over the network add to network congestion. Figure 15-10 on page 15-16 shows several strategies for minimizing remote method calls.

You can use the following strategies for minimizing remote method calls:

- Use local interfaces, where possible, to invoke methods on enterprise beans. Method calls to local interfaces pass parameters by reference. Objects passed by reference do not cause serialization and deserialization overhead or additional network calls.
- Group related EJB components and deploy them in a single JVM machine. Local interfaces can be accessed only by components executing in the same JVM machine.
- Use the Session Façade design pattern. This pattern consists of the following:
 - A session bean provides a remote interface to the client.
 - The remote interface provides high-level, coarse-grained, business services.
 - The session bean interacts with entity instances and other session beans (providing fine-grained services) using their local interfaces.
 - Each high-level business service encapsulates processing logic that makes one or more calls to the other beans through their local interfaces.
- Use the Data Transfer Object design pattern where applicable. Objects that are designed to pass only the data needed for a specific purpose are smaller than objects that represent the entire entity.

Optimizing Method Parameters

Some data inevitably must be transferred across the network. Whenever possible, follow these guidelines when you design methods and their parameters, to minimize processing time and unnecessary network traffic:

- Use Java technology primitives whenever possible. Primitives do not require deserialization, and use little bandwidth.
- When passing objects, pass only what is required. Use transient fields when possible because transient fields are not serialized. Use DTOs to pass everything that is needed in a single call.

Appendix A

Introducing Transactions

This appendix introduces basic transaction concepts.

Additional Resources



Additional resources – The following reference can provide additional details on the topics discussed in this appendix:

- Sun Microsystems, “JSR 220: Enterprise JavaBeans™, Version 3.0 EJB Core Contracts and Requirements, Chapter13.” [<https://sdhc3e.sun.com/ECom/EComActionServlet;jsessionid=CEAAE57A3BAB8A76D4555E3C5A1F4031>], accessed July 25, 2006.
- Sun Microsystems, “JSR 220: Enterprise JavaBeans™, Version 3.0 EJB 3.0 Simplified API.” [<https://sdhc3e.sun.com/ECom/EComActionServlet;jsessionid=CEAAE57A3BAB8A76D4555E3C5A1F4031>], accessed July 25, 2006.

Examining Transactions

Figure A-1 shows the steps an application takes to transfer funds between two bank accounts. The transfer-funds use case requires that at the end of this operation, the bank accounts are left in a consistent state. This is the case if the transfer is not carried out at all or if it is completed in full. It is not the case if the transfer is partially completed.

In Step 1, the application debits the funds from Account A. In Step 2, the application credits Account B. If Step 2 fails, the application must reverse the first step to return the two accounts to their pre-transaction states.

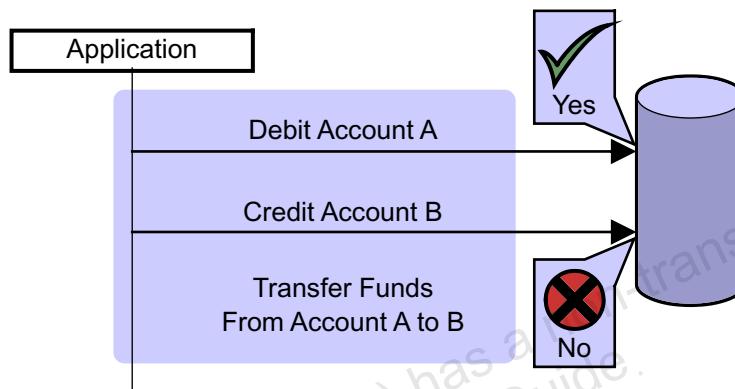


Figure A-1 Transfer of Funds

In general, a business use case requires one or more actions to a database or some other transactional resource, such as a transactional message queue. You should consider making all the operations of a single use case an atomic unit of work by default (that is, a single transaction).

If a single step in the transaction fails, all actions that had been performed prior to the failure must be reversed. This reversal is called a *roll back*.

In general, transaction management presents no particular problem for you if all the steps in the transaction take place on the same resource. This applies both inside and outside the Java EE technology environment. For example, all modern databases have built-in support for transaction management. They log the operations carried out during a transaction and, if a roll back is required, they reverse those operations with reference to the log. The JDBC API has mechanisms by which an application can control which operations are grouped into a single transaction.

Examining Transactions

Things are more complex when the operations of the transaction span multiple resources. Then the transactional states of those resources must be coordinated. Ideally, the application developer should not need to do this. Application servers have transaction management capabilities to help ease the burden on the developer. The transaction manager takes care of the coordination of resources (typically using a two-phase commit process). The application merely demarcates the transaction, (that is, indicate where it begins and ends).

Figure A-2 shows how you can apply transaction management to the same business case (transfer of funds).

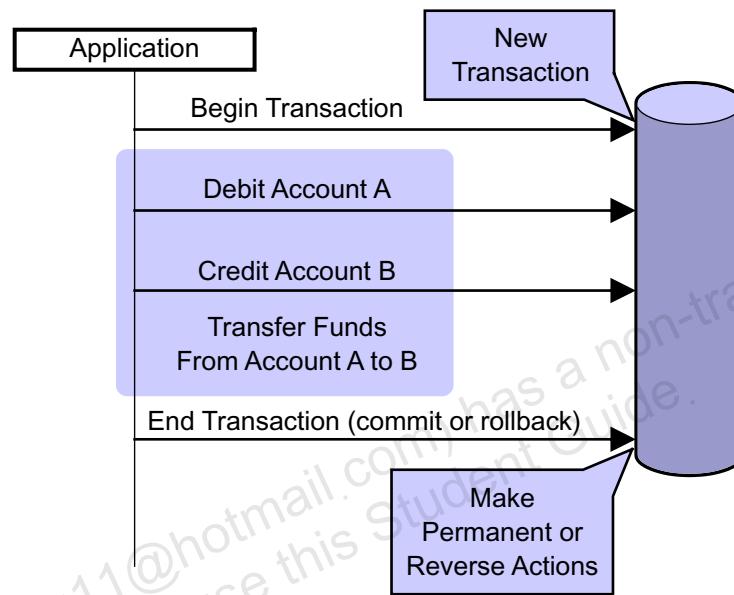


Figure A-2 Transfer of Funds Using Transaction Management

A transaction is a unit of work. A transaction consists of:

- A starting point that marks the beginning of a transaction
- One or more resource actions
- An end action consisting of one of the following:
 - Commit – Makes the changes permanent
 - Rollback – Reverses all the changes

The participants in a transaction are:

- An application that begins and ends a transaction
- A resource manager that services the transaction
 - Examples of resource managers are databases, messaging systems, and application servers
- A transaction manager to coordinate the transaction on those resources, if the transaction spans multiple resources

Transactions have the following properties:

- Atomicity – Either all actions of the transaction are completed or none are completed.
- Consistency – The result of a transaction is either a new set of valid data (commit) or return of the data to its original state (rollback).
- Isolation – A transaction's actions on data are not visible to other transactions until the transaction is committed.
- Durability – Committed data is permanent and survives system crashes and restarts.

Types of Transaction

Transaction technology defines the following three transaction types:

- Flat transactions
- Nested transactions
- Chained transactions

Note – Application servers based on the Java EE 1.3 specification are required to support flat transactions only.



Examining Transactions

Flat Transactions

Figure A-3 illustrates a flat transaction.

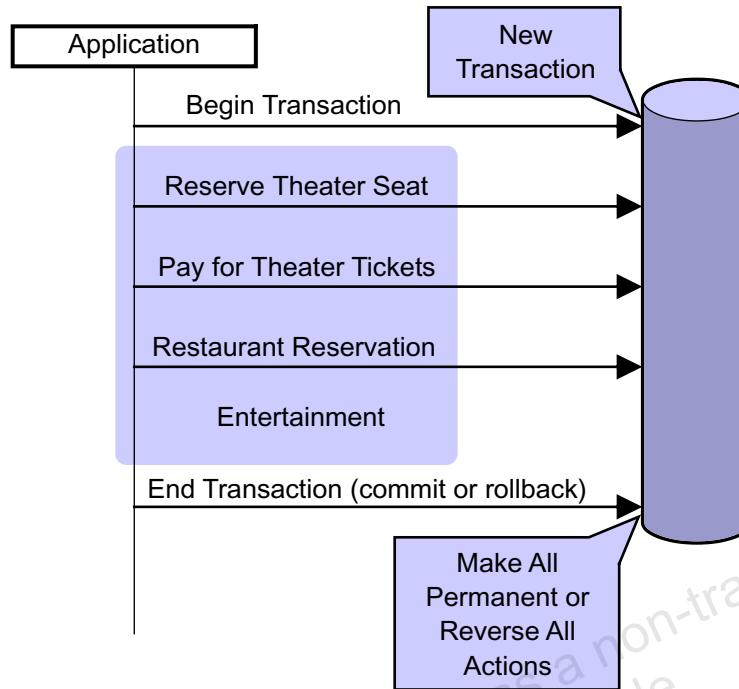


Figure A-3 Flat Transaction

A flat transaction consists of a sequence of data resource primitive operations (for example, select, update, insert, and delete) sandwiched between a begin and an end marker.

All enclosed primitive operations are considered an atomic unit. You can either commit or roll them back. A rollback does not affect operations performed on the data resource by the previous transaction.

Code A-1 shows an example of a flat transaction using pseudo code.

Code A-1 Flat Transaction Pseudo Code Example

```

begin transaction transferFunds
    update ...
    select ...
    update ...
end transaction transferFunds
  
```

Nested Transactions

A nested transaction contains other transactions nested within it. Each inner transaction can be rolled back without affecting the preceding subtransactions.

Figure A-4 shows an example of a nested transaction. The nested transaction contains two inner transactions nested within an outer transaction.

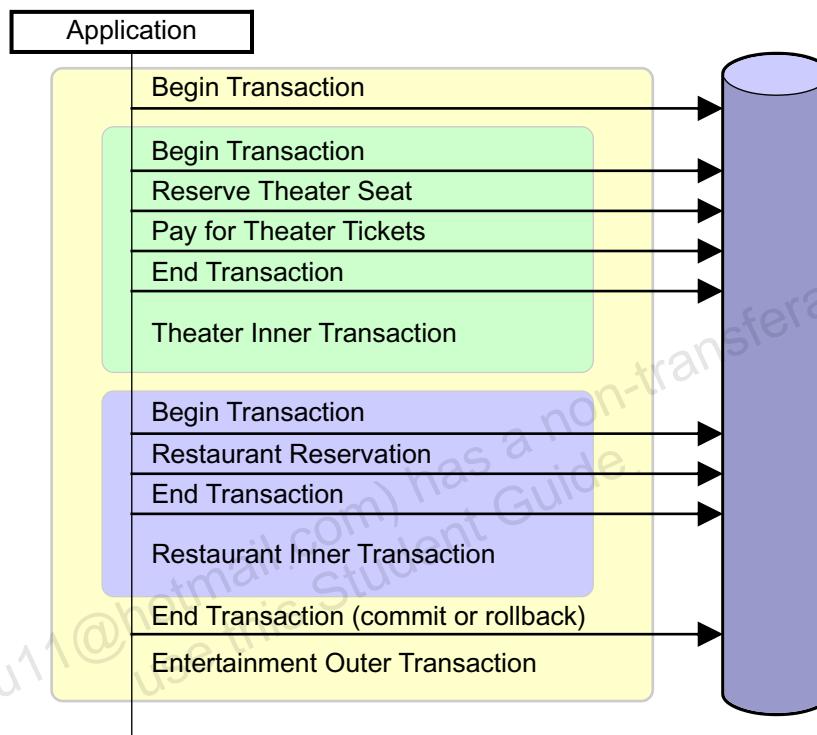


Figure A-4 Nested Transaction

A nested transaction permits an application to use an alternative business strategy to retry the subtransaction. For example, if the Restaurant inner transaction shown in Figure A-4 fails, the application attempts to reserve a table at an alternative restaurant.

Examining Transactions

Code A-2 shows an example of nested transactions using pseudo-code.

Code A-2 Nested Transaction Pseudo-Code Example

```
begin transaction entertainment
    begin transaction reserveTheatreTickets
        ...
        end transaction reserveTheatreTickets
    begin transaction reserveRestaurant
        ...
        end transaction reserveRestaurant
    end transaction entertainment
```

For the outer transaction to succeed, all inner transactions must succeed. If for any reason the outer transaction is rolled back, then all inner transactions are also rolled back.

Chained Transactions

Unlike a flat transaction, which has a single commit point, a chained transaction contains multiple breakpoints. A commit establishes a breakpoint. A rollback reverses all actions back to the last breakpoint. If the whole transaction fails, then the breakpoints do not prevent the complete set of operations from being rolled back.

Figure A-5 illustrates chained transactions.

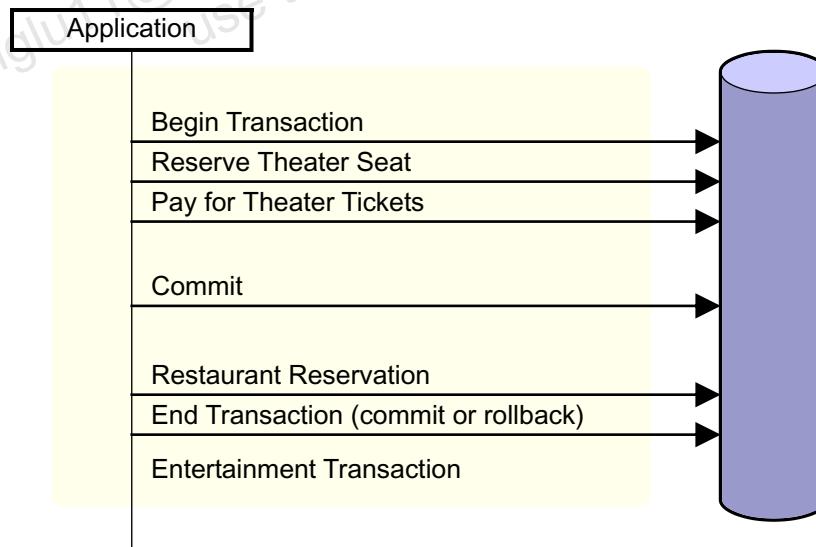


Figure A-5 Chained Transactions

Transaction-Related Concurrency Issues

If performance were not an issue in database applications, then transactions could be fully isolated from one another. The only way to ensure that this isolation occurs is for the database to apply the transactions one after the other, with each transaction waiting for the previous one to commit or roll back. Database designers call this *serialization* (which should not be confused with Java language serialization).

The problem is that serialization has a significant impact on performance. A database engine that has sufficient processing power to service 100 transactions at a time, for example, is servicing only one at a time, because the others are blocked. This means that full isolation is rarely possible in practice. Typically, you have to be satisfied with partial isolation. As the level of isolation decreases, there is an increased risk of *isolation anomalies*, that is, errors.

Figure A-6 shows the three most common problems you can encounter when multiple application threads access a database.

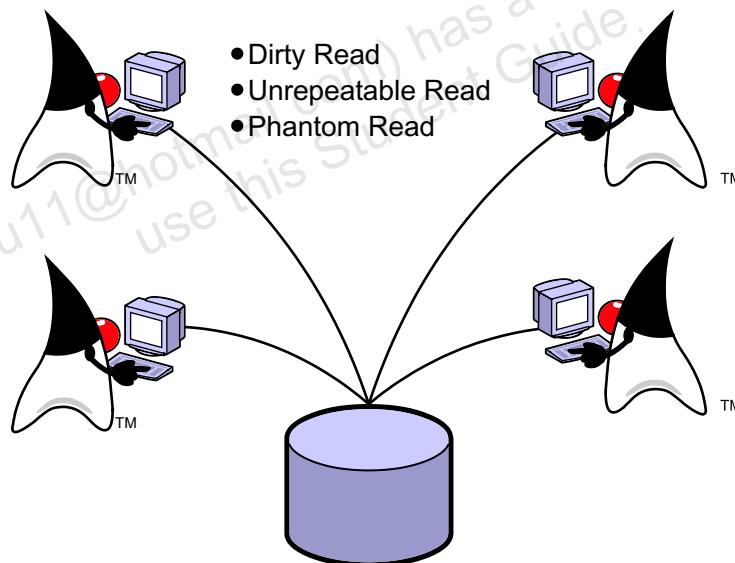


Figure A-6 Transaction-related Isolation Anomalies

These problems are caused by the transactional nature of database interactions. The changes made during a transaction are temporary until the transaction is complete.

Transaction-Related Concurrency Issues

- During concurrent database access, an application thread can access the temporary uncommitted data causing one of the following anomalies:
Dirty read anomaly – A dirty read occurs when transaction 1 writes a value, and transaction 2 reads the value while transaction 1 is still incomplete. If transaction 1 rolls back, then it restores the previous value in place of the value it wrote. Transaction 2, however, does not know this, and the application controlling transaction 2 ends up with data that does not match the database.

Allowing occasional dirty reads can give an enormous improvement in concurrency because any number of transactions can read while another transaction is writing. However, most applications would be badly damaged by the presence of dirty reads, and many database products do not allow the isolation level to be set this low.

- Unrepeatable read anomaly – This anomaly occurs when transaction 1 reads different data twice in the same transaction. If transaction 2 can write the data between transaction 1's pair of reads, then transaction 1 has read two different values for the same data. If the transactions were fully isolated, then transaction 2 would not have been allowed to start until transaction 1 had finished, so transaction 1 would have read the correct value twice.

Unrepeatable reads are bad, but rare. Unrepeatable reads can arise only when there are multiple reads of the same data in the same transaction. This is unusual in practice. Therefore, most databases allow the possibility of unrepeatable reads to improve performance, unless the databases are configured otherwise.

- Phantom read anomaly – This anomaly occurs when different data sets are returned by the same query that is executed twice in the same transaction. This is caused by another transaction that inserts data that satisfies the query between the reads.

The phantom read anomaly is similar to the unrepeatable read anomaly. It occurs when transaction 1 carries out two searches (for example, SELECT operations) in the same transaction. If transaction 2 is allowed to insert or delete data between transaction 1's two searches, then transaction 1 gets different results on the two occasions.

Again, although phantom reads are bad, they occur infrequently, in practice, because they can arise only where there are multiple SELECT operations in one transaction. It takes a high level of locking to avoid phantom reads, so most databases allow them by default.

Transaction Isolation Levels

You can solve concurrent database access problems by isolating the changes made during a transaction by one thread from other threads until the transaction is complete. This is known as isolation locking. Data resources provide you with configurable isolation options. The type of isolation is a compromise between efficiency and data integrity.

Different database vendors allow different levels of isolation to be set on their products. Some allow a great deal of flexibility, while others allow only one or two fixed levels of isolation. This variation in levels of isolation was recognized by the developers of the JDBC specification, who abstracted the various vendors' strategies into four basic isolation levels. Table A-1 shows the isolation levels defined by the JDBC technology, and the concurrency problems they solve.

The developer sets the isolation level using a JDBC API call. Not all the isolation levels in Table A-1 are available on all products. Also there are no API calls in the EJB specification or the Java EE technology specification for setting the isolation level. An API call can only be made at the level of the JDBC technology connection. In an application server, the likelihood is that the enterprise bean is not working on a real database connection, but instead it uses a virtual connection from the server's connection pool. That virtual connection might be used by many different enterprise beans. Thus, if your code sets the isolation level, it might affect other enterprise beans as well, possibly even in different applications. This means that the application server is not required to honor requests to change the isolation level. Therefore, you should avoid using API calls to change the isolation level in enterprise beans.

Transaction-Related Concurrency Issues

If you need to set isolation levels that are different from the database's default, you should investigate methods for doing this at the database level, rather than by using API calls.

Table A-1 Effect of Isolation Levels on Concurrency Problems

Isolation Level	Dirty Read	Unrepeatable Read	Phantom Read
Read uncommitted	Yes	Yes	Yes
Read committed	No	Yes	Yes
Repeatable read	No	No	Yes
Serializable	No	No	No

A yes indicates that the problem occurs.

Note – Isolation configurations are outside the Java EE technology specification and are resource specific.



Handling Distributed Transactions

Figure A-7 shows the transfer of funds from accounts stored in different databases, which involves debiting an account stored in one database and crediting an account stored in another database. Because the transfer of funds between two accounts is considered an atomic operation, the actions performed during this operation must take place within a single transaction.

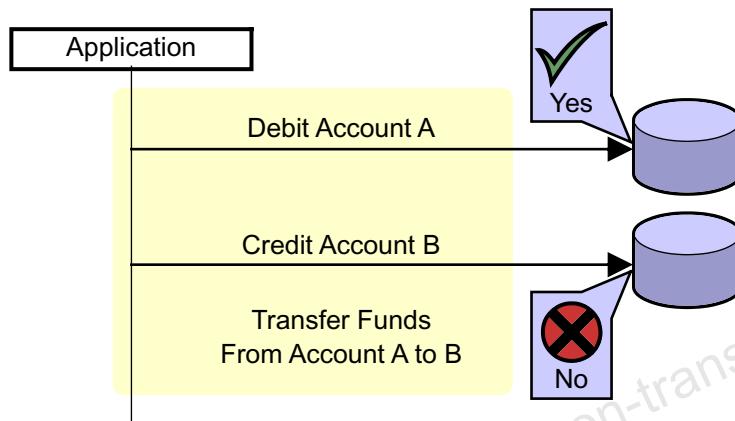


Figure A-7 Distributed Database Access

Transactions that involve multiple resources in a single transaction are referred to as distributed transactions. A failure in any resource must create a rollback in all other resources, under the control of a transaction manager.

Two-Phase Commit

The two-phase commit is an established strategy for managing distributed transactions. Figure A-8 illustrates the two-phase commit participants and protocol.

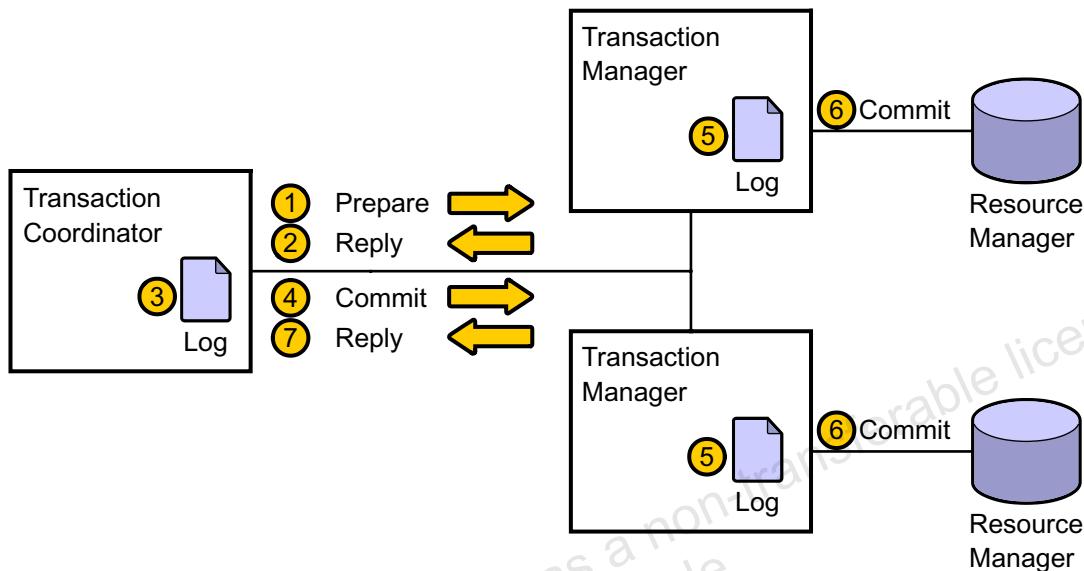


Figure A-8 Two-Phase Commit

The two-phase commit involves a single transaction coordinator and a transaction manager for each resource.

The following occurs during phase 1:

1. The transaction coordinator requests that all transaction managers prepare to commit.
2. The transaction managers reply.
3. The transaction coordinator logs the reply.

The following occurs during phase 2:

4. If all agree, the transaction coordinator sends the commit.
5. The transaction managers log the request.
6. The transaction managers commit.
7. The transaction managers reply confirming the commit.

Java Transaction API (JTA)

Figure A-9 illustrates the primary function of the JTA. JTA defines the interface between a transaction manager and an application client, an application server, and resource managers.

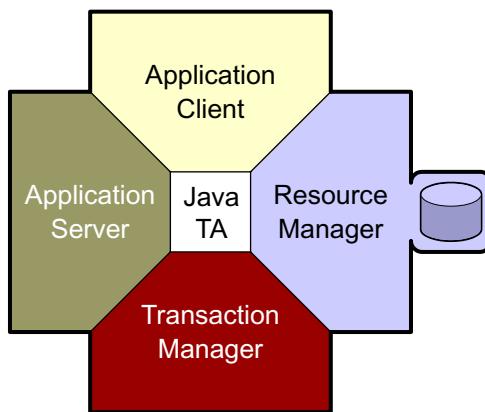


Figure A-9 Java Transaction API

The JTA specification defines interfaces that specify the interactions between the various parts of a transaction system: the application, the application server, the transaction manager, and the back-end resources. An application server vendor must provide implementations of most of these interfaces. Database driver vendors implement the others. The JTA specification is complicated. However, as an enterprise bean developer, you will mostly be concerned with the `javax.transaction.UserTransaction` interface. The interface's purpose is to support code-initiated transaction demarcation. For Bean Managed Transaction (BMT), you include code in your bean to call methods of this interface to begin and end a transaction to support code-initiated transaction control and demarcation.

Many EJB technology developers experience confusion about the relationship between JTA and Java Transaction Service (JTS). JTS is a Java technology implementation of the CORBA Object Transaction Service (OTS). Sun Microsystems provides a reference implementation of JTS that is widely used by developers who want to integrate Java technology applications with CORBA applications.

However, JTS is of little or no relevance to the enterprise bean developer. The application server might make use of JTS as its underlying transaction architecture, or it might not. JTS is not part of the Java EE technology specification (version 1.3), and application servers are not required to support it. An enterprise bean's access to the transaction management infrastructure is through JTA (or through the declarative transactions using CMT), not through JTS.

Appendix B

Integrating With Legacy Systems

Objectives

Upon completion of this module, you should be able to:

- Describe integration with legacy systems
- Examine the EIS connectivity module
- Examine JCA resource adaptors
- Use the Common Client Interface (CCI) API interfaces
- Use a message-driven bean resource adapter
- Integrate with legacy systems using the Common Object Request Broker Architecture (CORBA) protocol

Additional Resources



Additional resources – The following reference provides additional information on the topics described in this module:

- Sun Microsystems, “JSR 220: Enterprise JavaBeans™, Version 3.0 EJB Core Contracts and Requirements, Chapter15.” [<https://sdlc3e.sun.com/ECom/EComActionServlet;jsessionid=CEAAE57A3BAB8A76D4555E3C5A1F4031>], accessed July 25, 2006.
- Sun Microsystems, “J2EE Connector Architecture.” [<http://java.sun.com/j2ee/connector>], accessed July 25, 2006.

Introducing Integration With Legacy Systems

A Java EE application might require the services hosted in legacy systems. To obtain these services, a Java EE application developer must perform the following tasks:

- Request service from the legacy system using the legacy system's (proprietary) protocol
- For transaction-capable legacy systems, synchronize transaction with the legacy system
- For security sensitive legacy systems, propagate security credentials

This module describes the following two technology solution mechanisms that to a large extent abstract the difficulties associated with integrating with legacy systems:

- Java EE resource adapters
To use this approach, you must install a resource adapter for the legacy server in your application server.
- CORBA-protocol-based interfaces
To use this approach, the legacy system must provide the CORBA interfaces.

Examining the EIS Connectivity Module

Examining the EIS Connectivity Module

Figure B-1 illustrates Java EE components using a connectivity module to connect to an EIS.

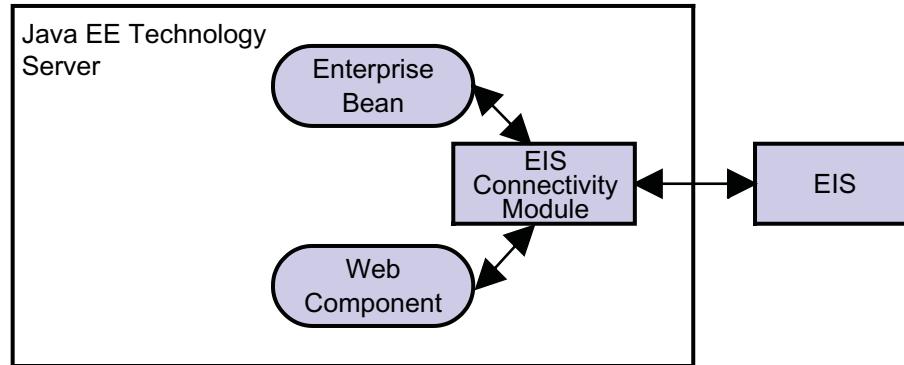


Figure B-1 EIS Connectivity Module

This section examines the requirements for the EIS connectivity module and compares four possible implementation alternatives for an EIS module.

Requirements

The following list summarizes the capabilities required of an EIS connectivity modules:

- Expose EIS services

Exposing EIS services is the primary task of an EIS connectivity module. The connectivity module act as a proxy objects to EIS resources. By providing a suitable API, connectivity modules shield Java EE application components from the complexity of communicating with EIS resources.

- Propagate security context

Most EIS services use access control that verify user credentials (for example, user identity and password) before they perform requested services. Enterprise bean developers should not have to supply credentials to EIS to establish a connection. For example, you are not required to supply credentials to establish JDBC software connections. The EIS connectivity module must have the capability to accept and pass to the EIS, user credentials supplied by the container, just as a JDBC driver does for JDBC software connection establishment.

- Propagate transaction context

An EIS service executes in a transaction context. When an enterprise bean method that is participating in a transaction invokes an EIS service, the transaction that is associated with the EIS service must be co-opted into the transaction associated with the invoking enterprise bean method. The contract between the EJB container and the EIS connectivity module must allow the transaction state to be communicated in both directions between the container and the EIS connectivity module. This enables the container to synchronize the commit or roll back of the EIS transaction with that of the enterprise bean method.

- Support resource pooling

EIS connectivity modules typically control access to scarce resources, such as network connections or file handles. These resources should be pooled for optimum performance and efficiency.

- Support deployment time configuration

An EIS connectivity module must provide for deployment time configuration. Ideally, this deployment time configuration must use EIS connectivity module-specific deployment descriptor elements.

Implementation Alternatives

Vendors who want to distribute self-contained software modules that enterprise bean developers can use to integrate with their legacy systems have the following options:

- Implement the connectivity in a class or a package of classes
- Implement the connectivity in a self-contained enterprise bean in its own JAR file
- Implement the connectivity in a proprietary resource adapter
- Implement the connectivity in a Java EE Connector Architecture (JCA) resource adapter

These choices are available irrespective of the nature of the integration software that might be using Java technology sockets, CORBA, XML messaging, or anything else. The developer of the integration software should choose the most basic strategy that is compatible with the integration method chosen.

The following sections evaluate the alternative connectivity implementations with respect to the following criteria:

- Integration approach
- Benefits
- Requirements fulfilment

Comparing Integration Approaches

Table B-1 compares the integration approach of the alternative implementations of EIS connectivity modules.

Table B-1 Comparing Integration Approaches of EIS Connectivity Alternatives

Implementation	Integration Approach
Basic class or package	The EIS vendor provides a class file or a JAR package. You import this class or package into the EJB technology JAR by reference, or you install it at the system level.
Enterprise bean	The EIS vendor provides a connectivity enterprise bean, which is self-contained in its own JAR file with its own DD.
Proprietary resource adapter	The EIS vendor supplies a proprietary resource adapter.
JCA resource adapter	The EIS vendor supplies a resource archive (RAR) package that conforms to the JCA specification.



Note – Proprietary resource adapters were used quite extensively before the JCA specification was published. Most vendors are now moving their proprietary code to JCA implementations.

Examining the EIS Connectivity Module

Comparing Benefits

Table B-2 compares the benefits of the alternative implementations of EIS connectivity modules.

Table B-2 Comparing Benefits of EIS Connectivity Alternatives

Implementation	Benefits
Basic class or package	This method is straightforward and easy to understand. Coding requirements are minimized for the integration operation.
Enterprise bean	Unlike a basic class or package, this implementation enables independent configuration through its DD.
Proprietary resource adapter	This method might integrate efficiently with the vendor's EIS system.
JCA resource adapter	Full integration with the EJB container enables security and transaction propagation and pooling. Can support CCI.

Comparing Requirements Fulfilment

Table B-3 compares the requirements fulfilment of the alternative implementations of EIS connectivity modules.

Table B-3 Comparing Requirements Fulfilment of EIS Connectivity Alternatives

Requirement	Class or Package	Enterprise Bean	Proprietary Resource Adapter	JCA Resource Adapter
EIS service API	Yes	Yes	Yes	Yes
Security context propagation	No	No	Implementation dependent	Yes
Transaction context propagation	No	No	Implementation dependent	Yes
Resource pooling	No	No	Implementation dependent	Yes
Configuration	No	Yes	Implementation dependent	Yes

Examining JCA Resource Adapters

The JCA specification contains the requirements for a JCA resource adapter. Figure B-2 shows the operational context of a resource adapter.

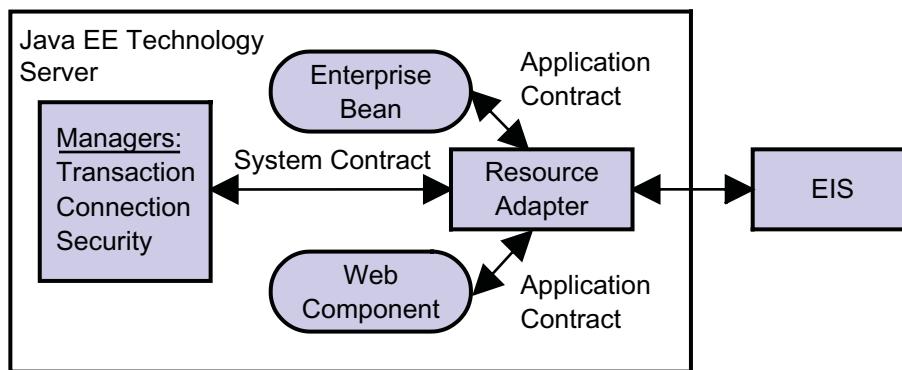


Figure B-2 JCA Resource Adapter

A resource adapter is a Java EE component that provides access to an EIS for Java EE application components, such as enterprise beans or web components. A resource adapter's primary function is to provide a uniform way for application components to interact with EISs.

EIS resource adapters implement the connector architecture interface. This interface consists of the following two APIs:

- Resource adapter system contracts
- Application contract

Resource Adapter System Contracts

Resource adapter system contracts are hidden from Java EE technology components. The contract provides an interface so that the EIS and the Java EE technology server interact. The system contract consists of the following contracts:

- The transaction management contract – Provides support for global transaction through X/Open (XA) protocol or local transactions managed by an EIS

- The connection management contract – Allows for connection pooling and other techniques to provide efficient use of the EIS resource by multiple clients
- The security management contract – Protects EIS information by providing authentication and secure communications between the EIS and the Java EE technology server

Application Contract

The application contract is the application-related interface between the enterprise bean and the resource adapter. The JCA specification defines an application interface called CCI. Following are the main features of the CCI interface:

- CCI is a set of interfaces that resemble the JDBC software interfaces. The Java EE components use this API to communicate with the EIS.
- CCI support is not compulsory:
 - Resource adapter vendors are encouraged to implement CCI.
 - Vendors can provide an alternative application contract.
 - Some resource adapters, with basic interaction semantics, do not require CCI.
- The use of CCI does not completely eliminate some API calls that are specific to the legacy system.

Using the CCI API Interfaces

The CCI API (`javax.resources.cci`) consists of the following interfaces:

- Connection-related interfaces that represent a connection factory and an application-level connection:
 - The `ConnectionFactory` interface provides factory methods to obtain `Connection` and `RecordFactory` objects.
 - The `Connection` interface represents a connection to the underlying EIS resource.
 - The `ConnectionSpec` interface is an implementing class that holds connection parameters.
 - The `LocalTransaction` interface enables a component to demarcate resource manager local transactions.
- Interaction-related interfaces that enable a component to drive an interaction, specified through an `InteractionSpec` object with an EIS instance:
 - The `Interaction` interface executes a specific function on the EIS resource.
 - The `InteractionSpec` interface is an implementing class that holds interaction parameters.
- Service endpoint message listener interface:
 - The `MessageListener` interface serves as a request-response message listener type that message endpoints can optionally implement. This allows an EIS to communicate with an endpoint using a request-response style.
- Data representation-related interfaces that are used to represent data structures involved in an interaction with an EIS instance:
 - The `Record` interface provides the base interface for return types or parameters of the `execute` method defined in the `Interaction` interface.
 - The `IndexedRecord` interface represents an ordered collection of `Record` instances based on the `java.util.List` interface.
 - The `MappedRecord` interface represents a key-value-based collection of `Record` instances based on the `java.util.Map` interface.
 - The `RecordFactory` interface provides factory methods to obtain `IndexedRecord` and `MappedRecord` objects.

- The Streamable interface enables a resource adapter to extract data from an input Record or set data into an output Record as a stream of bytes.
- The ResultSet interface is based on the JDBC ResultSet interface. The ResultSet interface extends the `java.sql.ResultSet` and `javax.resource.cci.Record` interfaces. A result set represents tabular data that is retrieved from an EIS instance by the execution of an interaction.
- The ResultSetMetaData interface provides information about the columns in a ResultSet instance.
- Metadata related-interfaces that provide basic meta information about a resource adapter implementation and an EIS connection:
 - The ConnectionMetaData interface provides information about an EIS instance connected through a Connection instance.
 - The ResourceAdapterMetaData interface provides information about the capabilities of a resource adapter implementation. This interface does not provide information about an EIS instance that is connected through a resource adapter.
 - The ResultSetInfo interface provides information on the support provided for ResultSet functionality by a connected EIS instance.
- Exceptions and warning classes:
 - The ResourceException class provides resource adapter specific error information.
 - The ResourceWarning provides information on the warnings related to interactions with EIS. A ResourceWarning is silently chained to an Interaction instance that has caused the warning to be reported.

Using the CCI API Interfaces

Figure 15-11 shows the relationships between the classes and interfaces of the CCI API.

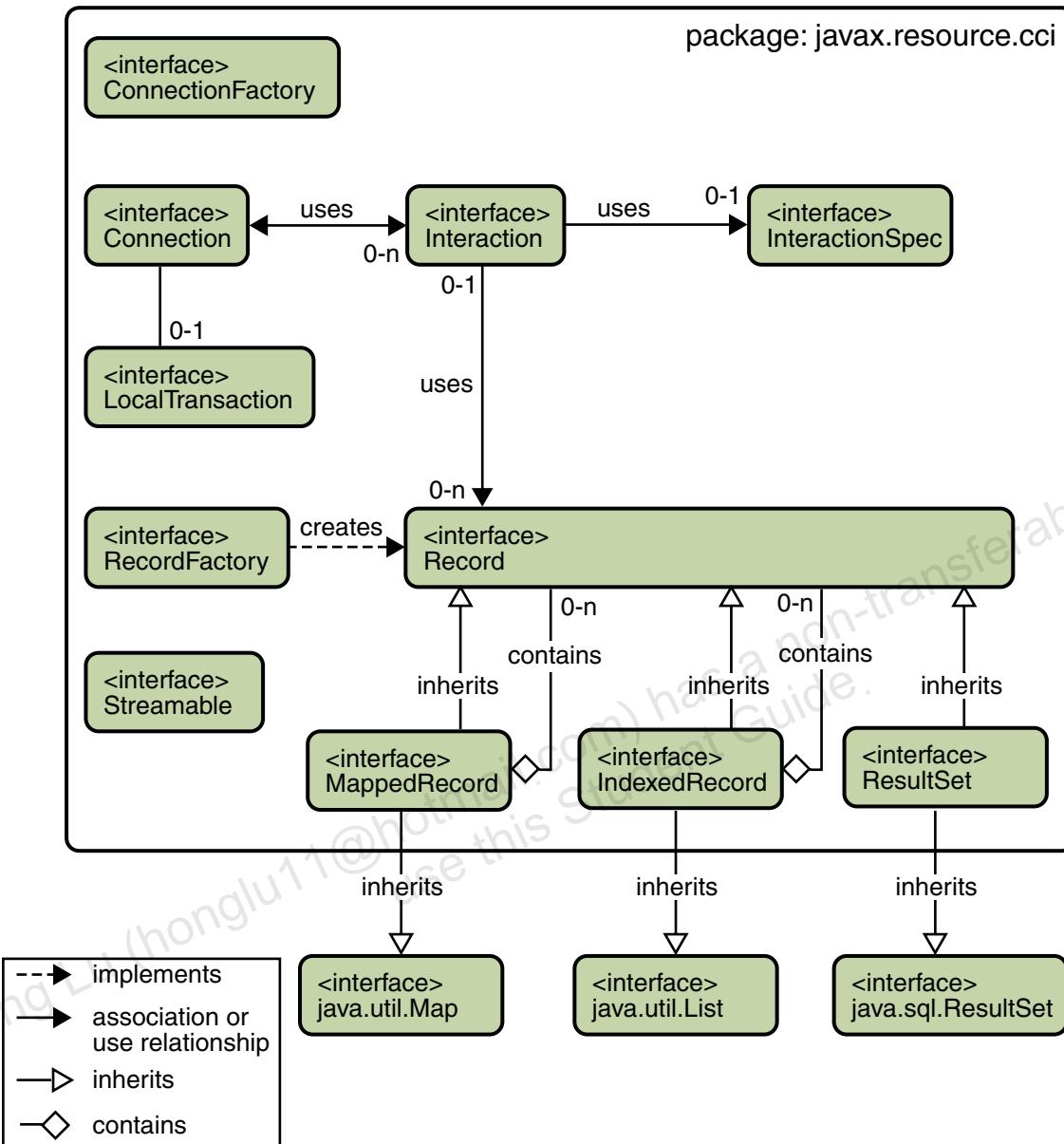


Figure 15-11 CCI Class Diagram

Figure B-3 shows a sequence of method calls that use the CCI interface to interact with a resource.

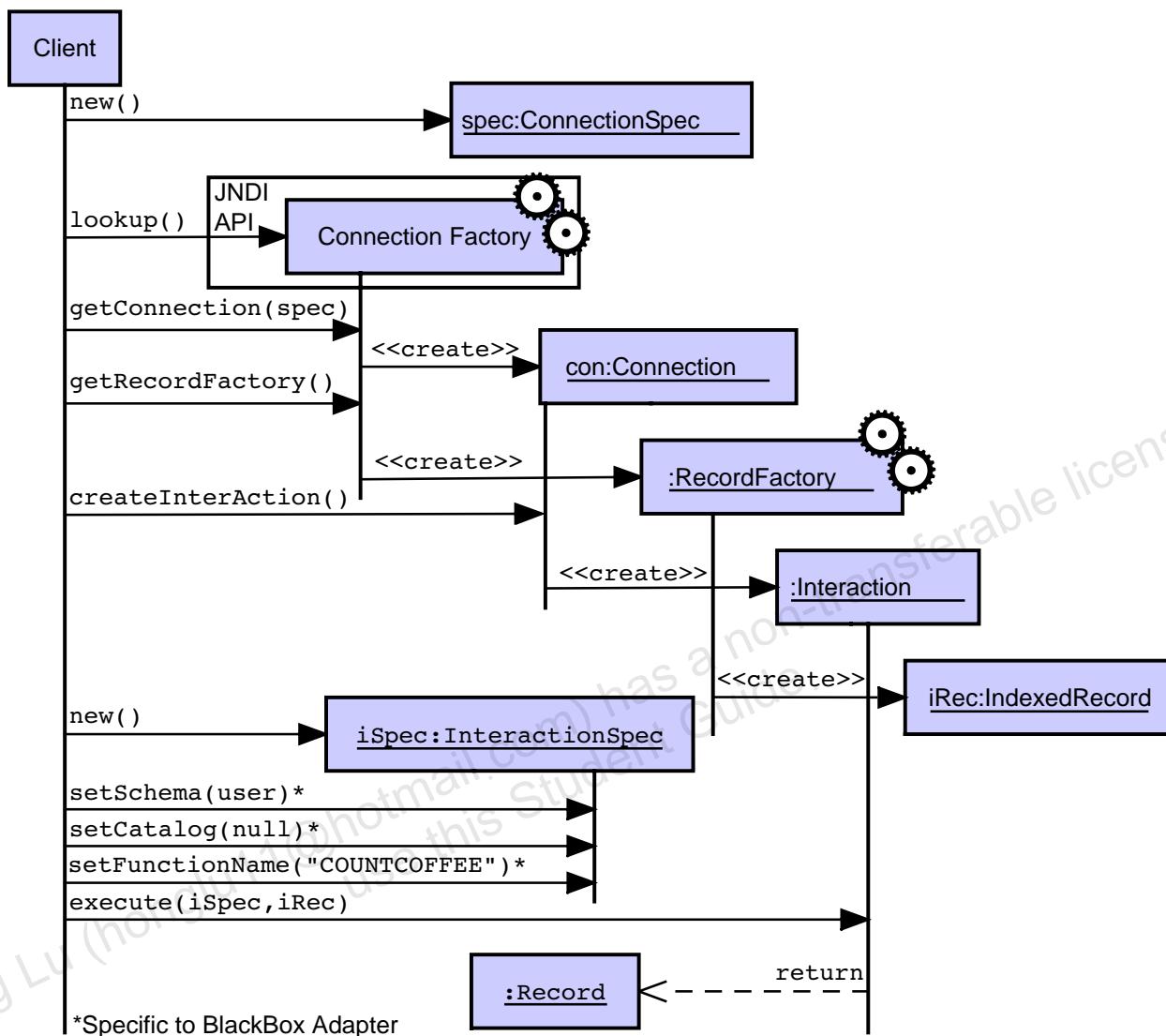


Figure B-3 Overview Sequence Diagram of CCI API

Using the CCI API Interfaces

In Figure B-3 on page B-15, the method calls are grouped into the following set of logical steps. Each step is described in detail in subsequent sections:

1. Create the ConnectionSpec object.
2. Use the ConnectionFactory object.
3. Use the Connection object.
4. Use the Interaction object.

Creating the ConnectionSpec Object

Figure B-4 shows the client creating a ConnectionSpec object.

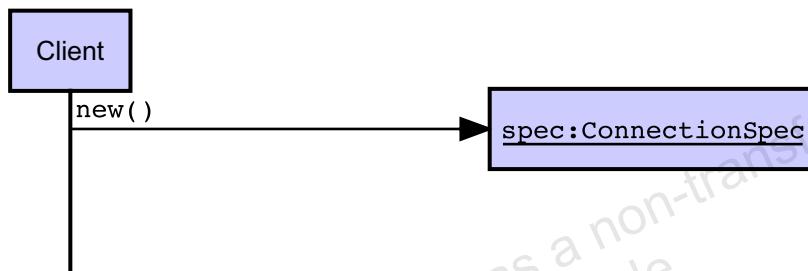


Figure B-4 Creating the ConnectionSpec Object

The following list summarizes the use of the ConnectionSpec object.

- Obtaining the object – The implementing class name is vendor specific. For example, the blackbox-tx resource adapter supplied with Java EE 1.3 SDK names the CciConnectionSpec class. You create the object by invoking the constructor.
- Function – The object provides a means for an application component to pass connection-request-specific properties to the ConnectionFactory when a connection is requested.
- Methods – The ConnectionSpec is an empty interface. The vendor specifies properties using attributes in the implementing class. Properties are set either by using a constructor or the set methods of the implementing class:

```
ConnectionSpec spec = new CciConnectionSpec(user, password); // create
con = cf.getConnection(spec); // using ConnectionSpec to pass properties
                             // to the ConnectionFactory object
```

Properties are read using get methods.

Using the ConnectionFactory Object

Figure B-5 shows how you can use the ConnectionFactory object.

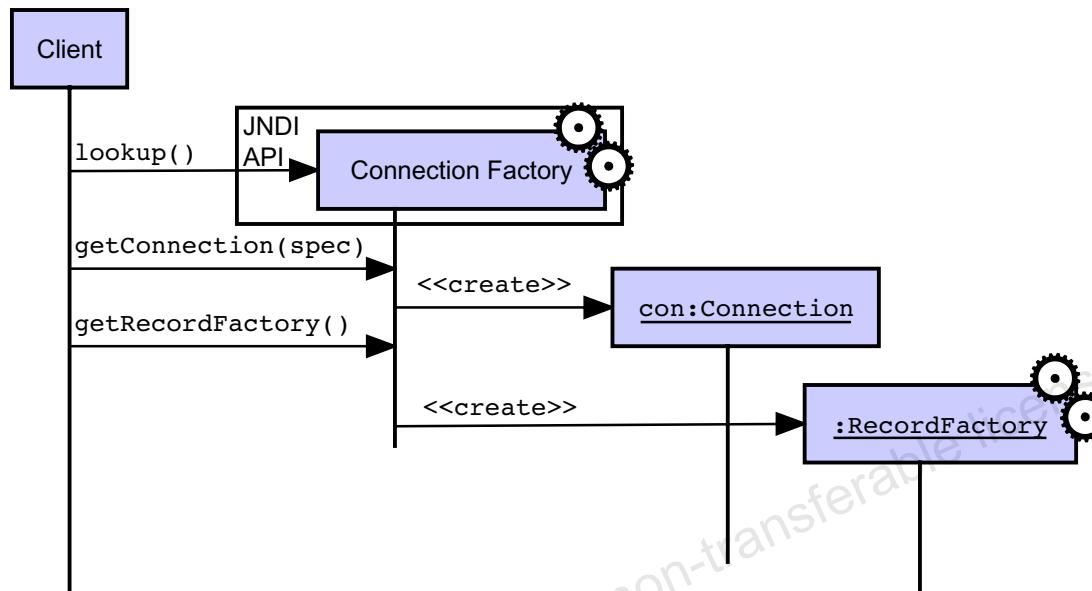


Figure B-5 Using the ConnectionFactory Object

The following list summarizes the use of the ConnectionFactory object

- Obtaining the object – The implementing class name is immaterial. You obtain the object through a naming service lookup.

```

Context ic = new InitialContext();
ConnectionFactory cf = (ConnectionFactory) ic.lookup
("java:comp/env/CCIEIS");

```

- Function – The object provides a means for an application component to obtain the Connection and RecordFactory objects .
- Methods:

- The `getConnection` method returns a `Connection` object.

```
Connection con = cf.getConnection(spec);
```

- The `getRecordFactory` method returns a `RecordFactory` object.

```
RecordFactory rf = cf.getRecordFactory();
```

Using the CCI API Interfaces

Using the Connection Object

Figure B-6 shows the use of the Connection object to create an Interaction object.

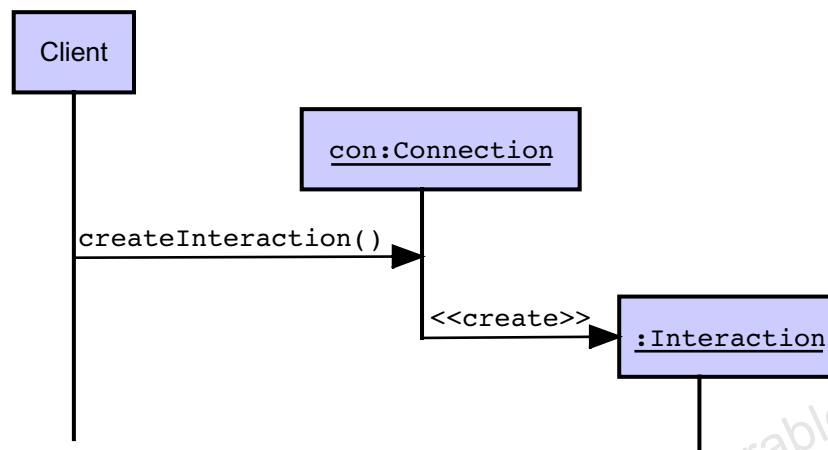


Figure B-6 Using the Connection Object

The following list summarizes the use of this object:

- Obtaining the object – The implementing class name is immaterial. You obtain the object by invoking the factory method `getConnection` on the `ConnectionFactory` object.

```
Connection con = cf.getConnection(spec);
```

- Function – The `Connection` object represents a connection to the underlying EIS resource. It provides factory methods to obtain objects that are used for interacting with the EIS resource.
- Methods – The `Connection` object provides several factory methods used by the application component. Of most interest are:
 - The `createInteraction` method, which returns an `Interaction` object.

```
Interaction ix = con.createInteraction();
```

- The `getLocalTransaction` method, which returns a `LocalTransaction` object.

Using the Interaction Object

Figure B-7 shows the use of an `Interaction` object to request a service on the EIS resource.

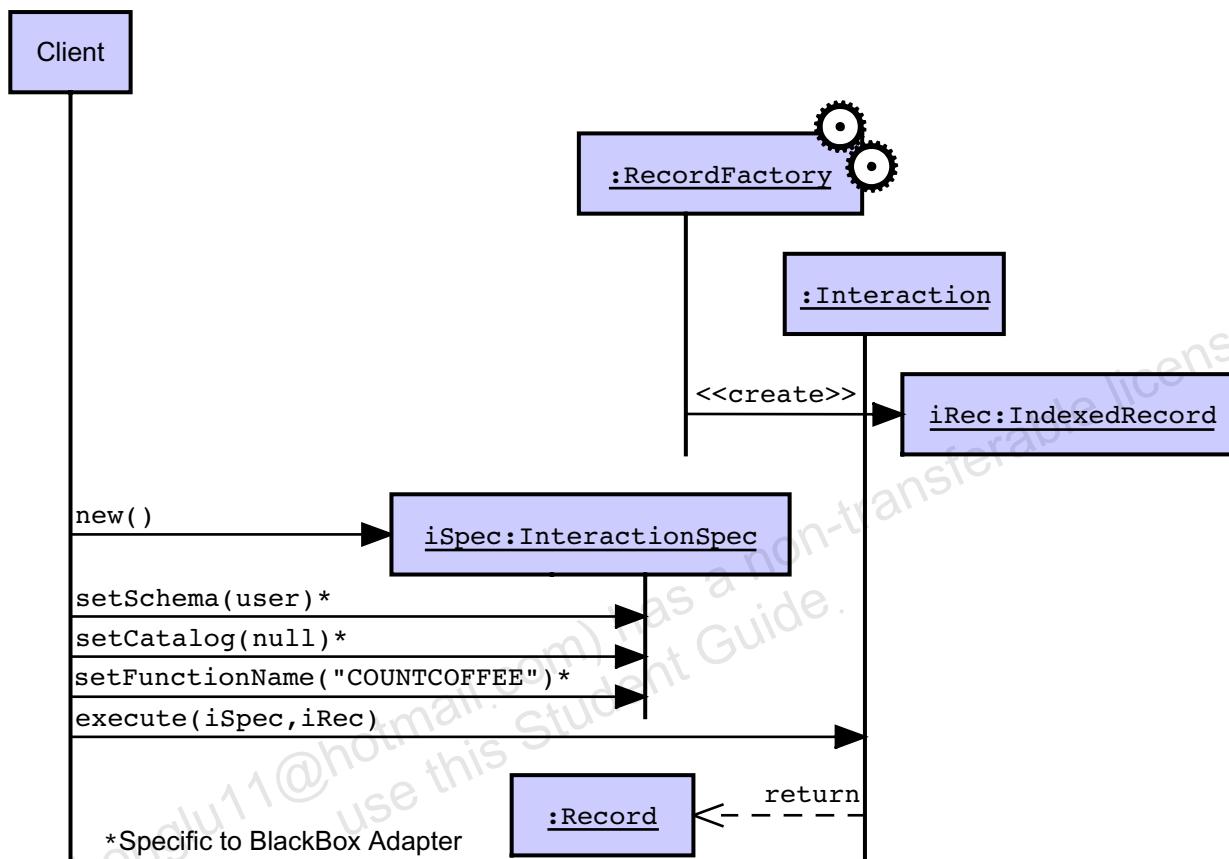


Figure B-7 Using the `Interaction` Object

Using the CCI API Interfaces

The following list summarizes the use of this object:

- Obtaining the object – The implementing class name is immaterial. You obtain the object by invoking the factory method `getInteraction` on the `Connection` object.

```
Interaction ix = con.createInteraction();
```

- Function – The `Interaction` object executes a specific function on the EIS resource.
- Methods – The `Interaction` object provides an `execute` method used for executing a specific function on the EIS resource.

```
Record oRec = ix.execute(iSpec, iRec);
```

The `execute` method takes two parameters: an `InteractionSpec` object and a `Record` object:

- The `InteractionSpec` object – You obtain the object by invoking the constructor. You must have the API of vendor's implementing class.

An `InteractionSpec` object holds properties, such as function name and mode pertaining to an application component's interaction with an EIS. Properties are set either by using a constructor or the `set` methods of the implementing class. Properties are read using `get` methods.

```
CciInteractionSpec iSpec = new CciInteractionSpec();
iSpec.setSchema(user);
iSpec.setCatalog(null);
iSpec.setFunctionName("COUNTCOFFEE");
```

- The `Record` object – You obtain a `Record` object by invoking one of the factory methods on the `RecordFactory` object.

The `RecordFactory` object can create two types of records. An `IndexedRecord` object represents an ordered collection of `Record` object instances based on the `java.util.List` interface. A `MappedRecord` object represents a key-value-based collection of `Record` instances based on the `java.util.Map` interface.

The `Record` interface is the base interface for return types or parameters of the `execute` method defined in the `Interaction` interface.

```
IndexedRecord iRec = rf.createIndexedRecord("InputRecord");
```

Examining a CCI Example

Code B-1 shows an example of CCI code.

Code B-1 CCI Code Sample

```
1 import java.math.BigDecimal;
2 import java.util.*;
3 import javax.resource.cci.*;
4 import javax.resource.ResourceException;
5 import javax.naming.*;
6 import com.sun.connector.cciblackbox.*;
7
8 public class SimpleConnector{
9     public static void main (String args[ ]) {
10         int count = -1;
11         try {
12             Context ic = new InitialContext();
13             String user = (String) ic.lookup("java:comp/env/user");
14             String password = (String)ic.lookup("java:comp/env/password");
15             ConnectionFactory cf =
16                 (ConnectionFactory) ic.lookup("java:comp/env/CCIEIS");
17             ConnectionSpec spec = new CciConnectionSpec(user, password);
18             Connection con = cf.getConnection(spec);
19             Interaction ix = con.createInteraction();
20             CciInteractionSpec iSpec = new CciInteractionSpec();
21             iSpec.setSchema(user);
22             iSpec.setCatalog(null);
23             iSpec.setFunctionName("COUNTCOFFEE");
24             RecordFactory rf = cf.getRecordFactory();
25             IndexedRecord iRec = rf.createIndexedRecord("InputRecord");
26             Record oRec = ix.execute(iSpec, iRec);
27             Iterator iterator = ((IndexedRecord)oRec).iterator();
28             while(iterator.hasNext()) {
29                 Object obj = iterator.next();
30                 if (obj instanceof Integer) {
31                     count = ((Integer)obj).intValue();
32                 }
33                 else if(obj instanceof BigDecimal) {
34                     count = ((BigDecimal)obj).intValue();
35                 }
36             }
37             con.close();
38             System.out.println("coffee count = " + count);
39         } catch (NamingException ex) {
```

Using the CCI API Interfaces

```
40         ex.printStackTrace();
41     } catch(ResourceException ex) {
42         ex.printStackTrace();
43     }
44 }
45
46
```

Using a Message-Driven Bean Resource Adapter: An Example

This section describes message-driven beans that use the *jmsra* resource adapter supplied with the Java EE SDK. The primary purpose of this section is to highlight the resource adapter dependent nature of the tasks involved in deploying a connector-based, message-driven bean.

The *jmsra* resource adapter is a JCA compliant connector that connects to the JMS provider supplied with the Java EE SDK. You can only map JMS message-driven beans to the *jmsra* resource adapter.

The following steps summarize the process of creating a connector-based, message-driven bean.

1. Create a message-driven bean class.

The *jmsra* resource adapter support is restricted to JMS message-driven beans. You can use any class that implements the `javax.ejb.MessageDriven` interface and the `javax.jms.MessageListener` interface. For example, you can use, unchanged, the `PlaceBidMDB` message-driven bean that is used in the Auction application.

2. Create the standard DD for the message-driven bean.

Code B-2 shows the DD segment required in the `ejb-jar.xml` file for a *jmsra* resource adapter based message-driven bean. The *jmsra* resource adapter does not require any connector specific DD elements apart from the standard DD elements required for JMS message-driven beans. Code B-2 is the unchanged standard DD for the `PlaceBidMDB` message-driven bean.

Code B-2 DD Segment for Message-Driven Bean Extracted From the `ejb-jar.xml` File

```
1 <enterprise-beans>
2   <message-driven>
3     <display-name>PlaceBidMDB</display-name>
4     <ejb-name>PlaceBidMDB</ejb-name>
5     <ejb-class>auctionsystem.ejbs.PlaceBidMDBBean</ejb-class>
6     <messaging-type>javax.jms.MessageListener</messaging-type>
7     <transaction-type>Container</transaction-type>
8     <activation-config>
9       <activation-config-property>
10        <activation-config-property-name>
```

Using a Message-Driven Bean Resource Adapter: An Example

```

11      destinationType
12      </activation-config-property-name>
13      <activation-config-property-value>
14          javax.jms.Queue
15      </activation-config-property-value>
16      </activation-config-property>
17  </activation-config>
18  <ejb-local-ref>
19      <ejb-ref-name>ejb/LocalAuctionManager</ejb-ref-name>
20      <ejb-ref-type>Session</ejb-ref-type>
21      <local-home>auctionsyste.ejbs.AuctionManagerLocalHome</local-
home>
22      <local>auctionsyste.ejbs.AuctionManagerLocal</local>
23      <ejb-link>AuctionManagerEJB</ejb-link>
24  </ejb-local-ref>
25  </message-driven>
26 </enterprise-beans>
27

```

3. Create the vendor specific DD for the message-driven bean.

When using the jmsra resource adapter with the Java EE SDK, you must supply a mdb-resource-adapter DD element in the vendor-specific DD (`sun-ejb-jar.xml`) file.

Code B-3 shows the message-driven bean portion of the vendor-specific DD `sun-ejb-jar.xml` file. This file contains the required updates to support the `PlaceBidMDB` message-driven bean using the jmsra resource adapter. The file contains the `resource-adapter-mid` attribute (of the `mdb-resource-adapter` DD element), which maps the resource adapter name `jmsra` to the physical queue name `BidsInQueue` that is specified by the `destination` attribute.

 **Note** – The mapping of the physical name of the JMS queue, including the use of the `mdb-resource-adapter` DD element to specify this mapping, is non-standard and is vendor specific.

Code B-3 Extract From the Vendor-Specific DD

```

1  <sun-ejb-jar>
2      <enterprise-beans>
3          <name>auctionBeansJAR</name>
4          <ejb>
5              <ejb-name>PlaceBidMDB</ejb-name>
6              <jndi-name>jms/placeBidQueue</jndi-name>

```

Using a Message-Driven Bean Resource Adapter: An Example

```
7 <ejb-ref>
8   <ejb-ref-name>ejb/LocalAuctionManager</ejb-ref-name>
9     <jndi-name>myAuctionManagerEJB</jndi-name>
10    </ejb-ref>
11    <mdb-connection-factory>
12      <jndi-name>jms/QueueConnectionFactory</jndi-name>
13        <default-resource-principal>
14          <name>j2ee</name>
15          <password>j2ee</password>
16        </default-resource-principal>
17      </mdb-connection-factory>
18      <mdb-resource-adapter>
19        <resource-adapter-mid>jmsra</resource-adapter-mid>
20        <activation-config>
21          <activation-config-property>
22            <activation-config-property-name>
23              destination
24            </activation-config-property-name>
25            <activation-config-property-value>
26              BidsInQueue
27            </activation-config-property-value>
28          </activation-config-property>
29        </activation-config>
30      </mdb-resource-adapter>
31    </ejb>
```

4. Deploy the resource adapter.

This step is not required for the jmsra resource adapter. It comes pre-deployed with the Java EE SDK.

5. Deploy the message-driven bean.

To perform this step, you package the message-driven bean class with the `ejb-jar.xml` file and the `sun-ejb-jar.xml` file. You then deploy the packaged message-driven bean.

Integrating With Legacy Systems Using the CORBA Protocol

This section investigates the integration of Java EE applications with legacy systems written in C, C++, and other CORBA protocol-compliant languages. This section examines the following topics:

- Interoperability between EJB technology and CORBA protocols
This section examines the compatibility between the EJB technology and CORBA protocols.
- Accessing an enterprise bean with a CORBA client
This section investigates the architecture that is associated with integrating an EJB application as a service provider to a CORBA-compliant legacy system.
- Accessing a CORBA EIS from an EJB component
This section investigates the architecture that is associated with integrating an EJB application with a CORBA EIS.

Examining the Interoperability Between the EJB Technology and CORBA Protocols

This section examines the compatibility between the EJB technology and CORBA protocols. Figure B-8 shows the protocols used by RMI-based and CORBA-based applications.

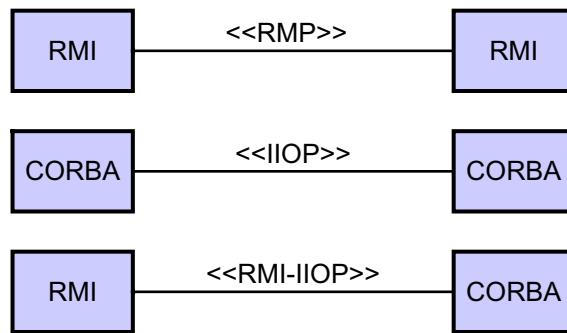


Figure B-8 RMI and CORBA Protocols

These protocols include:

- The Java technology Remote Method Protocol (RMP), which is the RMI transport mechanism for communication between objects in the Java programming language executing in different JVMs. RMP can pass objects either by references or by value.
- The IIOP is the CORBA transport mechanism for communicating between CORBA-compliant objects. Early versions of the IIOP were limited to passing objects by reference. The later versions support both pass-by-reference and pass-by-value.
- The Java RMI-IIOP technology (RMI-IIOP) is the implementation of the RMI transport protocol over IIOP. It combines the advantages of the Java Remote Method Invocation (Java RMI) interface for distributed systems programming with the Object Management Group's CORBA architecture for distributed computing. The RMI-IIOP provides flexibility by allowing developers to pass any Java technology object between application components either by reference or by value.

Accessing an Enterprise Bean With a CORBA Client

This section investigates the architecture that is associated with integrating an EJB application as a service provider to a CORBA-compliant legacy system. In this situation, the legacy system is the client and the EJB application becomes the server. Figure B-9 illustrates this scenario.

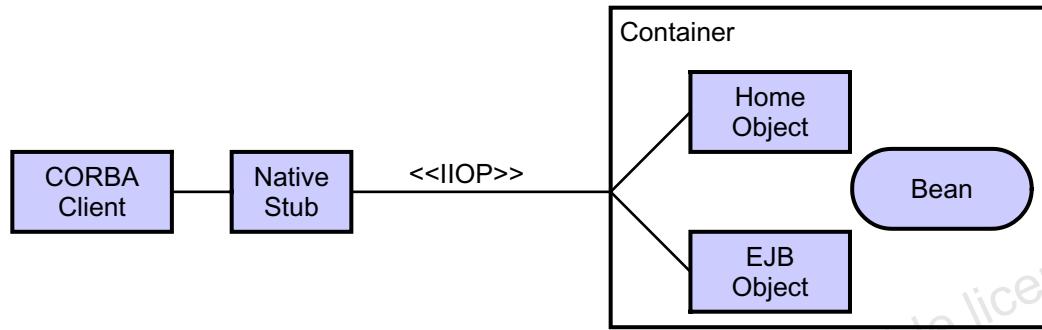


Figure B-9 Accessing an Enterprise Bean With a CORBA Client

A CORBA client requires a native stub to access an enterprise bean. The native stub provides the required networking code to communicate with the enterprise bean. The CORBA client believes that it is invoking methods on the enterprise bean. In reality, it invokes methods on the stub, which forwards the request to the enterprise bean and returns the reply sent by the enterprise bean to the CORBA client.

The following tasks are performed by the enterprise bean developer:

1. Obtain the remote home and remote component interface.
The remote home and remote component interfaces are created and published by the bean developer.
2. Generate RMI-IIOP skeletons and the Object Management Group - Interface Definition Language (OMG - IDL) interface for each interface.
The skeleton is the server-side counterpart to the stub. The skeleton forwards the requests from the stub to the enterprise bean.
You generate the skeleton and the `idl` file by executing the `rmic` compiler with the `-idl` option.
3. Ensure that the enterprise bean is deployed.
If required, deploy the enterprise bean in the Java EE technology application server.

The following tasks are performed by the native-client code developer.

1. Use an IDL-to-native language compiler to generate the client-side stub.

The instructions for this task are compiler specific.

2. Write the client code.

Incorporate the stub in the client code.

Figure B-10 shows the tasks performed to create the native stub.

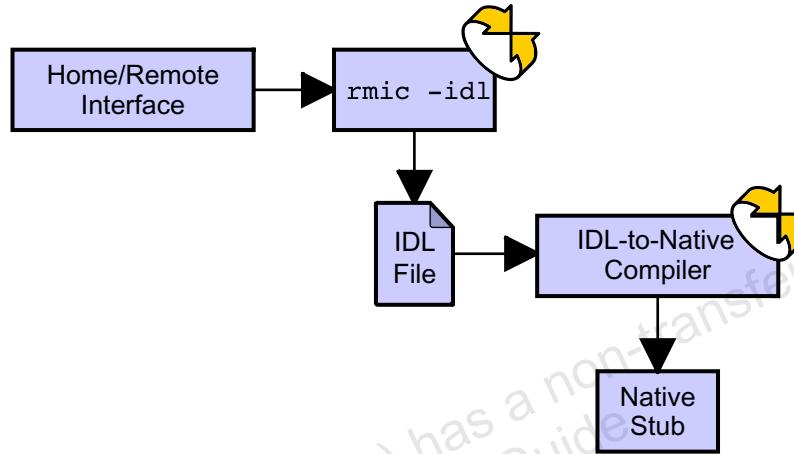


Figure B-10 Creating the Native Stub

Accessing a CORBA EIS From an EJB Component

Figure B-11 shows an enterprise bean accessing an EIS that provides a CORBA interface.

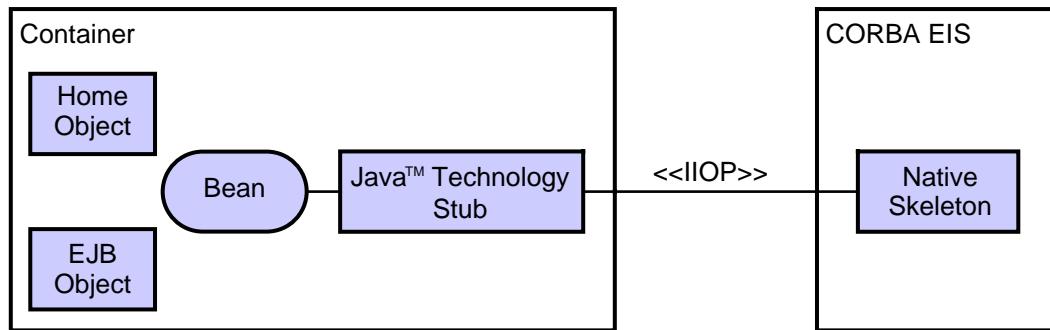


Figure B-11 Accessing a CORBA EIS From an EJB Component

To access a CORBA EIS, you need a Java technology stub. Figure B-12 shows the process of creating the Java technology stub.

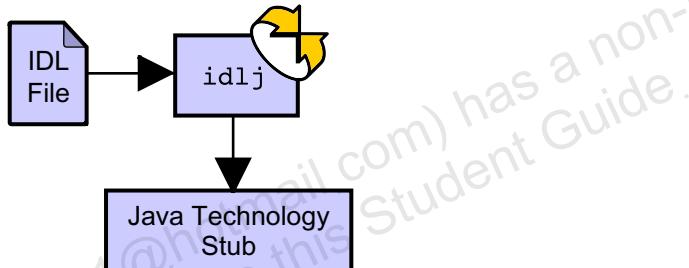


Figure B-12 Creating the Java Technology Stub

The CORBA EIS vendor supply an IDL interface for the EIS.

The following tasks are performed by the EJB component developer:

1. Obtain the IDL interface to the CORBA EIS.
The EIS vendor is responsible for creating and publishing the IDL interface to the EIS.
2. Generate the Java technology stub classes by compiling the IDL interface using a Java technology-to-IDL compiler (`idlj`).
3. Include code in the EJB component that accesses the EIS through the methods of the stub class.

Appendix C

Quick Reference for UML

This appendix is designed to serve as a quick reference for the Unified Modeling Language (UML) version 1.4, which was released in September, 2001.

Additional Resources



Additional resources – The following references provide additional details on the topics described in this appendix:

- Booch, Grady, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Reading: Addison Wesley Longman, Inc., 1999.
- Folwer, Martin, with Kendall Scott. *UML Distilled (2nd ed)*. Reading: Addison Wesley Longman, Inc., 2000.
- The Object Management Group. “OMG Unified Modeling Language Specification,” [<http://www.omg.org/technology/documents/formal/uml.htm>], version 1.4, September 2001.

Note – Additional UML resources are available online at:
<http://www.omg.org/uml/>.



UML Basics

Unified Modeling Language (UML) is a graphical language for modeling software systems. UML is not a programming language, it is a set of diagrams that can be used to specify, construct, visualize, and document software designs. Software engineers use UML diagrams to construct and explain their software designs just as a building architect uses blueprints to construct and explain their building designs. UML has diagrams to assist in every part of the application development process from requirements gathering through design, and into coding, testing, and deployment.

UML was developed in the early 1990's by three leaders in the object modeling world: Grady Booch, James Rumbaugh, and Ivar Jacobson. Their goal was to unify the major methods that they had previously developed to create a new standard for software modelling. UML is now the most commonly used modeling language. UML is currently maintained by the Object Management Group (OMG). The UML specification is available on the OMG web site at <http://www.omg.org/uml/>.

The UML is not a process for how to do analysis and design. UML is only a set of tools to use in a process. UML is frequently used with a process such as the Unified Software Development Process. Sun Microsystems OO-226, *Object Oriented Analysis and Design Using UML*, is a five day course that teaches effective methods of analysis and design using UML language, the USDP method, and software patterns.

UML defines nine standard types of diagrams. Table C-1 provides a list of these diagrams, an informal description, and best-use recommendations.

Table C-1 Types of UML Diagrams

Diagram Name	Description	Best Use
Use Case	A Use Case diagram is a simple diagram that shows who is using your system and what processes they will perform in the system.	The Use Case diagram is an extremely important diagram for the Requirements Gathering and Analysis work flows. Throughout the entire development, all work should be traceable back to the Use Case diagram.

UML Basics**Table C-1** Types of UML Diagrams (Continued)

Diagram Name	Description	Best Use
Class	A Class diagram shows a set of classes in the system and the associations and inheritance relationships between the classes. Class nodes might also contain a listing of attributes and operations.	The Class diagram is essential for showing the structure of the system and what needs to be programmed. Most UML case tools can generate code based on the Class diagram.
Object	An Object diagram shows specific object instances and the links between them. An Object diagram represents a snapshot of the system objects at a specific point in time.	The Object diagram can be used to clarify or validate the Class diagram.
Activity	An Activity diagram is essentially a flow chart with new symbols. This diagram represents the flow of activities in a process or algorithm.	Even in an OO system, it can sometimes be useful to consider processes without thinking in terms of objects. Activity diagrams are especially useful for modeling real world business systems during the Requirements Gathering workflow.
Collaboration	The Sequence diagram and the Collaboration diagram both show processes from an object oriented perspective. The main difference is that the layout of the Collaboration diagram puts more focus on the objects rather than the sequence.	Sequence diagrams are typically easier to read than Collaboration diagrams. Many people prefer to only use Sequence diagrams. A Collaboration diagram might be preferable if you want more focus on the objects than the sequence.
Sequence	Sequence diagrams show a process from an object oriented perspective by showing how a process is executed by a set of objects or actors.	The Sequence diagram is essential for assigning responsibilities to classes by considering how they can work together to implement the processes in the system.

Table C-1 Types of UML Diagrams (Continued)

Diagram Name	Description	Best Use
Statechart	A Statechart diagram shows how a particular object changes behavioral state as various events happen to it.	The Statechart diagram is very useful in understanding objects that change behavioral states in significant ways.
Component	A Component diagram shows the major software components of a system and how they can be integrated. Component diagrams can contain non-OO software components such as legacy procedural code and web documents.	Component diagrams can be a good way to show how all of the OO and non-OO components fit together in your system. It is also a good way to look at the high-level software structure of your system.
Deployment	A Deployment diagram shows the hardware nodes in the system.	The Deployment diagram is useful for seeing how a distributed system will be configured. Software components might be displayed inside the hardware nodes to show how they will be deployed.

General Elements

General Elements

In general, UML diagrams represent concepts, depicted as symbols (also called nodes), and relationships among concepts, depicted as paths (also called links) connecting the symbols. These nodes and links are specialized for the particular diagram. For example, in Class diagrams, the nodes represent object classes and the links represent associations between classes and generalization (inheritance) relationships.

There are other elements that augment these diagrams, including: packages, stereotypes, annotations, constraints, and tagged values.

Packages

In UML, packages enable you to arrange your modeling elements into groups. UML packages are a generic grouping mechanism and should not be directly associated with Java technology packages. However, you can use UML packages to model Java technology packages. Figure C-1 demonstrates a package diagram that contains a group of classes in a Class diagram.

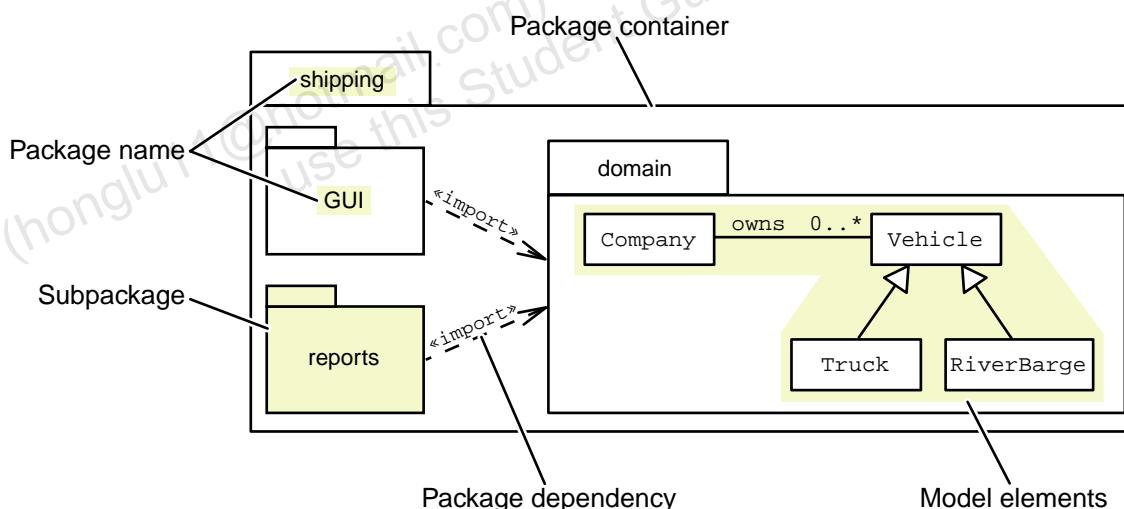


Figure C-1 Example Package

Mapping to Java Technology Packages

The mapping of UML packages to Java technology packages implies that the classes would contain the package declaration of package `shipping.domain`. For example, in the file `Vehicle.java`:

```
package shipping.domain;

public class Vehicle {
    // declarations
}
```

Figure C-1 on page C-6 also demonstrates a simple hierarchy of packages. The `shipping` package contains the `GUI`, `reports`, and `domain` subpackages. The dashed arrow from one package to another indicates that the package at the tail of the arrow uses (`imports`) elements in the package at the head of the arrow. For example, `reports` uses elements in the `domain` package as follows:

```
package shipping.reports;

import shipping.domain.*;

public class VehicleCapacityReport {
    // declarations
}
```

Note – In Figure C-1 on page C-6, the `shipping.GUI` and `shipping.reports` packages have their names in the body of the package box rather than in the head of the package box. This is done only when the diagram does not expose any of the elements in that package.



General Elements

Stereotypes

The designers of UML understood that they could not build a modeling language that would satisfy every programming language and every modeling need. They built several mechanisms into UML to enable modelers to create their own semantics for model elements (nodes and links). Figure C-2 shows the use of a stereotype tag «interface» to declare that the class node Set is a Java technology interface declaration. Stereotype tags can adorn relationships as well as nodes. There are over a hundred standard stereotypes. You can create your own stereotypes to model your own semantics.

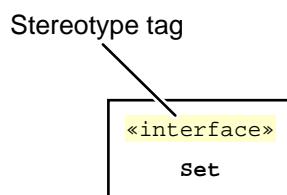


Figure C-2 Example Stereotype

Annotation

The designers of UML also built a method for annotating the diagrams into the UML language. Figure C-3 shows a simple annotation.

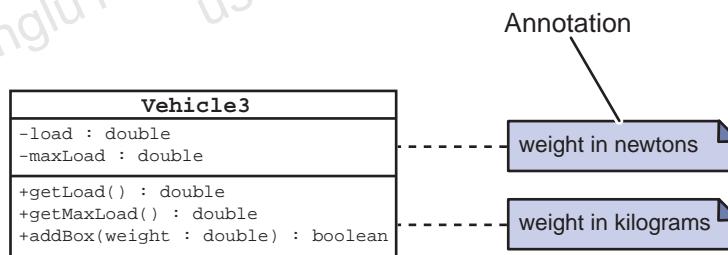


Figure C-3 Example Annotation

Annotations can contain notes about the diagram as a whole, notes about a particular node or link, or even notes about an element within a node. The dotted link from the annotation points to the element being annotated. If there is no link from the annotation, then the note is about the whole diagram.

Constraints

Constraints enable you to model certain conditions that apply to a node or link. Figure C-4 shows several constraints. The topmost constraint specifies that the Player objects must be stored in a persistent database. The middle constraint specifies that the captain and co-captain must also be members of the team's roster. The constraint on the bottom specifies the minimum number of players by gender.

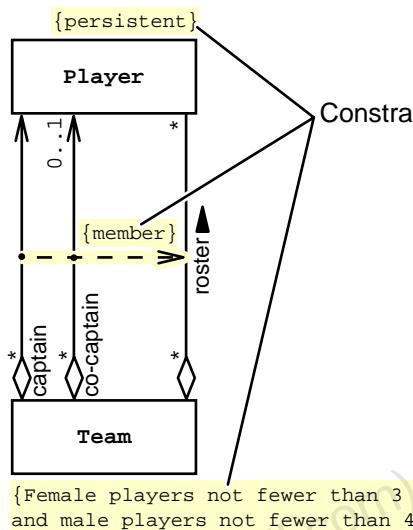


Figure C-4 Example Constraints

Tagged Values

Figure C-5 shows several examples of how tagged values enable you to add new properties to nodes in a diagram.

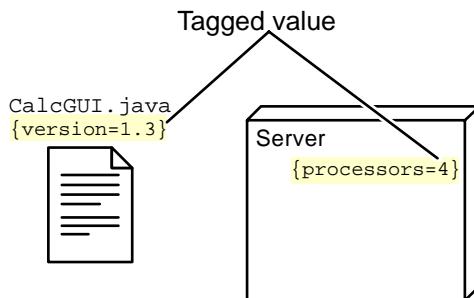


Figure C-5 Example Tagged Values

Use Case Diagrams

Use Case Diagrams

A Use Case diagram represents the functionality provided by the system to external users. The Use Case diagram is composed of actors, use case nodes, and their relationships. Actors can be humans or other systems.

Figure C-6 shows a simple banking Use Case diagram. An actor node can be denoted as a *stick figure* (as in the three Customer actors) or as a class node (see “Class Nodes”) with the stereotype of «actor». There can be a hierarchy of actors.

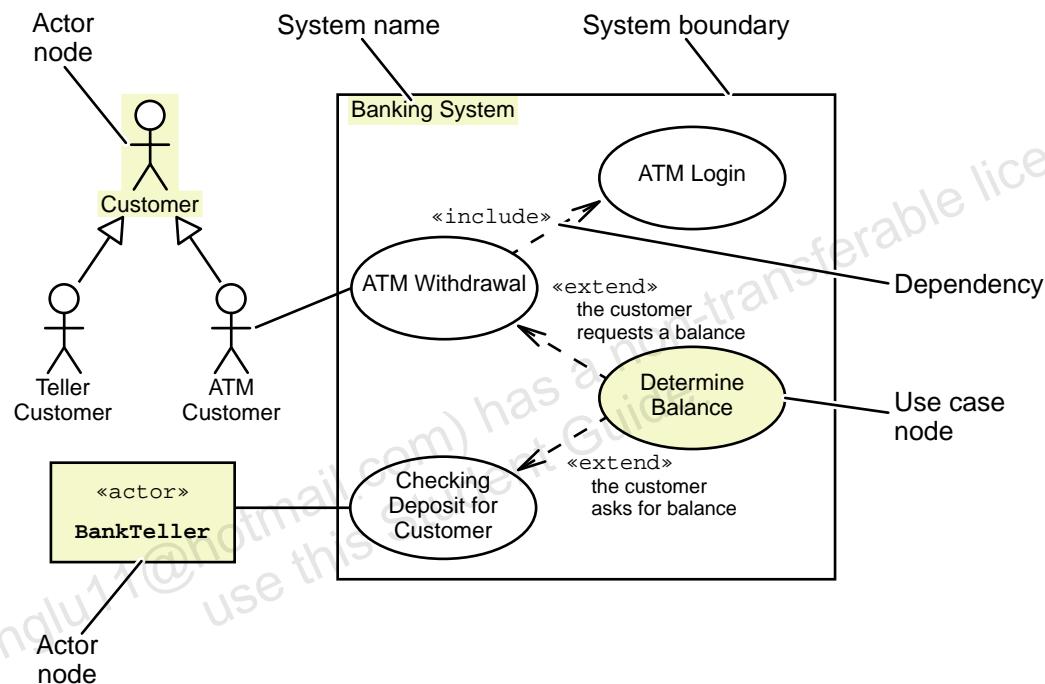


Figure C-6 An Example Use Case Diagram

A use case node is denoted by a labelled oval. The label indicates the activity summary that the system performs for the actor. Use case nodes are grouped into a system box, which is usually labelled in the top left corner. The relationship “actor uses the system to” is represented by the solid line from the actor to the use case node.

Use case nodes can depend on other use cases. For example, the “ATM Withdrawal” use case uses the “ATM Login” use case. Use case nodes can also extend other use cases to provide optional functionality. For example, the “Determine Balance” use case can be used to extend the “Checking Deposit for Customer” use case.

Class Diagrams

A Class diagram represents the static structure of a system. These diagrams are composed of classes and their relationships.

Class Nodes

Figure C-7 shows several *class nodes*. You do not have to model every aspect of a class every time that class is used in a diagram.

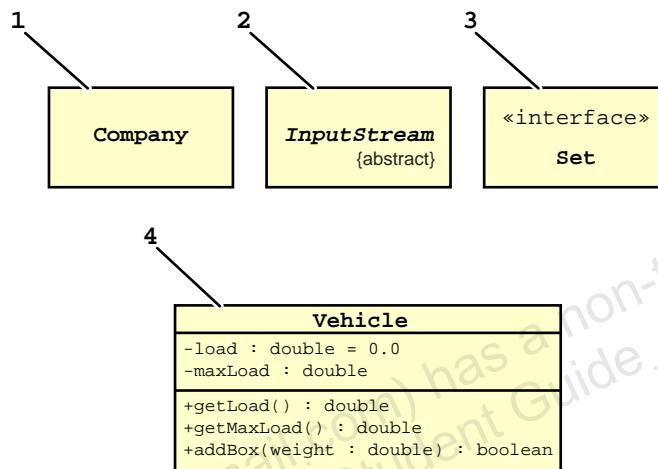


Figure C-7 Several Class Nodes

A class node can just be the name of the class, as in Examples 1, 2, and 3 of the figure. Example 1 is a concrete class, where no members are modeled. Example 2 is an abstract class (name is in italics). Example 3 is an interface. Example 4 is a concrete class, where members are modeled

Class Diagrams

Figure C-8 illustrates the elements of a class node.

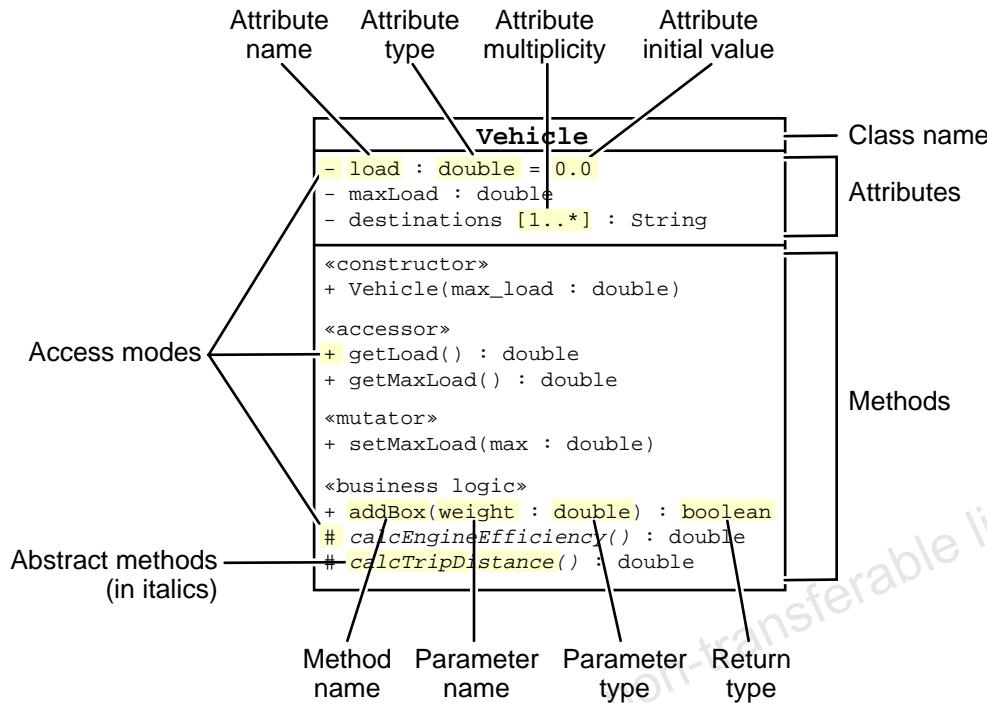


Figure C-8 Elements of a Class Node

A fully specified class node has three basic compartments:

- The name of the class in the top
- The set of attributes under the first bar

An attribute is specified by five elements, including access mode, name, multiplicity, data type, and initial value. With attributes, all of the elements are optional except for the name element.

- The set of methods under the second bar

A method is specified by four elements, including access mode, name, parameter list (a comma-delimited list of parameter name and type), and the return type. With methods, all but the name are optional, except for the name element. If the return value is not specified, then no value is returned (`void`). The name of an abstract method is marked in italics.

You can use stereotypes to group attributes or methods together. For example, you can separate accessor, mutator, and business logic methods from each other for clarity. And because there is no UML-specific notation for constructors, you can use the `«constructor»` stereotype to mark the constructors in your method compartment.

Table C-2 shows the valid UML access mode symbols.

Table C-2 UML Defined Access Modes and Their Symbols

Access Mode	Symbol
private	-
package private	~
protected	#
public	+

Figure C-9 shows an example class node with elements that have class (or static) scope. This is denoted by the underline under the element. For example, counter is a static data attribute and getTotalCount is a static method.

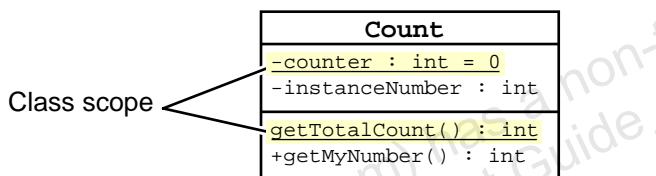


Figure C-9 An Example Class Node With Static Elements

You could write the Count class in the Java programming language as:

```
public class Count {
    private static int counter = 0;
    private int instanceNumber;

    public static int getTotalCount() {
        return counter;
    }
    public int getMyNumber() {
        return instanceNumber;
    }
}
```

Class Diagrams

Inheritance

Figure C-10 shows class inheritance through the generalization relationship arrow.

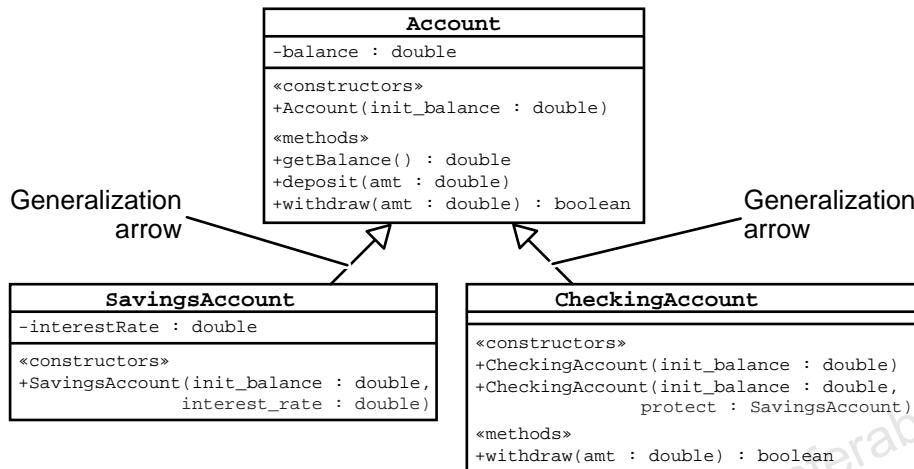


Figure C-10 Class Inheritance Relationship

Class inheritance is implemented in the Java programming language with the `extends` keyword. For example:

```

public class Account {
    // members
}

public class SavingsAccount extends Account {
    // members
}

public class CheckingAccount extends Account {
    // members
}
  
```

Interface Implementation

Figure C-11 shows how to use the “realization” arrow to model a class that is implementing an interface.

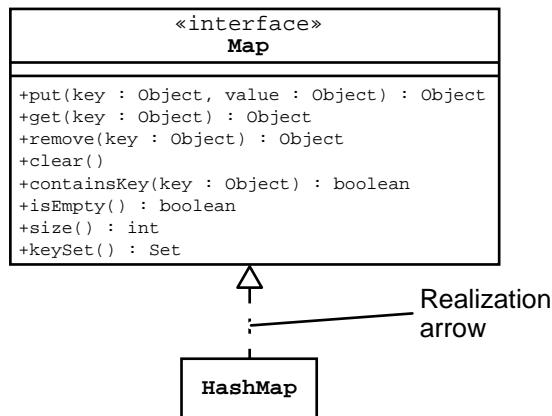


Figure C-11 An Example of a Class Implementing an Interface

An interface is implemented in the Java programming language with the `implements` keyword. For example:

```
public interface Map {  
    // declaration here  
}  
  
public class HashMap implements Map {  
    // definitions here  
}
```

Class Diagrams

Association, Roles, and Multiplicity

Figure C-12 shows an example association. An *association* is a link between two types of objects and implies the ability for the code to navigate from one object to another.

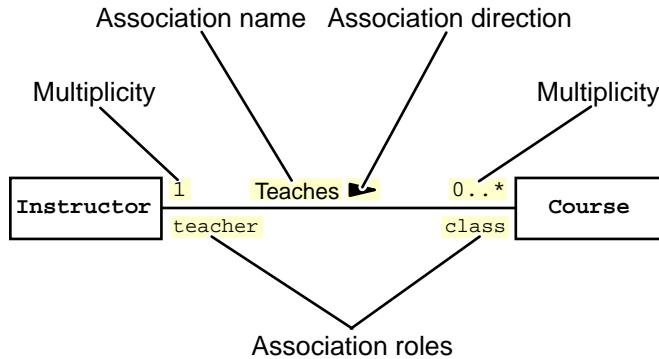


Figure C-12 Class Associations and Roles

In this diagram, “Teaches” is the name of the association with a directional arrow pointing to the right. This association can be read as “an instructor teaches a course.” You can also attach roles to each end of the association. In the figure, the “teacher” role indicates that the instructor is the teacher for a given course. All of these elements are optional if the association is obvious.

This example also demonstrates how many objects are involved in each role of the association. This is called *multiplicity*. In this example, there is only one teacher for every class, which is denoted by the “1” on the **Instructor** side of the association. Also any given teacher might teach zero or more courses, which is denoted by the “**0..***” on the **Course** side. You can leave out the multiplicity for a given role if it is always one. You can also abbreviate “zero or more” as just “*”.

You can express multiplicity values as follows:

- A range of values – For example, **2..10** means “at least 2 and up to 10
- A disjoint set of values – For example, **2,4,6,8,10**
- A disjoint set of values or ranges – For example, **1..3,6,8..11**

However, the most common values for multiplicity are exactly one (1 or left blank), zero or one (**0..1**), zero or more (*), or one or more (**1..***).

Associations are typically represented in the Java programming language as an attribute in the class at the tail of the relationship (specified by the direction indicator). If the multiplicity is greater than one, then some sort of collection or array is necessary to hold the elements.

Also, in Figure C-12 on page C-16 the association between an instructor and a course might be represented in the Instructor class as:

```
public class Instructor {  
    private Course[] classes = new Course[MAXIMUM];  
}
```

or as:

```
public class Instructor {  
    private List classes = new ArrayList();  
}
```

The latter representation is preferable if you do not know the maximum number of courses any given instructor might teach.

Aggregation and Composition

An *aggregation* is an association in which one object contains a group of parts that make up the *whole* object (see Figure C-13). For example, a car is an aggregation of an engine, wheels, body, and frame. Composition is a specialization of aggregation in which the parts cannot exist independently of the *whole* object. For example, a human is a composition of a head, two arms, two legs, and a torso. If you remove any of these parts without surgical intervention, the whole person is going to die.

Class Diagrams

In the example in Figure C-13, a sports league, defined as a sports event that occurs seasonally every year, is a composition of divisions and schedules. A division is an aggregation of teams. A team might exist independently of a particular season; in other words, a team might exist for several seasons (leagues) in a row. Therefore, a team might still exist even if a division is eliminated. Moreover, a game can only exist in the context of a particular schedule of a particular league.

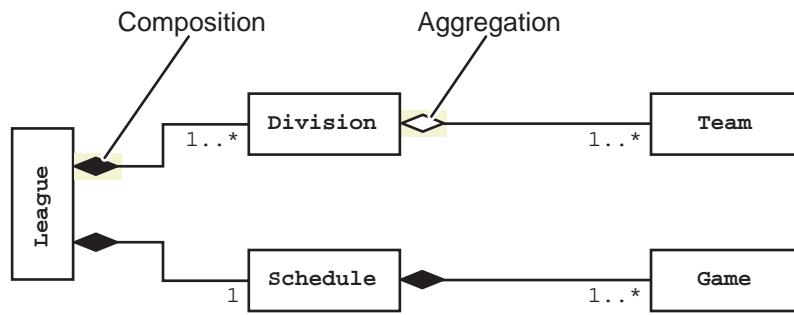


Figure C-13 Example Aggregation and Composition

Association Classes

An association between two classes might have properties and behavior of its own. Association classes are used to model this characteristic. For example, players might be required to register for a particular division within a sports league, as Figure C-14 shows. The association class is attached to the association link by a dashed line.

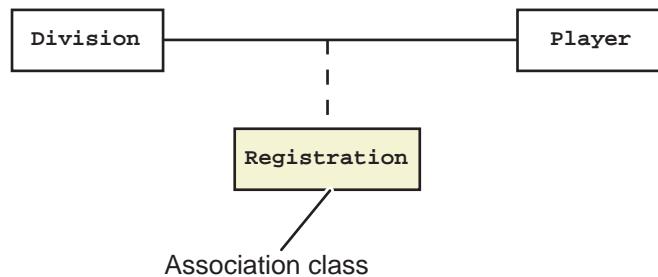


Figure C-14 A Simple Association Class

Figure C-15 shows an association class that is used by two associations and that has two private attributes. This example indicates that a Game object is associated with two opposing teams and each team will have a score and a flag specifying whether that team forfeited the game.

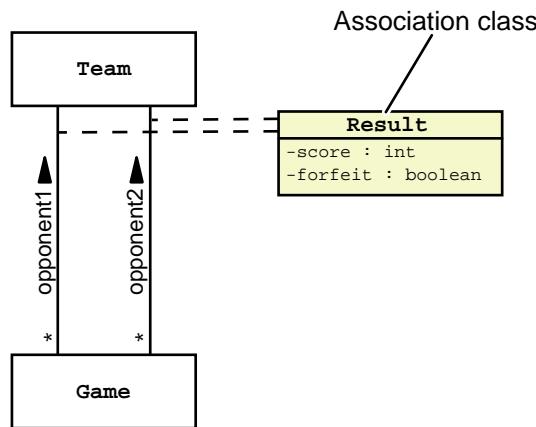


Figure C-15 A More Complex Association Class

Class Diagrams

You can use the Java programming language to represent an association class in several different ways. One way is to code the association class as a standard Java technology class. For example, registration could be coded as follows:

```
public class Registration {  
    private Division division;  
    private Player player;  
    // registration data  
    // methods...  
}  
public class Division {  
    // data  
    public Registration retrieveRegistration(Player p) {  
        // lookup registration info for player  
        return new Registration(this, p, ...);  
    }  
    // methods...  
}
```

Another technique is to code the association class attributes directly into one of the associated classes. For example, the Game class might include the score information as follows:

```
public class Game {  
    // first opponent and score details  
    private Team opponent1;  
    private int opponent1_score;  
    private boolean opponent1_forfeit;  
    // second opponent and score details  
    private Team opponent2;  
    private int opponent2_score;  
    private boolean opponent2_forfeit;  
    // methods...  
}
```

Other Association Elements

There are several other parts of associations. This section presents the constraints and qualifiers elements.

An association constraint enables you to augment the semantics of two or more associations by attaching a dependency arrow between them and tagging that dependency with a constraint. For example in Figure C-16, the captain and co-captain of a team are also members of the team's roster.

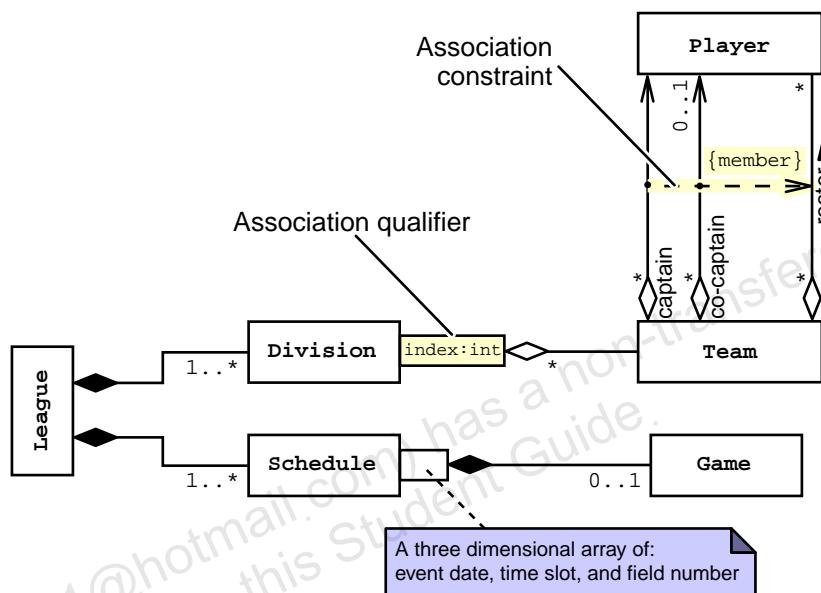


Figure C-16 Other Associations Elements

An association qualifier provides a modeling mechanism to state that an object at one end of the association can look up another object at the other end. For example, a particular game in a schedule is uniquely identified by an event date, such as Saturday August 12, 2000, a time-slot, such as 11:00 a.m. to 12:30 p.m., and a particular field number, such as field #2). One particular implementation might be a three dimension array (for example, Game[][][]), in which each qualifier element (date, time-slot, field#) is mapped to an integer index.

Object Diagrams

An Object diagram represents the static structure of a system at a particular instance in time. These diagrams are composed of object nodes, associations, and sometimes class nodes.

Figure C-17 shows a hierarchy of objects that represent a set of teams in a single division in a soccer sports league. This diagram shows one configuration of objects at a specific point of time in the system. Object nodes only show instance attributes because methods are elements of the class definition. Also, an Object diagram does need not to show every associated object, it just needs to be representative.

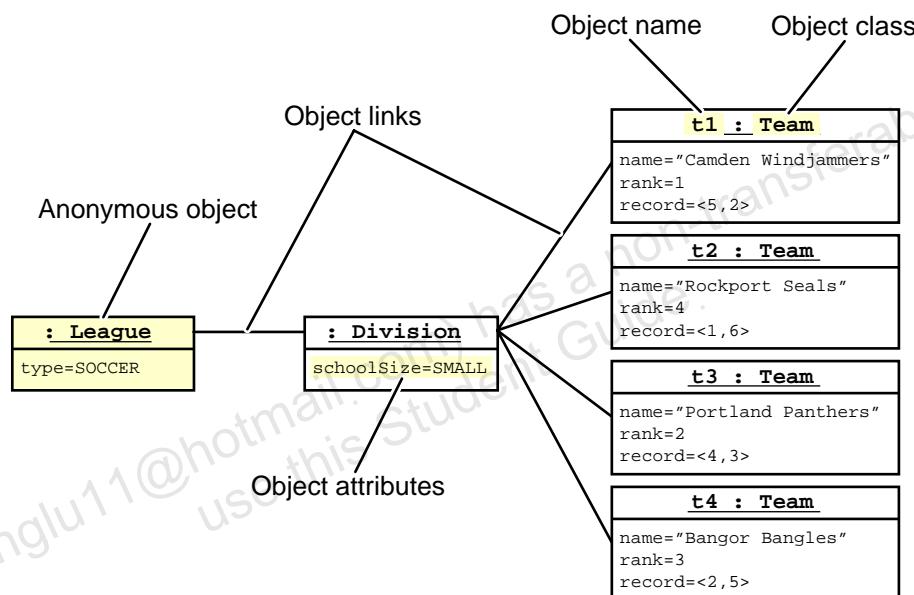


Figure C-17 An Example Object Diagram

Figure C-18 shows two objects, $c1$ and $c2$, with their instance data. They refer to the class node for Count, and the dependency arrow indicates that the object is an instance of the class Count. The objects do not include the counter attribute because it has class scope.

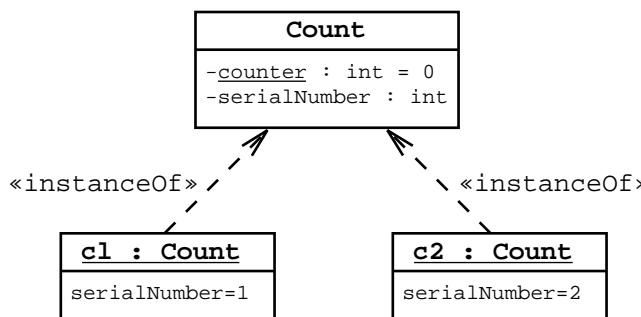


Figure C-18 An Example Object Diagram

Collaboration Diagrams

A Collaboration diagram represents a particular behavior shared by several objects. These diagrams are composed of objects, their links, and the message exchanges that accomplish the behavior.

Figure C-19 shows a Collaboration diagram in which an actor initiates a login sequence within a web application using a servlet.

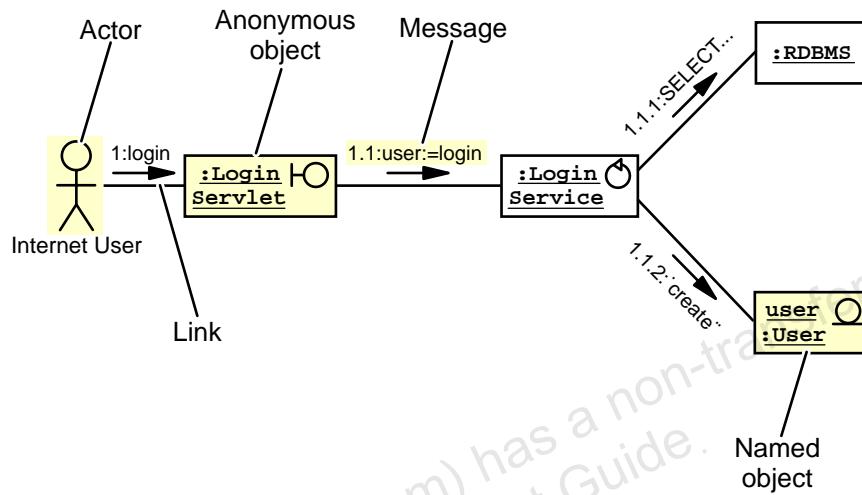


Figure C-19 An Example User-driven Collaboration Diagram

The servlet uses an object of the LoginService class to perform the lookup of the user name, verify the password, and create the User object. The links between objects show the dependencies and collaborations between these objects. Messages between objects are shown by the messages on the links. Messages are indicated by an arrow in the direction of the message, and a text string declares the type of message. The text of this message string is unrestricted. Messages are also labelled with a sequence number so you can see the order of the message calls.

Figure C-20 shows a more elaborate Collaboration diagram. In this diagram, some client object initiates an action on a session bean. This session bean then performs two database modifications within a single transaction context.

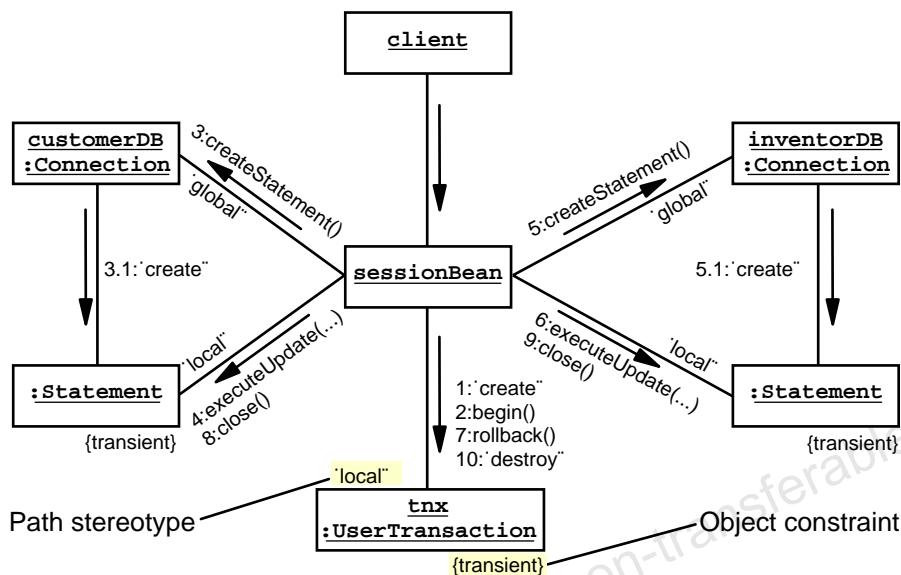


Figure C-20 Another Collaboration Diagram

You can label the links with a stereotype to indicate if the object is global or local to the call sequence. In this example, the connection objects are global, and the statement and transaction objects are local.

You can also label objects with a constraint to indicate if the object is transient.

Sequence Diagrams

Sequence Diagrams

A Sequence diagram represents a time sequence of messages exchanged between several objects to achieve a particular behavior. A Sequence diagram enables you to understand the flow of messages and events that occur in a certain process or collaboration. In fact, a Sequence diagram is just a time-ordered view of a Collaboration diagram (see page C-24).

Figure C-21 shows a Sequence diagram in which an actor initiates a login sequence within a web application which uses a servlet. This diagram is equivalent to the Collaboration diagram in Figure C-19 on page C-24. An important aspect of this type of diagram is that time is moving from top to bottom. The Sequence diagram shows the time-based interactions between a set of objects or roles. Roles can be named or anonymous and usually have a class associated with them. Roles can also be actors.

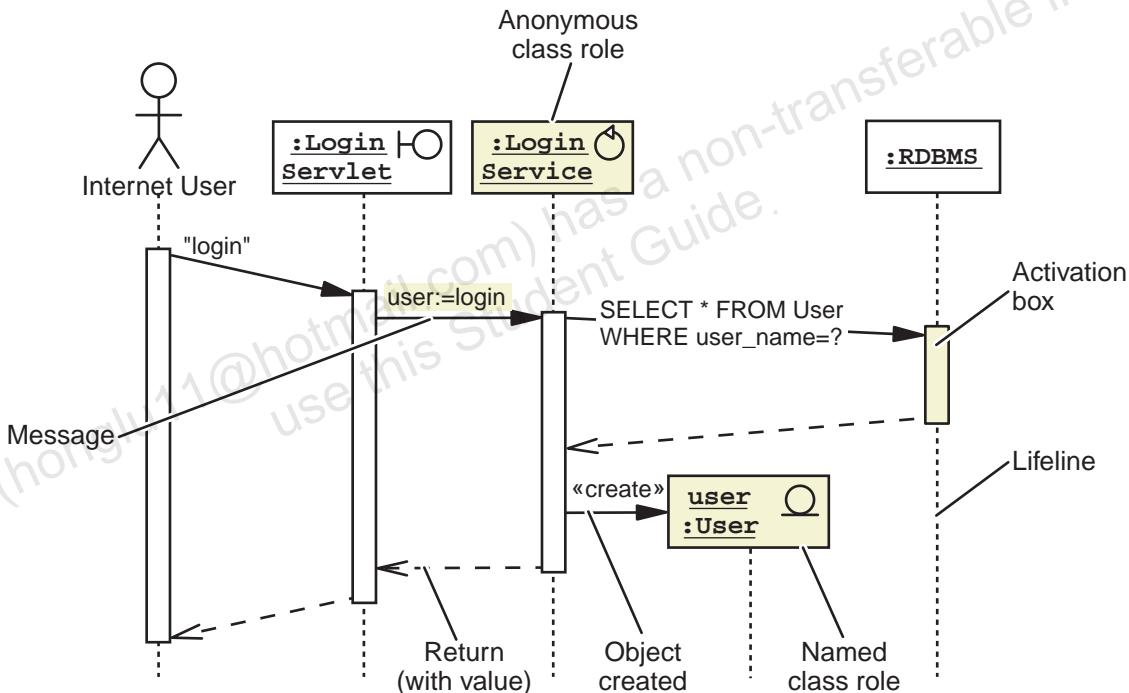


Figure C-21 An Example User-driven Sequence Diagram

Notice the message arrows between the servlet and the service object. The arrow is perfectly horizontal, which indicates that the message is probably implemented by the local method call. Notice that the message arrow between the actor and the servlet is angled, which indicates that the message is sent between components on different machines, such as an HTTP message from the user's web browser to the web container that handles the login servlet.

Figure C-22 shows a more elaborate Sequence diagram. This diagram is equivalent to the Collaboration diagram in Figure C-20 on page C-25.

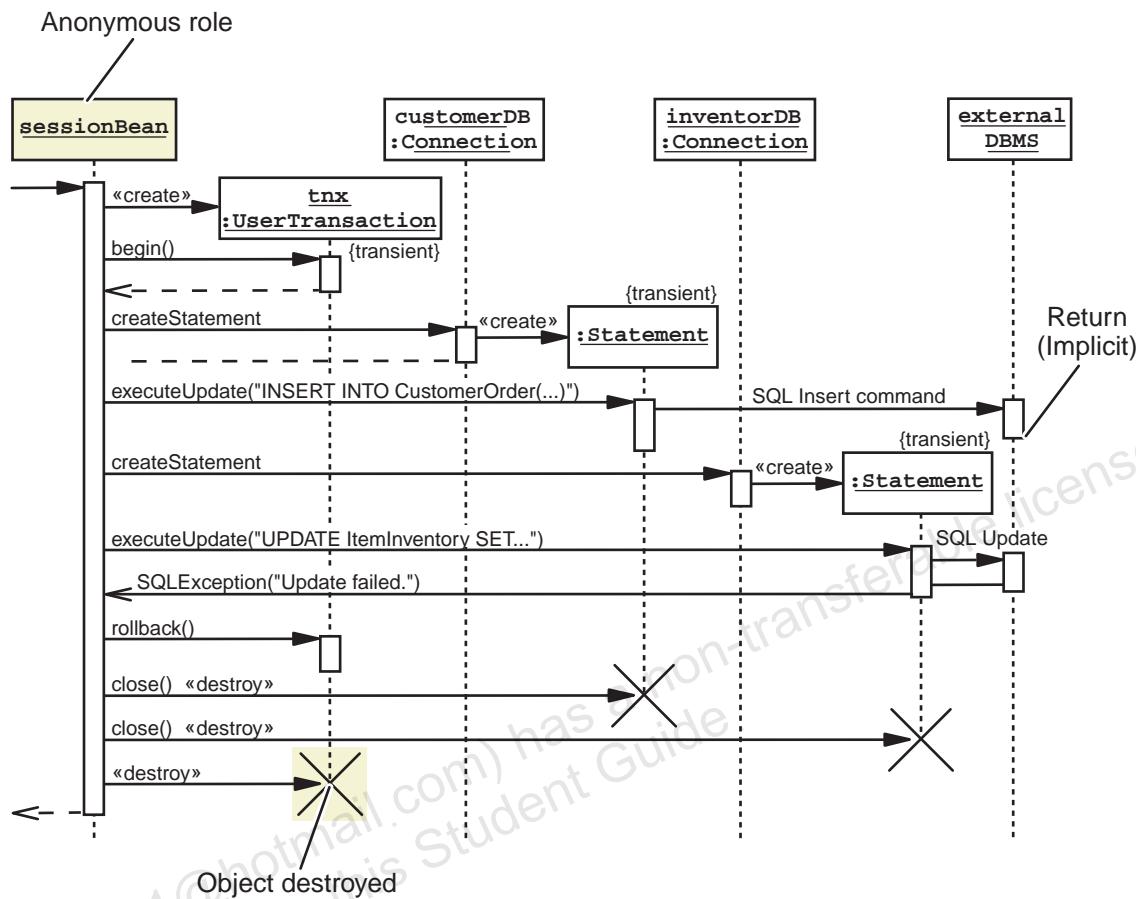


Figure C-22 Another Sequence Diagram

This example shows a few more details about Sequence diagrams. First, the return arrow is not always important. A return arrow is implicit at the end of the activation bar. Also, Sequence diagrams can show the creation and destruction of objects explicitly. Every role has a lifeline that extends from the base of the object node vertically. Roles at the top of the diagram existed before the entry message (into the left-most role). Roles that have a message arrow pointing to the head of the role node with the «create» message are created during the execution of the sequence. The destruction of an object is shown with a large cross that terminates the role's lifeline.

Note – Sequence diagrams can also show asynchronous messages. This type of message uses a solid line with stick arrow head. →



Statechart Diagrams

Statechart Diagrams

A Statechart diagram represents the states and responses of a class to external and internal triggers. You can also use a Statechart diagram to represent the life cycle of an object. The definition of an object state is dependent on the type of object and the level of depth you want to model.



Note – A Statechart diagram is recognized by several other names including State diagram and State Transition diagram.

Figure C-23 shows an example Statechart diagram. Every Statechart diagram should have an initial state (the state of the object at its creation) and a final state. By definition, no state can transition into the initial state and the final state cannot transition to any other state.

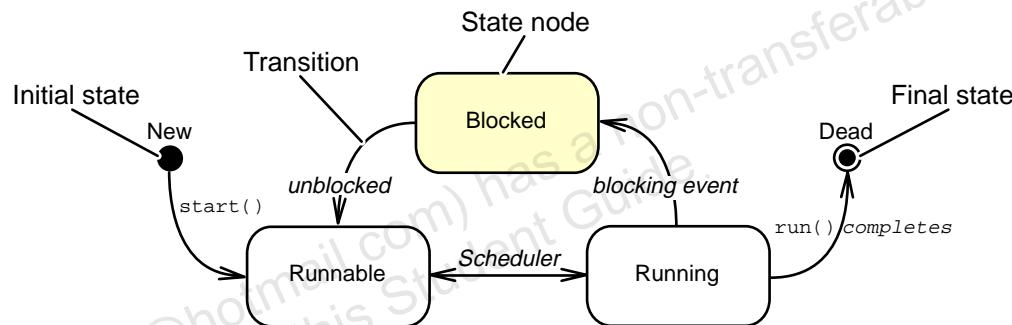


Figure C-23 An Example Statechart Diagram

There is no predefined way to implement a Statechart diagram. For complex behavior, you might consider using the State design pattern.

Transitions

A transition has five elements:

- Source state – The state affected by the transition
- Event trigger – The event whose reception by the object in the source state makes the transition eligible to fire, providing its guard condition is satisfied
- Guard condition – A Boolean expression used to determine if the state transition should be made when the event trigger occurs
- Action – A computation or operation performed on the object that makes the state transition
- Target state – The state that is active after the completion of the transition

Figure C-24 illustrates a detailed transition.

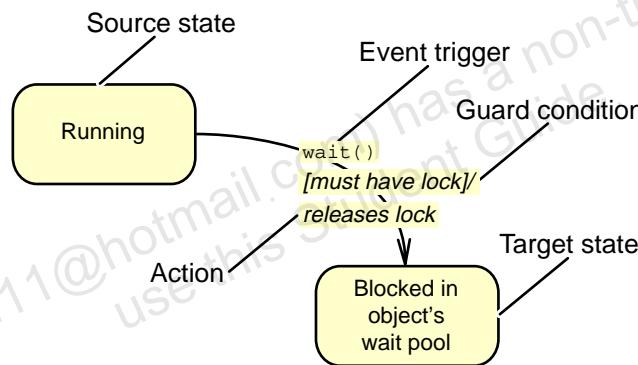


Figure C-24 An Example State Transition

Activity Diagrams

An Activity diagram represents the activities or actions of a process without regard to the objects that perform these activities.

Figure C-25 shows the elements of an Activity diagram. An Activity diagram is similar to a flow chart. There are activities and transitions between them. Every Activity diagram starts with a single start state and ends in a single stop state.

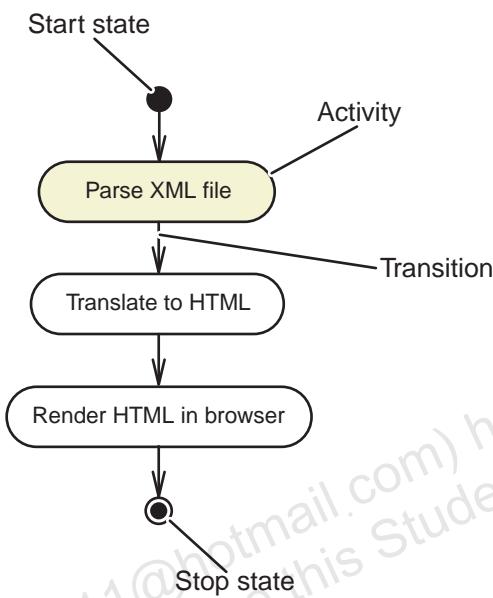


Figure C-25 Activities and Transitions

Figure C-26 demonstrates branching and looping in Activity diagrams. The diagram models the higher level activity of “Verify availability” of products in a purchase order. The top-level branch node forms a simple while loop. The guard on the transition below the branch is true if there are more products in the order to be processed. The “else” transition halts this activity.

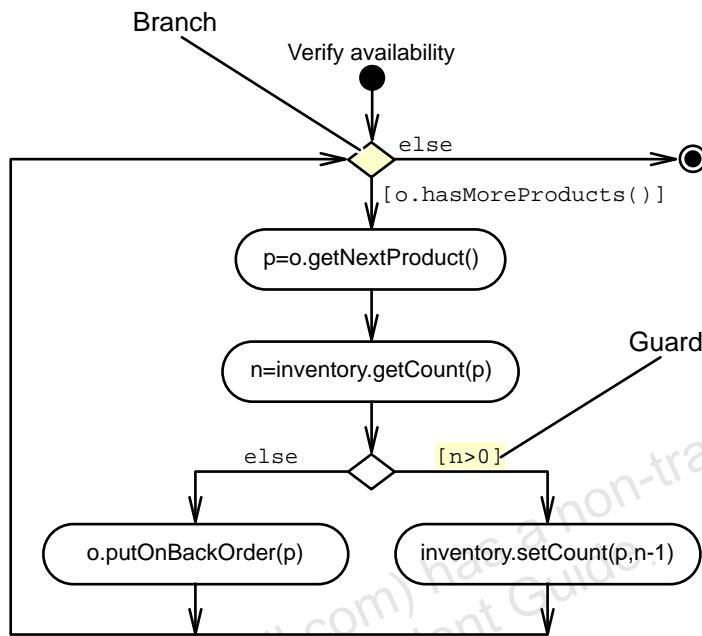


Figure C-26 Branching and Looping

Activity Diagrams

Figure C-27 shows a richer Activity diagram.

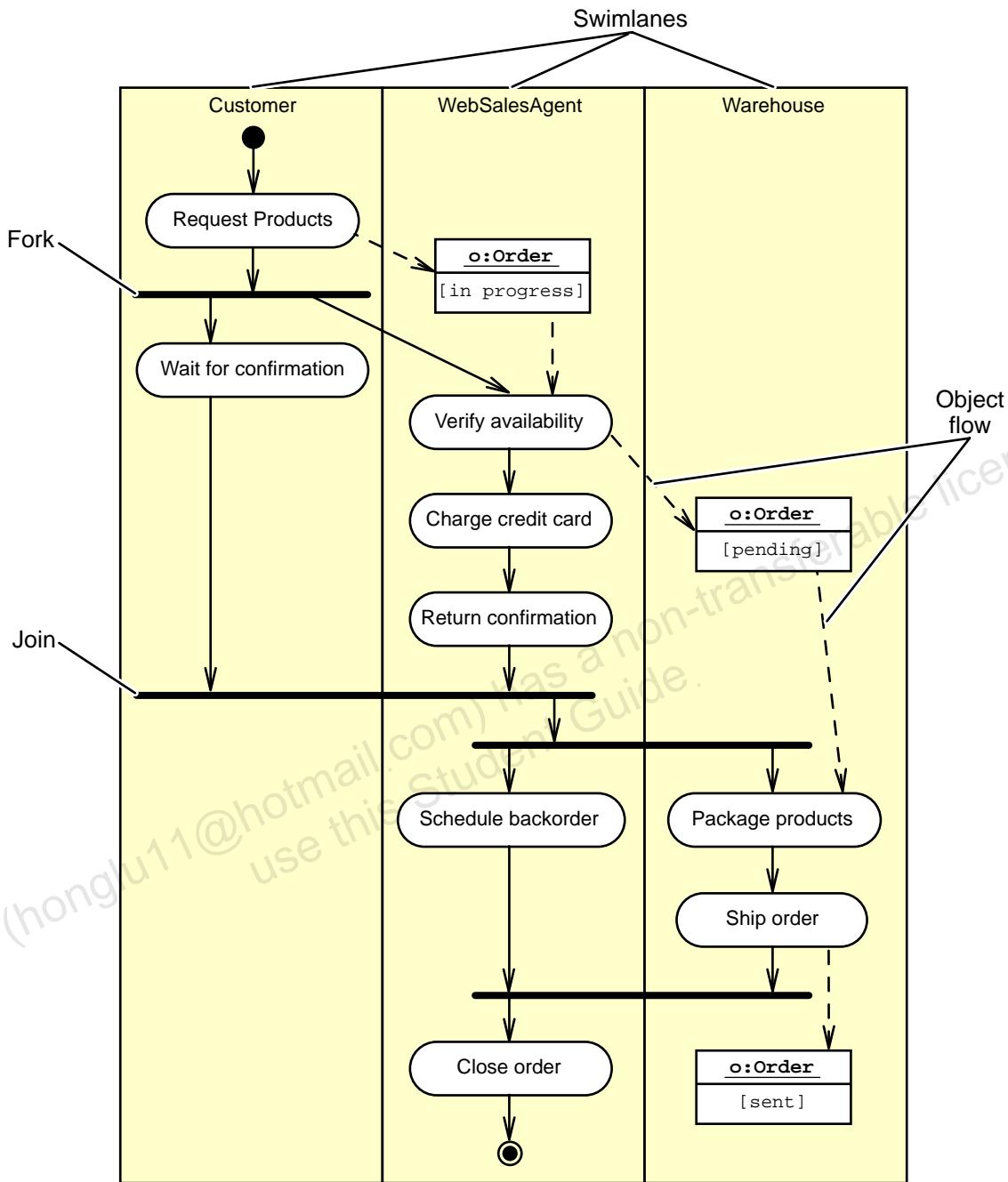


Figure C-27 An Example Activity Diagram

In this example, swim lanes are used to isolate the actor of a given set of activities. These actors might include humans, systems, or organization entities. This diagram also demonstrates the ability to model concurrent activities. For example, the Customer initiates the purchase of one or more products on the company's web site. The Customer then waits as the WebSalesAgent software begins to process the purchase order. The fork bar splits a single transition into two or more transitions. The corresponding join bar must contain the same number of inbound transitions.

Component Diagrams

Component Diagrams

A Component diagram represents the organization and dependencies among software implementation components.

Figure C-28 shows three types of icons that can represent software components. Example 1 is a generic icon. Example 2 is an icon that is used to represent a source file. Example 3 is an icon that represents a file that contains executable (or object) code.

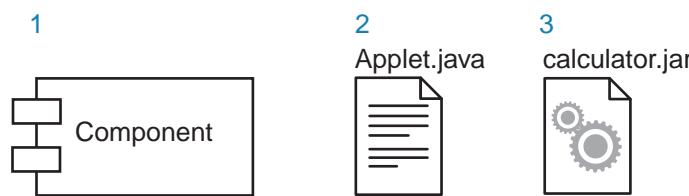


Figure C-28 Example Component Nodes

Figure C-29 shows the dependencies of packaging an HTML page that contains an applet. The HTML page depends on the JAR file, which is constructed from a set of class files. The Class files are compiled from the corresponding source files. You can use a tagged value to indicate the source control version numbers on the source files.

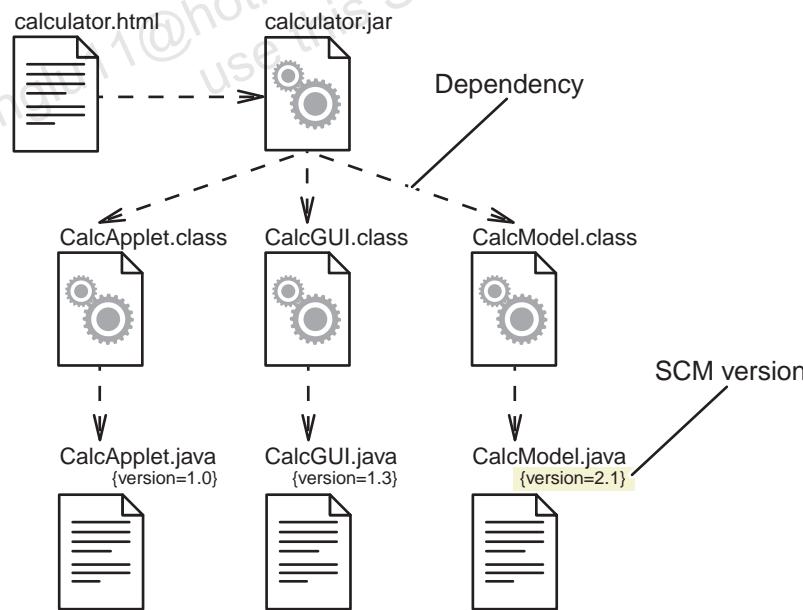


Figure C-29 An Example Component Diagram



Note – SCM is Source Control Management.

Figure C-30 shows another Component diagram. In this diagram, several components have an interface connector. The component attached to the connector implements the named interface. The component that has an arrow pointing to the connector depends on the fact that component realizes that interface.

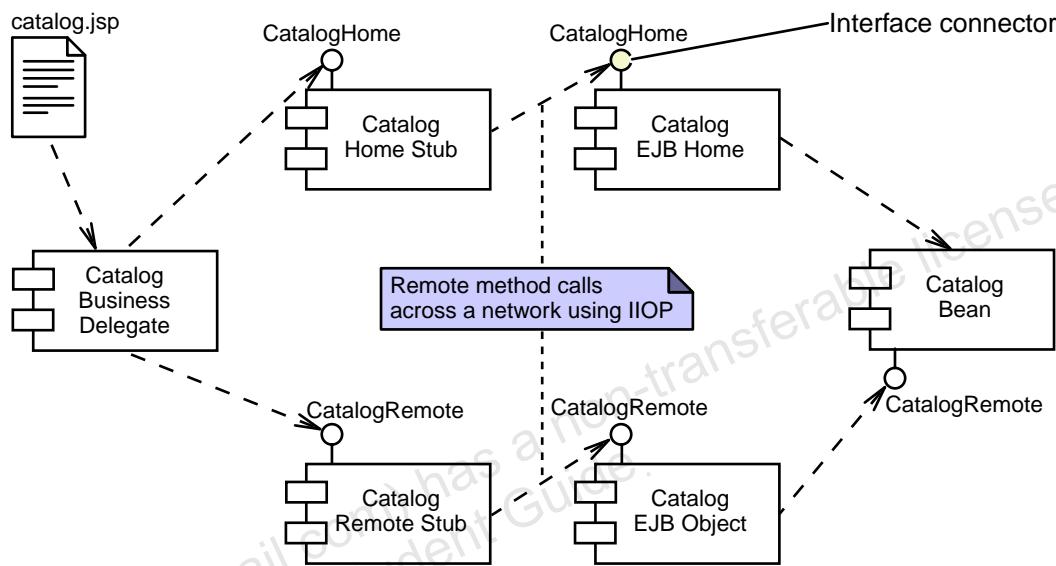


Figure C-30 A Component Diagram With Interfaces

In this Java EE technology example, the web tier includes a catalog JSP, which uses a catalog Business Delegate JavaBeans component to communicate to the EJB technology tier. Every enterprise bean must include two interfaces. The home interface enables the client to create new enterprise beans on the EJB server. The remote interface enables the client to call the business logic methods on the (remote) enterprise bean. The business delegate communicates with the catalog bean through local stub objects that implement the proper home and remote interfaces. These objects communicate over a network using the Internet Inter-ORB protocol with remote “skeletons.” In EJB technology terms, these objects are called EJBHome and EJBObject). These objects communicate directly with the catalog bean that implements the true business logic.

Deployment Diagrams

A Deployment diagram represents the network of processing resource elements and the configuration of software components on each physical element.

A Deployment diagram is composed of hardware nodes, software components, software dependencies, and communication relationships. Figure C-31 shows an example in which the client machine communicates with a web server using HTTP over TCP/IP.

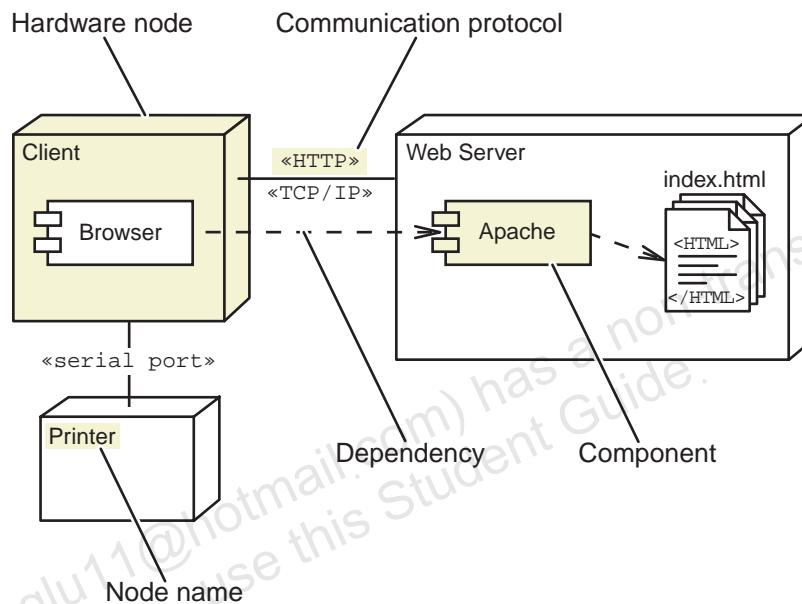


Figure C-31 An Example Deployment Diagram

The client is running a web browser which is communicating with an Apache web server. Therefore, the browser component depends upon the Apache component. Similarly, the Apache application depends upon the HTML files that it serves. The client machine is also connected to local printer using a parallel port.

You can use a Deployment diagram to show how the logical tiers of an application architecture are configured into a physical network.