Sun Microsystems

Enterprise JavaBeans™ Specification, Version 2.0

This is the specification of the Enterprise JavaBeans™ architecture. The Enterprise JavaBeans architecture is a component architecture for the development and deployment of component-based distributed business applications. Applications written using the Enterprise JavaBeans architecture are scalable, transactional, and multi-user secure. These applications may be written once, and then deployed on any server platform that supports the Enterprise JavaBeans specification.

EJB 2.0 Architects:

Linda G. DeMichiel, *Specification Lead*

L. Ümit Yalçinalp

Sanjeev Krishnan

Please send technical comments on this Draft to:

ejb-spec-comments@eng.sun.com

Version 2.0, Proposed Final Draft

October 23, 2000

Enterprise Java Beans (EJB (TM)) Specification ("Specification")

Version: 2.0

Status: Proposed Final Draft

Release: 23 October 2000

NOTICE

TRADEMARKS

DISCLAIMER OF WARRANTIES

COPYRIGHTS, TRADE SECRETS OR OTHER RIGHTS. This document does not represent any commitment to release or implement any portion of the Specification in any product.

THE SPECIFICATION COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS. CHANGES ARE PERIODICALLY ADDED TO THE INFORMATION THEREIN; THESE CHANGES WILL BE INCORPORATED INTO NEW VERSIONS OF THE SPECIFICATION, IF ANY. SUN MAY MAKE IMPROVEMENTS AND/OR CHANGES TO THE PRODUCT(S) AND/OR THE PROGRAM(S) DESCRIBED IN THE SPECIFICATION AT ANY TIME. Any use of such changes in the Specification will be governed by the then-current license for the applicable version of the Specification.

LIMITATION OF LIABILITY

TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL SUN OR ITS LICENSORS BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION, LOST REVENUE, PROFITS OR DATA, OR FOR SPECIAL, INDIRECT, CONSEQUENTIAL, INCIDENTAL OR PUNITIVE DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF OR RELATED TO ANY FURNISHING, PRACTICING, MODIFYING OR ANY USE OF THE SPECIFICATION, EVEN IF SUN AND/OR ITS LICENSORS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

You will indemnify, hold harmless, and defend Sun and its licensors from any claims based on your use of the Specification for any purposes other than those of internal evaluation, and from any claims that later versions or releases of any Specification furnished to you are incompatible with the Specification provided to you under this license.

RESTRICTED RIGHTS LEGEND

If this Software is being acquired by or on behalf of the U.S. Government or by a U.S. Government prime contractor or subcontractor (at any tier), then the Government's rights in the Software and accompanying documentation shall be only as set forth in this license; this is in accordance with 48 C.F.R. 227.7201 through 227.7202-4 (for Department of Defense (DoD) acquisitions) and with 48 C.F.R. 2.101 and 12.212 (for non-DoD acquisitions).

REPORT

You may wish to report any ambiguities, inconsistencies or inaccuracies you may find in connection with your evaluation of the Specification ("Feedback"). To the extent that you provide Sun with any Feedback, you hereby: (i) agree that such Feedback is provided on a non-proprietary and non-confidential basis, and (ii) grant Sun a perpetual, non-exclusive, worldwide, fully paid-up, irrevocable license, with the right to sublicense through multiple levels of sublicensees, to incorporate, disclose, and use without limitation the Feedback for any purpose related to the Specification and future versions, implementations, and test suites thereof.

Table of Contents

List of Figures

List of Tables

Chapter 1 Introduction

This is the specification of the Enterprise JavaBeans™ architecture. The Enterprise JavaBeans architecture is a component architecture for the development and deployment of component-based distributed business applications. Applications written using the Enterprise JavaBeans architecture are scalable, transactional, and multi-user secure. These applications may be written once, and then deployed on any server platform that supports the Enterprise JavaBeans specification.

1.1 Target audience

The target audiences for this specification are the vendors of transaction processing platforms, vendors of enterprise application tools, and other vendors who want to support the Enterprise JavaBeans™ (EJB) technology in their products.

Many concepts described in this document are system-level issues that are transparent to the Enterprise JavaBeans application programmer.

1.2 What is new in EJB 2.0

The Enterprise JavaBeans 2.0 architecture presented in this document is an extension of the Enterprise JavaBeans architecture designed and specified by Vlada Matena and Mark Hapner in the *Enterprise JavaBeans Specification, v1.1*. In this document we have extended Enterprise JavaBeans to include the following new functionality:

- *We have specified the integration of Enterprise JavaBeans with the Java Message Service, and introduced Message-driven beans. A message-driven bean is a stateless component that is invoked by the container as a result of the arrival of a JMS message. The goal of the message-driven bean model is to make developing an enterprise bean that is asynchronously invoked to handle the processing of incoming JMS messages as simple as developing the same functionality in any other JMS MessageListener.*

- *We have revised Enterprise JavaBeans container managed persistence for entity beans, and have added support for container managed relationships and support for the integration of persistence managers with EJB containers. We have specified new contracts for entity beans with container-managed persistence to address the limitations of the field-based approach to container-managed in persistence in earlier versions of the Enterprise JavaBeans specification. The new container managed persistence mechanisms are added to provide the following functionality:*

- *To support container managed relationships among entity beans and dependent object classes.*
- *To provide the basis for a portable finder query syntax.*
- *To support more efficient vendor implementations leveraging lazy loading mechanisms, dirty detection; to reduce memory footprints; to avoid data aliasing problems; etc.*
- *To provide a foundation for pluggable persistence managers.*

- *We have provided a declarative syntax for the definition of query methods for entity beans with container managed persistence that allows the implementation of finder and select methods to be provided by the persistence manager. The resulting Enterprise JavaBeans™ Query Language, EJB™ QL, provides for navigation across a network of enterprise beans and dependent objects defined by means of container-managed relationships.*

- *We have added select methods for the internal use of entity beans with container managed persistence. Select methods allow the selection of dependent objects, values, and related beans through EJB QL queries.*

- *We have added support for additional methods on the Home interface to implement business logic that is independent of a specific enterprise bean instance.*

- *We have added a run-as security identity functionality for enterprise beans. This functionality allows for the declarative specification of the principal to be used for the run-as identity of an enterprise bean in terms of its security role.*

- *We have defined an interoperability protocol based on CORBA/IIOP to allow invocations on session and entity beans from J2EE components that are deployed in products from different vendors.*

1.3 Acknowledgments

We would like to thank Vlada Matena and Mark Hapner for their generous help and encouragement and for the invaluable input that they have provided in the design of the Enterprise JavaBeans 2.0 architecture.

The Enterprise JavaBeans architecture is a broad effort that includes contributions from numerous groups at Sun and at partner companies. In particular, we would like to thank the members of the EJB 2.0 Expert Group for their contributions to this specification. The companies that have participated in the EJB 2.0 Expert Group include: Allaire, Art Technology Group, BEA, Bluestone Software, Forte, Fujitsu, GemStone, IBM, InLine, Inprise, IONA, iPlanet, Luna Information Systems, The ObjectPeople, Oracle, Persistence, Progress Software, Secant, Siemens, SilverStream, Software AG, Sun Microsystems, Sybase, Tibco, Vitria.

We would also like to thank our colleagues at Sun Microsystems—Joseph Fialli, Shel Finkelstein, Alan Frechette, Art Frechette, Ram Jeyaraman, Nick Kassem, Ron Monzillo, Vivek Nagar, Farrukh Najmi, Kevin Osborn, Ken Saks, Bill Shannon, Rahul Sharma, and Peter Walker—for their feedback on design and implementation issues and their suggestions to help improve the quality of the EJB 2.0 specification. We would like to thank Beth Stearns for her editorial assistance in the production of this document.

1.4 Organization

Chapter 2, "Goals" discusses the advantages of Enterprise JavaBeans architecture.

Chapter 3, "Roles and Scenarios" discusses the responsibilities of the Bean Provider, Application Assembler, Deployer, EJB Container and Server Providers, and System Administrators with respect to the Enterprise JavaBeans architecture.

Chapter 4, "Fundamentals" defines the scope of the Enterprise JavaBeans specification.

Chapters 5 through 7 define Session Beans: Chapter 5 discusses the client view, Chapter 6 presents the Session Bean component contract, and Chapter 7 outlines an example Session Bean scenario.

Chapters 8 through 13 define Entity Beans: Chapter 8 discusses the client view; Chapter 9 presents the Entity Bean component contract for container managed persistence; Chapter 10 presents EJB QL, the query language for Entity Beans with container managed persistence; Chapter 11 presents the Entity Bean component contract for bean managed persistence; Chapter 12 outlines an example Entity Bean scenario; and Chapter 13 specifies the EJB 1.1 Entity Bean component contract for container managed persistence.

Chapters 14 through 15 define Message-driven Beans: Chapter 14 presents the Message-driven Bean component contract, and Chapter 15 outlines an example Message-driven Bean scenario.

Chapters 16 through 20 discuss transactions; exceptions; distribution and interoperability; environment; and security.

Chapters 21 and 22 describe the format of the ejb-jar file and its deployment descriptor.

Chapter 23 defines the runtime APIs that a compliant EJB container must provide to the enterprise bean instances at runtime. The chapter also specifies the programming restrictions for portable enterprise beans.

Chapter 24 summarizes the responsibilities of the individual EJB Roles.

Chapter 25 is the Enterprise JavaBeans API Reference.

Chapter 26 provides a list of related documents.

1.5 Document conventions

The regular Times font is used for information that is prescriptive by the EJB specification.

The italic Times font is used for paragraphs that contain descriptive information, such as notes describing typical use, or notes clarifying the text with prescriptive specification.

```
The Courier font is used for code examples.
```

The Helvetica font is used to specify the BNF of EJB QL.

Chapter 2

Goals

2.1 Overall goals

The Enterprise JavaBeans (EJB) architecture has the following goals:

- *The Enterprise JavaBeans architecture will be the standard component architecture for building distributed object-oriented business applications in the Java™ programming language. The Enterprise JavaBeans architecture will make it possible to build distributed applications by combining components developed using tools from different vendors.*
- *The Enterprise JavaBeans architecture will make it easy to write applications: Application developers will not have to understand low-level transaction and state management details, multi-threading, connection pooling, and other complex low-level APIs.*
- *Enterprise JavaBeans applications will follow the Write Once, Run Anywhere™" philosophy of the Java programming language. An enterprise Bean can be developed once, and then deployed on multiple platforms without recompilation or source code modification.*
- *The Enterprise JavaBeans architecture will address the development, deployment, and runtime aspects of an enterprise application's life cycle.*

- *The Enterprise JavaBeans architecture will define the contracts that enable tools from multiple vendors to develop and deploy components that can interoperate at runtime.*
- *The Enterprise JavaBeans architecture will be compatible with existing server platforms. Vendors will be able to extend their existing products to support Enterprise JavaBeans.*
- *The Enterprise JavaBeans architecture will be compatible with other Java programming language APIs.*
- *The Enterprise JavaBeans architecture will provide interoperability between enterprise Beans and Java 2 Platform Enterprise Edition (J2EE) components as well as non-Java programming language applications.*
- *The Enterprise JavaBeans architecture will be compatible with the CORBA protocols.*

2.2 EJB Releases 1.0 and 1.1

Enterprise JavaBeans Release 1.0 focused on the following aspects:

- *Defined the distinct "EJB Roles" that are assumed by the component architecture.*
- *Defined the client view of enterprise Beans.*
- *Defined the enterprise Bean developer's view.*
- *Defined the responsibilities of an EJB Container provider and server provider; together these make up a system that supports the deployment and execution of enterprise Beans.*
- *Defined the format of the ejb-jar file, EJB's unit of deployment.*

Release 1.1 augmented this with a focus on the following:

- *Provided better support for application assembly and deployment.*
- *Specified in greater detail the responsibilities of the individual EJB roles.*

2.3 Goals for Release 2.0

Enterprise JavaBeans Release 2.0 focuses on the following aspects:

- *Define the integration of EJB with the Java Message Service.*
- *Provide improved support for the persistence of entity beans.*
- *Provide improved support for the management of relationships among enterprise beans.*

- *Provide a query syntax for entity bean finder methods.*
- *Provide support for additional methods in the home interface.*
- *Provide for network interoperability among EJB servers.*

Chapter 3

EJB Roles and Scenarios

3.1 EJB Roles

The Enterprise JavaBeans architecture defines seven distinct roles in the application development and deployment life cycle. Each EJB Role may be performed by a different party. The EJB architecture specifies the contracts that ensure that the product of each EJB Role is compatible with the product of the other EJB Roles. The EJB specification focuses on those contracts that are required to support the development and deployment of ISV-written enterprise Beans.

> *In some scenarios, a single party may perform several EJB Roles. For example, the Container Provider and the EJB Server Provider may be the same vendor. Or a single programmer may perform the two EJB Roles of the Enterprise Bean Provider and the Application Assembler.*

The following sections define the seven EJB Roles.

3.1.1 Enterprise Bean Provider

The Enterprise Bean Provider (Bean Provider for short) is the producer of enterprise beans. His or her output is an ejb-jar file that contains one or more enterprise beans. The Bean Provider is responsible for the Java classes that implement the enterprise bean's business methods; the definition of the bean's remote and home interfaces; and the bean's deployment descriptor. The deployment descriptor includes the structural information (e.g. the name of the enterprise bean class) of the enterprise bean and declares all the enterprise bean's external dependencies (e.g. the names and types of resources that the enterprise bean uses).

The Enterprise Bean Provider is typically an application domain expert. The Bean Provider develops reusable enterprise beans that typically implement business tasks or business entities.

The Bean Provider is not required to be an expert at system-level programming. Therefore, the Bean Provider usually does not program transactions, concurrency, security, distribution, or other services into the enterprise Beans. The Bean Provider relies on the EJB Container for these services.

A Bean Provider of multiple enterprise beans often performs the EJB Role of the Application Assembler.

3.1.2 Application Assembler

The Application Assembler combines enterprise beans into larger deployable application units. The input to the Application Assembler is one or more ejb-jar files produced by the Bean Provider(s). The Application Assembler outputs one or more ejb-jar files that contain the enterprise beans along with their application assembly instructions. The Application Assembler inserts the application assembly instructions into the deployment descriptors.

The Application Assembler can also combine enterprise beans with other types of application components (e.g. Java ServerPages™) when composing an application.

The EJB specification describes the case in which the application assembly step occurs *before* the deployment of the enterprise beans. However, the EJB architecture does not preclude the case that application assembly is performed *after* the deployment of all or some of the enterprise beans.

The Application Assembler is a domain expert who composes applications that use enterprise Beans. The Application Assembler works with the enterprise Bean's deployment descriptor and the enterprise Bean's client-view contract. Although the Assembler must be familiar with the functionality provided by the enterprise Bean's remote and home interfaces, he or she does not need to have any knowledge of the enterprise Bean's implementation.

3.1.3 Deployer

The Deployer takes one or more ejb-jar files produced by a Bean Provider or Application Assembler and deploys the enterprise beans contained in the ejb-jar files in a specific operational environment. The operational environment includes a specific EJB Server and Container.

The Deployer must resolve all the external dependencies declared by the Bean Provider (e.g. the Deployer must ensure that all resource manager connection factories used by the enterprise beans are present in the operational environment, and he or she must bind them to the resource manager connection factory references declared in the deployment descriptor), and must follow the application assembly instructions defined by the Application Assembler. To perform his or her role, the Deployer uses tools provided by the EJB Container Provider.

The Deployer's output are enterprise beans (or an assembled application that includes enterprise beans) that have been customized for the target operational environment, and that are deployed in a specific EJB Container.

The Deployer is an expert at a specific operational environment and is responsible for the deployment of enterprise Beans. For example, the Deployer is responsible for mapping the security roles defined by the Application Assembler to the user groups and accounts that exist in the operational environment in which the enterprise beans are deployed.

The Deployer uses tools supplied by the EJB Container Provider to perform the deployment tasks. The deployment process is typically two-stage:

- *The Deployer first generates the additional classes and interfaces that enable the container to manage the enterprise beans at runtime. These classes are container-specific.*
- *The Deployer performs the actual installation of the enterprise beans and the additional classes and interfaces into the EJB Container.*

In some cases, a qualified Deployer may customize the business logic of the enterprise Beans at their deployment. Such a Deployer would typically use the container tools to write relatively simple application code that wraps the enterprise Bean's business methods.

3.1.4 EJB Server Provider

The EJB Server Provider is a specialist in the area of distributed transaction management, distributed objects, and other lower-level system-level services. A typical EJB Server Provider is an OS vendor, middleware vendor, or database vendor.

The current EJB architecture assumes that the EJB Server Provider and the EJB Container Provider roles are the same vendor. Therefore, it does not define any interface requirements for the EJB Server Provider.

3.1.5 EJB Container Provider

The EJB Container Provider (Container Provider for short) provides

- The deployment tools necessary for the deployment of enterprise beans.
- The runtime support for the deployed enterprise bean instances.

From the perspective of the enterprise beans, the Container is a part of the target operational environment. The Container runtime provides the deployed enterprise beans with transaction and security management, network distribution of clients, scalable management of resources, and other services that are generally required as part of a manageable server platform.

The "EJB Container Provider's responsibilities" defined by the EJB architecture are meant to be requirements for the implementation of the EJB Container and Server. Since the EJB specification does not architect the interface between the EJB Container and Server, it is left up to the vendor how to split the implementation of the required functionality between the EJB Container and Server.

The expertise of the Container Provider is system-level programming, possibly combined with some application-domain expertise. The focus of a Container Provider is on the development of a scalable, secure, transaction-enabled container that is integrated with an EJB Server. The Container Provider insulates the enterprise Bean from the specifics of an underlying EJB Server by providing a simple, standard API between the enterprise Bean and the container. This API is the Enterprise JavaBeans component contract.

The Container Provider typically provides support for versioning the installed enterprise Bean components. For example, the Container Provider may allow enterprise Bean classes to be upgraded without invalidating existing clients or losing existing enterprise Bean objects.

The Container Provider typically provides tools that allow the system administrator to monitor and manage the container and the Beans running in the container at runtime.

3.1.6 Persistence Manager Provider

For Entity Beans with container-managed persistence, the Persistence Manager is responsible for the persistence of the Entity Beans installed in the container. The Persistence Manager Provider's tools are used at deployment time to generate code that moves data between the Entity Beans and a database or an existing application.

The Persistence Manager manages the persistent state of the entity beans and the dependent objects used by those beans, and the referential integrity of the relationships among the entity beans and dependent objects. The Persistence Manager is responsible for the execution of the finder and select methods for entity beans with container managed persistence.

The Persistence Manager interacts with the Container to receive notifications related to the lifecycle of the managed beans. The current EJB architecture, however, does not architect the full set of SPIs between the Container and the Persistence Manager: these interfaces are currenctly left to the Container Provider and Persistence Manager Provider.

3.1.7 System Administrator

The System Administrator is responsible for the configuration and administration of the enterprise's computing and networking infrastructure that includes the EJB Server and Container. The System Administrator is also responsible for overseeing the well-being of the deployed enterprise beans applications at runtime.

The EJB architecture does not define the contracts for system management and administration. The System Administrator typically uses runtime monitoring and management tools provided by the EJB Server and Container Providers to accomplish these tasks.

3.2 Scenario: Development, assembly, and deployment

Aardvark Inc. specializes in application integration. Aardvark developed the AardvarkPayroll *enterprise bean, which is a generic payroll access component that allows Java™ applications to access the payroll modules of the leading ERP systems. Aardvark packages the* AardvarkPayroll *enterprise bean in a standard ejb-jar file and markets it as a customizable enterprise bean to application developers. In the terms of the EJB architecture, Aardvark is the* ***Bean Provider*** *of the* AardvarkPayroll *bean.*

Wombat Inc. is a Web-application development company. Wombat developed an employee self-service application. The application allows a target enterprise's employees to access and update employee record information. The application includes the EmployeeService, EmployeeServiceAdmin, *and* EmployeeRecord *enterprise beans. The* EmployeeRecord *bean is a container-managed entity that allows deployment-time integration with an enterprise's existing human resource applications. In terms of the EJB architecture, Wombat is the* ***Bean Provider*** *of the* EmployeeService, EmployeeServiceAdmin, *and* EmployeeRecord *enterprise beans.*

In addition to providing access to employee records, Wombat would like to provide employee access to the enterprise's payroll and pension plan systems. To provide payroll access, Wombat licenses the AardvarkPayroll *enterprise bean from Aardvark, and includes it as part of the Wombat application. Because there is no available generic enterprise bean for pension plan access, Wombat decides that a suitable pension plan enterprise bean will have to be developed at deployment time. The pension plan bean will implement the necessary application integration logic, and it is likely that the pension plan bean will be specific to each Wombat customer.*

In order to provide a complete solution, Wombat also develops the necessary non-EJB components of the employee self-service application, such as the Java ServerPages™ (JSP) that invoke the enterprise beans and generate the HTML presentation to the clients. Both the JSP pages and enterprise beans are customizable at deployment time because they are intended to be sold to a number of target enterprises that are Wombat customers.

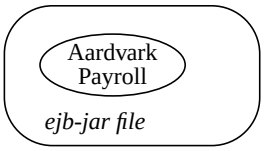
The Wombat application is packaged as a collection of JAR files. A single ejb-jar file contains all the enterprise beans developed by Wombat and also the AardvarkPayroll *enterprise bean developed by Aardvark; the other JAR files contain the non-EJB application components, such as the JSP components. The ejb-jar file contains the application assembly instructions describing how the enterprise beans are composed into an application. In terms of the EJB architecture, Wombat performs the role of the* ***Application Assembler****.*

Acme Corporation is a server software vendor. Acme developed an EJB Server and Container. In terms of the EJB architecture, Acme performs the ***EJB Container Provider*** *and* ***EJB Server Provider*** *roles.*

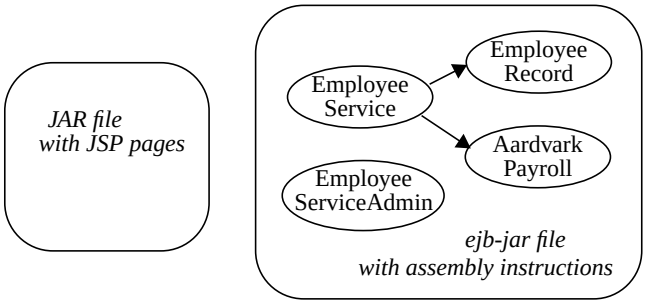
The ABC Enterprise wants to enable its employees to access and update employee records, payroll information, and pension plan information over the Web. The information is stored in ABC's ERP system. ABC buys the employee self-service application from Wombat. To host the application, ABC buys the EJB Container and Server from Acme. ABC's Information Technology (IT) department, with the help of Wombat's consulting services, deploys the Wombat self-service application. In terms of the EJB architecture, ABC's IT department and Wombat consulting services perform the ***Deployer*** *role. ABC's IT department also develops the* ABCPensionPlan *enterprise bean that provides the Wombat application with access to ABC's existing pension plan application.*

ABC's IT staff is responsible for configuring the Acme product and integrating it with ABC's existing network infrastructure. The IT staff is responsible for the following tasks: security administration, such as adding and removing employee accounts; adding employees to user groups such as the payroll department; and mapping principals from digital certificates that identify employees on VPN connections from home computers to the Kerberos user accounts that are used on ABC's intranet. ABC's IT staff also monitors the well-being of the Wombat application at runtime, and is responsible for servicing any error conditions raised by the application. In terms of the EJB architecture, ABC's IT staff performs the role of the ***System Administrator****.*

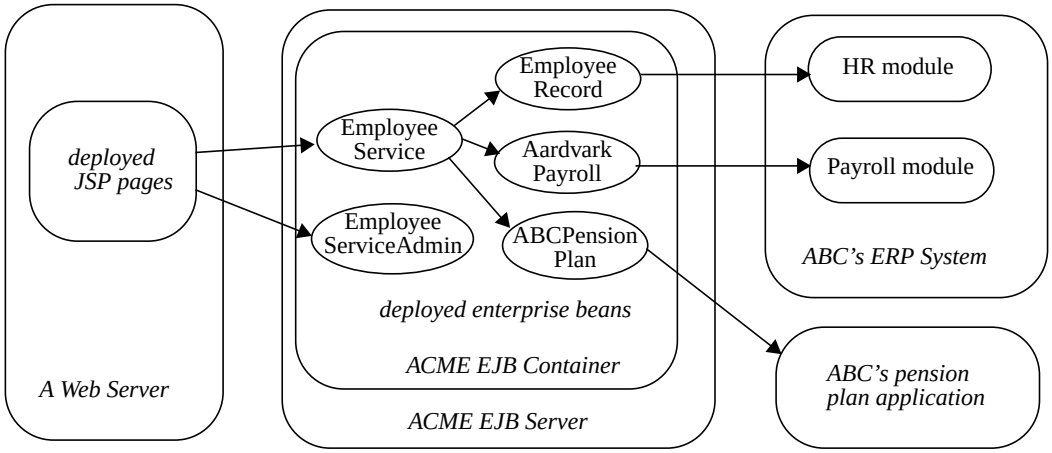
The following diagrams illustrates the products of the various EJB Roles.



(a) Aardvark's product is an ejb-jar file with an enterprise bean



(b) Wombat's product is an ejb-jar file with several enterprise beans assembled into an application. Wombat's product also includes non-EJB components.



(c) Wombat's application is deployed in ACME's EJB Container at the ABC enterprise.

The following table summarizes the EJB Roles of the organizations involved in the scenario.

Table 1 EJB Roles in the example scenarios

Organization	**EJB Roles**
Aardvark Inc.	Bean Provider
Wombat Inc.	Bean Provider Application Assembler
Acme Corporation	EJB Container Provider EJB Server Provider
ABC Enterprise's IT staff	Deployer Bean Provider (of `ABCPensionPlan`) System Administrator

Chapter 4 Overview

This chapter provides an overview of the Enterprise JavaBeans specification.

4.1 Enterprise Beans as components

Enterprise JavaBeans is an architecture for component-based distributed computing. Enterprise beans are components of distributed transaction-oriented enterprise applications.

4.1.1 Component characteristics

The essential characteristics of an enterprise bean are:

- An enterprise bean typically contains business logic that operates on the enterprise's data.
- An enterprise bean's instances are created and managed at runtime by a Container.
- An enterprise bean can be customized at deployment time by editing its environment entries.

- Various services information, such as a transaction and security attributes, are separate from the enterprise bean class. This allows the services information to be managed by tools during application assembly and deployment.
- Client access is mediated by the Container in which the enterprise Bean is deployed.
- If an enterprise Bean uses only the services defined by the EJB specification, the enterprise Bean can be deployed in any compliant EJB Container. Specialized containers can provide additional services beyond those defined by the EJB specification. An enterprise Bean that depends on such a service can be deployed only in a container that supports that service.
- An enterprise Bean can be included in an assembled application without requiring source code changes or recompilation of the enterprise Bean.
- The Bean Provider defines a client view of an enterprise Bean. The Bean developer can manually define the client view or it can be generated automatically by application development tools. The client view is unaffected by the container and server in which the Bean is deployed. This ensures that both the Beans and their clients can be deployed in multiple execution environments without changes or recompilation.

4.1.2 Flexible component model

The enterprise Bean architecture is flexible enough to implement components such as the following:

- An object that represents a stateless service.
- An object that represents a stateless service and whose invocation is asynchronous, driven by the arrival of JMS messages.
- An object that represents a conversational session with a particular client. Such session objects automatically maintain their conversational state across multiple client-invoked methods.
- An entity object that represents a business object that can be shared among multiple clients.

Enterprise beans are intended to be relatively coarse-grained business objects (e.g. purchase order, employee record). Fine-grained objects (e.g. line item on a purchase order, employee's address) should not be modeled as enterprise bean components.

Although the state management protocol defined by the Enterprise JavaBeans architecture is simple, it provides an enterprise Bean developer great flexibility in managing a Bean's state.

4.2 Enterprise JavaBeans contracts

This section provides an overview of the Enterprise JavaBeans contracts. The contracts are described in detail in the following chapters of this document.

4.2.1 Client-view contract

This is a contract between a client and a container. The client-view contract provides a uniform development model for applications using enterprise Beans as components. This uniform model enables the use of higher level development tools and allows greater reuse of components.

The enterprise bean client view is remotable—both local and remote programs can access an enterprise bean using the same view of the enterprise bean.

A client of an enterprise bean can be another enterprise bean deployed in the same or different Container. Or it can be an arbitrary Java program, such as an application, applet, or servlet. The client view of an enterprise bean can also be mapped to non-Java client environments, such as CORBA clients that are not written in the Java™ programming language.

The enterprise Bean Provider and the container provider cooperate to create the enterprise bean's client view. The client view includes:

- Home interface
- Remote interface
- Object identity
- Metadata interface
- Handle

The enterprise bean's **home interface** defines the methods for the client to create, remove, and find EJB objects of the same type (i.e., they are implemented by the same enterprise bean) as well as home business methods (business methods that are not specific to a particular bean instance). The home interface is specified by the Bean Provider; the Container creates a class that implements the home interface. The home interface extends the `javax.ejb.EJBHome` interface.

A client can locate an enterprise Bean home interface through the standard Java Naming and Directory Interface™ (JNDI) API.

An EJB object is accessible via the enterprise bean's **remote interface**. The remote interface defines the business methods callable by the client. The remote interface is specified by the Bean Provider; the Container creates a class that implements the remote interface. The remote interface extends the `javax.ejb.EJBObject` interface. The `javax.ejb.EJBObject` interface defines the operations that allow the client to access the EJB object's identity and create a persistent handle for the EJB object.

Each EJB object lives in a home, and has a unique identity within its home. For session beans, the Container is responsible for generating a new unique identifier for each session object. The identifier is not exposed to the client. However, a client may test if two object references refer to the same session object. For entity beans, the Bean Provider is responsible for supplying a primary key at entity object creation time[1]; the Container uses the primary key to identify the entity object within its home. A client may obtain an entity object's primary key via the `javax.ejb.EJBObject` interface. The client may also test if two object references refer to the same entity object.

A client may also obtain the enterprise bean's metadata interface. The metadata interface is typically used by clients who need to perform dynamic invocation of the enterprise bean. (Dynamic invocation is needed if the classes that provide the enterprise client view were not available at the time the client program was compiled.)

Message-driven beans have no home or remote interface and hence no client view in the sense of this section. A client can locate the JMS Destination to which it should send messages that are to be delivered to a message-driven bean by means of the standard JNDI API.

4.2.2 Component contract

This subsection describes the contract between an enterprise Bean and its Container, and, in the case of an enterprise Bean with container managed persistence, between an enterprise Bean and its Persistence Manager. The main requirements of the contract follow. (This is only a partial list of requirements defined by the specification.)

- The requirement for the Bean Provider to implement the business methods in the enterprise bean class. The requirement for the Container provider to delegate the client method invocation to these methods.

- For message-driven beans, the requirement for the Bean Provider to implement the `onMessage` method in the enterprise bean class. The requirement for the Container provider to invoke this method when a message has arrived for the bean to service.

- The requirement for the Bean Provider to implement the `ejbCreate<METHOD>`, `ejbPostCreate<METHOD>`,and `ejbRemove` methods, and to implement the `ejbFind<METHOD>` methods if the bean is an entity with bean-managed persistence. The requirement for the Container provider to invoke these methods during an EJB object creation, removal, and lookup.

- The requirement for the Bean Provider to provide abstract accessor methods for persistent fields and relationships for an entity with container-managed persistence. The requirement for the Persistence Manager Provider to provide the implementation of these methods.

- The requirement for the Bean Provider to provide a description of the relationships that are to be implemented by the Persistence Manager. The requirement for the Persistence Manager Provider to implement the relationships described by the Bean Provider.

[1] In some situations, the primary key type can be specified at deployment time (see subsection 9.10.1.3).

- The requirement for the Bean Provider to define the enterprise bean's home and remote interfaces, if the bean is an entity bean or a session bean. The requirement for the Container Provider to provide classes that implement these interfaces.

- For sessions, the requirement for the Bean Provider to implement the Container callbacks defined in the `javax.ejb.SessionBean` interface, and optionally the `javax.ejb.SessionSynchronization` interfaces. The requirement for the Container to invoke these callbacks at the appropriate times.

- For message-driven beans, the requirement for the Bean Provider to implement the Container callbacks defined in the `javax.ejb.MessageDrivenBean` interface.

- For entities, the requirement for the Bean Provider to implement the Container callbacks defined in the `javax.ejb.EntityBean` interface. The requirement for the Container to invoke these callbacks at the appropriate times.

- The requirement for the Persistence Manager Provider to implement persistence for entity beans with container-managed persistence.

- The requirement for the Container Provider to provide the `javax.ejb.SessionContext` interface to session bean instances, the `javax.ejb.EntityContext` interface to entity bean instances, and the `javax.ejb.MessageDrivenContext` interface to message-driven bean instances. The context interface allows the instance to obtain information from the container.

- The requirement for the Container to provide to the bean instances the JNDI context that contains the enterprise bean's environment.

- The requirement for the Container to manage transactions, security, and exceptions on behalf of the enterprise bean instances.

- The requirement for the Bean Provider to avoid programming practices that would interfere with the Container's runtime management of the enterprise bean instances.

- The requirement for the Bean Provider of entity beans with container-managed persistence to avoid programming practices that would interfere with the Persistence Manager's runtime management of the state of the entity beans.

4.2.3 Ejb-jar file

An **ejb-jar file** is a standard format used by EJB tools for packaging enterprise Beans with their declarative information. The ejb-jar file is intended to be processed by application assembly and deployment tools.

The ejb-jar file is a contract used both between the Bean Provider and the Application Assembler, and between the Application Assembler and the Deployer.

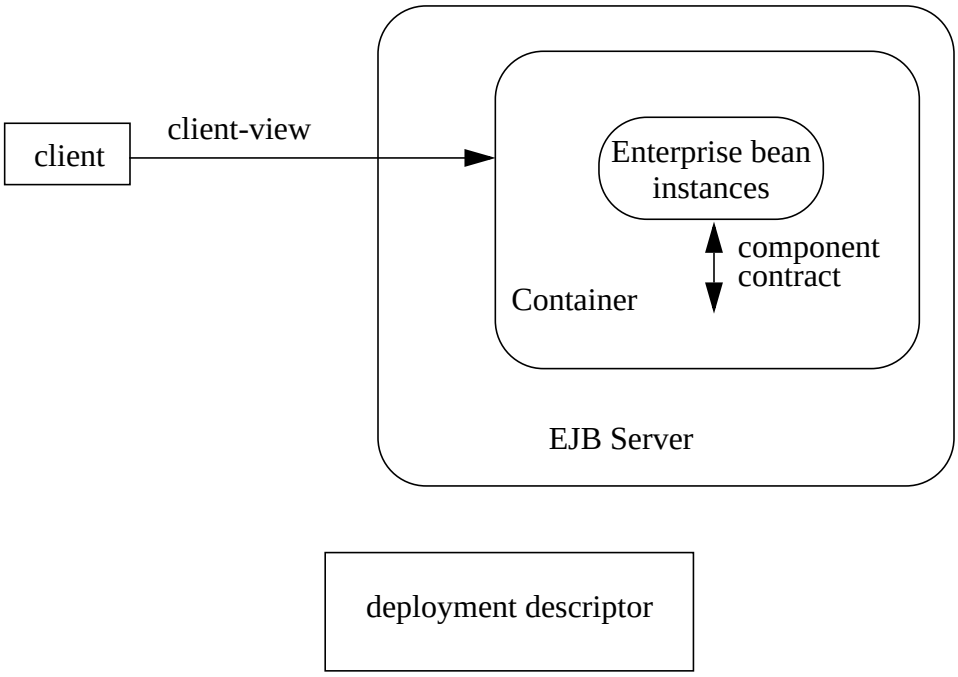
The ejb-jar file includes:

- Java class files for the enterprise Beans and their remote and home interfaces.
- An XML **deployment descriptor**. The deployment descriptor provides both the structural and application assembly information about the enterprise beans in the ejb-jar file. The application assembly information is optional. (Typically, only ejb-jar files with assembled applications include this information.)

4.2.4 Contracts summary

The following figure illustrates the Enterprise JavaBeans contracts.

Figure 1 Enterprise JavaBeans Contracts



Note that while the figure illustrates only a remote client running outside of the Container, the client-view API is also applicable to clients that are enterprise Beans deployed in the same Container.

4.3 Session, entity, and message-driven objects

The Enterprise JavaBeans architecture defines three types of enterprise bean objects:

- A session object.
- An entity object.
- A message-driven object.

4.3.1 Session objects

A typical session object has the following characteristics:

- *Executes on behalf of a single client.*
- *Can be transaction-aware.*
- *Updates shared data in an underlying database.*
- *Does not represent directly shared data in the database, although it may access and update such data.*
- *Is relatively short-lived.*
- *Is removed when the EJB Container crashes. The client has to re-establish a new session object to continue computation.*

A typical EJB Container provides a scalable runtime environment to execute a large number of session objects concurrently.

Session beans are intended to be stateful. The EJB specification also defines a ***stateless Session bean*** *as a special case of a Session Bean. There are minor differences in the API between stateful (normal) Session beans and stateless Session beans.*

4.3.2 Entity objects

A typical entity object has the following characteristics:

- *Provides an object view of data in the database.*
- *Allows shared access from multiple users.*
- *Can be long-lived (lives as long as the data in the database).*
- *The entity, its primary key, and its remote reference survive the crash of the EJB Container. If the state of an entity was being updated by a transaction at the time the container crashed, the*

entity's state is automatically reset to the state of the last committed transaction. The crash is not fully transparent to the client—the client may receive an exception if it calls an entity in a container that has experienced a crash.

A typical EJB Container and Server provide a scalable runtime environment for a large number of concurrently active entity objects.

4.3.3 Message-driven objects

A typical message-driven object has the following characteristics:

- *Executes on receipt of a single client message.*
- *Can be transaction-aware.*
- *May update shared data in an underlying database.*
- *Does not represent directly shared data in the database, although it may access and update such data.*
- *Is relatively short-lived.*
- *Is stateless.*
- *Is removed when the EJB Container crashes. The container has to re-establish a new message-driven object to continue computation.*

A typical EJB Container provides a scalable runtime environment to execute a large number of message-driven objects concurrently.

4.4 Standard mapping to CORBA protocols

To help interoperability for EJB environments that include systems from multiple vendors, the EJB 2.0 specification requires compliant implementations to support the interoperability protocol based on CORBA/IIOP for invocations from J2EE clients on Sessions Beans and Entity Beans through their Home and Remote interfaces. Implementations may support other remote invocation protocols in addition to IIOP.

Chapter 18 summarizes the support for distribution and the interoperability requirements of EJB 2.0.

Chapter 5 Client View of a Session Bean

This chapter describes the client view of a session bean. The session bean itself implements the business logic. The bean's container provides functionality for remote access, security, concurrency, transactions, and so forth.

While classes implemented by the container provide the client view of the session bean, the container itself is transparent to the client.

5.1 Overview

For a client, a session object is a non-persistent object that implements some business logic running on the server. One way to think of a session object is as a logical extension of the client program that runs on the server. A session object is not shared among multiple clients.

A client accesses a session object through the session bean's remote interface. The Java object that implements this remote interface is called a session **EJBObject.** A session EJBObject is a remote Java object accessible from a client through the standard Java™ APIs for remote object invocation [3].

From its creation until destruction, a session object lives in a container. Transparently to the client, the container provides security, concurrency, transactions, swapping to secondary storage, and other services for the session object.

Each session object has an identity which, in general, **does not** survive a crash and restart of the container, although a high-end container implementation can mask container and server crashes to the client.

The client view of a session bean is location-independent. A client running in the same JVM as the session object uses the same API as a client running in a different JVM on the same or different machine.

A client of an session bean can be another enterprise bean deployed in the same or different Container; or it can be an arbitrary Java program, such as an application, applet, or servlet. The client view of a session bean can also be mapped to non-Java client environments, such as CORBA clients that are not written in the Java programming language.

Multiple enterprise beans can be installed in a container. The container allows the clients to look up the home interfaces of the installed enterprise beans via JNDI. A session bean's home interface provides methods to create and remove the session objects of a particular session bean.

The client view of an session object is the same, irrespective of the implementation of the session bean and the container.

5.2 EJB Container

An EJB Container (container for short) is a system that functions as the "container" for enterprise beans. Multiple enterprise beans can be deployed in the same container. The container is responsible for making the home interfaces of its deployed enterprise beans available to the client through JNDI. Thus, the client can look up the home interface for a specific enterprise bean using JNDI.

5.2.1 Locating a session bean's home interface

A client locates a session bean's home interface using JNDI. For example, the home interface for the `Cart` session bean can be located using the following code segment:

```
Context initialContext = new InitialContext();
CartHome cartHome = (CartHome)javax.rmi.PortableRemoteObject.narrow(
     initialContext.lookup("java:comp/env/ejb/cart"),
     CartHome.class);
```

A client's JNDI name space may be configured to include the home interfaces of enterprise beans installed in multiple EJB Containers located on multiple machines on a network. The actual locations of an enterprise bean and EJB Container are, in general, transparent to the client using the enterprise bean.

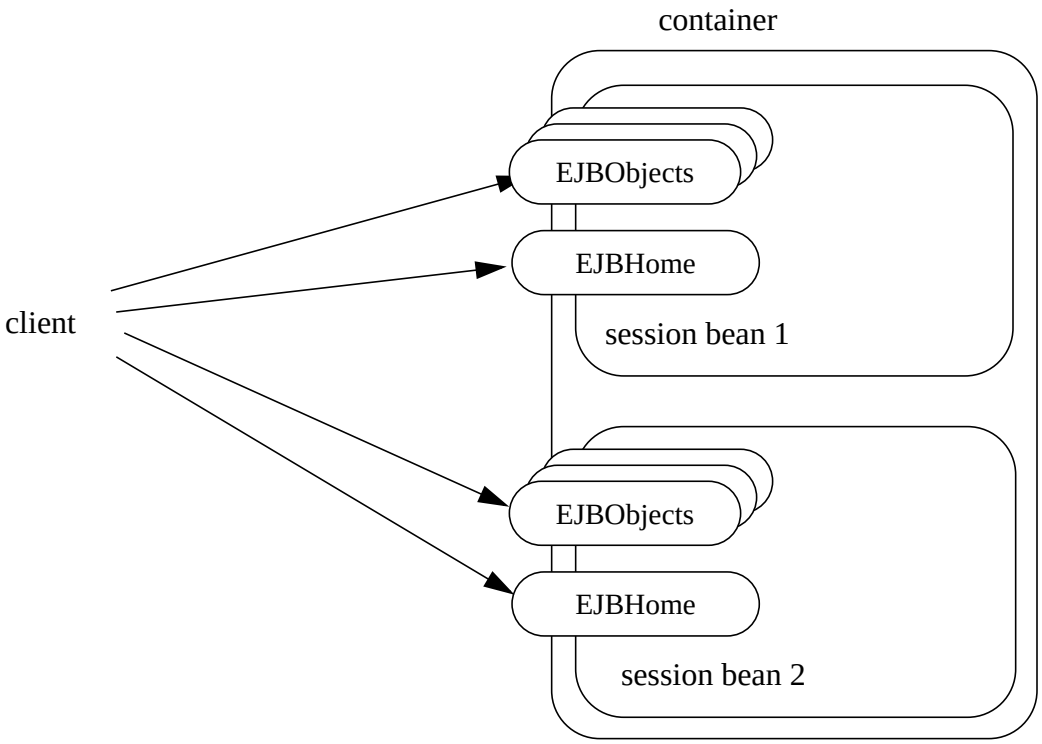
The lifecycle of the distributed object implementing the home interface (the EJBHome object) is Container-specific. A client application should be able to obtain a home interface, and then use it multiple times, during the client application's lifetime.

A client can pass a home interface object reference to another application. The receiving application can use the home interface in the same way that it would use a home interface object reference obtained via JNDI.

5.2.2 What a container provides

The following diagram illustrates the view that a container provides to clients of session beans.

Figure 2 Client View of session beans deployed in a Container



5.3 Home interface

A Container implements the home interface of the enterprise bean installed in the container. The object that implements a session bean's home interface is called a session EJBHome object. The container makes the session beans' home interfaces available to the client through JNDI.

The home interface allows a client to do the following:

- Create a new session object.

- Remove a session object.
- Get the `javax.ejb.EJBMetaData` interface for the session bean. The `javax.ejb.EJBMetaData` interface is intended to allow application assembly tools to discover information about the session bean, and to allow loose client/server binding and client-side scripting.
- Obtain a handle for the home interface. The home handle can be serialized and written to stable storage. Later, possibly in a different JVM, the handle can be deserialized from stable storage and used to obtain back a reference of the home interface.

5.3.1 Creating a session object

A home interface defines one or more `create<METHOD>(...)` methods, one for each way to create a session object. The arguments of the **create** methods are typically used to initialize the state of the created session object.

The following example illustrates a home interface that defines two `create<METHOD>(...)` methods:

```
public interface CartHome extends javax.ejb.EJBHome {
     Cart create(String customerName, String account)
          throws RemoteException, BadAccountException,
               CreateException;
     Cart createLargeCart(String customerName, String account)
          throws RemoteException, BadAccountException,
               CreateException;
}
```

The following example illustrates how a client creates a new session object using a `create<METHOD>(...)` method of the `CartHome` interface:

```
cartHome.create(“John”, “7506”);
```

5.3.2 Removing a session object

A client may remove a session object using the `remove()` method on the `javax.ejb.EJBObject` interface, or the r`emove(Handle handle)` method of the `javax.ejb.EJBHome` interface.

Because session objects do not have primary keys that are accessible to clients, invoking the `javax.ejb.EJBHome.remove(Object primaryKey)` method on a session results in the `javax.ejb.RemoveException`.

5.4 EJBObject

A client never directly accesses instances of the session bean's class. A client always uses the session bean's remote interface to access a session bean's instance. The class that implements the session bean's remote interface is provided by the container; its instances are called session `EJBObject`s.

A session EJBObject supports:

- The business logic methods of the object. The session EJBObject delegates invocation of a business method to the session bean instance.
- The methods of the `javax.ejb.EJBObject` interface. These methods allow the client to:
  - Get the session object's home interface.
  - Get the session object's handle.
  - Test if the session object is identical with another session object.
  - Remove the session object.

The implementation of the methods defined in the `javax.ejb.EJBObject` interface is provided by the container. They are not delegated to the instances of the session bean class.

5.5 Session object identity

Session objects are intended to be private resources used only by the client that created them. For this reason, session objects, from the client's perspective, appear anonymous. In contrast to entity objects, which expose their identity as a primary key, session objects hide their identity. As a result, the `EJBObject.getPrimaryKey()` method results in a `java.rmi.RemoteException`, and the `EJBHome.remove(Object primaryKey)` method results in a `javax.ejb.RemoveException` if called on a session bean. If the `EJBMetaData.getPrimaryKeyClass()`method is invoked on a `EJBMetaData` object for a Session bean, the method throws the `java.lang.RuntimeException`.

Since all session objects hide their identity, there is no need to provide a finder for them. The home interface of a session bean must not define any finder methods.

A session object handle can be held beyond the life of a client process by serializing the handle to persistent store. When the handle is later deserialized, the session object it returns will work as long as the session object still exists on the server. (An earlier timeout or server crash may have destroyed the session object.)

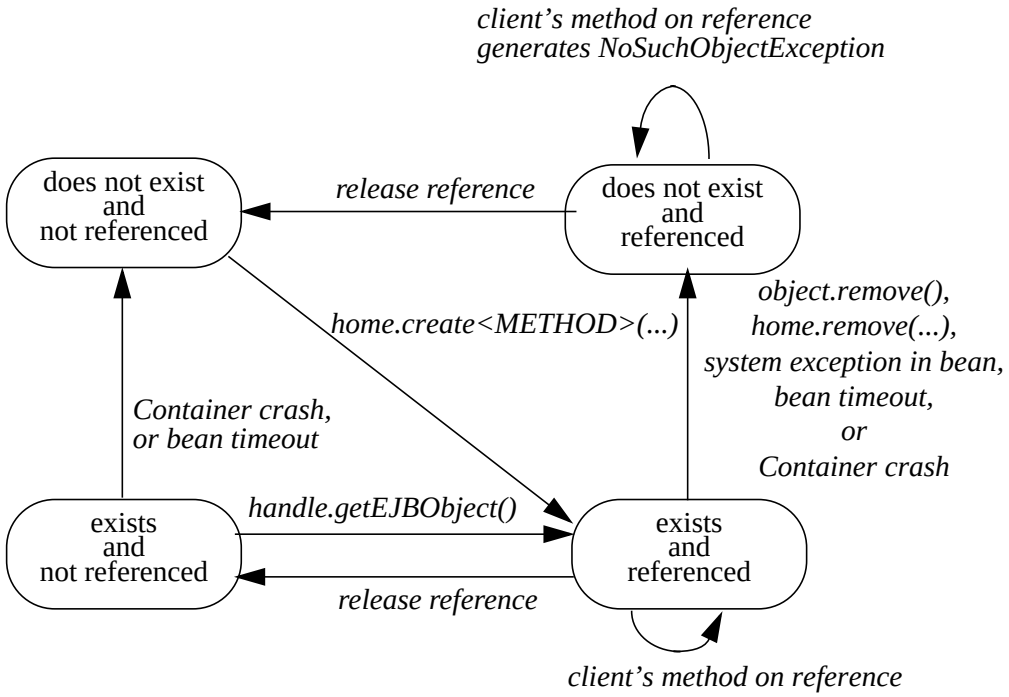
The client code must use the `javax.rmi.PortableRemoteObject.narrow(...)`method to convert the result of the `getEJBObject()` method invoked on a handle to the remote interface type.

A handle is not a capability, in the security sense, that would automatically grant its holder the right to invoke methods on the object. When a reference to a session object is obtained from a handle, and then a method on the session object is invoked, the container performs the usual access checks based on the caller's principal.

5.6 Client view of session object's life cycle

From a client point of view, the life cycle of a session object is illustrated below.

Figure 3 Lifecycle of a session object.



A session object does not exist until it is created. When a client creates a session object, the client has a reference to the newly created session object's remote interface.

A client that has a reference to a session object can then do any of the following:

- Invoke business methods defined in the session object's remote interface.
- Get a reference to the session object's home interface.
- Get a handle for the session object.

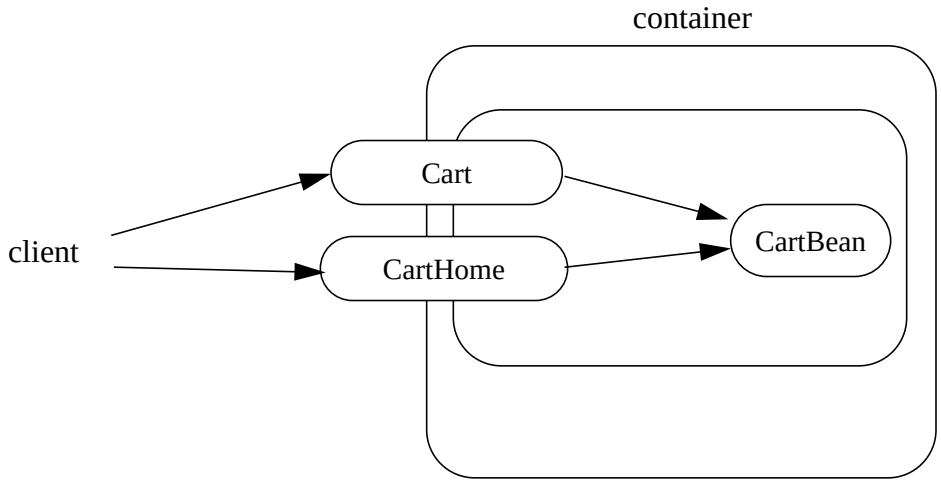
- Pass the reference as a parameter or return value within the scope of the client.
- Remove the session object. A container may also remove the session object automatically when the session object's lifetime expires.

It is invalid to reference a session object that does not exist. Attempted invocations on a session object that does not exist result in `java.rmi.NoSuchObjectException`.

5.7 Creating and using a session object

An example of the session bean runtime objects is illustrated by the following diagram:

Figure 4 Session Bean Example Objects



A client creates a `Cart` session object (which provides a shopping service) using a `create<METHOD>(...)` method of the Cart's home interface. The client then uses this session object to fill the cart with items and to purchase its contents.

Suppose that the end-user wishes to start the shopping session, suspend the shopping session temporarily for a day or two, and later complete the session. The client might implement this feature by getting the session object's handle, saving the serialized handle in persistent storage, then using it later to reestablish access to the original `Cart`.

For the following example, we start by looking up the Cart's home interface in JNDI. We then use the home interface to create a `Cart` session object and add a few items to it:

```
CartHome cartHome = (CartHome)javax.rmi.PortableRemoteObject.narrow(
     initialContext.lookup(...), CartHome.class);
Cart cart = cartHome.createLargeCart(...);
cart.addItem(66);
cart.addItem(22);
```

Next we decide to complete this shopping session at a later time so we serialize a handle to this cart session object and store it in a file:

```
Handle cartHandle = cart.getHandle();
// serialize cartHandle, store in a file...
```

Finally we deserialize the handle at a later time, re-create the reference to the cart session object, and purchase the contents of the shopping cart:

```
Handle cartHandle = ...; // deserialize from a file...
Cart cart = (Cart)javax.rmi.PortableRemoteObject.narrow(
              cartHandle.getEJBObject(), Cart.class);
cart.purchase();
cart.remove();
```

5.8 Object identity

5.8.1 Stateful session beans

A stateful session object has a unique identity that is assigned by the container at create time.

A client can determine if two object references refer to the same session object by invoking the `isIdentical(EJBObject otherEJBObject)` method on one of the references.

The following example illustrates the use of the `isIdentical` method for a stateful session object.

```
FooHome fooHome = ...; // obtain home of a stateful session bean
Foo foo1 = fooHome.create(...);
Foo foo2 = fooHome.create(...);

if (foo1.isIdentical(foo1)) {// this test must return true
    ...
}

if (foo1.isIdentical(foo2)) {// this test must return false
    ...
}
```

5.8.2 Stateless session beans

All session objects of the same stateless session bean within the same home have the same object identity, which is assigned by the container. If a stateless session bean is deployed multiple times (each deployment results in the creation of a distinct home), session objects from different homes will have a different identity.

The `isIdentical(EJBObject otherEJBObject)` method always returns true when used to compare object references of two session objects of the same stateless session bean.

The following example illustrates the use of the `isIdentical` method for a stateless session object.

```
FooHome fooHome = ...; // obtain home of a stateless session bean
Foo foo1 = fooHome.create();
Foo foo2 = fooHome.create();

if (foo1.isIdentical(foo1)) {// this test returns true
    ...
}

if (foo1.isIdentical(foo2)) {// this test returns true
    ...
}
```

5.8.3 getPrimaryKey()

The object identifier of a session object is, in general, opaque to the client. The result of `getPrimaryKey()` on a session EJBObject reference results in `java.rmi.RemoteException`.

5.9 Type narrowing

A client program that is intended to be interoperable with all compliant EJB Container implementations must use the `javax.rmi.PortableRemoteObject.narrow(...)` method to perform type-narrowing of the client-side representations of the home and remote interface.

Note: Programs using the cast operator for narrowing the remote and home interfaces are likely to fail if the Container implementation uses RMI-IIOP as the underlying communication transport.

Chapter 6 Session Bean Component Contract

This chapter specifies the contract between a session bean and its container. It defines the life cycle of the session bean instances.

This chapter defines the developer's view of session bean state management and the container's responsibility for managing session bean state.

6.1 Overview

A session bean instance is an instance of the session bean class. It holds the session object's state.

By definition, a session bean instance is an extension of the client that creates it:

- Its fields contain a **conversational state** on behalf of the session object's client. This state describes the conversation represented by a specific client/session object pair.
- It typically reads and updates data in a database on behalf of the client. Within a transaction, some of this data may be cached in the instance.
- Its lifetime is controlled by the client.

A container may also terminate a session bean instance's life after a deployer-specified time-out or as a result of the failure of the server on which the bean instance is running. For this reason, a client should be prepared to recreate a new session object if it loses the one it is using.

Typically, a session object's conversational state is not written to the database. A session bean developer simply stores it in the session bean instance's fields and assumes its value is retained for the lifetime of the instance.

On the other hand, the session bean must explicitly manage cached database data. A session bean instance must write any cached database updates prior to a transaction completion, and it must refresh its copy of any potentially stale database data at the beginning of the next transaction.

6.2 Goals

The goal of the session bean model is to make developing a session bean as simple as developing the same functionality directly in a client.

The container manages the life cycle of the session bean instances. It notifies the instances when bean action may be necessary, and it provides a full range of services to ensure that the session bean implementation is scalable and can support a large number of clients.

The remainder of this section describes the session bean life cycle in detail and the protocol between the bean and its container.

6.3 A container's management of its working set

To efficiently manage the size of its working set, a session bean container may need to temporarily transfer the state of an idle stateful session bean instance to some form of secondary storage. The transfer from the working set to secondary storage is called instance **passivation**. The transfer back is called **activation**.

A container may only passivate a session bean instance when the instance is **not** in a transaction.

To help the container manage its state, a session bean is specified at deployment as having one of the following state management modes:

- STATELESS—the session bean instances contain no conversational state between methods; any instance can be used for any client.
- STATEFUL—the session bean instances contain conversational state which must be retained across methods and transactions.

6.4 Conversational state

The conversational state of a STATEFUL session object is defined as the session bean instance's field values, plus the transitive closure of the objects from the instance's fields reached by following Java object references.

In advanced cases, a session object's conversational state may contain open resources, such as open sockets and open database cursors. A container cannot retain such open resources when a session bean instance is passivated. A developer of such a session bean must close and open the resources in the `ejbPassivate` and `ejbActivate` notifications.

6.4.1 Instance passivation and conversational state

The Bean Provider is required to ensure that the `ejbPassivate` method leaves the instance fields ready to be serialized by the Container. The objects that are assigned to the instance's non-`transient` fields after the `ejbPassivate` method completes must be one of the following:

- A serializable object[2].
- A `null`.
- An enterprise bean's remote interface reference, even if the stub class is not serializable.
- An enterprise bean's home interface reference, even if the stub class is not serializable.
- A reference to the `SessionContext` object, even if it is not serializable.
- A reference to the environment naming context (that is, the `java:comp/env` JNDI context) or any of its subcontexts.
- A reference to the `UserTransaction` interface.
- A reference to a resource manager connection factory.
- An object that is not directly serializable, but becomes serializable by replacing the references to an enterprise bean's remote and home interfaces, the references to the `SessionContext` object, the references to the `java:comp/env` JNDI context and its subcontexts, and the references to the `UserTransaction` interface by serializable objects during the object's serialization.

This means, for example, that the Bean Provider must close all JDBC™ connections in `ejbPassivate` *and assign the instance's fields storing the connections to* `null`.

[2] Note that the Java Serialization protocol dynamically determines whether or not an object is serializable. This means that it is possible to serialize an object of a serializable subclass of a non-serializable declared field type.

The last bulleted item covers cases such as storing Collections of remote interfaces in the conversational state.

The Bean Provider must assume that the content of transient fields may be lost between the `ejbPassivate` and `ejbActivate` notifications. Therefore, the Bean Provider should not store in a `transient` field a reference to any of the following objects: `SessionContext` object; environment JNDI naming context and any its subcontexts; home and remote interfaces; and the `UserTransaction` interface.

The restrictions on the use of transient fields ensure that Containers can use Java Serialization during passivation and activation.

The following are the requirements for the Container.

The container performs the Java programming language Serialization (or its equivalent) of the instance's state after it invokes the `ejbPassivate` method on the instance.

The container must be able to properly save and restore the reference to the remote and home interfaces of the enterprise beans stored in the instance's state even if the classes that implement the object references are not serializable.

The container may use, for example, the object replacement technique that is part of the java.io.ObjectOutputStream and java.io.ObjectInputStream protocol to externalize the remote and home references.

If the session bean instance stores in its conversational state an object reference to the `javax.ejb.SessionContext` interface passed to the instance in the `setSessionContext(...)` method, the container must be able to save and restore the reference across the instance's passivation. The container can replace the original `SessionContext` object with a different and functionally equivalent `SessionContext` object during activation.

If the session bean instance stores in its conversational state an object reference to the `java:comp/env` JNDI context or its subcontext, the container must be able to save and restore the object reference across the instance's passivation. The container can replace the original object with a different and functionally equivalent object during activation.

If the session bean instance stores in its conversational state an object reference to the `UserTransaction` interface, the container must be able to save and restore the object reference across the instance's passivation. The container can replace the original object with a different and functionally equivalent object during activation.

The container may destroy a session bean instance if the instance does not meet the requirements for serialization after `ejbPassivate`.

While the container is not required to use the Serialization protocol for the Java programming language to store the state of a passivated session instance, it must achieve the equivalent result. The one exception is that containers are not required to reset the value of `transient` fields during activation[3]. Declaring the session bean's fields as `transient` is, in general, discouraged.

6.4.2 The effect of transaction rollback on conversational state

A session object's conversational state is not transactional. It is not automatically rolled back to its initial state if the transaction in which the object has participated rolls back.

If a rollback could result in an inconsistency between a session object's conversational state and the state of the underlying database, the bean developer (or the application development tools used by the developer) must use the `afterCompletion` notification to manually reset its state.

6.5 Protocol between a session bean instance and its container

Containers themselves make no actual service demands on the session bean instances. The container makes calls on a bean instance to provide it with access to container services and to deliver notifications issued by the container.

6.5.1 The required *SessionBean* interface

All session beans must implement the `SessionBean` interface.

The bean's container calls the `setSessionContext` method to associate a session bean instance with its context maintained by the **container**. Typically, a session bean instance retains its session context as part of its conversational state.

The `ejbRemove` notification signals that the instance is in the process of being removed by the container. In the `ejbRemove` method, the instance typically releases the same resources that it releases in the `ejbPassivate` method.

The `ejbPassivate` notification signals the intent of the container to passivate the instance. The `ejbActivate` notification signals the instance it has just been reactivated. Because containers automatically maintain the conversational state of a session bean instance when it is passivated, most session beans can ignore these notifications. Their purpose is to allow session beans to maintain those open resources that need to be closed prior to an instance's passivation and then reopened during an instance's activation.

6.5.2 The *SessionContext* interface

A container provides the session bean instances with a `SessionContext`, which gives the session bean instance access to the instance's context maintained by the container. The `SessionContext` interface has the following methods:

- The `getEJBObject` method returns the session bean's remote interface.

[3] This is to allow the Container to swap out an instance's state through techniques other than the Java Serialization protocol. For example, the Container's Java Virtual Machine implementation may use a block of memory to keep the instance's variables, and the Container swaps the whole memory block to the disk instead of performing Java Serialization on the instance.

- The `getEJBHome` method returns the session bean's home interface.
- The `getCallerPrincipal` method returns the `java.security.Principal` that identifies the invoker of the bean instance's EJB object.
- The `isCallerInRole` method tests if the session bean instance's caller has a particular role.
- The `setRollbackOnly` method allows the instance to mark the current transaction such that the only outcome of the transaction is a rollback. Only instances of a session bean with container-managed transaction demarcation can use this method.
- The `getRollbackOnly` method allows the instance to test if the current transaction has been marked for rollback. Only instances of a session bean with container-managed transaction demarcation can use this method.
- The `getUserTransaction` method returns the `javax.transaction.UserTransaction` interface. The instance can use this interface to demarcate transactions and to obtain transaction status. Only instances of a session bean with bean-managed transaction demarcation can use this method.

6.5.3 The optional *SessionSynchronization* interface

A session bean class can optionally implement the `javax.ejb.SessionSynchronization` interface. This interface provides the session bean instances with transaction synchronization notifications. The instances can use these notifications, for example, to manage database data they may cache within transactions.

The `afterBegin` notification signals a session bean instance that a new transaction has begun. The container invokes this method before the first business method within a transaction (which is not necessarily at the beginning of the transaction). The `afterBegin` notification is invoked with the transaction context. The instance may do any database work it requires within the scope of the transaction.

The `beforeCompletion` notification is issued when a session bean instance's client has completed work on its current transaction but prior to committing the resource managers used by the instance. At this time, the instance should write out any database updates it has cached. The instance can cause the transaction to roll back by invoking the `setRollbackOnly` method on its session context.

The `afterCompletion` notification signals that the current transaction has completed. A completion status of `true` indicates that the transaction has committed; a status of `false` indicates that a rollback has occurred. Since a session bean instance's conversational state is not transactional, it may need to manually reset its state if a rollback occurred.

All container providers must support `SessionSynchronization`. It is optional only for the bean implementor. If a bean class implements `SessionSynchronization`, the container must invoke the `afterBegin`, `beforeCompletion` and `afterCompletion` notifications as required by the specification.

Only a stateful Session bean with container-managed transaction demarcation may implement the `SessionSynchronization` interface. A stateless Session bean must not implement the `SessionSynchronization` interface.

There is no need for a Session bean with bean-managed transaction to rely on the synchronization call backs because the bean is in control of the commit—the bean knows when the transaction is about to be committed and it knows the outcome of the transaction commit.

6.5.4 Business method delegation

The session bean's remote interface defines the business methods callable by a client. The session bean's remote interface is implemented by the session EJBObject class generated by the container tools. The session EJBObject class delegates an invocation of a business method to the matching business method that is implemented in the session bean class.

6.5.5 Session bean's *ejbCreate<METHOD>(...)* methods

A client creates a session bean instance using one of the `create<METHOD>` methods defined in the session bean's home interface. The session bean's home interface is provided by the bean developer; its implementation is generated by the deployment tools provided by the container provider.

The container creates an instance of a session bean in three steps. First, the container calls the bean class' `newInstance` method to create a new session bean instance. Second, the container calls the `setSessionContext` method to pass the context object to the instance. Third, the container calls the instance's `ejbCreate<METHOD>` method whose signature matches the signature of the `create<METHOD>` method invoked by the client. The input parameters sent from the client are passed to the `ejbCreate<METHOD>` method.

Each session bean class must have at least one `ejbCreate<METHOD>` method. The number and signatures of a session bean's `create<METHOD>` methods are specific to each session bean class.

Since a session bean represents a specific, private conversation between the bean and its client, its create parameters typically contain the information the client uses to customize the bean instance for its use.

6.5.6 Serializing session bean methods

A container serializes calls to each session bean instance. Most containers will support many instances of a session bean executing concurrently; however, each instance sees only a serialized sequence of method calls. Therefore, a session bean does not have to be coded as reentrant.

The container must serialize all the container-invoked callbacks (that is, the methods `ejbPassivate`, `beforeCompletion`, and so on), and it must serialize these callbacks with the client-invoked business method calls.

Clients are not allowed to make concurrent calls to a stateful session object. If a client-invoked business method is in progress on an instance when another client-invoked call, from the same or different client, arrives at the same instance of a stateful session bean class, the container may throw the `java.rmi.RemoteException` to the second client[4]. This restriction does not apply to a stateless session bean because the container routes each request to a different instance of the session bean class.

6.5.7 Transaction context of session bean methods

The implementation of a business method defined in the remote interface is invoked in the scope of a transaction determined by the transaction attribute specified in the deployment descriptor.

A session bean's `afterBegin` and `beforeCompletion` methods are always called with the same transaction context as the business methods executed between the `afterBegin` and `beforeCompletion` methods.

A session bean's `newInstance, setSessionContext`, `ejbCreate`, `ejbRemove`, `ejbPassivate`, `ejbActivate,` and `afterCompletion` methods are called with an unspecified transaction context. Refer to Subsection 16.6.5 for how the Container executes methods with an unspecified transaction context.

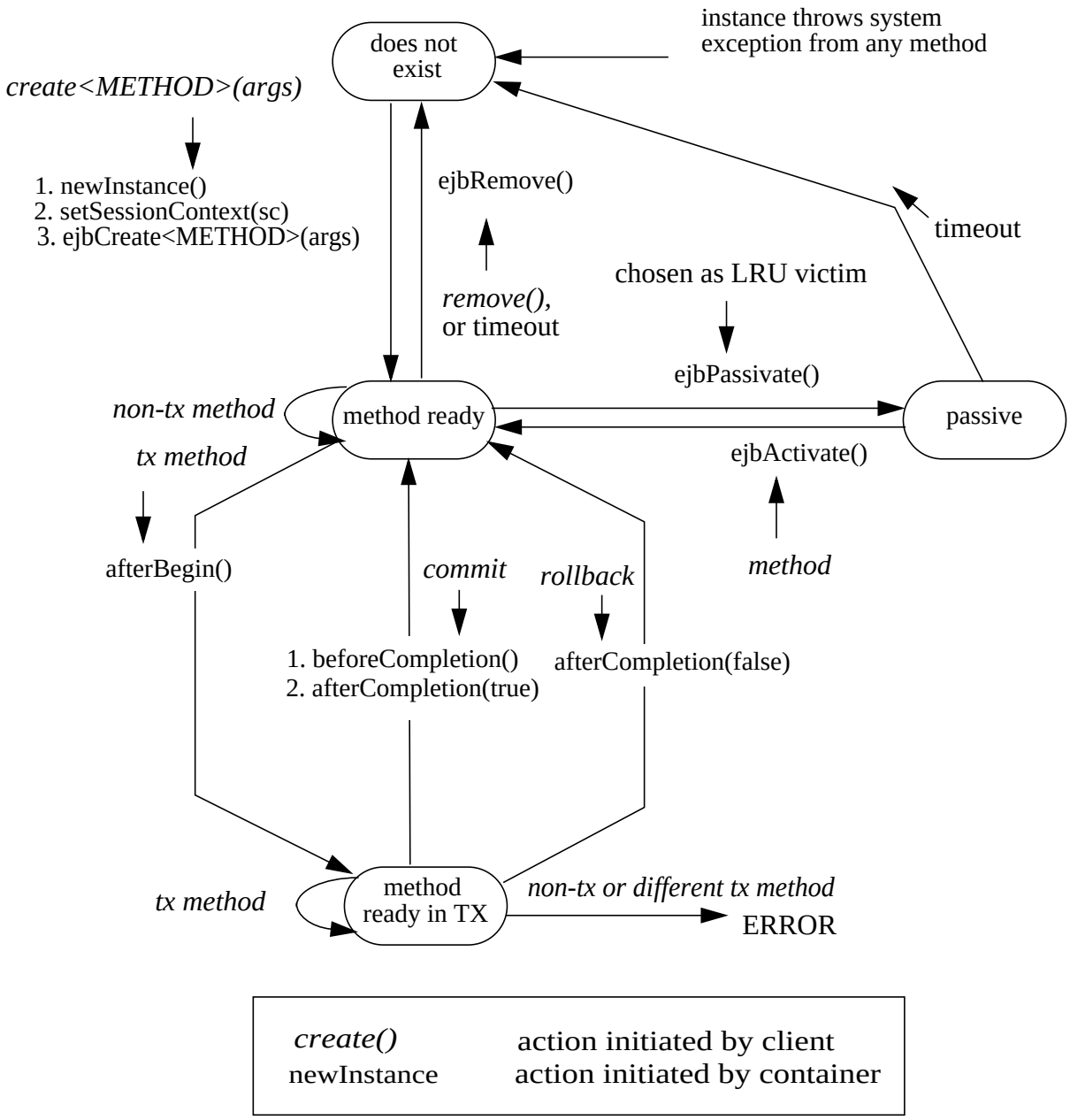
For example, it would be wrong to perform database operations within a session bean's `ejbCreate` *or* `ejbRemove` *method and to assume that the operations are part of the client's transaction. The* `ejbCreate` *and* `ejbRemove` *methods are not controlled by a transaction attribute because handling rollbacks in these methods would greatly complicate the session instance's state diagram.*

6.6 STATEFUL Session Bean State Diagram

The following figure illustrates the life cycle of a STATEFUL session bean instance.

[4] In certain special circumstances (e.g., to handle clustered web container architectures), the container may instead queue or serialize such concurrent requests. Clients, however, cannot rely on this behavior.

Figure 5 Lifecycle of a STATEFUL Session bean instance



The following steps describe the life cycle of a STATEFUL session bean instance:

- A session bean instance's life starts when a client invokes a `create<METHOD>(...)` method on the session bean's home interface. This causes the container to invoke `newIn-`

stance()` on the session bean class to create a new session bean instance. Next, the container calls `setSessionContext()` and `ejbCreate<METHOD>(...)` on the instance and returns the remote reference of the session object to the client. The instance is now in the method ready state.

- The session bean instance is now ready for client's business methods. Based on the transaction attributes in the session bean's deployment descriptor and the transaction context associated with the client's invocation, a business method is executed either in a transaction context or with an unspecified transaction context (shown as tx method and non-tx method in the diagram). See Chapter 16 for how the container deals with transactions.

- A non-transactional method is executed while the instance is in the method ready state.

- An invocation of a transactional method causes the instance to be included in a transaction. When the session bean instance is included in a transaction, the container issues the `afterBegin()` method on it. The `afterBegin` is delivered to the instance before any business method is executed as part of the transaction. The instance becomes associated with the transaction and will remain associated with the transaction until the transaction completes.

- Session bean methods invoked by the client in this transaction can now be delegated to the bean instance. An error occurs if a client attempts to invoke a method on the session object and the deployment descriptor for the method requires that the container invoke the method in a different transaction context than the one with which the instance is currently associated or in an unspecified transaction context.

- If a transaction commit has been requested, the transaction service notifies the container of the commit request before actually committing the transaction, and the container issues a `beforeCompletion` on the instance. When `beforeCompletion` is invoked, the instance should write any cached updates to the database. If a transaction rollback had been requested instead, the rollback status is reached without the container issuing a `beforeCompletion`. The container may not call the `beforeCompletion` method if the transaction has been marked for rollback (nor does the instance write any cached updates to the database).

- The transaction service then attempts to commit the transaction, resulting in either a commit or rollback.

- When the transaction completes, the container issues `afterCompletion` on the instance, specifying the status of the completion (either commit or rollback). If a rollback occurred, the bean instance may need to reset its conversational state back to the value it had at the beginning of the transaction.

- The container's caching algorithm may decide that the bean instance should be evicted from memory (this could be done at the end of each method, or by using an LRU policy). The container issues `ejbPassivate` on the instance. After this completes, the container saves the instance's state to secondary storage. A session bean can be passivated only between transactions, and not within a transaction.

- While the instance is in the passivated state, the Container may remove the session object after the expiration of a timeout specified by the deployer. All object references and handles for the

session object become invalid. If a client attempts to invoke the session object, the Container will throw the `java.rmi.NoSuchObjectException` to the client.

- If a client invokes a session object whose session bean instance has been passivated, the container will activate the instance. To activate the session bean instance, the container restores the instance's state from secondary storage and issues `ejbActivate` on it.

- The session bean instance is again ready for client methods.

- When the client calls `remove` on the home or remote interface to remove the session object, the container issues `ejbRemove()` on the bean instance. This ends the life of the session bean instance and the associated session object. Any subsequent attempt by its client to invoke the session object causes the `java.rmi.NoSuchObjectException` to be thrown. (This exception is a subclass of `java.rmi.RemoteException`). The `ejbRemove()` method cannot be called when the instance is participating in a transaction. An attempt to remove a session object while the object is in a transaction will cause the container to throw the `javax.ejb.RemoveException` to the client. Note that a container can also invoke the `ejbRemove()` method on the instance without a client call to `remove` the session object after the lifetime of the EJB object has expired.

Notes:

1. The Container must call the `afterBegin`, `beforeCompletion`, and `afterCompletion` methods if the session bean class implements, directly or indirectly, the `SessionSynchronization` interface. The Container does not call these methods if the session bean class does not implement the `SessionSynchronization` interface.

6.6.1 Operations allowed in the methods of a stateful session bean class

Table 2 defines the methods of a stateful session bean class from which the session bean instances can access the methods of the `javax.ejb.SessionContext` interface, the `java:comp/env` environment naming context, resource managers, and other enterprise beans.

If a session bean instance attempts to invoke a method of the `SessionContext` interface, and that access is not allowed in Table 2, the Container must throw the `java.lang.IllegalStateException`.

If a session bean instance attempts to access a resource manager or an enterprise bean, and that access is not allowed in Table 2, the behavior is undefined by the EJB architecture.

Table 2 Operations allowed in the methods of a stateful session bean

Bean method	**Bean method can perform the following operations**	
	Container-managed transaction demarcation	**Bean-managed transaction demarcation**
constructor	-	-
setSessionContext	SessionContext methods: *getEJBHome* JNDI access to java:comp/env	SessionContext methods: *getEJBHome* JNDI access to java:comp/env
ejbCreate ejbRemove ejbActivate ejbPassivate	SessionContext methods: *getEJBHome*, *getCallerPrincipal*, *isCallerInRole*, *getEJBObject* JNDI access to java:comp/env Resource manager access Enterprise bean access	SessionContext methods: *getEJBHome*, *getCallerPrincipal*, *isCallerInRole*, *getEJBObject*, *getUserTransaction* UserTransaction methods JNDI access to java:comp/env Resource manager access Enterprise bean access
business method from remote interface	SessionContext methods: *getEJBHome*, *getCallerPrincipal*, *getRollbackOnly*, *isCallerInRole*, *setRollbackOnly*, *getEJBObject* JNDI access to java:comp/env Resource manager access Enterprise bean access	SessionContext methods: *getEJBHome*, *getCallerPrincipal*, *isCallerInRole*, *getEJBObject*, *getUserTransaction* UserTransaction methods JNDI access to java:comp/env Resource manager access Enterprise bean access
afterBegin beforeCompletion	SessionContext methods: *getEJBHome*, *getCallerPrincipal*, *getRollbackOnly*, *isCallerInRole*, *setRollbackOnly*, *getEJBObject* JNDI access to java:comp/env Resource manager access Enterprise bean access	N/A (a bean with bean-managed transaction demarcation cannot implement the SessionSynchronization interface)
afterCompletion	SessionContext methods: *getEJBHome*, *getCallerPrincipal*, *isCallerInRole*, *getEJBObject* JNDI access to java:comp/env	

Notes:

- The `ejbCreate<METHOD>`, `ejbRemove`, `ejbPassivate`, and `ejbActivate` methods of a session bean with container-managed transaction demarcation execute with an unspecified transaction context. Refer to Subsection 16.6.5 for how the Container executes methods with an unspecified transaction context.

Additional restrictions:

- The `getRollbackOnly` and `setRollbackOnly` methods of the `SessionContext` interface should be used only in the session bean methods that execute in the context of a transaction. The Container must throw the `java.lang.IllegalStateException` if the methods are invoked while the instance is not associated with a transaction.

The reasons for disallowing the operations in Table 2 follow:

- Invoking the `getEJBObject` methods is disallowed in the session bean methods in which there is no session object identity established for the instance.
- Invoking the `getCallerPrincipal` and `isCallerInRole` methods is disallowed in the session bean methods for which the Container does not have a client security context.
- Invoking the `getRollbackOnly` and `setRollbackOnly` methods is disallowed in the session bean methods for which the Container does not have a meaningful transaction context, and to all session beans with bean-managed transaction demarcation.
- Accessing resource managers and enterprise beans is disallowed in the session bean methods for which the Container does not have a meaningful transaction context or client security context.
- The `UserTransaction` interface is unavailable to enterprise beans with container-managed transaction demarcation.

6.6.2 Dealing with exceptions

A `RuntimeException` thrown from any method of the session bean class (including the business methods and the callbacks invoked by the Container) results in the transition to the "does not exist" state. Exception handling is described in detail in Chapter 17.

From the client perspective, the corresponding session object does not exist any more. Subsequent invocations through the remote interface will result in `java.rmi.NoSuchObjectException`.

6.6.3 Missed `ejbRemove()` calls

The Bean Provider cannot assume that the Container will always invoke the `ejbRemove()` method on a session bean instance. The following scenarios result in `ejbRemove()` not being called on an instance:

- A crash of the EJB Container.
- A system exception thrown from the instance's method to the Container.
- A timeout of client inactivity while the instance is in the `passive` state. The timeout is specified by the Deployer in an EJB Container implementation specific way.

If the session bean instance allocates resources in the `ejbCreate<METHOD>(...)` method and/or in the business methods, and normally releases the resources in the `ejbRemove()` method, these resources will not be automatically released in the above scenarios. The application using the session bean should provide some clean up mechanism to periodically clean up the unreleased resources.

For example, if a shopping cart component is implemented as a session bean, and the session bean stores the shopping cart content in a database, the application should provide a program that runs periodically and removes "abandoned" shopping carts from the database.

6.6.4 Restrictions for transactions

The state diagram implies the following restrictions on transaction scoping of the client invoked business methods. The restrictions are enforced by the container and must be observed by the client programmer.

- A session bean instance can participate in at most a single transaction at a time.
- If a session bean instance is participating in a transaction, it is an error for a client to invoke a method on the session object such that the transaction attribute in the deployment descriptor would cause the container to execute the method in a different transaction context or in an unspecified transaction context. The container throws the `java.rmi.RemoteException` to the client in such a case.
- If a session bean instance is participating in a transaction, it is an error for a client to invoke the `remove` method on the session object's remote or home interface object. The container must detect such an attempt and throw the `javax.ejb.RemoveException` to the client. The container should not mark the client's transaction for rollback, thus allowing the client to recover.

6.7 Object interaction diagrams for a STATEFUL session bean

This section contains object interaction diagrams (OID) that illustrates the interaction of the classes.

6.7.1 Notes

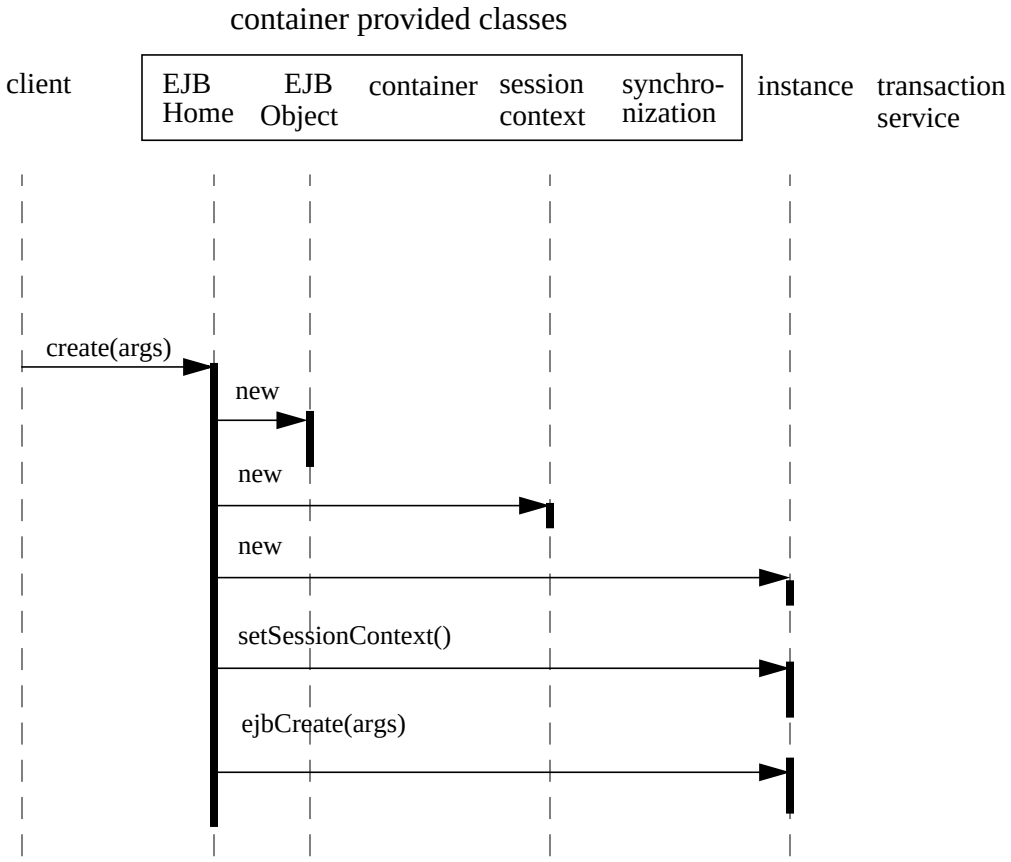
The object interaction diagrams illustrate a box labeled "container-provided classes." These are either classes that are part of the container, or classes that were generated by the container tools. These classes communicate with each other through protocols that are container-implementation specific. Therefore, the communication between these classes is not shown in the diagrams.

The classes shown in the diagrams should be considered as an illustrative implementation rather than as a prescriptive one.

6.7.2 Creating a session object

The following diagram illustrates the creation of a session object.

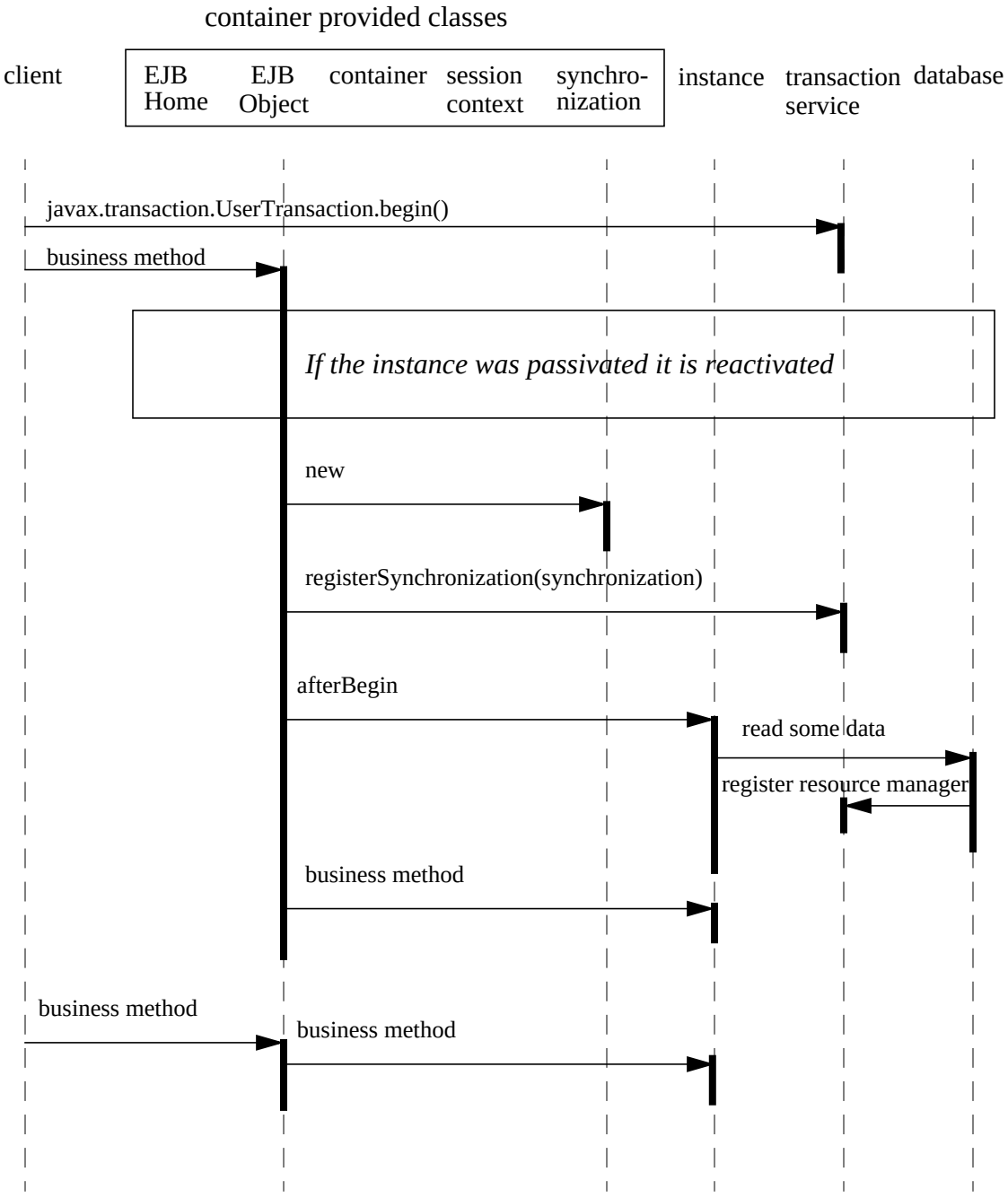
Figure 6 OID for Creation of a session object



6.7.3 Starting a transaction

The following diagram illustrates the protocol performed at the beginning of a transaction.

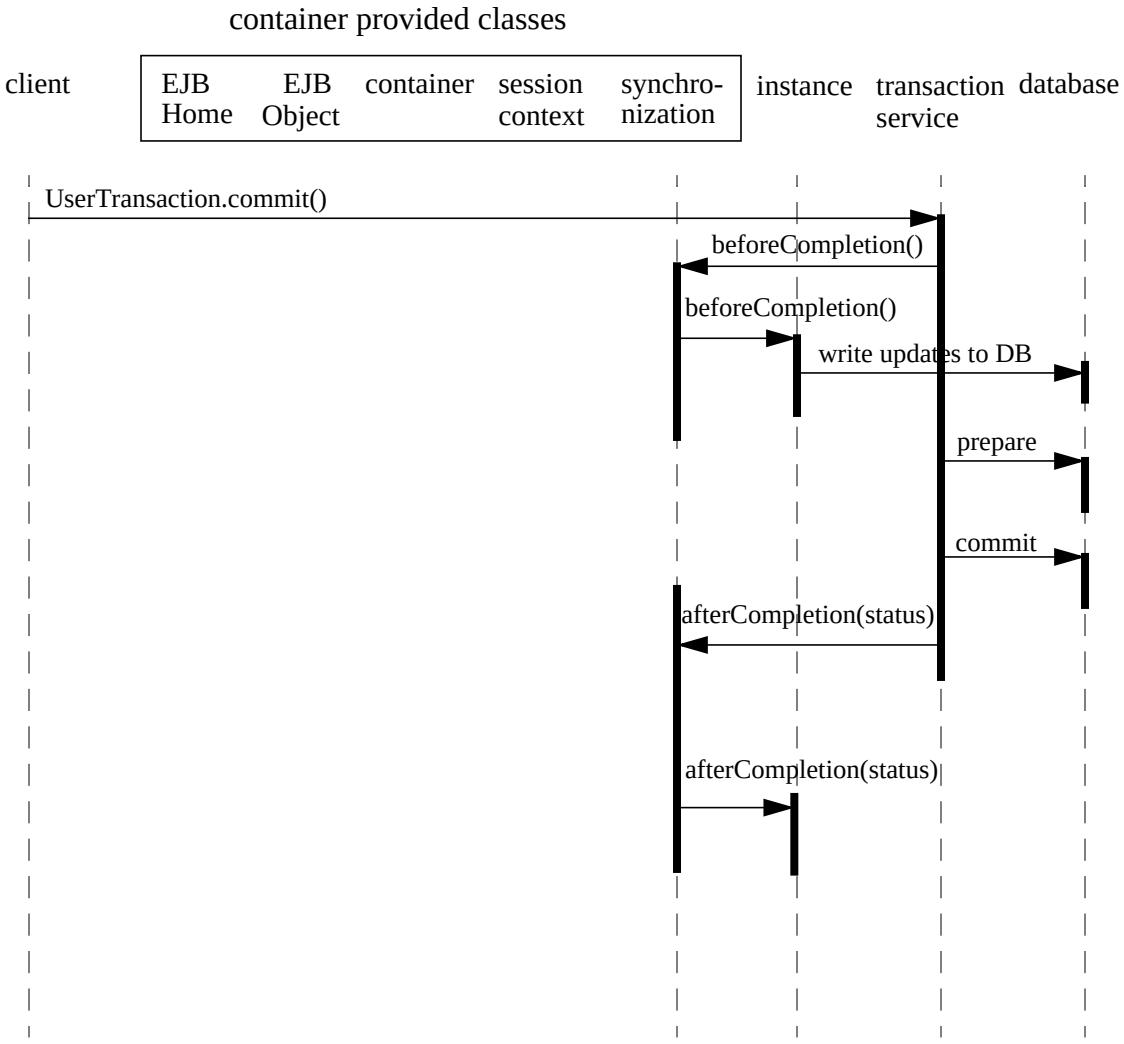
Figure 7 OID for session object at start of a transaction.



6.7.4 Committing a transaction

The following diagram illustrates the transaction synchronization protocol for a session object.

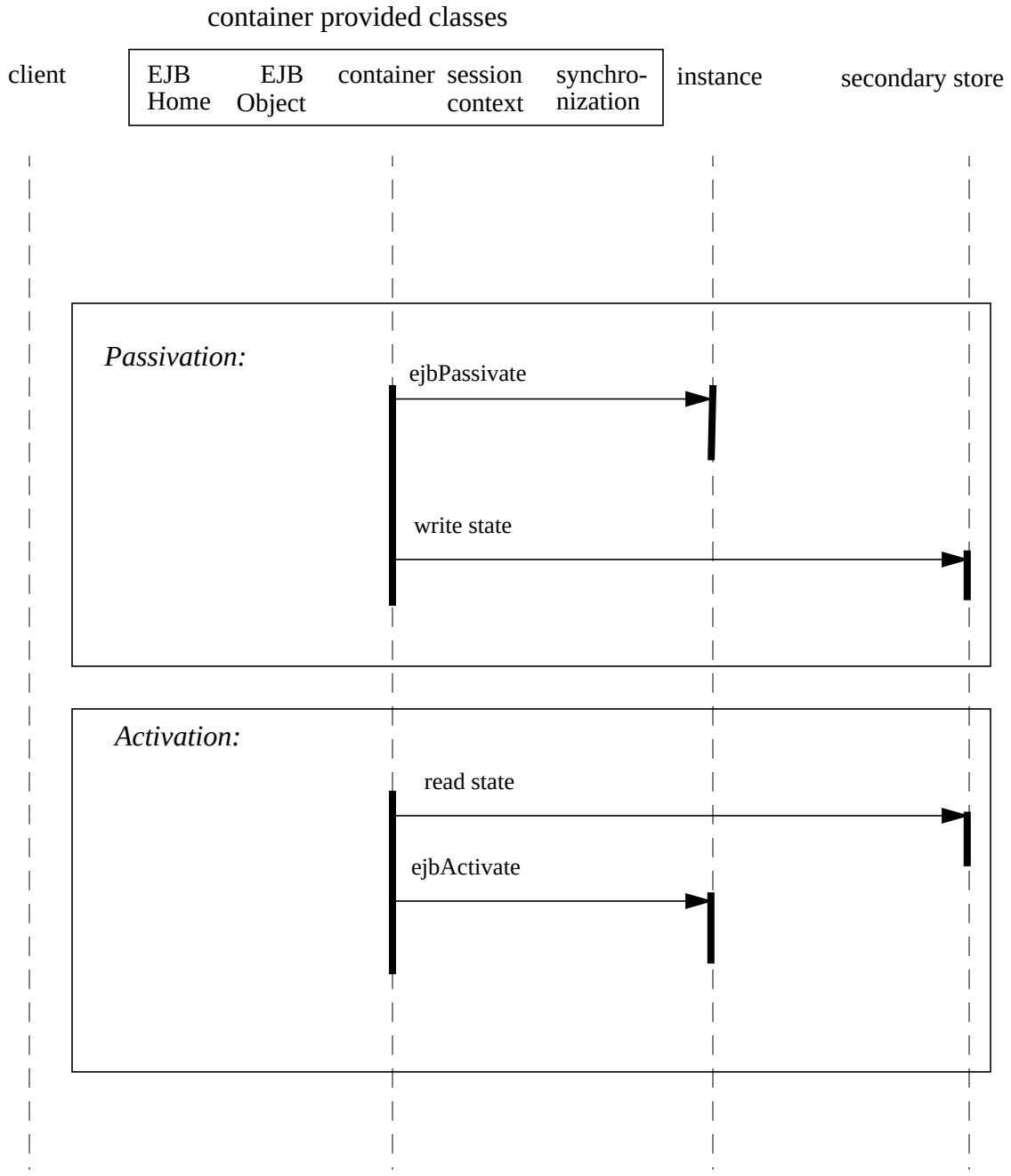
Figure 8 OID for session object transaction synchronization



6.7.5 Passivating and activating an instance between transactions

The following diagram illustrates the passivation and reactivation of a session bean instance. Passivation typically happens spontaneously based on the needs of the container. Activation typically occurs when a client calls a method.

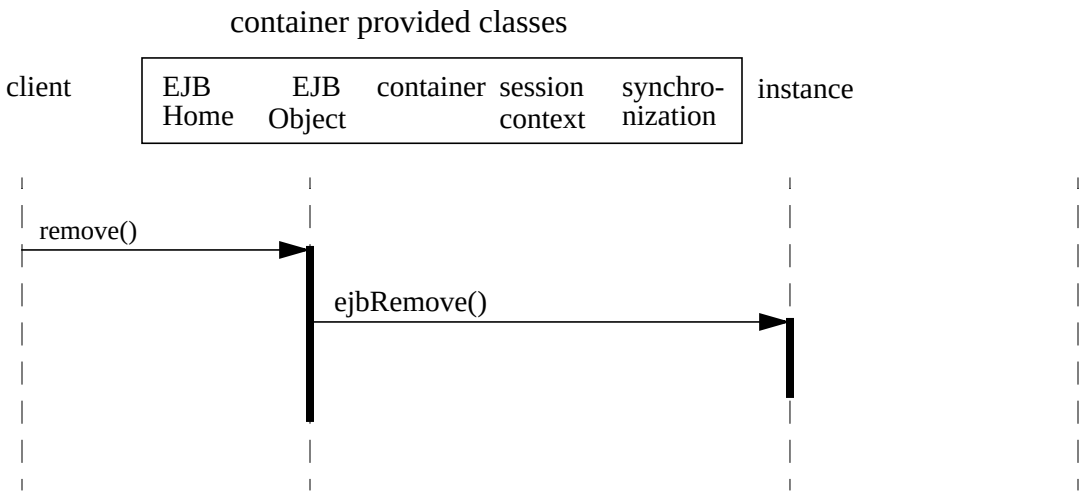
Figure 9 OID for passivation and activation of a session object



6.7.6 Removing a session object

The following diagram illustrates the removal of a session object.

Figure 10 OID for the removal of a session object



6.8 Stateless session beans

Stateless session beans are session beans whose instances have no conversational state. This means that all bean instances are equivalent when they are not involved in servicing a client-invoked method.

The term "stateless" signifies that an instance has no state for a specific client. However, the instance variables of the instance can contain the state across client-invoked method calls. Examples of such states include an open database connection and an object reference to an EJB object.

The home interface of a stateless session bean must have one `create` method that takes no arguments and returns the session bean's remote interface. There can be no other `create` methods in the home interface. The session bean class must define a single `ejbCreate` method that takes no arguments.

Because all instances of a stateless session bean are equivalent, the container can choose to delegate a client-invoked method to any available instance. This means, for example, that the Container may delegate the requests from the same client within the same transaction to different instances, and that the Container may interleave requests from multiple transactions to the same instance.

A container only needs to retain the number of instances required to service the current client load. Due to client "think time," this number is typically much smaller than the number of active clients. Passivation is not needed for stateless sessions. The container creates another stateless session bean instance if one is needed to handle an increase in client work load. If a stateless session bean is not needed to handle the current client work load, the container can destroy it.

Because stateless session beans minimize the resources needed to support a large population of clients, depending on the implementation of the container, applications that use stateless session beans may scale somewhat better than those using stateful session beans. However, this benefit may be offset by the increased complexity of the client application that uses the stateless beans.

Clients use the `create` and `remove` methods on the home interface of a stateless session bean in the same way as on a stateful session bean. To the client, it appears as if the client controls the life cycle of the session object. However, the container handles the `create` and `remove` calls without necessarily creating and removing an EJB instance.

There is no fixed mapping between clients and stateless instances. The container simply delegates a client's work to any available instance that is method-ready.

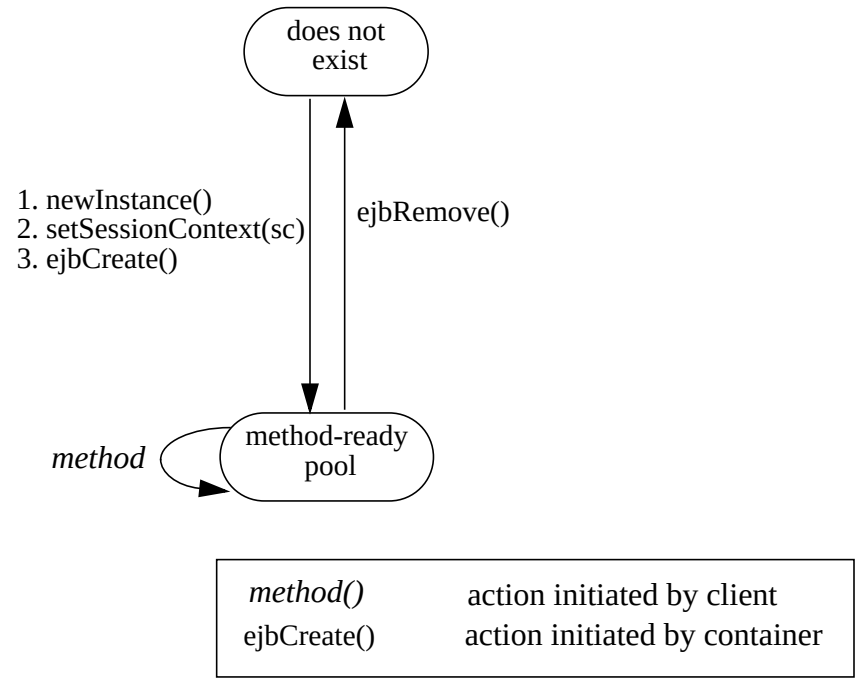
A stateless session bean must not implement the `javax.ejb.SessionSynchronization` interface.

6.8.1 Stateless session bean state diagram

When a client calls a method on a stateless session object, the container selects one of its **method-ready** instances and delegates the method invocation to it.

The following figure illustrates the life cycle of a STATELESS session bean instance.

Figure 11 Lifecycle of a STATELESS Session bean



The following steps describe the lifecyle of a session bean instance:

- A stateless session bean instance's life starts when the container invokes `newInstance()` on the session bean class to create a new instance. Next, the container calls `setSessionContext()` followed by `ejbCreate()` on the instance. The container can perform the instance creation at any time—there is no relationship to a client's invocation of the `create()` method.
- The session bean instance is now ready to be delegated a business method call from any client.
- When the container no longer needs the instance (usually when the container wants to reduce the number of instances in the method-ready pool), the container invokes ejbRemove() on it. This ends the life of the stateless session bean instance.

6.8.2 Operations allowed in the methods of a stateless session bean class

Table 3 defines the methods of a stateless session bean class in which the session bean instances can access the methods of the `javax.ejb.SessionContext` interface, the `java:comp/env` environment naming context, resource managers, and other enterprise beans.

If a session bean instance attempts to invoke a method of the `SessionContext` interface, and the access is not allowed in Table 3, the Container must throw the `java.lang.IllegalStateException.`

If a session bean instance attempts to access a resource manager or an enterprise bean and the access is not allowed in Table 3, the behavior is undefined by the EJB architecture.

Table 3 Operations allowed in the methods of a stateless session bean

Bean method	**Bean method can perform the following operations**	
	Container-managed transaction demarcation	**Bean-managed transaction demarcation**
constructor	-	-
setSessionContext	SessionContext methods: *getEJBHome* JNDI access to java:comp/env	SessionContext methods: *getEJBHome* JNDI access to java:comp/env
ejbCreate ejbRemove	SessionContext methods: *getEJBHome*, *getEJBObject* JNDI access to java:comp/env	SessionContext methods: *getEJBHome*, *getEJBObject, getUserTransaction* JNDI access to java:comp/env
business method from remote interface	SessionContext methods: *getEJBHome*, *getCallerPrincipal*, *getRollback-Only, isCallerInRole*, *setRollback-Only, getEJBObject* JNDI access to java:comp/env Resource manager access Enterprise bean access	SessionContext methods: *getEJBHome*, *getCallerPrincipal*, *isCallerInRole, getEJBObject, getUserTransaction* UserTransaction methods JNDI access to java:comp/env Resource manager access Enterprise bean access

Additional restrictions:

- The `getRollbackOnly` and `setRollbackOnly` methods of the `SessionContext` interface should be used only in the session bean methods that execute in the context of a transaction. The Container must throw the `java.lang.IllegalStateException` if the methods are invoked while the instance is not associated with a transaction.

The reasons for disallowing operations in Table 3:

- Invoking the `getEJBObject` method is disallowed in the session bean methods in which there is no session object identity associated with the instance.
- Invoking the `getCallerPrincipal` and `isCallerInRole` methods is disallowed in the session bean methods for which the Container does not have a client security context.
- Invoking the `getRollbackOnly` and `setRollbackOnly` methods is disallowed in the session bean methods for which the Container does not have a meaningful transaction context, and for all session beans with bean-managed transaction demarcation.

- Accessing resource managers and enterprise beans is disallowed in the session bean methods for which the Container does not have a meaningful transaction context or client security context.
- The `UserTransaction` interface is unavailable to session beans with container-managed transaction demarcation.

6.8.3 Dealing with exceptions

A `RuntimeException` thrown from any method of the enterprise bean class (including the business methods and the callbacks invoked by the Container) results in the transition to the "does not exist" state. Exception handling is described in detail in Chapter 17.

From the client perspective, the session object continues to exist. The client can continue accessing the session object because the Container can delegate the client's requests to another instance.

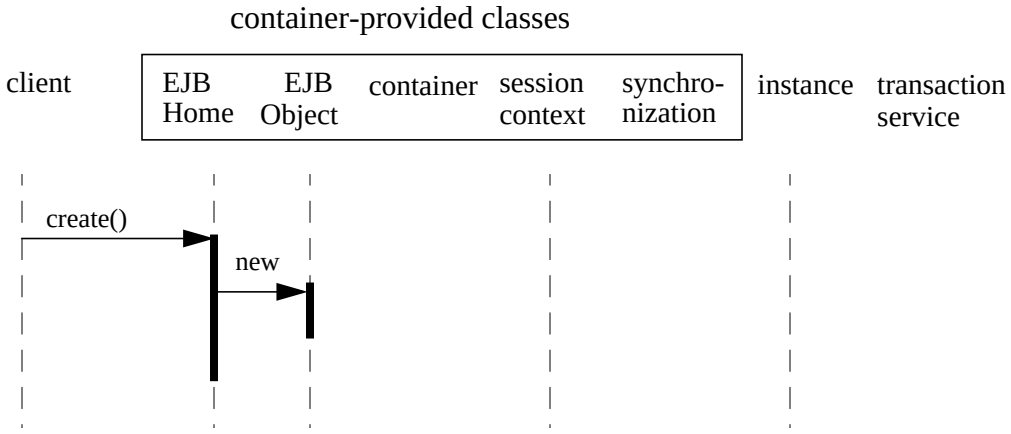
6.9 Object interaction diagrams for a STATELESS session bean

This section contains object interaction diagrams that illustrates the interaction of the classes.

6.9.1 Client-invoked *create()*

The following diagram illustrates the creation of a stateless session object.

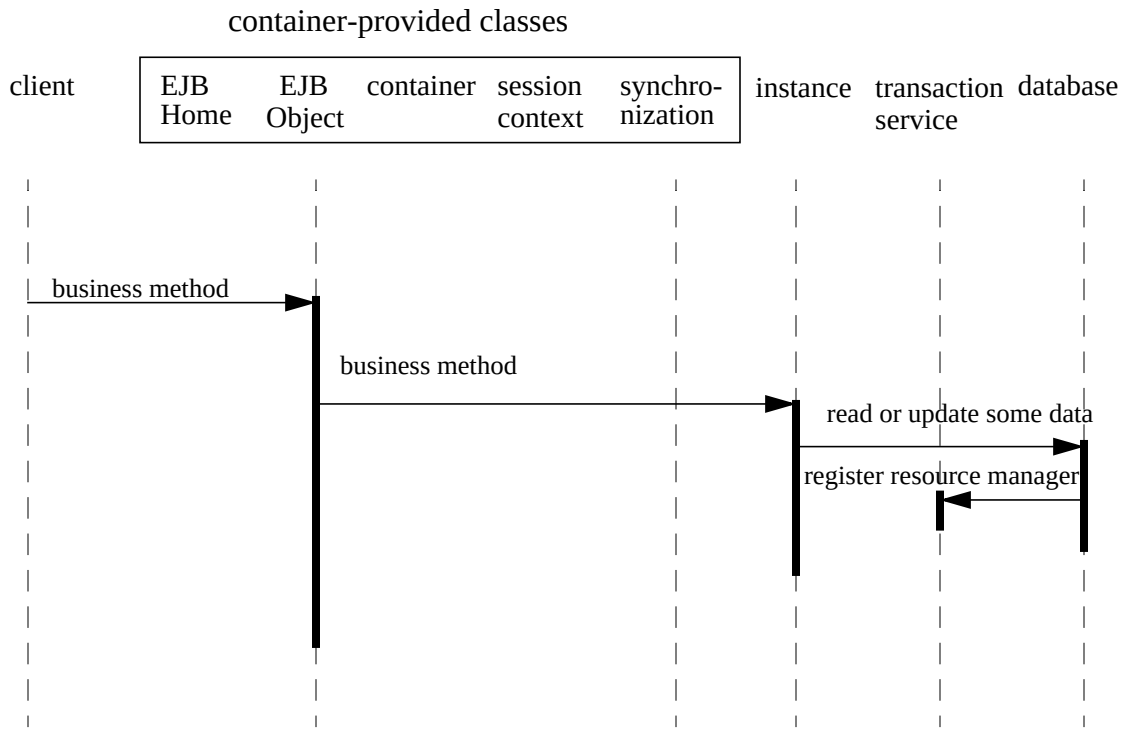
Figure 12 OID for creation of a STATELESS session object



6.9.2 Business method invocation

The following diagram illustrates the invocation of a business method.

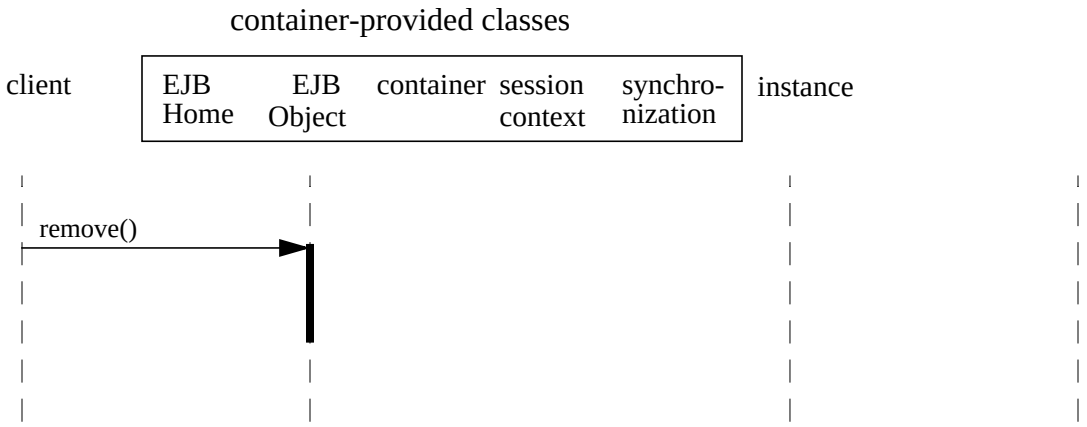
Figure 13 OID for invocation of business method on a STATELESS session object



6.9.3 Client-invoked *remove()*

The following diagram illustrates the destruction of a stateless session object.

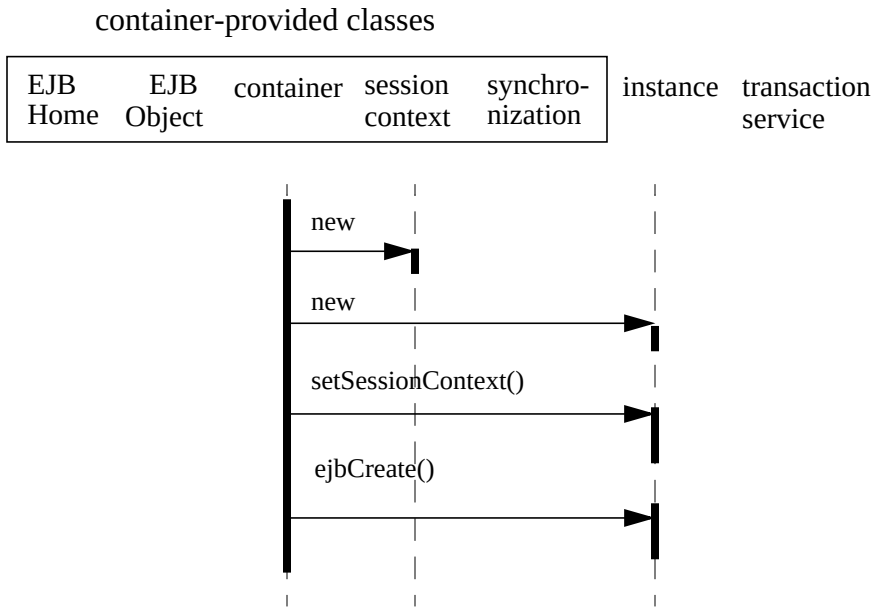
Figure 14 OID for removal of a STATELESS session object



6.9.4 Adding instance to the pool

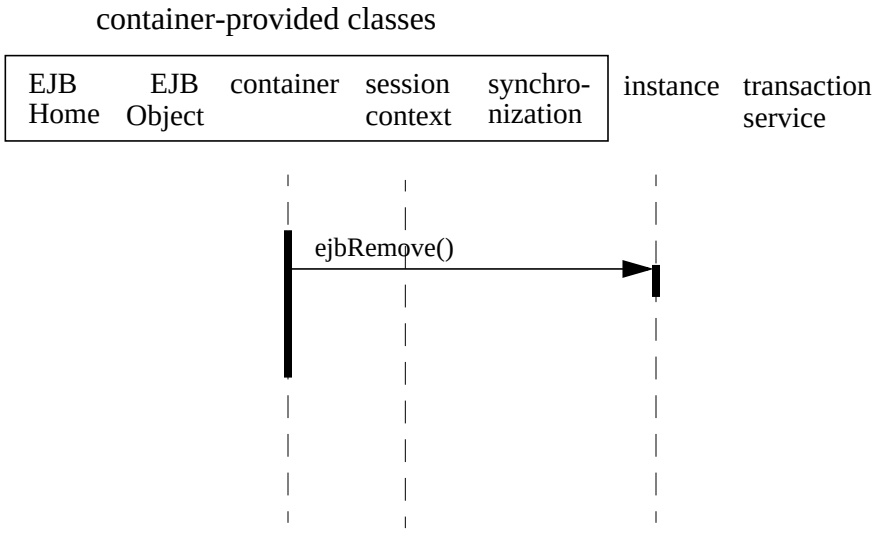
The following diagram illustrates the sequence for a container adding an instance to the method-ready pool.

Figure 15 OID for Container Adding Instance of a STATELESS session bean to a method-ready pool



The following diagram illustrates the sequence for a container removing an instance from the method-ready pool.

Figure 16 OID for a Container Removing an Instance of a STATELESS Session bean from ready pool



6.10 The responsibilities of the bean provider

This section describes the responsibilities of session bean provider to ensure that a session bean can be deployed in any EJB Container.

6.10.1 Classes and interfaces

The session bean provider is responsible for providing the following class files:

- Session bean class.
- Session bean's remote interface.
- Session bean's home interface.

6.10.2 Session bean class

The following are the requirements for session bean class:

- The class must implement, directly or indirectly, the `javax.ejb.SessionBean` interface.

- 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 may, but is not required to, implement the session bean's remote interface[5].
- The class must implement the business methods and the `ejbCreate` methods.
- If the class is a stateful session bean, it may optionally implement the `javax.ejb.SessionSynchronization` interface.
- The session bean class may have superclasses and/or superinterfaces. If the session bean has superclasses, then the business methods, the `ejbCreate<METHOD>` methods, the methods of the `SessionBean` interface, and the methods of the optional `SessionSynchronization` interface may be defined in the session bean class, or in any of its superclasses.
- The session bean class is allowed to implement other methods (for example helper methods invoked internally by the business methods) in addition to the methods required by the EJB specification.

6.10.3 *ejbCreate<METHOD>* methods

The session bean class must define one or more `ejbCreate<METHOD>(...)` methods whose signatures must follow these rules:

- The method name must have `ejbCreate` as its prefix.
- The method must be declared as `public`.
- The method must not be declared as `final` or `static`.
- The return type must be `void`.
- The method arguments must be legal types for RMI/IIOP.
- The throws clause may define arbitrary application exceptions, possibly including the `javax.ejb.CreateException`.

Compatibility Note: EJB 1.0 allowed the ejbCreate method to throw the `java.rmi.RemoteException` to indicate a non-application exception. This practice was deprecated in EJB 1.1—an EJB 1.1 or EJB 2.0 compliant enterprise bean should throw the javax.ejb.EJBException or another RuntimeException to indicate non-application exceptions to the Container (see Section 17.2.2).

[5] If the session bean class does implement the remote interface, care must be taken to avoid passing of `this` as a method argument or result. This potential error can be avoided by choosing not to implement the remote interface in the session bean class.

6.10.4 Business methods

The session bean class may define zero or more business methods whose signatures must follow these rules:

- The method names can be arbitrary, but they must not start with "ejb" to avoid conflicts with the callback methods used by the EJB architecture.
- The business method must be declared as `public`.
- The method must not be declared as `final` or `static`.
- The argument and return value types for a method must be legal types for RMI/IIOP.
- The throws clause may define arbitrary application exceptions.

Compatibility Note: EJB 1.0 allowed the business methods to throw the `java.rmi.RemoteException` to indicate a non-application exception. This practice was deprecated in EJB 1.1—an EJB 1.1 or EJB 2.0 compliant enterprise bean should throw the javax.ejb.EJBException or another RuntimeException to indicate non-application exceptions to the Container (see Section 17.2.2).

6.10.5 Session bean's remote interface

The following are the requirements for the session bean's remote interface:

- The interface must extend the `javax.ejb.EJBObject` interface.
- The methods defined in this interface must follow the rules for RMI/IIOP. This means that their argument and return values must be of valid types for RMI/IIOP, and their throws clauses must include the `java.rmi.RemoteException`.
- The remote interface is allowed to have superinterfaces. Use of interface inheritance is subject to the RMI/IIOP rules for the definition of remote interfaces.
- For each method defined in the remote interface, there must be a matching method in the session bean's class. The matching method must have:
  - The same name.
  - The same number and types of arguments, and the same return type.
  - All the exceptions defined in the throws clause of the matching method of the session bean class must be defined in the throws clause of the method of the remote interface.

6.10.6 Session bean's home interface

The following are the requirements for the session bean's home interface:

- The interface must extend the `javax.ejb.EJBHome` interface.

- The methods defined in this interface must follow the rules for RMI/IIOP. This means that their argument and return values must be of valid types for RMI/IIOP, and that their throws clauses must include the `java.rmi.RemoteException`.

- The home interface is allowed to have superinterfaces. Use of interface inheritance is subject to the RMI/IIOP rules for the definition of remote interfaces.

- A session bean's home interface must define one or more `create<METHOD>(...)` methods. A stateless session bean must define exactly one `create()` method with no arguments.

- Each `create` method must be named "**create**<**METHOD**>", and it must match one of the `ejbCreate<METHOD>` methods defined in the session bean class. The matching `ejbCreate<METHOD>` method must have the same number and types of arguments. (Note that the return type is different.) The methods for a stateless session bean must be named "**create**" and "**ejbCreate**".

- The return type for a `create<METHOD>` method must be the session bean's remote interface type.

- All the exceptions defined in the throws clause of an `ejbCreate<METHOD>` method of the session bean class must be defined in the throws clause of the matching `create<METHOD>` method of the home interface.

- The throws clause must include `javax.ejb.CreateException`.

6.11 The responsibilities of the container provider

This section describes the responsibilities of the container provider to support a session bean. The container provider is responsible for providing the deployment tools and for managing the session bean instances at runtime.

Because the EJB specification does not define the API between deployment tools and the container, we assume that the deployment tools are provided by the container provider. Alternatively, the deployment tools may be provided by a different vendor who uses the container vendor's specific API.

6.11.1 Generation of implementation classes

The deployment tools provided by the container are responsible for the generation of additional classes when the session bean is deployed. The tools obtain the information that they need for generation of the additional classes by introspecting the classes and interfaces provided by the enterprise bean provider and by examining the session bean's deployment descriptor.

The deployment tools must generate the following classes:

- A class that implements the session bean's home interface (session EJBHome class).

- A class that implements the session bean's remote interface (session EJBObject class).

The deployment tools may also generate a class that mixes some container-specific code with the session bean class. This code may, for example, help the container to manage the bean instances at runtime. The tools can use subclassing, delegation, and code generation.

The deployment tools may also allow the generation of additional code that wraps the business methods and is used to customize the business logic to an existing operational environment. For example, a wrapper for a `debit` function on the `AccountManager` bean may check that the debited amount does not exceed a certain limit.

6.11.2 Session EJBHome class

The session EJBHome class, which is generated by the deployment tools, implements the session bean's home interface. This class implements the methods of the `javax.ejb.EJBHome` interface and the `create<METHOD>` methods specific to the session bean.

The implementation of each `create<METHOD>(...)` method invokes a matching `ejbCreate<METHOD>(...)` method.

6.11.3 Session EJBObject class

The Session EJBObject class, which is generated by the deployment tools, implements the session bean's remote interface. It implements the methods of the `javax.ejb.EJBObject` interface and the business methods specific to the session bean.

The implementation of each business method must activate the instance (if the instance is in the passive state) and invoke the matching business method on the instance.

6.11.4 Handle classes

The deployment tools are responsible for implementing the handle classes for the session bean's home and remote interfaces.

6.11.5 EJBMetaData class

The deployment tools are responsible for implementing the class that provides meta-data to the client view contract. The class must be a valid RMI Value class and must implement the `javax.ejb.EJBMetaData` interface.

6.11.6 Non-reentrant instances

The container must ensure that only one thread can be executing an instance at any time. If a client request arrives for an instance while the instance is executing another request, the container must throw the `java.rmi.RemoteException` to the second request.

Note that a session object is intended to support only a single client. Therefore, it would be an application error if two clients attempted to invoke the same session object.

One implication of this rule is that an application cannot make loopback calls to a session bean instance.

6.11.7 Transaction scoping, security, exceptions

The container must follow the rules with respect to transaction scoping, security checking, and exception handling, as described in Chapters 16, 20, and 17, respectively.

6.11.8 SessionContext

The container must implement the `SessionContext.getEJBContext()` method such that the bean instance can use the Java language cast to convert the returned value to the session bean's remote interface type. Specifically, the bean instance does not have to use the `PortableRemoteObject.narrow(...)` method for the type conversion.

Chapter 7 Example Session Scenario

This chapter describes an example development and deployment scenario of a session bean. We use the scenario to explain the responsibilities of the bean provider and those of the container provider.

The classes generated by the container provider's tools in this scenario should be considered illustrative rather than prescriptive. Container providers are free to implement the contract between a session bean and its container in a different way, provided that it achieves an equivalent effect (from the perspectives of the bean provider and the client-side programmer).

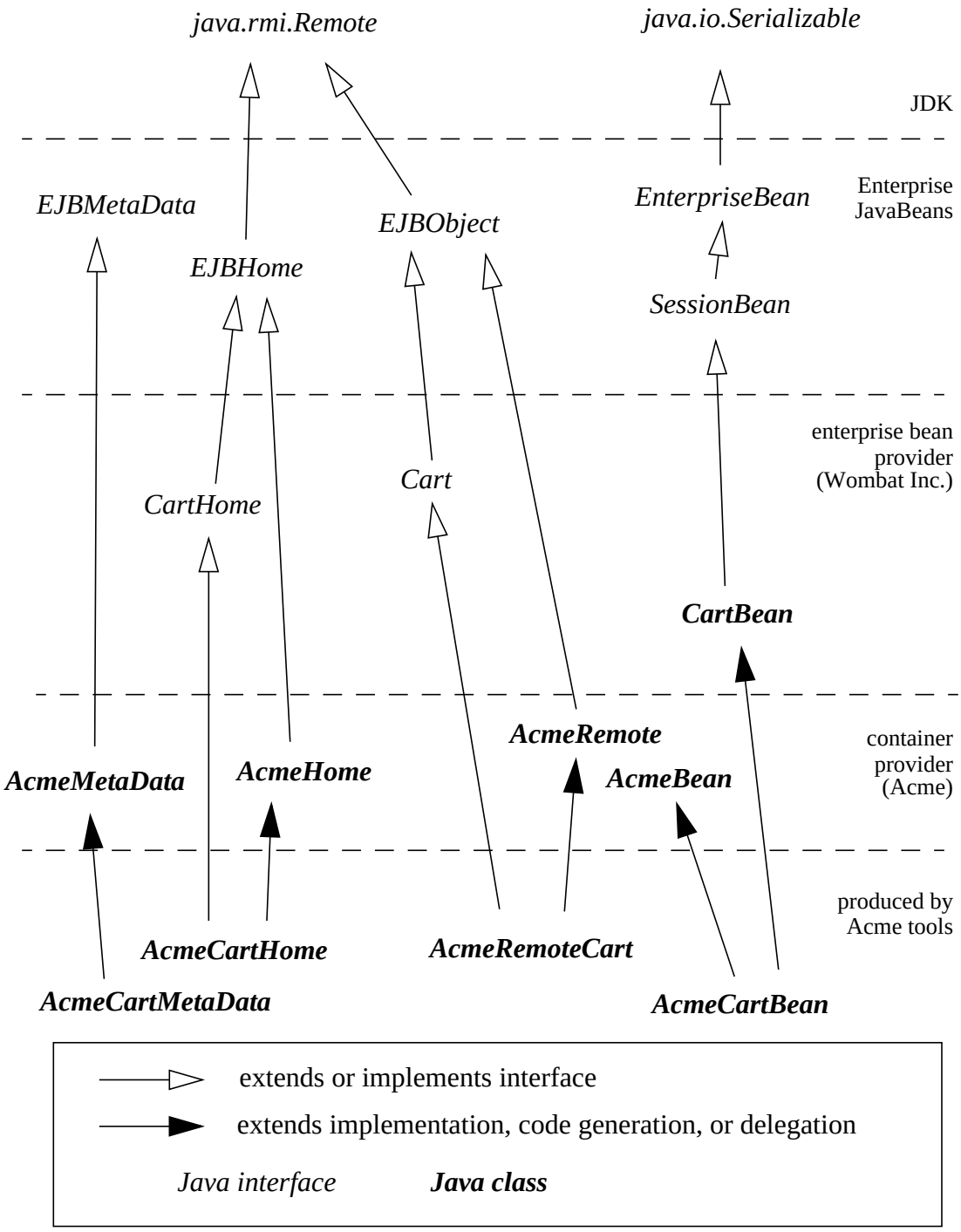
7.1 Overview

Wombat Inc. has developed the `CartBean` *session Bean. The CartBean is deployed in a container provided by the Acme Corporation.*

7.2 Inheritance relationship

An example of the inheritance relationship between the interfaces and classes is illustrated in the following diagram:

Figure 17 Example of Inheritance Relationships Between EJB Classes



7.2.1 What the session Bean provider is responsible for

Wombat Inc. is responsible for providing the following:

- *Define the session Bean's remote interface (Cart). The remote interface defines the business methods callable by a client. The remote interface must extend the javax.ejb.EJBObject interface, and follow the standard rules for a RMI-IIOP remote interface. The remote interface must be defined as public.*

- *Write the business logic in the session Bean class (CartBean). The enterprise Bean class may, but is not required to, implement the enterprise Bean's remote interface (Cart). The enterprise Bean must implement the javax.ejb.SessionBean interface, and define the ejbCreate<METHOD>(...) method(s) invoked at EJB object creation.*

- *Define a home interface (CartHome) for the enterprise Bean. The home interface must be defined as public, extend the javax.ejb.EJBHome interface, and follow the standard rules for RMI-IIOP remote interfaces.*

- *Define a deployment descriptor specifying any declarative metadata that the session Bean provider wishes to pass with the Bean to the next stage of the development/deployment workflow.*

7.2.2 Classes supplied by container provider

The following classes are supplied by the container provider Acme Corp:

The AcmeHome class provides the Acme implementation of the javax.ejb.EJBHome methods.

The AcmeRemote class provides the Acme implementation of the javax.ejb.EJBObject methods.

The AcmeBean class provides additional state and methods to allow Acme's container to manage its session Bean instances. For example, if Acme's container uses an LRU algorithm, then AcmeBean may include the clock count and methods to use it.

The AcmeMetaData class provides the Acme implementation of the javax.ejb.EJBMetaData methods.

7.2.3 What the container provider is responsible for

The tools provided by Acme Corporation are responsible for the following:

- *Generate the class (AcmeRemoteCart) that implements the session bean's remote interface. The tools also generate the classes that implement the communication protocol specific artifacts for the remote interface.*

- *Generate the implementation of the session Bean class suitable for the Acme container (AcmeCartBean). AcmeCartBean includes the business logic from the CartBean class mixed with the services defined in the AcmeBean class. Acme tools can use inheritance, delegation, and code generation to achieve a mix-in of the two classes.*

- *Generate the class (AcmeCartHome) that implements the session bean's home interface. The tools also generate the classes that implement the communication protocol specific artifacts for the home interface.*

- *Generate the class (AcmeCartMetaData) that implements the javax.ejb.EJBMetaData interface for the Cart Bean.*

Many of the above classes and tools are container-specific (i.e., they reflect the way Acme Corp implemented them). Other container providers may use different mechanisms to produce their runtime classes, and these classes will likely be different from those generated by Acme's tools.

Chapter 8 Client View of an Entity

This chapter describes the client view of an entity bean. It is actually a contract fulfilled by the Container in which the entity bean is deployed. Only the business methods are supplied by the enterprise bean itself.

Although the client view of the deployed entity beans is provided by classes implemented by the container, the container itself is transparent to the client.

8.1 Overview

For a client, an entity bean is a component that represents an object-oriented view of some entities stored in a persistent storage, such as a database, or entities that are implemented by an existing enterprise application.

A client accesses an entity bean through the entity bean's remote and home interfaces. The container provides classes that implement the entity bean's remote and home interfaces. The objects that implement the home and remote objects are remote Java objects, and are accessible from a client through the standard Java™ APIs for remote object invocation [3].

From its creation until its destruction, an entity object lives in a container. Transparently to the client, the container provides security, concurrency, transactions, persistence, and other services for the entity objects that live in the container. The container is transparent to the client—there is no API that a client can use to manipulate the container.

Multiple clients can access an entity object concurrently. The container in which the entity bean is deployed properly synchronizes access to the entity object's state using transactions.

Each entity object has an identity which, in general, survives a crash and restart of the container in which the entity object has been created. The object identity is implemented by the container with the cooperation of the enterprise bean class.

The client view of an entity bean is location independent. A client running in the same JVM as an entity bean instance uses the same API to access the entity bean as a client running in a different JVM on the same or different machine.

A client of an entity object can be another enterprise bean deployed in the same or different Container; or a client can be an arbitrary Java program, such as an application, applet, or servlet. The client view of an entity bean can also be mapped to non-Java client environments, such as CORBA clients not written in the Java programming language.

Multiple enterprise beans can be deployed in a container. For each entity bean deployed in a container, the container provides a class that implements the entity bean's **home interface.** The home interface allows the client to create, find, and remove entity objects within the enterprise bean's home as well as execute home business methods, which are not specific to a particular entity bean object. A client can look up the entity bean's home interface through JNDI; it is the responsibility of the container to make the entity bean's home interface available in the JNDI name space.

A client view of an entity bean is the same, irrespective of the implementation of the entity bean and its container. This ensures that a client application is portable across all container implementations in which the entity bean might be deployed.

8.2 EJB Container

An EJB Container (Container for short) is a system that functions as a runtime container for enterprise beans.

Multiple enterprise beans can be deployed in a single container. For each entity bean deployed in a container, the container provides a **home interface** that allows the client to create, find, and remove entity objects that belong to the entity bean. The home interface may also provide home business methods, which are not specific to a particular entity bean object. The container makes the entity beans' home interfaces (defined by the bean provider and implemented by the container provider) available in the JNDI name space for clients.

An EJB Server may host one or multiple EJB Containers. The containers are transparent to the client: there is no client API to manipulate the container, and there is no way for a client to tell in which container an enterprise bean is installed.

8.2.1 Locating an entity bean's home interface

A client locates an entity bean's home interface using JNDI. For example, the home interface for the Account entity bean can be located using the following code segment:

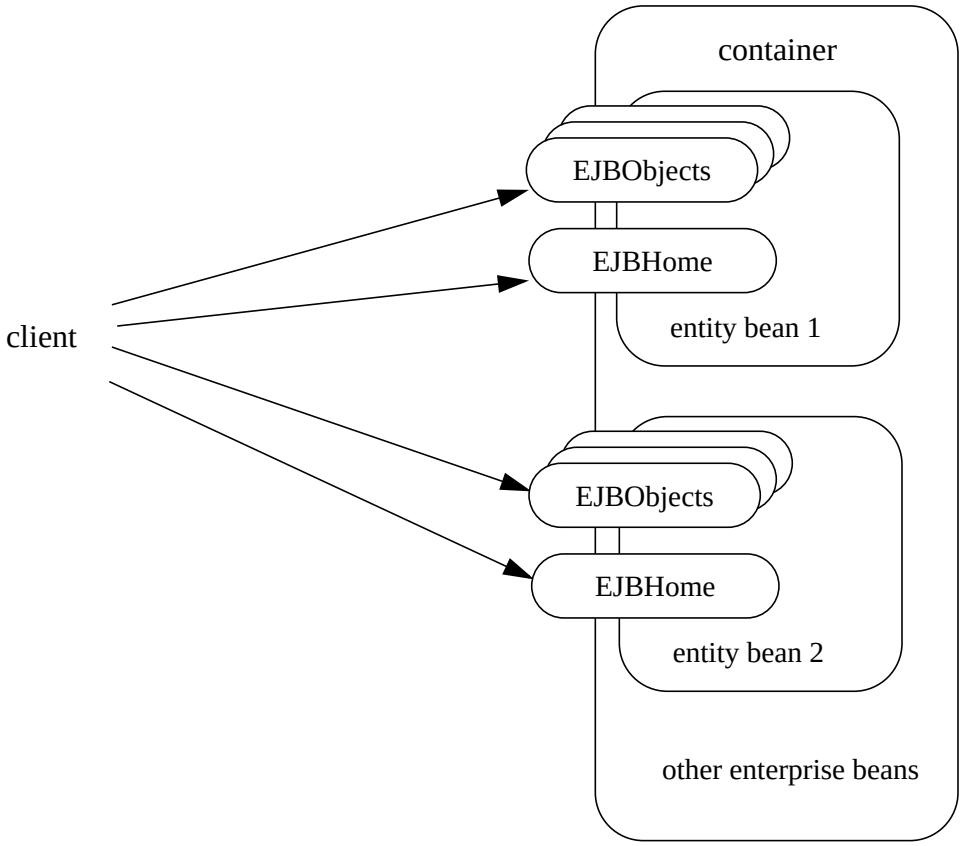
```
Context initialContext = new InitialContext();
AccountHome accountHome = (AccountHome)
         javax.rmi.PortableRemoteObject.narrow(
              initialContext.lookup("java:comp/env/ejb/accounts"),
                                   AccountHome.class);
```

A client's JNDI name space may be configured to include the home interfaces of enterprise beans deployed in multiple EJB Containers located on multiple machines on a network. The actual location of an EJB Container is, in general, transparent to the client.

8.2.2 What a container provides

The following diagram illustrates the view that a container provides to the clients of the entity beans deployed in the container.

Figure 18 Client view of entity beans deployed in a container



8.3 Entity bean's home interface

The container provides the implementation of the home interface for each entity bean deployed in the container. The container makes the home interface of every entity bean deployed in the container accessible to the clients through JNDI. An object that implements an entity bean's home interface is called an **EJBHome** object.

The entity bean's home interface allows a client to do the following:

- Create new entity objects within the home.
- Find existing entity objects within the home.
- Remove an entity object from the home.

- Execute a home business method.
- Get the javax.ejb.EJBMetaData interface for the entity bean. The javax.ejb.EJBMetaData interface is intended to allow application assembly tools to discover the meta-data information about the entity bean. The meta-data information allows loose client/server binding and scripting.
- Obtain a handle for the home interface. The home handle can be serialized and written to stable storage; later, possibly in a different JVM, the handle can be deserialized from stable storage and used to obtain a reference to the home interface.

An entity bean's home interface must extend the `javax.ejb.EJBHome` interface and follow the standard rules for Java programming language remote interfaces.

8.3.1 *create* methods

An entity bean's home interface can define zero or more `create<METHOD>(...)` methods, one for each way to create an entity object. The arguments of the `create` methods are typically used to initialize the state of the created entity object. The name of each create method starts with the prefix "**create**".

The return type of a `create<METHOD>` method is the entity bean's remote interface.

The throws clause of every `create<METHOD>`method includes the `java.rmi.RemoteException` and the `javax.ejb.CreateException`. It may include additional application-level exceptions.

The following home interface illustrates three possible `create` methods:

```
public interface AccountHome extends javax.ejb.EJBHome {
     public Account create(String firstName, String lastName,
          double initialBalance)
           throws RemoteException, CreateException;
     public Account create(String accountNumber,
          double initialBalance)
           throws RemoteException, CreateException,
              LowInitialBalanceException;
     public Account createLargeAccount(String firstname,
          String lastname, double initialBalance)
           throws RemoteException, CreateException;
      ...
}
```

The following example illustrates how a client creates a new entity object:

```
AccountHome accountHome = ...;
Account account = accountHome.create("John", "Smith", 500.00);
```

8.3.2 *finder* methods

An entity bean's home interface defines one or more `finder` methods[6], one for each way to find an entity object or collection of entity objects within the home. The name of each finder method starts with the prefix "**find**", such as `findLargeAccounts(...)`. The arguments of a finder method are used by the entity bean implementation to locate the requested entity objects. The return type of a finder method must be the entity bean's remote interface, or a type representing a collection of objects that implement the entity bean's remote interface (see Subsections 9.6.6 and 11.1.8).

The throws clause of every finder method includes the `java.rmi.RemoteException` and the `javax.ejb.FinderException`.

The home interface of every entity bean includes the `findByPrimaryKey(primaryKey)` method, which allows a client to locate an entity object using a primary key. The name of the method is always `findByPrimaryKey`; it has a single argument that is the same type as the entity bean's primary key type, and its return type is the entity bean's remote interface. There is a unique `findByPrimaryKey(primaryKey)` method for an entity bean; this method must not be overloaded. The implementation of the `findByPrimaryKey(primaryKey)` method must ensure that the entity object exists.

The following example shows the `findByPrimaryKey` method:

```
public interface AccountHome extends javax.ejb.EJBHome {
     ...
     public Account findByPrimaryKey(String AccountNumber)
          throws RemoteException, FinderException;
}
```

The following example illustrates how a client uses the `findByPrimaryKey` method:

```
AccountHome = ...;
Account account = accountHome.findByPrimaryKey("100-3450-3333");
```

8.3.3 *remove* methods

The `javax.ejb.EJBHome` interface defines several methods that allow the client to remove an entity object.

```
public interface EJBHome extends Remote {
     void remove(Handle handle) throws RemoteException,
          RemoveException;
     void remove(Object primaryKey) throws RemoteException,
          RemoveException;
}
```

[6] The *findByPrimaryKey(primaryKey)* method is mandatory for all Entity Beans.

After an entity object has been removed, subsequent attempts to access the entity object by a client result in the `java.rmi.NoSuchObjectException`.

8.3.4 *home* methods

An entity bean's home interface may define one or more home methods. Home methods are methods that the bean provider supplies for business logic that is not specific to an entity bean instance.

Home methods can have arbitrary method names, but they must not start with "**create**", "**find**", or "**remove**". The arguments of a home method are used by the entity bean implementation in computations that do not depend on a specific entity bean instance. The method arguments and return value types must be legal types for RMI-IIOP.

The throws clause of every home method includes the `java.rmi.RemoteException`. It may also include additional application-level exceptions.

The following example shows two home methods:

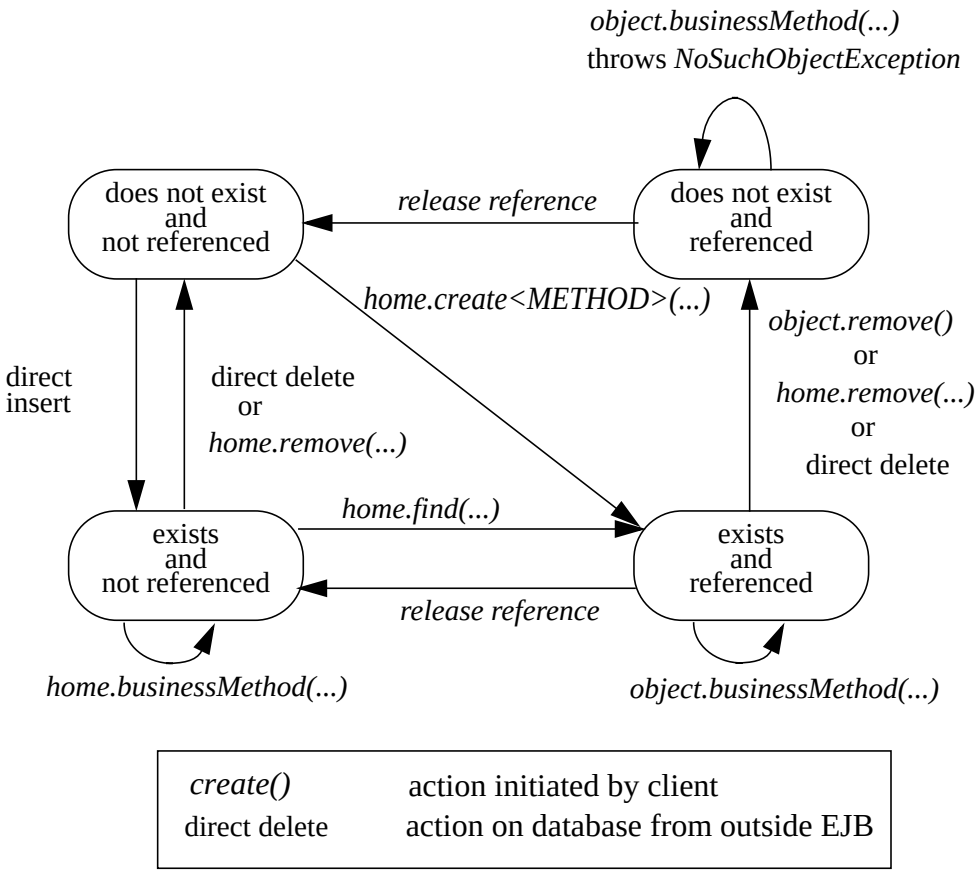
```
public interface EmployeeHome extends javax.ejb.EJBHome {
     ...
     // this method returns a living index depending on
     // the state and the base salary of an employee:
     // the method is not specific to an instance
     public float livingIndex(String state, float Salary)
          throws RemoteException;

     // this method adds a bonus to all of the employees
     // based on a company profit sharing index
     public void addBonus(float company_share_index)
          throws RemoteException, ShareIndexOutOfRangeException;

     ...
}
```

8.4 Entity object's life cycle

This section describes the life cycle of an entity object from the perspective of a client.

The following diagram illustrates a client's point of view of an entity object life cycle. (The term **referenced** in the diagram means that the client program has a reference to the entity object's remote interface.)

Figure 19 Client View of Entity Object Life Cycle



An entity object does not exist until it is created. Until it is created, it has no identity. After it is created, it has identity. A client creates an entity object using the entity bean's home interface whose class is implemented by the container. When a client creates an entity object, the client obtains a reference to the newly created entity object.

In an environment with legacy data, entity objects may "exist" before the container and entity bean are deployed. In addition, an entity object may be "created" in the environment via a mechanism other than by invoking a `create<METHOD>(...)` method of the home interface (e.g. by inserting a database record), but still may be accessible by a container's clients via the finder methods. Also, an entity object may be deleted directly using other means than the `remove()` operation (e.g. by deletion of a database record). The "direct insert" and "direct delete" transitions in the diagram represent such direct database manipulation.

A client can get a reference to an existing entity object's remote interface in any of the following ways:

- Receive the reference as a parameter in a method call (input parameter or result).
- Find the entity object using a finder method defined in the entity bean's home interface.
- Obtain the reference from the entity object's handle. (see Section 8.7)

A client that has a reference to an entity object's remote interface can do any of the following:

- Invoke business methods on the entity object through the remote interface.
- Obtain a reference to the enterprise Bean's home interface.
- Pass the reference as a parameter or return value of a remote method call.
- Obtain the entity object's primary key.
- Obtain the entity object's handle.
- Remove the entity object.

All references to an entity object that does not exist are invalid. All attempted invocations on an entity object that does not exist result in an `java.rmi.NoSuchObjectException` being thrown.

All entity objects are considered **persistent objects**. The lifetime of an entity object is not limited by the lifetime of the Java Virtual Machine process in which the entity bean instance executes. While a crash of the Java Virtual Machine may result in a rollback of current transactions, it does not destroy previously created entity objects nor invalidate the references to the remote and home interfaces held by clients.

Multiple clients can access the same entity object concurrently. Transactions are used to isolate the clients' work from each other.

8.5 Primary key and object identity

Every entity object has a unique identity within its home. If two entity objects have the same home and the same primary key, they are considered identical.

The Enterprise JavaBeans architecture allows a primary key class to be any class that is a legal Value Type in RMI-IIOP, subject to the restrictions defined in Subsections 9.7.12 and 11.2.10. The primary key class may be specific to an entity Bean class (i.e., each entity bean class may define a different class for its primary key, but it is possible that multiple entity beans use the same primary key class).

A client that holds a reference to an entity object's remote interface can determine the entity object's identity within its home by invoking the `getPrimaryKey()` method on the reference. The object identity associated with a reference does not change over the lifetime of the reference. (That is, `getPrimaryKey()` always returns the same value when called on the same entity object reference.)

A client can test whether two entity object references refer to the same entity object by using the `isIdentical(EJBObject)` method. Alternatively, if a client obtains two entity object references from the same home, it can determine if they refer to the same entity by comparing their primary keys using the `equals` method.

The following code illustrates using the `isIdentical` method to test if two object references refer to the same entity object:

```
Account acc1 = ...;
Account acc2 = ...;

if (acc1.isIdentical(acc2)) {
    acc1 and acc2 are the same entity object
} else {
    acc2 and acc2 are different entity objects
}
```

A client that knows the primary key of an entity object can obtain a reference to the entity object by invoking the `findByPrimaryKey(key)` method on the entity bean's home interface.

Note that the Enterprise JavaBeans architecture does not specify "object equality" (i.e. use of the `==` operator) for entity object references. The result of comparing two object references using the Java programming language `Object.equals(Object obj)` method is unspecified. Performing the `Object.hashCode()` method on two object references that represent the entity object is not guaranteed to yield the same result. Therefore, a client should always use the `isIdentical` method to determine if two entity object references refer to the same entity object.

8.6 Entity Bean's remote interface

A client accesses an entity object through the entity bean's remote interface. An entity bean's remote interface must extend the `javax.ejb.EJBObject` interface. A remote interface defines the business methods that are callable by clients.

The following example illustrates the definition of an entity bean's remote interface:

```
public interface Account extends javax.ejb.EJBObject {
    void debit(double amount)
        throws java.rmi.RemoteException,
            InsufficientBalanceException;
    void credit(double amount)
        throws java.rmi.RemoteException;
    double getBalance()
        throws java.rmi.RemoteException;
}
```

The `javax.ejb.EJBObject` interface defines the methods that allow the client to perform the following operations on an entity object's reference:

- Obtain the home interface for the entity object.

- Remove the entity object.
- Obtain the entity object's handle.
- Obtain the entity object's primary key.

The container provides the implementation of the methods defined in the `javax.ejb.EJBObject` interface. Only the business methods are delegated to the instances of the enterprise bean class.

Note that the entity object does not expose the methods of the `javax.ejb.EnterpriseBean` interface to the client. These methods are not intended for the client—they are used by the container to manage the enterprise bean instances.

8.7 Entity bean's handle

An entity object's handle is an object that identifies the entity object on a network. A client that has a reference to an entity object's remote interface can obtain the entity object's handle by invoking the `getHandle()` method on the remote interface.

Since a handle class extends `java.io.Serializable`, a client may serialize the handle. The client may use the serialized handle later, possibly in a different process or even system, to re-obtain a reference to the entity object identified by the handle.

The client code must use the `javax.rmi.PortableRemoteObject.narrow(...)`method to convert the result of the `getEJBObject()` method invoked on a handle to the entity bean's remote interface type.

The lifetime and scope of a handle is specific to the handle implementation. At the minimum, a program running in one JVM must be able to obtain and serialize the handle, and another program running in a different JVM must be able to deserialize it and re-create an object reference. An entity handle is typically implemented to be usable over a long period of time—it must be usable at least across a server restart.

Containers that store long-lived entities will typically provide handle implementations that allow clients to store a handle for a long time (possibly many years). Such a handle will be usable even if parts of the technology used by the container (e.g. ORB, DBMS, server) have been upgraded or replaced while the client has stored the handle. Support for this "quality of service" is not required by the EJB specification.

An EJB Container is not required to accept a handle that was generated by another vendor's EJB Container.

The use of a handle is illustrated by the following example:

```
// A client obtains a handle of an account entity object and
// stores the handle in stable storage.
//
ObjectOutputStream stream = ...;
Account account = ...;
Handle handle = account.getHandle();
stream.writeObject(handle);

// A client can read the handle from stable storage, and use the
// handle to resurrect an object reference to the
// account entity object.
//
ObjectInputStream stream = ...;
Handle handle = (Handle) stream.readObject(handle);
Account account = (Account)javax.rmi.PortableRemoteObject.narrow(
         handle.getEJBObject(), Account.class);
account.debit(100.00);
```

A handle is not a capability, in the security sense, that would automatically grant its holder the right to invoke methods on the object. When a reference to an object is obtained from a handle, and then a method on the object is invoked, the container performs the usual access checks based on the caller's principal.

8.8 Entity home handles

The EJB specification allows the client to obtain a handle for the home interface. The client can use the home handle to store a reference to an entity bean's home interface in stable storage, and re-create the reference later. This handle functionality may be useful to a client that needs to use the home interface in the future, but does not know the JNDI name of the home interface.

A handle to a home interface must implement the `javax.ejb.HomeHandle` interface.

The client code must use the `javax.rmi.PortableRemoteObject.narrow(...)`method to convert the result of the `getEJBHome()` method invoked on a handle to the home interface type.

The lifetime and scope of a handle is specific to the handle implementation. At the minimum, a program running in one JVM must be able to serialize the handle, and another program running in a different JVM must be able to deserialize it and re-create an object reference. An entity handle is typically implemented to be usable over a long period of time—it must be usable at least across a server restart.

8.9 Type narrowing of object references

A client program that is intended to be interoperable with all compliant EJB Container implementations must use the `javax.rmi.PortableRemoteObject.narrow(...)` method to perform type-narrowing of the client-side representations of the home and remote interface.

Note: Programs that use the cast operator to narrow the remote and home interfaces are likely to fail if the Container implementation uses RMI-IIOP as the underlying communication transport.

Chapter 9

Entity Bean Component Contract for Container Managed Persistence

The entity bean component contract for container managed persistence is the contract between an entity bean, its container, and its persistence manager. It defines the life cycle of the entity bean instances, the model for method delegation of the client-invoked business methods, and the model for the management of the entity bean's persistent state and relationships. The main goal of this contract is to ensure that an entity bean component using container managed persistence is portable across all compliant EJB Containers.

This chapter defines the enterprise Bean Provider's view of this contract and responsibilities of the Container Provider and Persistence Manager Provider for managing the life cycle of the enterprise bean instances and their persistent state and relationships.

9.1 Overview

In accordance with the architecture for container managed persistence, the Bean Provider develops a set of beans and dependent object classes for an application and determines the relationships among them. For each bean, the bean provider specifies an abstract persistence schema that defines the methods for accessing the bean's container-managed fields and relationships. The bean provider likewise specifies an abstract persistence schema for each dependent object class that is related to the bean. The bean or dependent object class accesses these fields and relationships at runtime by means of the methods defined for its abstract persistence schema.

The persistent fields and relationships of the abstract persistence schema are specified in the deployment descriptor that is produced by the bean provider. The deployer, using the persistence manager provider's tools, determines how the persistent fields and relationships are mapped to a database or other persistent store, and generates the necessary additional classes and interfaces that enable the persistence manager to manage the persistent fields and relationships of the beans and dependent objects at runtime. The persistence for these fields and relationships is provided by the persistence manager at runtime.

> *The entity bean component contract for container managed persistence has been substantially changed in the EJB 2.0 specification. Entity beans that use the EJB 1.1 component contract for container managed persistence must still be supported in EJB 2.0 containers. However, the contracts are separate, and the bean provider must choose one or the other. The EJB 1.1 entity bean contract for container managed persistence is defined in Chapter 13 "EJB 1.1 Entity Bean Component Contract for Container Managed Persistence" .*

9.2 Data independence between the Client View, the Entity Bean View, and the Persistence View

When designing an entity bean with container managed persistence, the Bean Provider must be mindful of the distinction between the client view of the entity bean and the entity bean's view of its persistent state. In particular, there need be no direct relationship between the two. While the EJB component model provides a separation between the client view of a bean (as presented by its home and remote interfaces) and the entity bean instance (which provides the implementation of the client view), the EJB architecture for container managed persistence adds to this a separation between the entity bean instance (as defined by the bean provider) and its persistent state. The container managed persistence architecture thus provides not only a layer of data independence between the client view of a bean and the bean instance, but also between the bean instance and its persistent representation. This allows an entity bean to be evolved independently from its clients, without requiring the redefinition or recompilation of those clients, and it allows an entity bean to be redeployed across different persistence managers and different persistent data stores, without requiring the redefinition or recompilation of the entity bean class.

Chapter 8 describes the Client View of an Entity Bean. This view is no different for an entity bean with container managed persistence than for an entity bean with bean managed persistence.

This chapter describes the component contract for an entity bean with container managed persistence, and how data independence is maintained between the entity bean instance and its persistent representation. It describes this contract from three viewpoints: from the viewpoint of the entity bean provider, the persistence manager, and the container. It also describes the responsibilities of the bean provider in maintaining the independence of the client view of the entity bean from its persistence view.

The component contract for bean managed persistence is described in Chapter 11.

9.3 Container-managed entity persistence

An entity bean implements an object view of a business entity or set of business entities stored in an underlying database or implemented by an existing enterprise application (for example, by a mainframe program or by an ERP application). The data access protocol for transferring the state of the entity between the entity bean instances and the underlying database or application is referred to as **persistence**.

In container-managed persistence, unlike in bean-managed persistence, the Bean Provider does not write database access calls in the entity bean. Instead, persistence is handled by a Persistence Manager that is available to the Container at runtime. The entity Bean Provider must specify in the deployment descriptor those persistent fields and relationships for which the Persistence Manager Provider's tools must generate data access calls. The Persistence Manager Provider's tools are then used at the entity bean's deployment time to generate the necessary database access calls. The deployment descriptor for the entity bean indicates that the entity bean uses container-managed persistence. The Bean Provider of an entity bean with container-manager persistence must code all persistent data access by using the accessor methods that are defined in the bean's abstract persistence schema. The implementation of the persistent fields and relationships, as well as all data access, is deferred to the Persistence Manager.

A dependent object class is a persistent helper class that is used in container managed relationships by one or more entity beans to further model their persistent state. A dependent object is used internally by the entity bean(s) and is itself not directly exposed through the client view. The Bean Provider specifies in the deployment descriptor those persistent fields and relationships of the dependent object class for which the persistence manager provider's tools must generate data access calls.

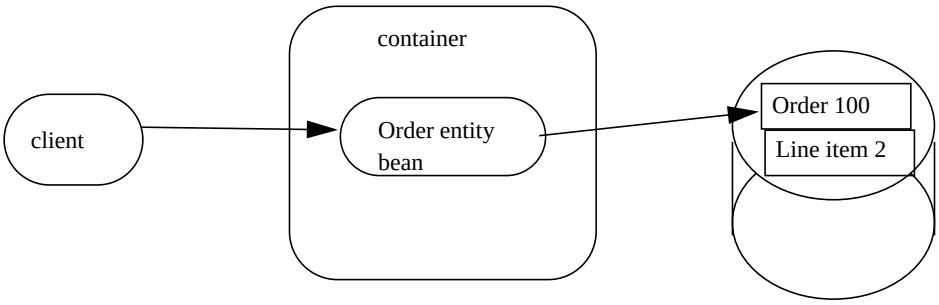
It is the responsibility of the Deployer to map the abstract persistence schema of a bean or dependent object class into the physical schema used by the underlying data store (e.g., into a relational schema) by using the Persistence Manager Provider's tools. The Deployer uses the deployment descriptor as input to the Persistence Manager's tools to perform this mapping. The Persistence Manager's tools typically are also used to generate the concrete implementation of the entity bean and dependent object classes, including the code that delegates calls to the methods of an entity bean or dependent object class's abstract persistence schema to the runtime persistent data access layer that is generated by the Persistence Manager Provider's tools. Typically, the Persistence Manager Provider's tools use the ejb-jar file produced by the Bean Provider or Assembler to generate a new ejb-jar file that includes the concrete implementations of the entity bean and dependent object classes. This new ejb-jar file then serves as input to the Container Provider's tools.

The EJB deployment descriptor describes *logical* relationships involving beans and dependent object classes. It does not provide a mechanism for specifying how the abstract persistence schemas of the entity bean and dependent object classes are to be mapped to an underlying database. This is the responsibility of the Deployer, who, using the Persistence Manager Provider's tools, uses the logical relationships that are specified in the deployment descriptor to map to the physical relationships that are specific to the underlying resource or its implementation. It is the responsibility of the Persistence Manager to manage the mapping between the logical and physical relationships at runtime and to manage the referential integrity of the relationships.

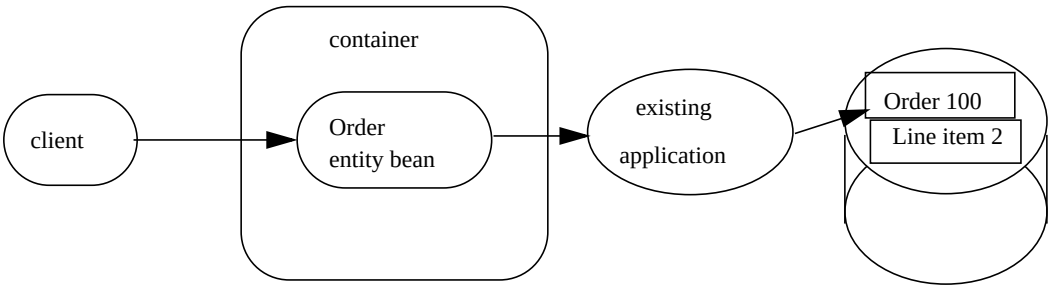
The advantage of using container-managed persistence is that the entity bean can be logically independent of the data source in which the entity is stored. The Persistence Manager Provider's tools can generate classes that use JDBC or SQLJ to access the entity state in a relational database; classes that implement access to a non-relational data source, such as an IMS database; or classes that implement function calls to existing enterprise applications. These tools are typically specific to each data source.

Figure 20 Client view of underlying data sources accessed through entity bean

(a) Entity bean is an object view of a collection of related records in the database



(b) Entity bean is an object view of an existing application



9.3.1 Granularity of entity beans

This section provides guidelines to the Bean Provider for the modeling of business objects as entity beans.

In general, an entity bean should represent an independent business object that has an independent identity and lifecycle, and is referenced by multiple enterprise beans and/or clients.

A dependent object can be characterized as follows. An object D is a dependent object, if D is created by another object, if D can only be accessed through another object (and not remotely), or if D's existence or removal is dependent on some other object: in other words, if D's lifecycle is managed by some other object.

A *dependent object* should not be implemented as an entity bean. Instead, a dependent object is better implemented as a Java class (or several classes) and included with the entity bean or beans on which it depends.

For example, whereas a purchase order might be implemented as an entity bean, the individual line items on the purchase order might be implemented as dependent objects, not as entity beans. An employee record might be implemented as an entity bean, but the employee address and phone number might be implemented as dependent objects, rather than as entity beans.

The state of an entity object that has dependent objects is often stored in multiple records in multiple database tables.

In addition, the Bean Provider must take the following into consideration when making a decision on the granularity of an entity object:

> *Every method call to an entity object via the remote and home interface is potentially a remote call. Even if the calling and called entity bean are collocated in the same JVM, the call must go through the container, which must create copies of all the parameters that are passed through the interface by value (i.e., all parameters that do not extend the* `java.rmi.Remote` *interface). The container is also required to check security and apply the declarative transaction attribute on the inter-component calls. The overhead of an inter-component call will likely be prohibitive for object interactions that are too fine-grained.*

9.4 The entity bean provider's view of persistence

An entity bean with container managed persistence consists of its class and a set of related dependent object classes; a remote interface which defines its client view business methods; a home interface which defines its create, remove, home, and finder methods; and its abstract persistence schema as specified in the deployment descriptor.

A client of an entity bean can control the lifecycle of a bean by using the bean's home interface and can manipulate the bean as a business entity by using the methods defined by its remote interface. The home and remote interfaces of a bean define its client view.

The abstract persistence schema of an entity bean consists of a set of cmp- and cmr-fields that define the entity bean's persistent state and relationships. The abstract persistence schema defines the set of accessor methods for the persistent fields and relationships of the entity bean. The persistent fields and relationships themselves are maintained by the persistence manager.

It is the responsibility of the bean provider to specify the abstract persistence schema of the entity bean in the deployment descriptor. This responsibility is discussed further in Section 9.4.14.

9.4.1 The entity bean provider's programming contract

The bean provider must observe the following programming contract when defining an entity bean class that uses container managed persistence:

- The Bean Provider must define the entity bean class as an abstract class. The persistence manager provides the implementation class that is used for the entity bean at runtime.

- The container managed persistent fields and container managed relationship fields must *not* be defined in the entity bean class. From the perspective of the Bean Provider, the container managed persistent fields and container managed relationship fields are *virtual* fields only, and are accessed through get and set accessor methods. The implementation of the container managed persistent fields and container managed relationship fields is supplied by the Persistence Manager.

- The container managed persistent fields and container managed relationship fields of the entity bean must be specified in the deployment descriptor using the `cmp-field` and `cmr-field` elements respectively. The names of these fields must begin with a lowercase letter.

- The Bean Provider must define the accessor methods for the container managed persistent fields and container managed relationship fields as get and set methods, as for a JavaBean. The implementation of the accessor methods is supplied by the Persistence Manager.

- The accessor methods must be public, must be abstract, and must bear the name of the container managed persistent field (`cmp-field`) or container managed relationship field (`cmr-field`) that is specified in the deployment descriptor, and in which the first letter is uppercased, prefixed by "**get**" or "**set**".

- The accessor methods for container managed relationship fields that reference other enterprise beans must be defined in terms of the remote interfaces of those beans, as described in Section 9.4.2.

- The accessor methods for container managed relationship fields for one-to-many or many-to-many relationships must utilize one of the following Collection interfaces: `java.util.Collection` or `java.util.Set`[7]. The Collection interfaces used by a bean are specified in the deployment descriptor. The implementation of the collection classes used for the container managed relationship fields is supplied by the Persistence Manager.

[7] We expect to include java.util.List and java.util.Map in a later version of this specification.

- The accessor methods of the bean must not be exposed in the client interface of the bean except in the cases that are defined in Section 9.4.11. This restriction is to ensure the data independence between the client view of the bean and the persistent state that is managed by the persistence manager, and to allow flexibility to the Persistence Manager in loading and maintaining the persistent state of the bean.

- Once the primary key for an entity bean has been set, the Bean Provider must not attempt to change it by use of the bean's set accessor methods on the primary key fields.

- The Bean Provider must ensure that the Java types assigned to the cmp-fields are restricted to the following: Java primitive types, Java serializable types, and references of enterprise beans' remote or home interfaces.[8]

9.4.2 The entity bean provider's view of persistent relationships

A bean may have relationships both with other beans and with dependent object classes.

Relationships may be one-to-one, one-to-many, or many-to-many relationships.

Relationships may be either bidirectional or unidirectional. If a relationship is bidirectional, it can be navigated in both directions, whereas a unidirectional relationship can be navigated in one direction only. Bidirectional container managed relationships can exist only among beans and dependent objects whose abstract persistence schemas are defined in the same deployment descriptor and which thus are managed by the same Persistence Manager.

A bean or dependent object class may have a unidirectional relationship with a target entity bean that is "remote" (i.e., whose abstract persistence schema is not defined in the same deployment descriptor). This includes entity beans with bean managed persistence, EJB 1.1 entity beans with container managed persistence, and entity beans that are defined in another deployment descriptor and ejb-jar file. A unidirectional relationship is implemented with a cmr-field on the entity bean or dependent object from which navigation can take place, and no related cmr-field on the entity bean or dependent object that is the target of the relationship. Unidirectional relationships are typically used for relationships with "remote" beans as well as when the Bean Provider wishes to restrict the visibility of a relationship.

The bean developer navigates or manipulates logical relationships by using the get and set accessor methods for the container managed relationship fields and the `java.util.Collection` API for collection-valued container managed relationship fields.

The bean provider must consider the type and cardinality of relationships when the beans and dependent object classes are programmed.

- In a relationship between beans, the get method must return either the remote interface of the related bean or a collection (more precisely, either `java.util.Collection` or `java.util.Set`), in which the members of the collection must be the remote interfaces of the related beans. The set method for the relationship must take as an argument the remote

[8] The Bean Provider should, however, avoid the use of storing references to enterprise beans' remote or home interfaces in cmp-fields in favor of the use of container managed relationships.

interface of the related bean or a collection whose members are the remote interfaces of the related beans.

- In a relationship between a bean and dependent object class, the get method that accesses the bean from the dependent object class must return the remote interface of the related bean (or a collection whose members must be the remote interfaces of the related beans), and the set method defined on the dependent object class must take as an argument the remote interface of the related bean (or a collection whose members must be the remote interfaces of the related beans). The get method that accesses the dependent object class from the bean must return the dependent object class type (or a collection whose members must be of the dependent object class type), and the set method defined on the bean must take as an argument the dependent object class type (or a collection of the same).

- In a relationship between dependent object classes, the formal types of the arguments and results of the get and set methods must be the respective dependent object classes (or collections of the same).

In EJB 1.1, the bean provider had to supply the code to explicitly locate related beans through the JNDI lookup of their home interfaces and the execution of their finder methods. In EJB 2.0 container managed persistence, the bean provider accesses related beans by means of container managed relationships defined in terms of the remote interfaces of the related beans. The responsibility to locate the homes of the related beans and to execute their finder methods is shifted to the persistence manager, which automatically provides the related objects (beans or dependent objects) at runtime.

9.4.3 The view of dependent classes

Two types of dependent classes must be distinguished: *dependent object classes* and *dependent value classes*. This chapter has up to now considered only dependent object classes.

A *dependent object class* is defined in much the same way as an entity bean class: as an abstract class, with an abstract persistence schema that defines the dependent object class's relationships with entity beans and other dependent object classes. A dependent object class can participate in both unidirectional and bidirectional container managed relationships. Dependent object classes can be shared across multiple entity beans (of the same or different type) within the same ejb-jar file, and across multiple relationships of the same entity bean. For example, an Order entity bean might use the dependent object class `Address` for both the cmr-field `shippingAddress` and the cmr-field `billingAddress`.

A dependent object class instance (or a collection of dependent object class instances) can be the value of a cmr-field; a dependent object class instance (or a collection of dependent object class instances) cannot be the value of a cmp-field.

A dependent object class must not be exposed through the remote interface of an entity bean. It is recommended that the dependent object classes not be serializable.[9]

[9] If the dependent object class is serializable, care must be taken to avoid use of the dependent object class as a method argument or result in the remote interface. This potential error can be avoided by not implementing the dependent object class as serializable.

A *dependent value class* is a concrete class. A dependent value class can be the value of a cmp-field, but it cannot be the value of a cmr-field. A dependent value class may be a legacy class that the bean provider wishes to use internally within an entity bean with container managed persistence, and/or it may be a class that the bean provider chooses to expose through the remote interface of the entity bean. A dependent value class must be serializable. The internal structure of a dependent value class is *not* described in the EJB deployment descriptor.

9.4.4 The entity bean provider's programming contract for dependent object classes

The bean provider must observe the following programming contract when defining a dependent object class:

- The bean provider must define the dependent object class as an abstract class. The Persistence Manager provides the implementation class for the dependent object class that is used at runtime.

- The container managed persistent fields and container managed relationship fields must *not* be defined in the dependent object class. From the perspective of the Bean Provider, the container managed persistent fields and container managed relationship fields are *virtual* fields only, and are accessed through get and set accessor methods. The implementation of the container managed persistent fields and container managed relationship fields is supplied by the Persistence Manager.

- The container managed persistent fields and container managed relationship fields of the dependent object class must be specified in the deployment descriptor using the `cmp-field` and `cmr-field` elements respectively. The names of these fields must begin with a lowercase letter.

- The Bean Provider must define the accessor methods for the container managed persistent fields and container managed relationship fields as get and set methods, as for a JavaBean. The implementation of the accessor methods is supplied by the Persistence Manager.

- The accessor methods must be public, must be abstract, and must bear the name of the container managed persistent field (`cmp-field`) or container managed relationship field (`cmr-field`) that is specified in the deployment descriptor, and in which the first letter is uppercased, prefixed by "**get**" or "**set**".

- The accessor methods for container managed relationship fields that reference enterprise beans must be defined in terms of the remote interfaces of those beans, as described in Section 9.4.2.

- The accessor methods for container managed relationship fields for one-to-many or many-to-many relationships must utilize one of the following Collection interfaces: `java.util.Collection or java.util.Set`[10]. The Collection interfaces used by a dependent object class are specified in the deployment descriptor. The implementation of the collection classes used for the container managed relationship fields is supplied by the Persistence Manager.

[10] We expect to include java.util.List and java.util.Map in a later version of this specification.

- Once the primary key for a dependent object has been set, the Bean Provider must not attempt to change it by use of the dependent object's set accessor methods on the primary key fields.

- If it is possible for the application to result in a loopback call to a dependent object, the Bean Provider should program the dependent object class to handle loopback calls.

9.4.4.1 Creation protocol for dependent objects

Because the container-managed dependent object classes are abstract, a special API is needed so that the Bean Provider can obtain a new instance of a dependent object class at runtime.

The Bean Provider defines the following methods, which enable an instance of a dependent object class to be created and initialized from an entity bean or dependent object class.

- The Bean Provider defines an abstract `create<METHOD>(...)` method on any entity bean or dependent object class which may need to create instances of the dependent object class. The result type of the `create<METHOD>(...)` method is the dependent object class.

  The `create<METHOD>(...)` methods must be declared as `public` and must not be declared as `final` or `static`.

  The Bean Provider must not expose the `create<METHOD>(...)` methods of an entity bean in the remote interface of the bean.

  It is the responsibility of the Persistence Manager to provide the concrete implementation of the `create<METHOD>(...)` methods. The Persistence Manager's `create<METHOD>(...)` method must create a new instance of the dependent object class and invoke the Bean Provider's corresponding `ejbCreate<METHOD>(...)` method on the new instance, followed by the matching `ejbPostCreate<METHOD>(...)` method, passing the `create<METHOD>(...)` parameters to those matching methods. Prior to invoking the `ejbCreate<METHOD>(...)` method provided by the Bean Provider, the Persistence Manager must ensure that the values that will be initially returned by the instance's get methods for container managed fields will be the Java language defaults (e.g. 0 for integer, `null` for pointers), except for collection-valued cmr-fields, which will have the empty collection (or set) as their value.

  Since an entity bean may have relationships with multiple dependent object classes, and a dependent object class may have relationships with multiple entity beans, it is recommended that the names of the `create<METHOD>(...)` methods be chosen carefully to avoid naming conflicts. For example, a typical naming protocol might be to name such create methods as `create<DEPENDENTNAME>(...)`, where `DEPENDENTNAME` is the value of the `dependent-name` deployment descriptor element for the dependent object class.

- The Bean Provider defines an `ejbCreate<METHOD>(...)` method on the abstract dependent object class. The `ejbCreate<METHOD>(...)` method performs the initialization of the dependent object class instance. It is invoked by the Persistence Manager. If the primary key fields of the dependent object class are defined by the Bean Provider, the Bean Provider must use the `ejbCreate<METHOD>(...)` method to set the values of those fields. In general, the Bean Provider's responsibility is to initialize the instance in the `ejbCreate<METHOD>(...)` method from the input arguments, using the get and set accessor methods, such that when the `ejbCreate<METHOD>(...)` method returns, the persistent representation of the instance can be created. The Bean Provider is guaranteed that the values that will be initially returned by the dependent object class instance's get methods will be the

Java language defaults (e.g. 0 for integer, `null` for pointers), except for collection-valued cmr-fields, which will have the empty collection (or set) as their value. The Bean Provider must not attempt to modify the values of cmr-fields or create instances of dependent object classes in an `ejbCreate<METHOD(...)` method; this should be done in the `ejbPostCreate<METHOD(...)` method instead.

- The Bean Provider defines an `ejbPostCreate<METHOD>(...)` method on the abstract dependent object class. This method is invoked by the Persistence Manager after the `ejbCreate<METHOD>(...)` method completes and after the identity of the dependent object class instance has been established. If the primary key of the dependent object class is defined by the Persistence Manager rather than the Bean Provider, it is the responsibility of the Persistence Manager to have established the identity of the dependent object instance before the `ejbPostCreate<METHOD>(...)` method is called. The Bean Provider can make use of the `ejbPostCreate<METHOD>(...)` method to complete the initialization of the dependent object class instance, including the creation of related objects and the setting of the values of cmr-fields.

For each `create<METHOD>(...)` method defined on any entity bean or dependent object class there must be a matching `ejbCreate<METHOD>(...)` and `ejbPostCreate<METHOD>(...)` method on the dependent object class. The matching methods must have the same number and type of arguments, and the same return type (i.e., the dependent object class).

The throws clause of the `create<METHOD>(...)`method must include the `javax.ejb.CreateException`. The throws clause may define arbitrary application specific exceptions.

All the exceptions defined in the throws clause of the matching `ejbCreate<METHOD>` and `ejbPostCreate<METHOD>` methods of the dependent object class must be included in the `throws` clause of the matching `create<METHOD>(...)` methods (i.e., the set of exceptions defined for a `create<METHOD>(...)` method must be a superset of the union of exceptions defined for the `ejbCreate<METHOD>` and `ejbPostCreate<METHOD>` methods).

The dependent object class instance must have a primary key value that is unique across all instances of the dependent object class. However, it is legal to reuse the primary key of a previously removed dependent object instance. The Persistence Manager's `create<METHOD>(...)` method may, but is not required to, detect the attempt to create a dependent object instance with a duplicate primary key, and throw the `DuplicateKeyException`.

If a dependent object is created, but not assigned to any container managed relationship field, it will be created in the data store by the Persistence Manager, but will not be reachable by navigation. An EJB QL query must be used to access such a "detached" dependent object. See Chapter 10 "EJB QL: EJB Query Language for Container Managed Persistence Query Methods" .

9.4.4.2 Removal of dependent objects

The Bean Provider can specify the removal of dependent objects in two ways:

- By the use of a `remove()` method on the dependent object class.
- By the use of a `cascade-delete` specification in the deployment descriptor.

Detaching a dependent object that has already been entered into the data source from all of the container-managed relationships in which it participates does not cause the persistent representation of the dependent object to be removed. The dependent object will continue to exist in the data store as a "detached" dependent object. It will not be reachable by navigation but can be accessed by an EJB QL query.

The Bean Provider must define an abstract `remove()` method on every dependent object class. The Persistence Manager provides the concrete implementation of the `remove()` method. When the remove method is invoked on a dependent object, the persistence manager's implementation of the `remove()` method must remove that dependent object from all relationships in which it participates and remove its persistent representation.

When a dependent object has been removed from a relationship because of its persistent removal, accessor methods for the relationship will return null (in the case of a one-to-one or many-to-one relationship) or a collection from which the dependent object has been removed (in the case of a one-to-many or many-to-many relationship).

The Persistence Manager must detect the attempt to invoke an accessor method on a deleted dependent object and raise the `java.lang.IllegalStateException`. The Persistence Manager must detect the attempt to assign an object whose persistent representation has been deleted as the value of a cmr-field of another object (whether as an argument to a set accessor method or as an argument to a method of the `java.util.Collection` API) and raise the `java.lang.IllegalArgument-Exception`.

More than one relationship may be affected by the persistent removal of a dependent object, as in the following example. Once the shipping address object used by the Order bean has been removed, the billing address accessor method will also return null.

```
public void changeAddress()
     Address a = createAddress();
     setShippingAddress(a);
     setBillingAddress(a);
     //both relationships now reference the same dependent
     //object instance
     getShippingAddress.remove();
     if (getBillingAddress() == null) //it must be
          ...
     else ...
          // this is impossible....
```

The `remove()` method causes only the dependent object on which it is invoked to be removed. It does not cause the deletion to be cascaded to other objects. In order for the deletion of one object to be automatically cascaded to another object, use of the `cascade-delete` mechanism should be specified.

The `cascade-delete` deployment descriptor element is used to specify that, within a particular relationship, the lifetime of one or more dependent objects is dependent upon the lifetime of another object (an entity bean or dependent object). The `cascade-delete` element can only be used to specify the cascaded deletion of a dependent object, not an entity bean. To delete an entity bean, the remove method of the entity bean's home or remote interface must be used.

The `cascade-delete` deployment descriptor element is contained within the `ejb-relationship-role` element. The `cascade-delete` element can only be specified within an `ejb-relationship-role` element in which the `role-source` element specifies a dependent object class.

The `cascade-delete` element can only be specified for an `ejb-relationship-role` element contained in an `ejb-relation` element if the *other* `ejb-relationship-role` element in the `ejb-relation` element specifies a `multiplicity` of `One`. That is, the deletion of one object can only be cascaded to the deletion of other objects if the first object is in a one-to-one or one-to-many relationship with those other objects.

If an entity bean or dependent object is deleted, and the `cascade-delete` deployment descriptor element is specified for a cmr-field of that bean or dependent object class, then the removal is cascaded to cause the removal of the related object or objects. As with the `remove()` operation, when a dependent object has been removed from a relationship because of a cascaded delete, accessor methods for relationships that formerly referenced that object will return null (in the case of a one-to-one or many-to-one relationship) or a collection from which the dependent object has been removed (in the case of a one-to-many or many-to-many relationship).

The use of `cascade-delete` causes only the object or objects in the relationship for which it is specified to be deleted. It does not cause the deletion to be further cascaded to other objects, unless they are participants in relationship roles for which `cascade-delete` has also been specified.

9.4.5 Identity of dependent object class instances

From the viewpoint of the Bean Provider, instances of a dependent object class have a persistent identity that is maintained by the Persistence Manager.

For example, the Bean Provider may use the dependent object class `Address` as the target of two distinct unidirectional relationships of the entity bean `Order`: one to represent a shipping address and one to represent a billing address. If the same dependent object class instance is set as the value of both the `shippingAddress` and `billingAddress` cmr-fields, as in the example below, it is the responsibility of the persistence manager to ensure that the same object is returned by `getShippingAddress()` and `getBillingAddress()`, and the identity of the object is maintained, including across `ejbLoad()` and `ejbStore()` operations and across transactions.

```
public void setAddresses(){
     Address a = createAddress();
     ...
     setShippingAddress(a);
     setBillingAddress(a);
     a.setZipcode("94303");
     // the single address object is modified
     ...
}
```

The Persistence Manager maintains the identity of a dependent object class instance on the basis of the primary key fields of the dependent object as specified by the Bean Provider.

The primary key of a dependent object class may or may not be visible in the dependent object class, depending on the way in which it is specified. The Bean Provider specifies the primary key of a dependent object as described in Section 9.10.2. Once it has been set, the Bean Provider must not attempt to change the value of a visible primary key field of a dependent object.

Note that depending on the mapping of the abstract persistence schema of the dependent object class to the physical schema of the underlying data store, a visible primary key field may also be used by the Persistence Manager in the implementation of a container-managed relationship (i.e., it may serve as the implementation of a cmr-field). This case corresponds to that of a compound primary key which contains a foreign key in a relational database. In this case, the Persistence Manager must ensure that the visible primary key field is updated when changes are made to the relationship.

When a new instance of a dependent object whose primary keys are visible in the dependent object class is created, the Bean Provider must use the `ejbCreate<METHOD>(...)` method of the dependent object class to set all the primary key fields of the dependent object before the dependent object can participate in a relationship, e.g. be used in a set accessor method for a cmr-field. The Bean Provider must not reset a primary key value after it has been set.

9.4.6 Semantics of assignment for relationships

The assignment operations for container managed relationships have a special semantics that is determined by the referential integrity semantics for the relationship multiplicity.

In the case of a one-to-one relationship, when the Bean Provider uses a set accessor method to assign an object from a cmr-field of a given *relationship type* (as defined by the `ejb-relation` and `ejb-relationship-role` deployment descriptor elements) in one instance to a cmr-field of the *same relationship type* in another instance, the object is effectively *moved* and the value of the source cmr-field is set to null in the same transaction context.

In the case of a one-to-many or many-to-many relationship, either the `java.util.Collection` API or a set accessor method may be used to manipulate the contents of a collection-valued cmr-field. These two approaches are discussed below.

9.4.6.1 Use of the java.util.Collection API to update relationships

The methods of the `java.util.Collection` API for the container managed collections used for collection-valued cmr-fields have the usual semantics, with the following exception: the `add` and `addAll` methods applied to container managed collections in one-to-many relationships have a special semantics that is determined by the referential integrity of one-to-many relationships.

- If the argument to the `add` method is already an element of a collection-valued relationship field of the *same relationship type* as the target collection (as defined by the `ejb-relation` and `ejb-relationship-role` deployment descriptor elements), it is removed from this first relationship and added, in the same transaction context, to the target relationship (i.e., it is in effect moved from one collection of the relationship type to the other). For example, if there is a one-to-many relationship between field offices and sales representatives, adding a sales representative to a new field office will have the effect of removing him from his current field office. If the argument to the `add` method is not an element of a collection-valued relationship

of the same relationship type, it is simply added to the target collection and not removed from its current collection, if any.

- The `addAll` method, when applied to a target collection in a one-to-many relationship, has similar semantics, applied to the members of its collection argument individually.

Note that in the case of a many-to-many relationship, however, adding an element or elements to the contents of a collection-valued cmr-field has no effect on the source collection, if any. For example, if there is a many-to-many relationship between customers and sales representatives, a customer can be added to the set of customers handled by a particular sales representative without affecting the set of customers handled by any other sales representative.

When the `java.util.Collection` API is used to manipulate the contents of container managed relationship fields, the argument to any Collection method defined with a single Object parameter must be of the element type of the collection defined for the target cmr-field. The argument for any collection-valued parameter must be a `java.util.Collection` (or `Set`), all of whose elements are of the element type of the collection defined for the target cmr-field. If an argument is not of the correct type for the relationship, the Persistence Manager must throw the `java.lang.IllegalArgumentException`.

The Bean Provider must exercise caution when using an Iterator over a collection in a container managed relationship. In particular, the Bean Provider must not modify the container managed collection while the iteration is in progress in any way that causes elements to be added or removed, other than by the `java.util.Iterator.remove()` method. If elements are added or removed from the underlying container managed collection used by an iterator other than by the `java.util.Iterator.remove()` method, the persistence manager should throw the `java.lang.IllegalStateException` on the next operation on the iterator.

The following example illustrates how operations on container managed relationships that affect the contents of a collection-valued cmr-field viewed through an iterator should be avoided. Because there is a one-to-many relationship between field offices and sales representatives, adding a sales representative to a new field office causes him or her to be removed from the current field office.

```
Collection nySalesreps = nyOffice.getSalesreps();
Collection sfSalesreps = sfOffice.getSalesreps();

Iterator i = nySalesreps.iterator();
Salesrep salesrep;

// the wrong way to transfer the salesrep
while (i.hasNext()) {
     salesrep = i.next();
     sfSalesreps.add(salesrep); // removes salesrep from nyOffice
}

// this is a correct and safe way to transfer the salesrep
while (i.hasNext()) {
     salesrep = i.next();
     i.remove();
     sfSalesreps.add(salesrep);
}
```

9.4.6.2 Use of set accessor methods to update relationships

The semantics of a set accessor method, when applied to a collection-valued cmr-field, is also determined by the referential integrity semantics associated with the multiplicity of the relationship.

In the case of a one-to-many relationship, if a collection of objects is assigned from a cmr-field of a given relationship type in one instance to a cmr-field of the same relationship type in another instance, the objects in the collection are effectively *moved*. The contents of the collection of the target instance are replaced with the contents of the collection of the source instance, but the *identity* of the collection object containing the instances in the relationship does not change. The source cmr-field contains the same collection object as before (i.e., the identity of the collection object is preserved), but the collection is empty. The Bean Provider can thus use the set method to move objects between the collections referenced by cmr-fields of the same relationship type in different instances. The set accessor method, when applied to a cmr-field in a one-to-many relationship thus has the semantics of the `java.util.Collection` methods `clear`, followed by `addAll`, applied to the target collection; and `clear`, applied to the source collection. It is the responsibility of the persistence manager to transfer the contents of the collection instances in the same transaction context.

In the following example, the telephone numbers associated with the billing address of an Order bean instance are transferred to the shipping address. Both the billing address and shipping address are of the same dependent object class, `Address`. The dependent object class `Address` is related to the dependent object class `TelephoneNumber` in a one-to-many relationship. The example illustrates how a Bean Provider uses the set method to move a set of dependent object class instances.

```
public void changeTelephoneNumber() {
     Address a = getShippingAddress();
     Address b = getBillingAddress();
     Collection c = b.getTelephoneNumbers();
     a.setTelephoneNumbers(b.getTelephoneNumbers());
     if (c.isEmpty()) {//must be true...
     ..
}
```

In the case of a many-to-many relationship, if the value of a cmr-field is assigned to a cmr-field of the same relationship type in another instance, the objects in the collection of the first instance are assigned as the contents of the cmr-field of the second instance. The contents of the collections are shared, but not the collections themselves and the identity of the collection objects is unchanged. The set accessor method, when applied to a cmr-field in a many-to-many relationship thus has the semantics of the `java.util.Collection` methods `clear`, followed by `addAll`, applied to the target collection.

For example, if there is a many-to-many relationship between customers and sales representatives, assigning the set of customers of one sales representative to the another sales representative will result in both sales representatives handling the customers. If the second sales representative originally handled a different set of customers, those customers will no longer be handled by that sales representative.

```
public void shareCustomers(SalesRep rep) {
     setCustomers(rep.getCustomers());
     // the customers are shared among the salesreps
}
```

The following section, 9.4.7 “Assignment rules for relationships” , defines the semantics of assignment for relationships in further detail and provides further examples.

9.4.7 Assignment rules for relationships

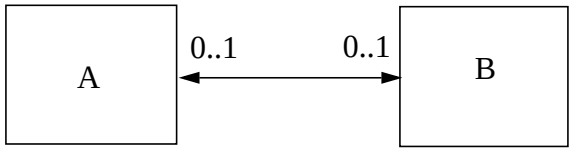
This section defines the semantics of assignment and collection manipulation in one-to-one, one-to-many, and many-to-many container managed relationships.

The figures make use of two dependent object classes, A and B. Instances of `A` are typically designated as `a1`,...,`an`; instances of `B` as `b1`,...,`bm`. Class `A` has accessor methods `getB` and `setB` for navigable relationships with `B`: `getB` returns an instance of `B` or a collection of instances of `B`, depending on the multiplicity of the relationship. Similarly, class `B` has accessor methods `getA` and `setA` for navigable relationships with `A`.

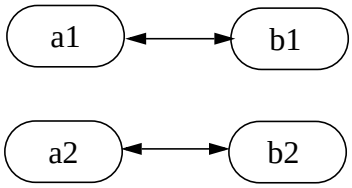
All changes in each subsection are assumed to be applied to the figure labeled "Before change" at the beginning of the subsection (i.e., changes are not cumulative). The results of changes are designated graphically as well as in conditional expressions written in the Java™ programming language.

9.4.7.1 One-to-one bidirectional relationships

A and B are in a one-to-one bidirectional relationship:



Before change:



Before change:

```
B b1 = a1.getB();
B b2 = a2.getB();
```

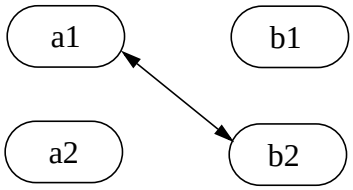
Change:

```
a1.setB(a2.getB());
```

Expected result:

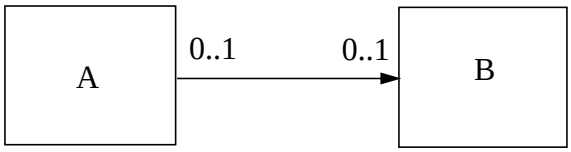
```
(a1.getB() == b2) && (a2.getB() == null) && (b1.getA() == null) &&
(b2.getA() == a1)
```

After change:

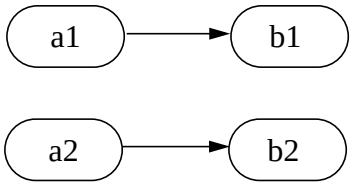


9.4.7.2 One-to-one unidirectional relationships

A and B are in a one-to-one unidirectional relationship:



Before change:



Before change:

```
B b1 = a1.getB();
B b2 = a2.getB();
```

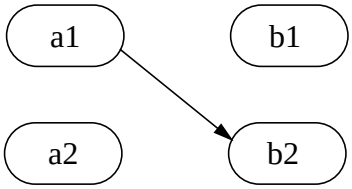
Change:

```
a1.setB(a2.getB());
```

Expected result:

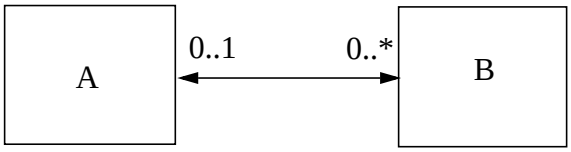
```
(a1.getB() == b2) && (a2.getB() == null)
```

After change:

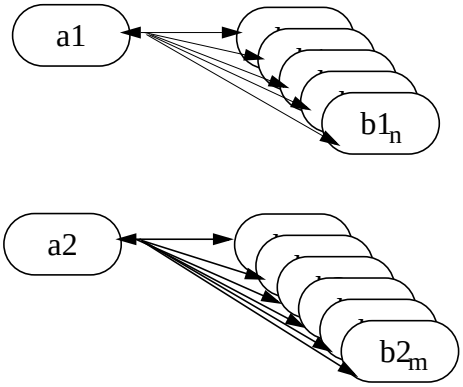


9.4.7.3 One-to-many bidirectional relationships

A and B are in a one-to-many bidirectional relationship:



Before change:



Before change:

```
Collection b1 = a1.getB();
Collection b2 = a2.getB();
B b11, b12, ... , b1n; // members of b1
B b21, b22, ... , b2m; // members of b2
```

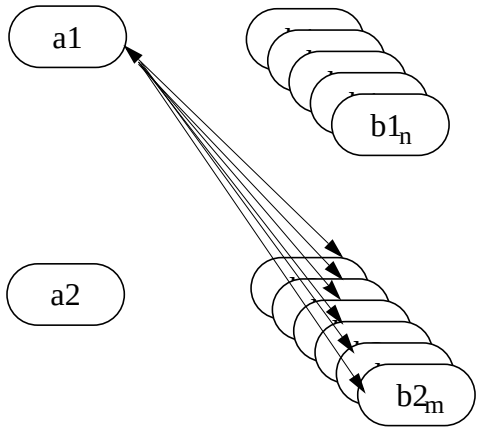
Change:

```
a1.setB(a2.getB());
```

Expected result:

```
(a2.getB().isEmpty()) &&
(b2.isEmpty()) &&
(b1 == a1.getB()) &&
(b2 == a2.getB()) &&
(a1.getB().contains(b21)) &&
(a1.getB().contains(b22)) &&  ... &&
(a1.getB().contains(b2m)) &&
(b11.getA() == null) &&
(b12.getA() == null) &&  ... &&
(b1n.getA() == null) &&
(b21.getA() == a1) &&
(b22.getA() == a1) && ...&&
(b2m.getA() == a1)
```

After change:



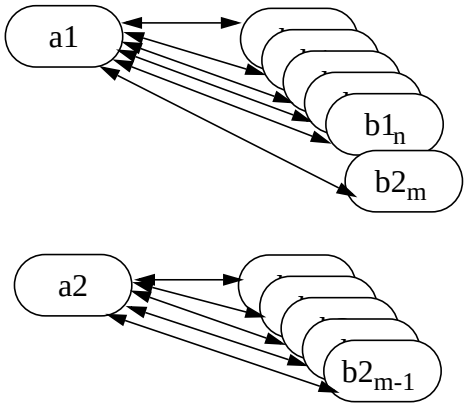
Change:

```
b2m.setA(b1n.getA());
```

Expected result:

```
(b1.contains(b11)) &&
(b1.contains(b12)) &&  ... &&
(b1.contains(b1n)) &&
(b1.contains(b2m)) &&
(b2.contains(b21)) &&
(b2.contains(b22)) &&  ... &&
(b2.contains(b2m_1)) &&
(b11.getA() == a1) &&
(b12.getA() == a1) &&  ... &&
(b1n.getA() == a1) &&
(b21.getA() == a2) &&
(b22.getA() == a2) &&  ... &&
(b2m_1.getA() == a2) &&
(b2m.getA() == a1)
```

After change:



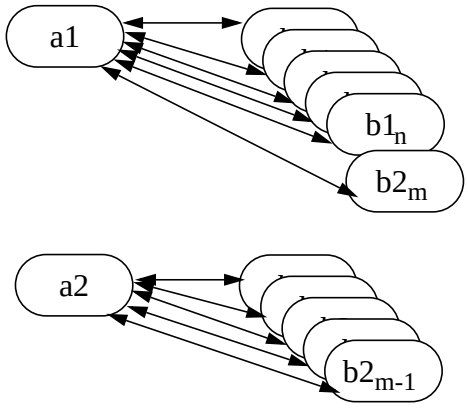
Change:

```
a1.getB().add(b2m);
```

Expected result:

```
(b1.contains(b11)) &&
(b1.contains(b12)) &&  ... &&
(b1.contains(b1n)) &&
(b1.contains(b2m)) &&
(b2.contains(b21)) &&
(b2.contains(b22)) &&  ... &&
(b2.contains(b2m_1)) &&
(b11.getA() == a1) &&
(b12.getA() == a1) &&  ... &&
(b1n.getA() == a1) &&
(b21.getA() == a2) &&
(b22.getA() == a2) &&  ... &&
(b2m_1.getA() == a2) &&
(b2m.getA() == a1)
```

After change:



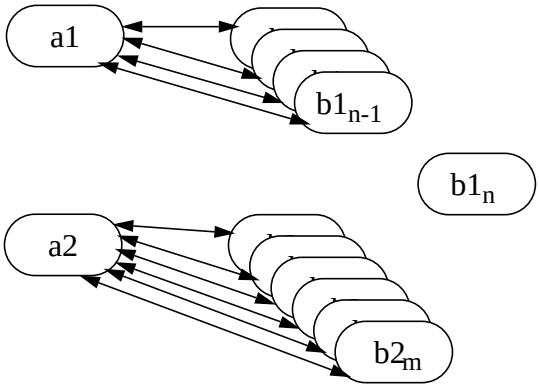
Change:

```
a1.getB().remove(b1n);
```

Expected result:

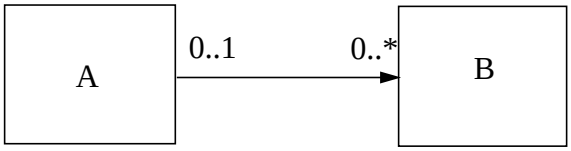
```
(b1n.getA() == null) &&
(a1.getB() == b1) &&
(b1.contains(b11)) &&
(b1.contains(b12)) &&  ... &&
(b1.contains(b1n_1)) &&
!(b1.contains(b1n))
```

After change:

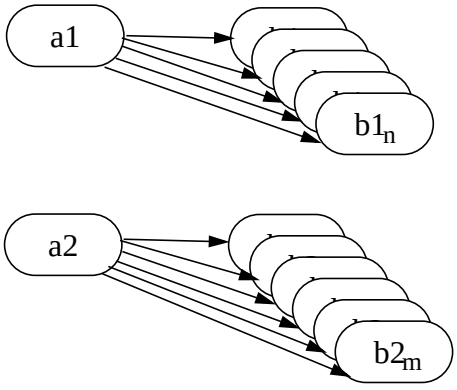


9.4.7.4 One-to-many unidirectional relationships

A and B are in a one-to-many unidirectional relationship:



Before change:



Before change:

```
Collection b1 = a1.getB();
Collection b2 = a2.getB();
B b11, b12, ... , b1n; // members of b1
B b21, b22, ... , b2m; // members of b2
```

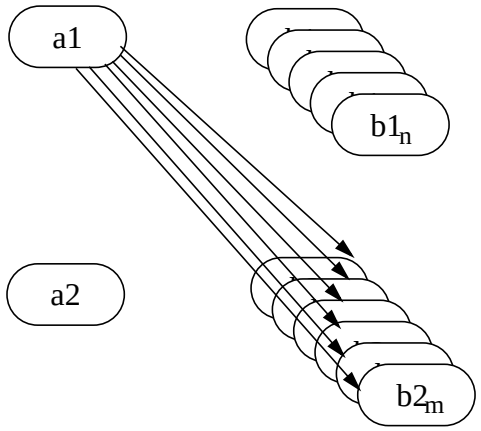
Change:

```
a1.setB(a2.getB());
```

Expected result:

```
(a2.getB().isEmpty()) &&
(b2.isEmpty()) &&
(b1 == a1.getB()) &&
(b2 == a2.getB()) &&
(a1.getB().contains(b21)) &&
(a1.getB().contains(b22)) &&  ... &&
(a1.getB().contains(b2m))
```

After change:



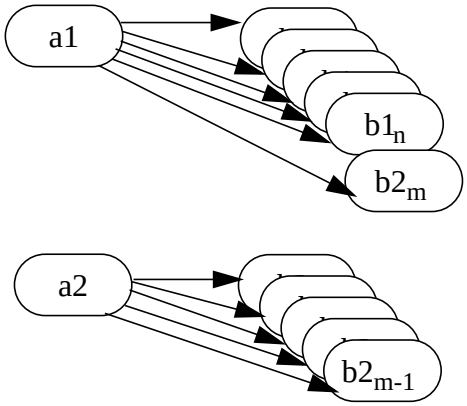
Change:

```
a1.getB().add(b2m);
```

Expected result:

```
(a1.getB() == b1) &&
(b1.contains(b2m))
```

After change:



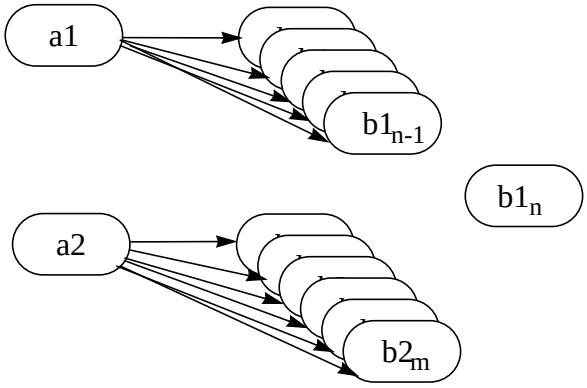
Change:

```
a1.getB().remove(b1n);
```

Expected result:

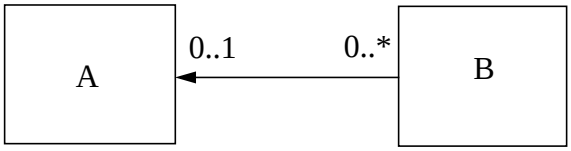
```
(a1.getB().contains(b11)) &&
(a1.getB().contains(b12)) &&  ... &&
(a1.getB().contains(b1n_1)) &&
!(a1.getB().contains(b1n)) &&
```

After change:

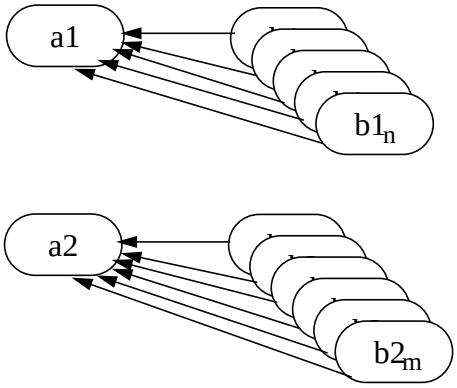


9.4.7.5 Many-to-one unidirectional relationships

A and B are in a many-to-one unidirectional relationship:



Before change:



Before change:

```
B b11, b12, ... , b1n;
B b21, b22, ... , b2m;
// the following is true
// (b11.getA() == a1) && ... && (b1n.getA() == a1) &&
// (b21.getA() == a2) && ... && (b2m.getA() == a2)
```

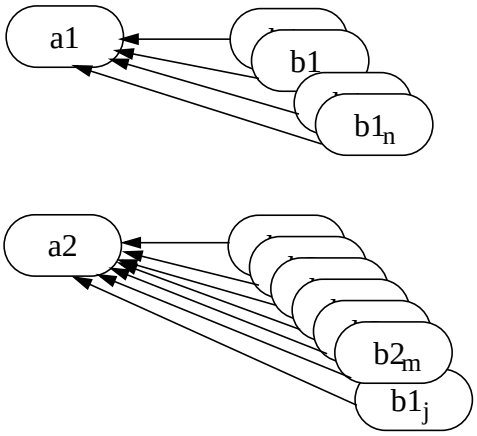
Change:

```
b1j.setA(b2k.getA());
```

Expected result:

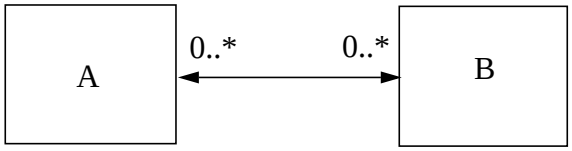
```
(b11.getA() == a1) &&
(b12.getA() == a1) &&
...
(b1j.getA() == a2) &&
...
(b1n.getA() == a1) &&
(b21.getA() == a2) &&
(b22.getA() == a2) &&
...
(b2k.getA() == a2) &&
...
(b2m.getA() == a2)
```

After change:

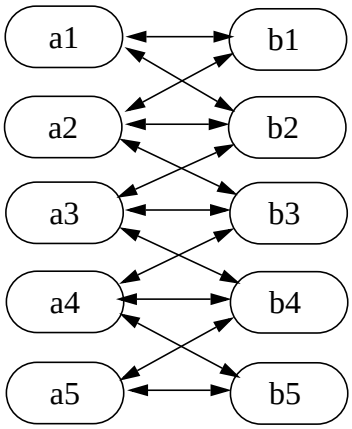


9.4.7.6 Many-to-many bidirectional relationships

A and B are in a many-to-many bidirectional relationship:



Before change:



Before change the following holds:

```
(a1.getB().contains(b1)) &&
(a1.getB().contains(b2)) &&
(a2.getB().contains(b1)) &&
(a2.getB().contains(b2)) &&
(a2.getB().contains(b3)) &&
(a3.getB().contains(b2)) &&
(a3.getB().contains(b3)) &&
(a3.getB().contains(b4)) &&
(a4.getB().contains(b3)) &&
(a4.getB().contains(b4)) &&
(a4.getB().contains(b5)) &&
(a5.getB().contains(b4)) &&
(a5.getB().contains(b5)) &&
(b1.getA().contains(a1)) &&
(b1.getA().contains(a2)) &&
(b2.getA().contains(a1)) &&
(b2.getA().contains(a2)) &&
(b2.getA().contains(a3)) &&
(b3.getA().contains(a2)) &&
(b3.getA().contains(a3)) &&
(b3.getA().contains(a4)) &&
(b4.getA().contains(a3)) &&
(b4.getA().contains(a4)) &&
(b4.getA().contains(a5)) &&
(b5.getA().contains(a4)) &&
(b5.getA().contains(a5)) &&
```

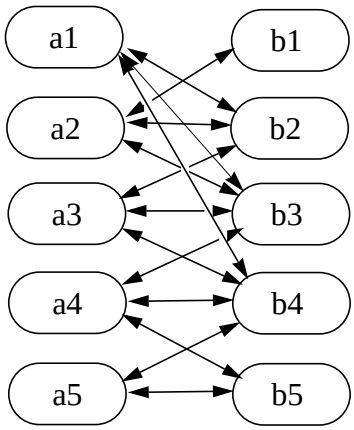
Change:

```
a1.setB(a3.getB());
```

Expected result:

```
(a1.getB().contains(b2)) &&
(a1.getB().contains(b3)) &&
(a1.getB().contains(b4)) &&
(a3.getB().contains(b2)) &&
(a3.getB().contains(b3)) &&
(a3.getB().contains(b4)) &&
(b1.getA().contains(a2)) &&
(b2.getA().contains(a1)) &&
(b2.getA().contains(a2)) &&
(b2.getA().contains(a3)) &&
(b3.getA().contains(a1)) &&
(b3.getA().contains(a2)) &&
(b3.getA().contains(a3)) &&
(b3.getA().contains(a4)) &&
(b4.getA().contains(a1)) &&
(b4.getA().contains(a3)) &&
(b4.getA().contains(a4)) &&
(b4.getA().contains(a5))
```

After change:



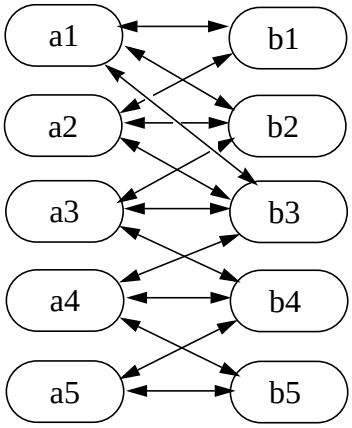
Change:

```
a1.getB().add(b3);
```

Expected result:

```
(a1.getB().contains(b1)) &&
(a1.getB().contains(b2)) &&
(a1.getB().contains(b3)) &&
(b3.getA().contains(a1)) &&
(b3.getA().contains(a2)) &&
(b3.getA().contains(a3)) &&
(b3.getA().contains(a4)) &&
```

After change:



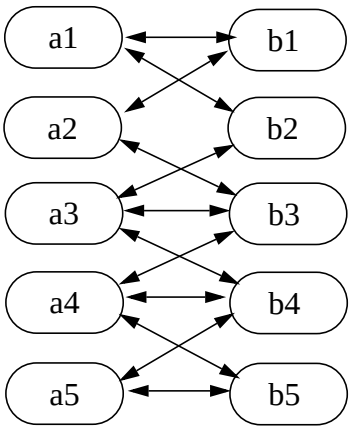
Change:

```
a2.getB().remove(b2);
```

Expected result:

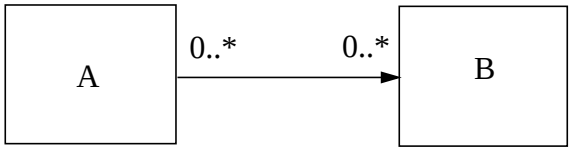
```
(a2.getB().contains(b1)) &&
(a2.getB().contains(b3)) &&
(b2.getA().contains(a1)) &&
(b2.getA().contains(a3))
```

After change:

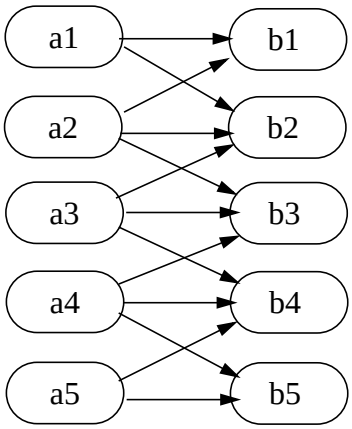


9.4.7.7 Many-to-many unidirectional relationships

A and B are in a many-to-many unidirectional relationship:



Before change:



Before change the following holds:

```
(a1.getB().contains(b1)) &&
(a1.getB().contains(b2)) &&
(a2.getB().contains(b1)) &&
(a2.getB().contains(b2)) &&
(a2.getB().contains(b3)) &&
(a3.getB().contains(b2)) &&
(a3.getB().contains(b3)) &&
(a3.getB().contains(b4)) &&
(a4.getB().contains(b3)) &&
(a4.getB().contains(b4)) &&
(a4.getB().contains(b5)) &&
(a5.getB().contains(b4)) &&
(a5.getB().contains(b5)) &&
```

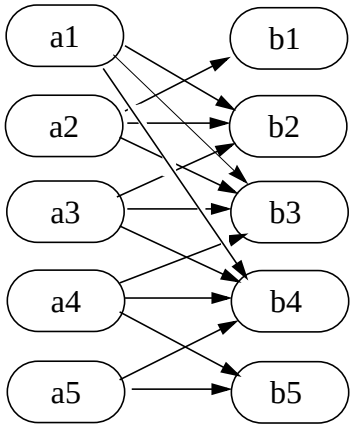
Change:

```
a1.setB(a3.getB());
```

Expected Result:

```
(a1.getB().contains(b2)) &&
(a1.getB().contains(b3)) &&
(a1.getB().contains(b4)) &&
(a3.getB().contains(b2)) &&
(a3.getB().contains(b3)) &&
(a3.getB().contains(b4)) &&
```

After change:



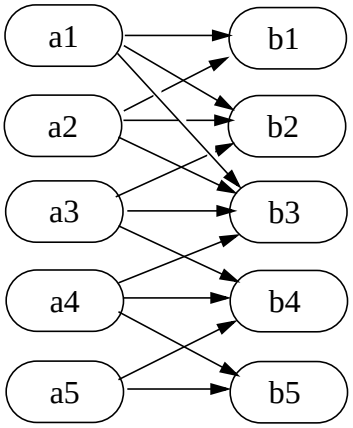
Change:

```
a1.getB().add(b3);
```

Expected result:

```
(a1.getB().contains(b1)) &&
(a1.getB().contains(b2)) &&
(a1.getB().contains(b3))
```

After change:



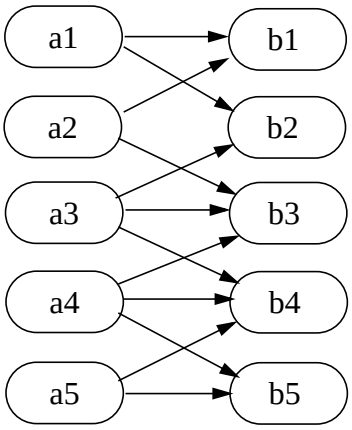
Change:

```
a2.getB().remove(b2);
```

Expected result:

```
(a2.getB().contains(b1)) &&
(a2.getB().contains(b3))
```

After change:



9.4.8 Collections managed by the Persistence Manager

The collections that are used in the representation of one-to-many and many-to-many container managed relationships are implemented and managed by the Persistence Manager. The following semantics apply to these collections:

- It is the responsibility of the Persistence Manager to preserve the runtime identity of the collection objects used in container-managed relationships.
- There is no constructor available to the Bean Provider for the collections that are maintained by the Persistence Manager.
- If there are no related values for a given container managed relationship, the get accessor method for that cmr-field returns an empty collection (and not `null`).
- It is the responsibility of the Persistence Manager to raise the `java.lang.IllegalArgumentException` if the Bean Provider attempts to assign `null` as the value of a collection-valued cmr-field by means of the set accessor method.
- It is the responsibility of the Persistence Manager to ensure that when the `java.util.Collection` API is used to manipulate the contents of container managed relationship fields, the argument to any Collection method defined with a single Object parameter must be of the element type of the collection defined for the target cmr-field. The argument for any collection-valued parameter must be a `java.util.Collection` (or `Set`), all of whose elements are of the element type of the collection defined for the target cmr-field. If an argument is not of the correct type for the relationship, the Persistence Manager must throw the `java.lang.IllegalArgumentException`.

9.4.9 Dependent value classes

A dependent value class can only be the value of a cmp-field, not a cmr-field. A dependent value class cannot have a member that is a dependent object class. A dependent value class can always be assigned to a cmp-field of the corresponding dependent value class type. The get accessor method for a cmp-field that corresponds to a dependent value class returns a copy of the dependent value class instance. The assignment of a dependent value class value to a cmp-field causes the value to be copied to the target cmp-field.

Dependent value classes that are referred to through container managed persistent fields must be serializable.

Descriptors for dependent value classes must not be specified in the deployment descriptor.

9.4.10 Non-persistent state

The Bean Provider may use instance variables in the entity bean instance to maintain the non-persistent state of the entity bean, e.g., a JMS connection.

The Bean Provider can use instance variables to store values that depend on the persistent state of the entity bean or its dependent objects, although this use is not encouraged. The Bean Provider must use the `ejbLoad()` method to resynchronize the values of any instance variables that depend on the entity bean's persistent state, such as references to dependent objects or collections of dependent objects. In general, any non-persistent state that depends on the persistent state of an entity bean or its dependent objects should be recomputed during the `ejbLoad()` method.

9.4.11 The relationship between the persistence view and the client view

The Enterprise JavaBeans architecture defines a component model in which the client view of the component hides the details of the internal implementation of the enterprise bean class. Typically, after designing the client view of an entity bean, the bean provider will design the internals of the component, deciding on its abstract persistence schema. The classes that are exposed by the remote interface of the bean may or may not be related to the classes that require persistence. In designing the abstract persistence schema of the bean, the Bean Provider should therefore keep in mind the following:

- The classes and relationships that are exposed by the remote interface are decoupled from the persistence layer. Instances of these classes are passed to and from the client *by value*.
- The classes and relationships that are defined in the abstract persistence schema are *persistent* in nature. The concrete representation of these classes and relationships (including the Collection types) is determined by the persistence manager that is used in the given deployment environment.
- Because the persistence manager is free to optimize the delivery of persistent data to the bean instance (for example, by the use of lazy loading strategies), the instances of the dependent object classes that are defined in the abstract persistence schema and the contents of collections managed by the persistence manager may not be fully materialized.

The Bean Provider must not expose the dependent object classes or the persistent `Collection` classes that are used in container managed relationships through the remote interface of the bean.

This means that the get and set methods of the entity bean's abstract persistence schema must not be exposed through the remote interface of the entity bean except in the following cases:

- When the relationship is defined as a one-to-one or many-to-one relationship between two entity beans.
- When the get and set accessor methods are methods for a cmp-field. Set accessor methods corresponding to primary key fields, however, should not be exposed in the remote interface of the bean.

Dependent value classes *can* be exposed in the remote interface and can be included in the client ejb-jar file.

The Bean Provider is also free to expose get and set methods that do not correspond to cmp-fields or cmr-fields.

Although dependent object classes cannot be exposed in the remote interface, the Bean Provider can use the accessor methods to obtain instances of these persistent classes (including the collection classes that correspond to relationships), and can copy data to or from these instances to instances of the classes that are exposed in the remote interface.

9.4.12 Mapping data to a persistent store

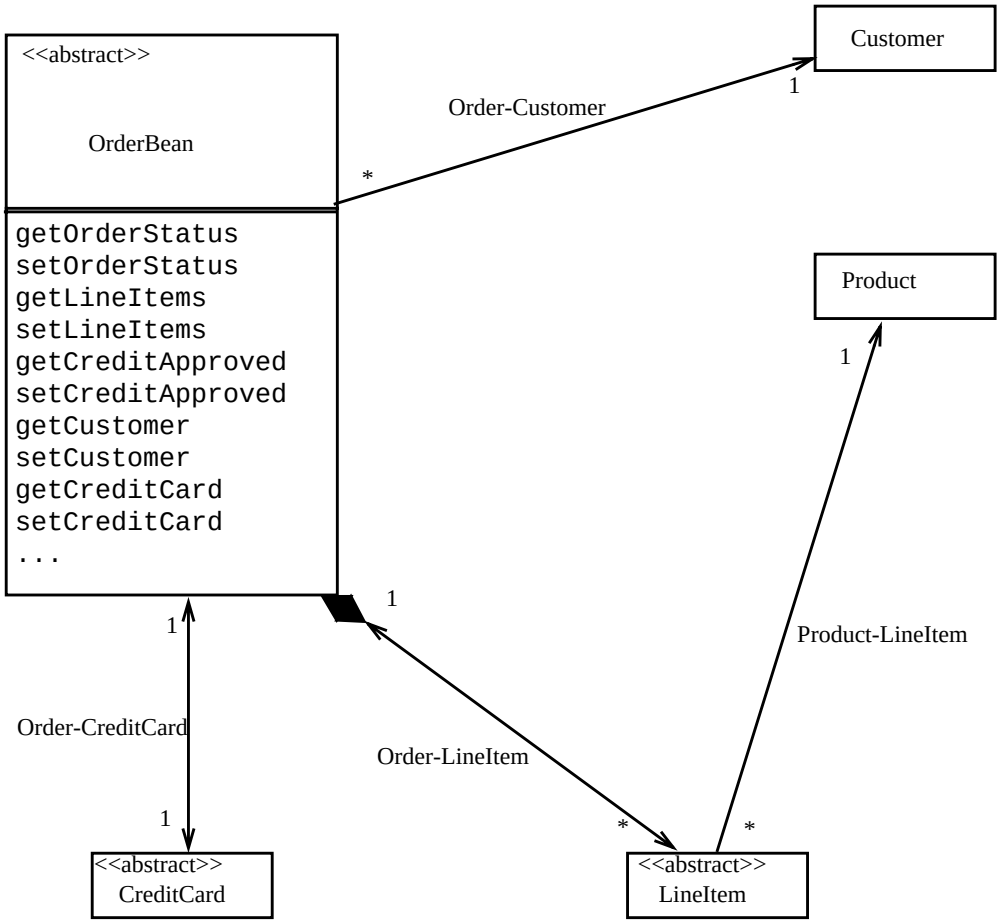
This specification does not prescribe how the abstract persistence schema of an entity bean or dependent object class should be mapped to a relational (or other) schema of a persistent store, or define how such a mapping is described.

If the Bean Provider needs to characterize the mapping between an abstract persistence schema and a database schema, we recommend the convention that an auxiliary deployment descriptor be created in the same directory as the EJB 2.0 deployment descriptor and that the ID mechanism be used to make references to the elements of the EJB 2.0 deployment descriptor in order to add information where needed. See Section 21.5.

9.4.13 Example

Figure 21 illustrates an entity bean `Order` with relationships to line items, credit cards, customers and products. It shows the abstract persistence schema and relationships. The accessor methods for the dependent object classes are not shown. Sample code for the `OrderBean` class follows the figure.

Figure 21 Relationship example



The sample code below illustrates how the relationships and the accessor methods of the Order entity bean's abstract persistence schema are used. The code also illustrates the use of the dependent object classes `LineItem` and `CreditCard`. The dependent value class `ClientLineItem` is used only in the client view. The dependent value class `AddressToShip` is used in both the client view and as the value of a cmp-field of `LineItem`.

```
package com.acme.order;

import java.util.Collection;
import java.util.Vector;
import java.util.Date;
...

public abstract class OrderBean implements javax.ejb.EntityBean{

    private EntityContext context;

// define status codes for processing

     static final int BACKORDER = 1;
     static final int SHIPPED = 2;
     static final int UNSHIPPED = 3;

//
    ....

// getters and setters for the cmp fields


    public abstract int getOrderStatus();
    public abstract void setOrderStatus(int orderStatus);

    public abstract boolean getCreditApproved();
    public abstract void setCreditApproved(boolean creditapproved);

    public abstract Date getOrderDate();
    public abstract void setOrderDate(Date orderDate);


// getters and setters for the relationship fields

    public abstract Collection getLineItems();
    public abstract void setLineItems(Collection lineitems);

    public abstract Customer getCustomer();
    public abstract void setCustomer(Customer customer);

    public abstract CreditCard getCreditCard();
    public abstract void setCreditCard(CreditCard creditcard);

// methods to create instances of dependent object classes


     public abstract CreditCard createCreditCard(String num,
         String type) throws javax.ejb.CreateException;
     public abstract LineItem createLineItem()
                   throws javax.ejb.CreateException;
```

```
// business methods.

    // Remote method setCustomerForOrder:

    public void setCustomerForOrder(Customer customer,
                            String creditcardNum,
                            String creditCardType,
                            String expiration)
         throws InvalidCreditCardException{

    // CreditCard is a dependent object created by the entity bean

         try {
              CreditCard card = createCreditCard(creditcardNum,
                                          creditCardType);
              card.setExpires(convertToDate(expiration));
              card.setCustomer(customer);
              setCustomer(customer);
              setCreditCard(card);
         } catch (javax.ejb.CreateException e) {
              throw new InvalidCreditCardException();
         }

    }


    // remote method addLineItem:

    // This method is used to add a line item.
    // Internally the bean code creates the persistent dependent
    // object and adds it to the collection of line items that
    // are already created.

    public void addLineItem(Product product,
                            integer quantity,
                            AddressToShip address)
        throws InsufficientInfoException, LineItemCreateException{

        // create a new line item

        if (validAddress(address)) {
              // AddressToShip is a legacy class. It is a dependent
              // value class that is available both in the client and
              // in the entity bean and that is serializable.
              // AddressToShip is used as the value of a cmp-field
              // of LineItem.
          try {
            LineItem litem = createLineItem();
          } catch (javax.ejb.CreateException e) {
              throw new LineItemCreateException();
              }
              litem.setProduct(product);
            litem.setQuantity(quantity);
            litem.setTax(calculateTax(product.getPrice(),
                                      quantity,
                                      address));
            litem.setStatus(UNSHIPPED);
              //set the address for the line item to be shipped
```

```
            litem.setAddress(address);

              // The entity bean uses a special dependent value class
              // to represent the dates related to order status.
              // This class holds shipment date, expected shipment
              // date, credit approval date, and inventory dates which
              // are internal to the order fulfillment process.
              // Not all information represented in this class
              // will be available to the client when
              // the client requests to see the line items.
            Dates dates = new Dates();
            litem.setDates(dates);
            getLineItems().add(litem);
        } else {
            throw new InsufficientInfoException();
        }
    }

    // remote method getOrderLineItems:

    // This method returns a view of the line items in the
    // order to the client. It makes only relevant
    // information visible to the client and hides the
    // internal details of the representation of the line items

    public Collection getOrderLineItems() {
        Vector clientlineitems = new Vector();
        Collection  lineitems = getLineItems();
        java.util.Iterator iterator = lineitems.iterator();

         // ClientLineItem is a value class that is available in the
         // client and represents the client view of the persistent
         // dependent object LineItem. It is not an abstract class.
         // The entity bean provider abstracts information from the
         // persistent representation of the line item to construct
         // the client view

        ClientLineItem clitem;

         while (iterator.hasNext()) {
              LineItem litem = (LineItem)iterator.next();
              clitem = new ClientLineItem();
              // Only the name of the product is available in the
              // client view

              clitem.setProduct(litem.getProduct().getName());
              clitem.setQuantity(litem.getQuantity());

              // The client view gets a specific descriptive message
              // depending on the line item status.
              clitem.setCurrentStatus(
                   statusCodeToString(litem.getStatus()));

              // Address is not copied to the client view.
              // as this class includes other information with
              // respect to the order handing that should not be
              // available to the client. Only the relevant info
              // is copied.
```

```
                int lineitemStatus = litem.getStatus();
                if ( lineitemStatus == BACKORDER) {
                   clitem.setShipDate(
                     litem.getDates().getExpectedShipDate());
                } else if (lineitemStatus == SHIPPED) {
                    clitem.setShipDate(
                         litem.getDates().getShippedDate());
              }

                //Add the new line item
                clientlineitems.add(clitem);
          }
          // Return the value objects to the client
          return clientlineitems;

      }

       // other methods
}
```

9.4.14 The Bean Provider's view of the deployment descriptor

The deployment descriptor provides information about the relationships among beans and dependent object classes and their abstract persistence schemas.

The persistent fields (cmp-fields) of both entity beans and dependent object classes, as well as the relationships in which entity beans and dependent object classes participate (cmr-fields), must be declared in the deployment descriptor.

The deployment descriptor provides the following information about entity beans and relationships:

- An `ejb-name` element for each entity bean. The `ejb-name` must be unique within the ejb-jar file.
- A set of `dependent` elements, which describe the dependent object classes that participate in container managed relationships. The cmp-fields of dependent object classes must be declared in the deployment descriptor and each `dependent` element must be given a `dependent-name`. The `dependent-name` must be unique within the ejb-jar file and must not be the same as any `ejb-name` within the ejb-jar file. The Bean Provider may optionally designate the primary fields for the dependent object using the `pk-field` elements.
- A set of `ejb-entity-ref` elements, which describe remote entity beans and which provide names for remote entity beans by means of `remote-ejb-name` elements. Remote entity beans are the entity beans that participate in relationships but whose abstract persistence schemas are not available in the ejb-jar file. This includes entity beans with bean managed persistence, EJB 1.1 entity beans with container managed persistence, and entity beans that are defined in another ejb-jar file. The `ejb-entity-ref` element provides a named abstraction on top of the `ejb-ref` element. A `remote-ejb-name` element must be unique within the ejb-jar file and must not be the same as any `ejb-name` or `dependent-name` within the ejb-jar file.

- A set of `ejb-relation` elements, each of which contains a pair of `ejb-relationship-role` elements to describe the two roles in the relationship.[11]

- Each `ejb-relationship-role` element describes a relationship role: its name, its multiplicity within a relation, and its navigability. It specifies the name of the `cmr-field` that is used from the perspective of the relationship participant (a bean or its dependent object class). Each relationship role refers to an entity bean or a dependent object class by means of an `ejb-name`, `remote-ejb-name` or `dependent-name` element contained in the `role-source` element. The Bean Provider must ensure that the content of each `role-source` element refers to an existing entity bean, entity bean reference, or dependent object class.

[11] The relation names and the relationship role names are not used in the code provided by the bean provider.

The following example shows a deployment descriptor segment that defines the abstract persistence schema for a set of related entity beans and dependent object classes. The deployment descriptor elements for container managed persistence and relationships are described further in Chapter 21.

```
<ejb-jar>

...

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

<dependents>
     <dependent>
          <description>Line item dependent class</description>
          <dependent-class>
          com.acme.order.LineItem
          </dependent-class>
          <dependent-name>LineItem</dependent-name>
          <cmp-field><field-name>quantity</field-name></cmp-field>
          <cmp-field><field-name>tax</field-name></cmp-field>
          <cmp-field><field-name>status</field-name></cmp-field>
          <cmp-field><field-name>address</field-name></cmp-field>
     </dependent>
     <dependent>
          <description>CreditCard dependent class </description>
          <dependent-class>com.acme.order.CreditCard
          </dependent-class>
          <dependent-name>CreditCard</dependent-name>
          <cmp-field><field-name>number</field-name></cmp-field>
          <cmp-field><field-name>type</field-name></cmp-field>
          <cmp-field><field-name>approved</field-name></cmp-field>
          <cmp-field><field-name>expires</field-name></cmp-field>
          <pk-field>number</pk-field>
     </dependent>
</dependents>

<relationships>
     <!-- Since OrderEJB and CustomerEJB are entity beans whose
abstract persistence schemas are included in this ejb-jar file, there
is no need to supply an ejb-entity-ref element to refer to them in the
relationships. Instead the relationships use the ejb-names of
OrderEJB and CustomerEJB to specify the role source. ProductEJB, how-
ever, is remote, and an ejb-entity-ref element must therefore be spec-
ified so that it can be used in relationships.
-->


     <ejb-entity-ref>
          <description>
          This is a reference descriptor for a Product bean
          </description>
          <remote-ejb-name>ProductEJB</remote-ejb-name>
          <home>com.commercewarehouse.catalog.ProductHome</home>
          <remote>com.commercewarehouse.catalog.Product</remote>
          <ejb-link>Product</ejb-link>
     </ejb-entity-ref>
```

```
<!--
ONE-TO-MANY: Order LineItem
-->

     <ejb-relation>
          <ejb-relation-name>Order-LineItem</ejb-relation-name>
          <ejb-relationship-role>
               <ejb-relationship-role-name>
               order-has-lineitems
               </ejb-relationship-role-name>
               <multiplicity>One</multiplicity>
               <role-source>
                    <ejb-name>OrderEJB</ejb-name>
               </role-source>
               <cmr-field>
                    <cmr-field-name>lineItems</cmr-field-name>
                    <cmr-field-type>java.util.Collection
                    </cmr-field-type>
               </cmr-field>
          </ejb-relationship-role>

          <ejb-relationship-role>
               <ejb-relationship-role-name>lineitem_belongsto_order
               </ejb-relationship-role-name>
               <multiplicity>Many</multiplicity>
               <cascade-delete/>
               <role-source>
                    <dependent-name>LineItem<dependent-name>
               </role-source>
               <cmr-field>
                    <cmr-field-name>order</cmr-field-name>
               </cmr-field>
          </ejb-relationship-role>
     </ejb-relation>

<!--
ONE-TO-MANY unidirectional relationship:
Product is not aware of its relationship with LineItem
-->

     <ejb-relation>
          <ejb-relation-name>Product-LineItem</ejb-relation-name>

          <ejb-relationship-role>
               <ejb-relationship-role-name>
               product-has-lineitems
               </ejb-relationship-role-name>
               <multiplicity>One</multiplicity>
               <role-source>
                    <ejb-name>ProductEJB</ejb-name>
               </role-source>
                  <!-- since Product does not know about LineItem
                       there is no cmr field in Product for accessing
                         Lineitem
                  -->
          </ejb-relationship-role>

          <ejb-relationship-role>
```

```
                <ejb-relationship-role-name>
                line-item-product
                </ejb-relationship-role-name>
                <multiplicity>Many</multiplicity>
                <role-source>
                     <dependent-name>LineItem</dependent-name>
                </role-source>
                <cmr-field>
                     <cmr-field-name>product</cmr-field-name>
                </cmr-field>
           </ejb-relationship-role>

     </ejb-relation>


<!--
ONE-TO-MANY: Order Customer:
-->

     <ejb-relation>
          <ejb-relation-name>Order-Customer</ejb-relation-name>

          <ejb-relationship-role>
               <ejb-relationship-role-name>
               customer-has-orders
               </ejb-relationship-role-name>
               <multiplicity>One</multiplicity>
               <role-source>
                    <ejb-name>CustomerEJB</ejb-name>
               </role-source>
               <cmr-field>
                    <cmr-field-name>orders</cmr-field-name>
                    <cmr-field-type>java.util.Collection
                    </cmr-field-type>
               </cmr-field>
          </ejb-relationship-role>

          <ejb-relationship-role>
               <ejb-relationship-role-name>
               order-belongsto-customer
               </ejb-relationship-role-name>
               <multiplicity>Many</multiplicity>
               <role-source>
                    <ejb-name>OrderEJB
                    </ejb-name>
               </role-source>
               <cmr-field>
                    <cmr-field-name>customer</cmr-field-name>
               </cmr-field>
          </ejb-relationship-role>

     </ejb-relation>


<!--
ONE-TO-ONE: Order CreditCard:
-->

     <ejb-relation>
```

```
            <ejb-relation-name>Order-CreditCard</ejb-relation-name>

            <ejb-relationship-role>
                 <ejb-relationship-role-name>order-paidby-creditcard
                 </ejb-relationship-role-name>
                 <multiplicity>One</multiplicity>
                 <role-source>
                      <ejb-name>OrderEJB</ejb-name>
                 </role-source>
                 <cmr-field>
                      <cmr-field-name>creditcard</cmr-field-name>
                 </cmr-field>
            </ejb-relationship-role>

            <ejb-relationship-role>
                 <ejb-relationship-role-name>creditcard-pays-order
                 </ejb-relationship-role-name>
                 <multiplicity>One</multiplicity>
            <role-source>
                 <dependent-name>CreditCard</dependent-name>
            </role-source>
            <cmr-field>
                 <cmr-field-name>order</cmr-field-name>
            </cmr-field>
            </ejb-relationship-role>

       </ejb-relation>

  </relationships>

  ...

  </ejb-jar>
```

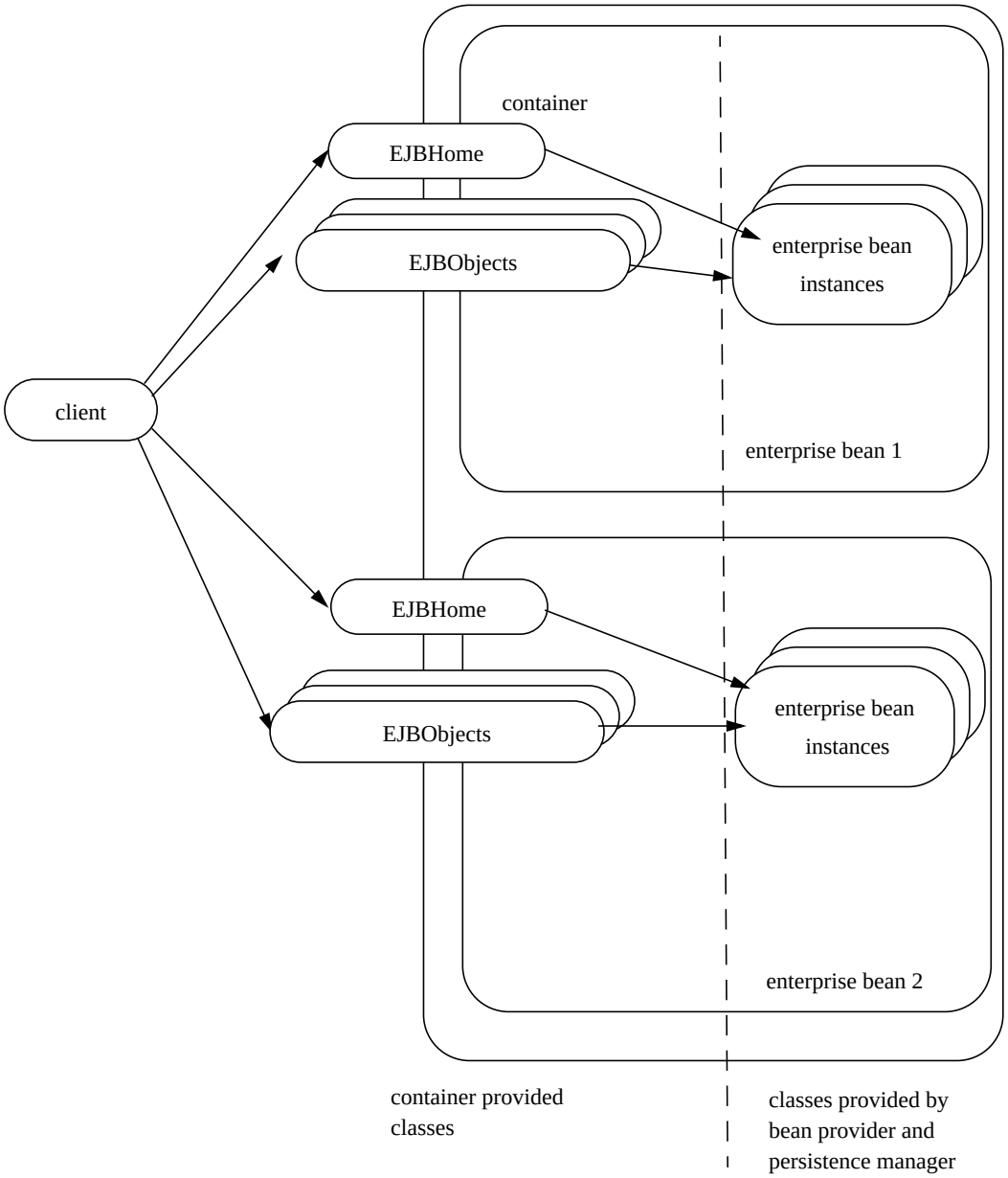
9.5 The entity bean component contract

This section specifies the container managed persistence contract between an entity bean, its container, and its persistence manager.

9.5.1 Runtime execution model of entity beans

This subsection describes the runtime model and the classes used in the description of the contract between an entity bean, its container, and its persistence manager. Figure 22 shows an overview of the runtime model.

Figure 22 Overview of the entity bean runtime execution model



An **enterprise bean** is an object whose class is provided by the Bean Provider. The class of an entity bean with container managed persistence is abstract; the concrete bean class is generated by the persistence manager provider's tools at deployment time.

An entity **EJBObject** is an object whose class was generated at deployment time by the Container Provider's tools. The entity EJBObject class implements the entity bean's remote interface. A client never references an entity bean instance directly—a client always references an entity EJBObject whose class is generated by the Container Provider's tools.

An entity **EJBHome** object provides the life cycle operations (create, remove, find) for its entity objects as well as home business methods, which are business methods that are not specific to an entity bean instance. The class for the entity EJBHome object is generated by the Container Provider's tools at deployment time. The entity EJBHome object implements the entity bean's home interface that was defined by the Bean Provider.

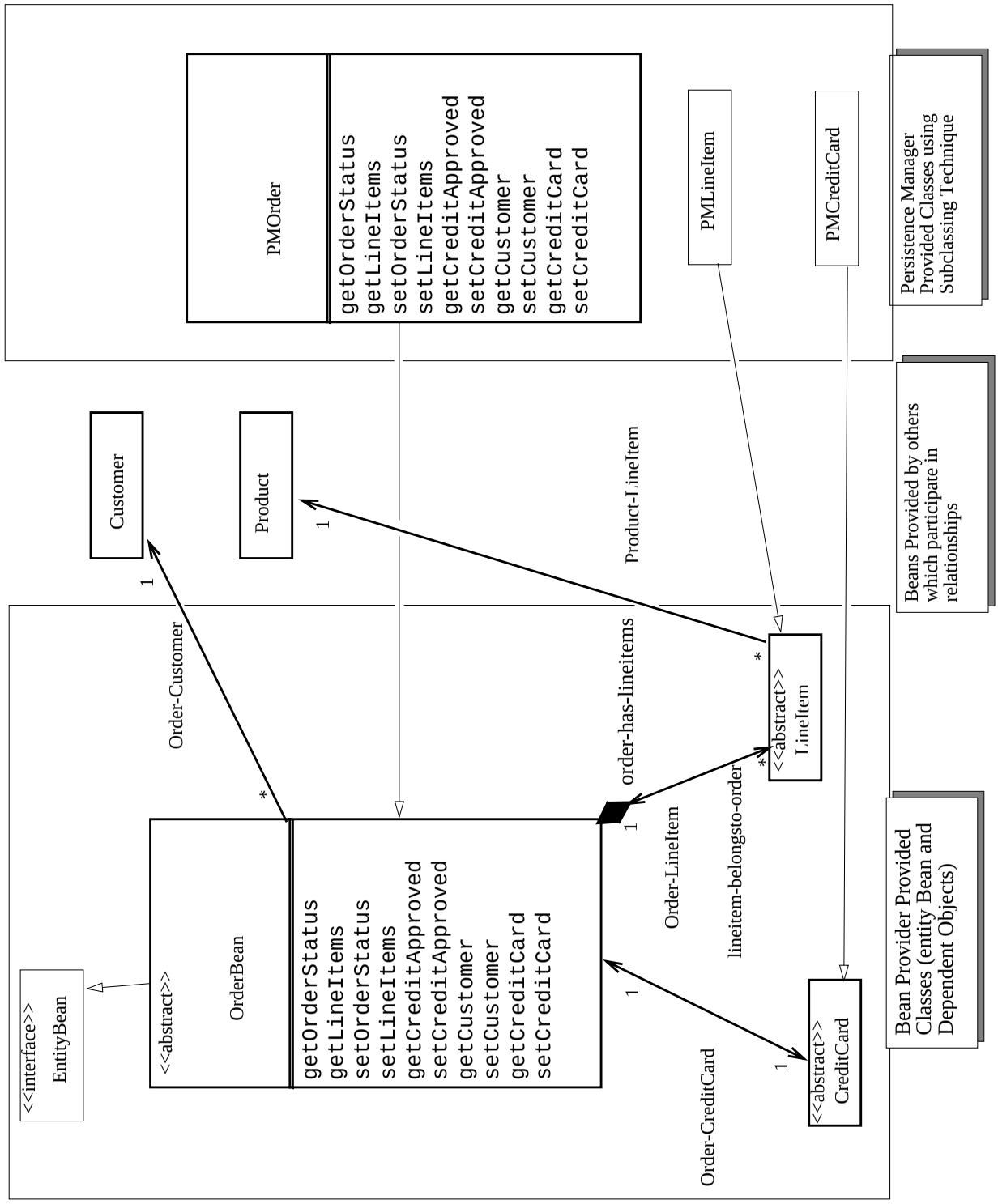
9.5.2 Relationships among the classes provided by the bean provider and persistence manager

The entity bean provider is responsible for providing the entity bean class as an abstract class. Any dependent object classes are also provided by the entity bean provider as abstract classes. The persistence manager provider tools are responsible for providing the concrete entity bean and dependent object classes.

The classes provided by the persistence manager tools are responsible for managing the relationships of the entity beans and dependent object classes and for managing the access to their persistent state. The persistence manager provider tools are also responsible for providing the implementations of the `java.util.Collection` classes that are used in maintaining the container managed relationships.

Figure 23 illustrates these relationships.

Figure 23 Relationships among the classes



9.5.3 Persistence Manager responsibilities

9.5.3.1 Container-managed fields

An entity bean with container managed persistence relies on the Persistence Manager Provider's tools to generate methods that perform persistent data access on behalf of the entity bean instances. The generated methods transfer data between an entity bean instance and the underlying resource manager. The generated methods also implement the creation, removal, and lookup of the entity object in the underlying database.

The Persistence Manager is responsible for implementing the entity bean classes and dependent object classes by providing the implementation of the get and set accessor methods of their abstract persistence schemas. In order to implement these methods, the Persistence Manager must also manage the mapping between primary keys or handles and EJBObjects. The Persistence Manager must be capable of persisting references to enterprise bean remote and home interfaces (for example, by storing primary keys or handles). The Persistence Manager Provider is allowed to use Java serialization to store the container-managed fields.

The Persistence Manager is responsible for transferring data between the entity bean and the underlying data source. The data is transferred by the Persistence Manager as a result of the execution of the entity bean's methods. Because of the requirement that all data access occur through the accessor methods, the Persistence Manager can implement both eager and lazy loading and storing schemes. Synchronization between the Persistence Manager and the Container is achieved before or after the execution of the `ejbCreate<METHOD>`, `ejbRemove`, `ejbLoad`, and `ejbStore` methods. These contracts are described in Section 9.6.

9.5.3.2 Container-managed relationships

The Persistence Manager maintains the relationships among beans and dependent object classes.

- For a relationship with a bean, it is the responsibility of the Persistence Manager to materialize the remote object for the bean based on the information provided in the deployment descriptor. This eliminates the need for the bean provider to look up and execute a finder method for the related bean.

- It is the responsibility of the Persistence Manager to maintain the referential integrity of the container managed relationships in accordance with the semantics of the relationship type as specified in the deployment descriptor. For example, if a bean (or dependent object) is added to a collection corresponding to the container managed relationship field of another bean (or dependent object), the container managed relationship field of the first bean (or dependent object) must also be updated by the Persistence Manager in the same transaction context. Section 9.4.6 describes this semantics.

- It is the responsibility of the Persistence Manager to throw the `java.lang.IllegalArgumentException` when the argument to a set method in a relationship is a collection con taining instances of the wrong type, or when an argument to a method of the `java.util.Collection` API used to manipulate a collection-valued container managed relationship field is an instance of the wrong type or a collection that contains instances of the wrong type (see Section 9.4.6).

9.5.3.3 Container-managed dependent object classes

The Persistence Manager must provide the implementation of the dependent object classes that the bean provider provides as abstract classes.

The Persistence Manager must implement the `create<METHOD>(...)` methods that are defined on the entity bean's class and dependent object classes to enable the Bean Provider to create instances of dependent object classes at runtime. It is the responsibility of the Persistence Manager to ensure that the values that will be initially returned by the dependent object class instance's get methods for container-managed fields will be the Java language defaults (e.g. 0 for integer, `null` for pointers), except for collection-valued cmr-fields, which must have the empty collection (or set) as their value, and that any non-persistent fields in the dependent object are set to their Java language defaults.

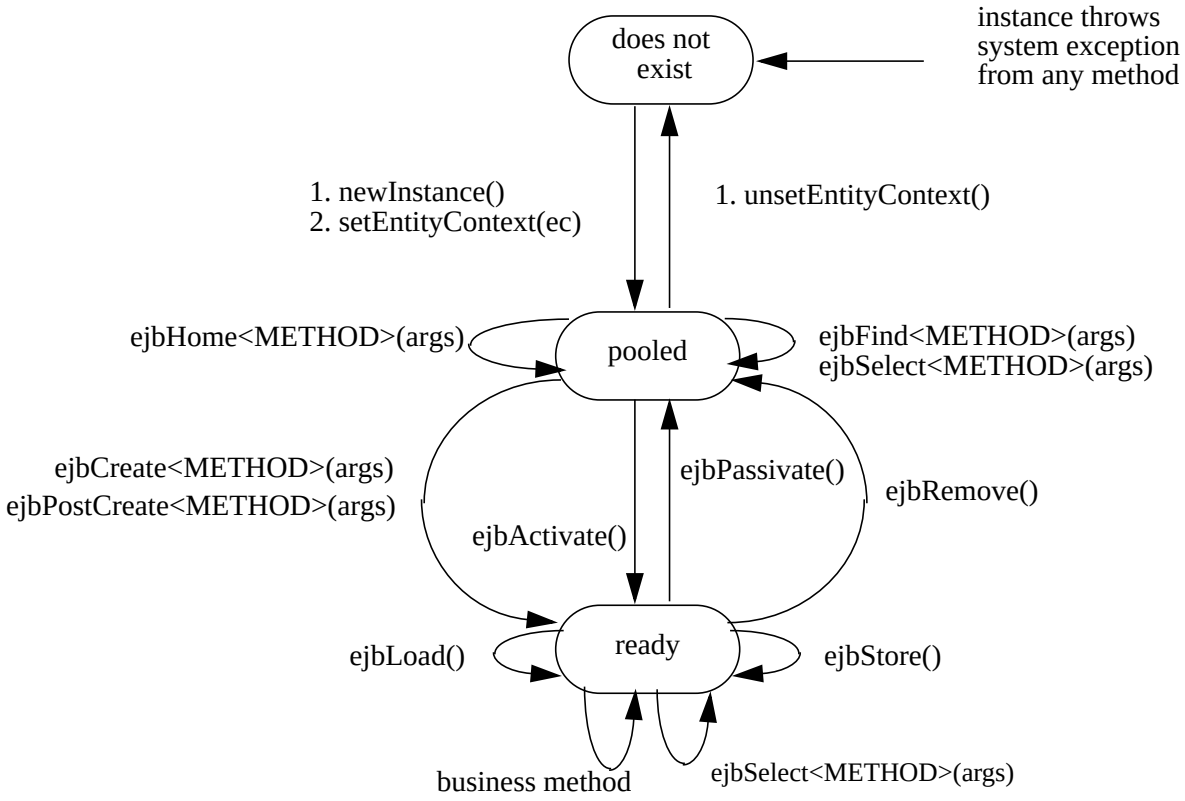
It is the responsibility of the Persistence Manager to implement the `remove()` methods that are defined on the dependent object classes to enable the Bean Provider to delete instances of dependent object classes at runtime.

9.6 Instance life cycle contract between the bean, the container, and the persistence manager

This section describes the part of the component contract between the entity bean, the container, and the persistence manager that relates to the management of the entity bean instance's lifecycle.

9.6.1 Instance life cycle

Figure 24 Life cycle of an entity bean instance.



An entity bean instance is in one of the following three states:

- It does not exist.
- Pooled state. An instance in the pooled state is not associated with any particular entity object identity.
- Ready state. An instance in the ready state is assigned an entity object identity.

The following steps describe the life cycle of an entity bean instance:

- An entity bean instance's life starts when the container creates the instance using `newInstance()`. The container then invokes the `setEntityContext()` method to pass the instance a reference to the `EntityContext` interface. The `EntityContext` interface

allows the instance to invoke services provided by the container and to obtain the information about the caller of a client-invoked method.

- The instance enters the pool of available instances. Each entity bean has its own pool. While the instance is in the available pool, the instance is not associated with any particular entity object identity. All instances in the pool are considered equivalent, and therefore any instance can be assigned by the container to any entity object identity at the transition to the ready state. While the instance is in the pooled state, the container may use the instance to execute any of the entity bean's finder methods (shown as `ejbFind<METHOD>(...)` in the diagram) or any of the entity bean's home methods (shown `ejbHome<METHOD>(...)` in the diagram). The instance does **not** move to the ready state during the execution of a finder or a home method. An `ejbSelect<METHOD>(...)` method may be called by an entity bean's home method while the instance is in the pooled state.

- An instance transitions from the pooled state to the ready state when the container selects that instance to service a client call to an entity object. There are two possible transitions from the pooled to the ready state: through the `ejbCreate<METHOD>(...)` and `ejbPostCreate<METHOD>(...)` methods, or through the `ejbActivate()` method. The container invokes the `ejbCreate<METHOD>(...)` and `ejbPostCreate<METHOD>(...)` methods when the instance is assigned to an entity object during entity object creation (i.e., when the client invokes a create method on the entity bean's home object). The container invokes the `ejbActivate()` method on an instance when an instance needs to be activated to service an invocation on an existing entity object—this occurs because there is no suitable instance in the ready state to service the client's call.

- When an entity bean instance is in the ready state, the instance is associated with a specific entity object identity. While the instance is in the ready state, the container can synchronize the state of the instance with the state of the entity in the underlying data source whenever it determines the need to, in the process invoking the `ejbLoad()` and `ejbStore()` methods zero or more times. A business method can be invoked on the instance zero or more times. Invocations of the `ejbLoad()` and `ejbStore()` methods can be arbitrarily mixed with invocations of business methods. An `ejbSelect<METHOD>` method can be called by a business method (or `ejbLoad()` or `ejbStore()` method) while the instance is in the ready state.

- The container can choose to passivate an entity bean instance within a transaction. To passivate an instance, the container first invokes the `ejbStore` method to allow the instance to prepare itself for the synchronization of the database state with the instance's state, and to allow the persistence manager to store the instance's state to the database, and then the container invokes the `ejbPassivate` method to return the instance to the pooled state.

- Eventually, the container will transition the instance to the pooled state. There are three possible transitions from the ready to the pooled state: through the `ejbPassivate()` method, through the `ejbRemove()` method, and because of a transaction rollback for `ejbCreate()`, `ejbPostCreate()`, or `ejbRemove()` (not shown in Figure 24). The container invokes the `ejbPassivate()` method when the container wants to disassociate the instance from the entity object identity without removing the entity object. The container invokes the `ejbRemove()` method when the container is removing the entity object (i.e., when the client invoked the `remove()` method on the entity object's remote interface or one of the `remove()` methods on the entity bean's home interface). If `ejbCreate()`, `ejb-`

`PostCreate()`, or `ejbRemove()` is called and the transaction rolls back, the container will transition the bean instance to the pooled state.

- When the instance is put back into the pool, it is no longer associated with an entity object identity. The container can assign the instance to any entity object within the same entity bean home.

- The container can remove an instance in the pool by calling the `unsetEntityContext()` method on the instance.

Notes:

1. The `EntityContext` interface passed by the container to the instance in the `setEntityContext` method is an interface, not a class that contains static information. For example, the result of the `EntityContext.getPrimaryKey()` method might be different each time an instance moves from the pooled state to the ready state, and the result of the `getCallerPrincipal()` and `isCallerInRole(...)` methods may be different in each business method.

2. A `RuntimeException` thrown from any method of the entity bean class (including the business methods and the callbacks invoked by the container) results in the transition to the "does not exist" state. The container must not invoke any method on the instance after a `RuntimeException` has been caught. From the client perspective, the corresponding entity object continues to exist. The client can continue accessing the entity object through its remote interface because the container can use a different entity bean instance to delegate the client's requests. Exception handling is described further in Chapter 17.

3. The container is not required to maintain a pool of instances in the pooled state. The pooling approach is an example of a possible implementation, but it is not the required implementation. Whether the container uses a pool or not has no bearing on the entity bean coding style.

9.6.2 Bean Provider's entity bean instance's view

The following describes the entity bean instance's view of the contract *as seen by the Bean Provider*:

The entity Bean Provider is responsible for implementing the following methods in the abstract entity bean class:

- A public constructor that takes no arguments.

- `public void setEntityContext(EntityContext ic);`

  A container uses this method to pass a reference to the `EntityContext` interface to the entity bean instance. If the entity bean instance needs to use the `EntityContext` interface during its lifetime, it must remember the `EntityContext` interface in an instance variable.

  This method executes with an unspecified transaction context (Refer to Subsection 16.6.5 for how the Container executes methods with an unspecified transaction context). An identity of an entity object is not available during this method. The entity bean must not attempt to access

its persistent state using the accessor methods or attempt to create instances of dependent object classes during this method.

The instance can take advantage of the `setEntityContext()` method to allocate any resources that are to be held by the instance for its lifetime. Such resources cannot be specific to an entity object identity because the instance might be reused during its lifetime to serve multiple entity object identities.

- `public void unsetEntityContext();`

  A container invokes this method before terminating the life of the instance.

  This method executes with an unspecified transaction context. An identity of an entity object is not available during this method. The entity bean must not attempt to access its persistent state using the accessor methods or attempt to create instances of dependent object classes during this method.

  The instance can take advantage of the `unsetEntityContext()` method to free any resources that are held by the instance. (These resources typically had been allocated by the `setEntityContext()` method.)

- `public` *`PrimaryKeyClass`* `ejbCreate<METHOD>(...);`

  There are zero[12] or more `ejbCreate<METHOD>(...)` methods, whose signatures match the signatures of the `create<METHOD>(...)` methods of the entity bean home interface. The container invokes an `ejbCreate<METHOD>(...)` method on an entity bean instance when a client invokes a matching `create<METHOD>(...)` method to create an entity object.

  The entity Bean Provider's responsibility is to initialize the instance in the `ejbCreate<METHOD>(...)` methods from the input arguments, using the get and set accessor methods, such that when the `ejbCreate<METHOD>(...)` method returns, the persistent representation of the instance can be created. The entity Bean Provider is guaranteed that the values that will be initially returned by the instance's get methods for container managed fields will be the Java language defaults (e.g. 0 for integer, `null` for pointers), except for collection-valued cmr-fields, which will have the empty collection (or set) as their value. The entity Bean Provider must not attempt to modify the values of cmr-fields or create instances of dependent object classes in an `ejbCreate<METHOD(...)` method; this should be done in the `ejbPostCreate<METHOD(...)` method instead.

  The entity object created by the `ejbCreate<METHOD>` method must have a unique primary key. This means that the primary key must be different from the primary keys of all the existing entity objects within the same home. However, it is legal to reuse the primary key of a previously removed entity object. The `ejbCreate<METHOD>(...)` methods must be defined to return the primary key class type. The implementation of the Bean Provider's `ejbCreate<METHOD>(...)` methods should be coded to return a `null`.[13]

  An `ejbCreate<METHOD>(...)` method executes in the transaction context determined by the transaction attribute of the matching `create<METHOD>(...)` method, as described in

[12] An entity enterprise Bean has no *ejbCreate*`<METHOD>`*(...)* and *ejbPostCreate*`<METHOD>`*(...)* methods if it does not define any create methods in its home interface. Such an entity enterprise Bean does not allow the clients to create new EJB objects. The enterprise Bean restricts the clients to accessing entities that were created through direct database inserts.

[13] The above requirement is to allow the creation of an entity bean with bean-managed persistence by subclassing an entity bean with container-managed persistence. The Java language rules for overriding methods in subclasses requires the signatures of the *ejbCreate*`<METHOD>`*(...)* methods in the subclass and the superclass to be the same.

subsection 16.6.2. The database insert operations are performed by the persistence manager within the same transaction context after the Bean Provider's `ejbCreate<METHOD>(...)` method completes.

- `public void ejbPostCreate<METHOD>(...);`

  For each `ejbCreate<METHOD>(...)` method, there is a matching `ejbPostCreate<METHOD>(...)` method that has the same input parameters but whose return value is `void`. The container invokes the matching `ejbPostCreate<METHOD>(...)` method on an instance after it invokes the `ejbCreate<METHOD>(...)` method with the same arguments. The entity Bean Provider is guaranteed that the persistent representation of the entity has been created before the `ejbPostCreate<METHOD>(...)` method has been called. The instance can discover the primary key by calling `getPrimaryKey()` on its entity context object.

  The entity object identity is available during the `ejbPostCreate<METHOD>(...)` method. The instance may, for example, obtain the remote interface of the associated entity object and pass it to another enterprise bean as a method argument.

  The entity Bean Provider may use the `ejbPostCreate<METHOD>(...)` to create instances of dependent object classes and set the values of cmr-fields to complete the initialization of the entity bean instance.

  An `ejbPostCreate<METHOD>(...)` method executes in the same transaction context as the previous `ejbCreate<METHOD>(...)` method.

- `public void ejbActivate();`

  The container invokes this method on the instance when the container picks the instance from the pool and assigns it to a specific entity object identity. The `ejbActivate()` method gives the entity bean instance the chance to acquire additional resources that it needs while it is in the ready state.

  This method executes with an unspecified transaction context. The entity bean must not attempt to access its persistent state using the accessor methods or attempt to create instances of dependent object classes during this method.

  The instance can obtain the identity of the entity object via the `getPrimaryKey()` or `getEJBObject()` method on the entity context. The instance can rely on the fact that the primary key and entity object identity will remain associated with the instance until the completion of `ejbPassivate()` or `ejbRemove()`.

- `public void ejbPassivate()`;

  The container invokes this method on an instance when the container decides to disassociate the instance from an entity object identity, and to put the instance back into the pool of available instances. The `ejbPassivate()` method gives the instance the chance to release any resources that should not be held while the instance is in the pool. (These resources typically had been allocated during the `ejbActivate()` method.) If the Bean Provider has assigned dependent objects to instance variables (i.e., to non-container-managed fields of the bean instance), the Bean Provider should discard the references by setting the instance variables to null.

  This method executes with an unspecified transaction context. The entity bean must not attempt to access its persistent state using the accessor methods or attempt to create instances of dependent object classes during this method.

The instance can still obtain the identity of the entity object via the `getPrimaryKey()` or `getEJBObject()` method of the `EntityContext` interface.

- `public void ejbRemove();`

  The container invokes the `ejbRemove()` method on an entity bean instance in response to a client-invoked `remove` operation on the entity bean's home or remote interface. The instance is in the ready state when `ejbRemove()` is invoked and it will be entered into the pool when the method completes.

  The entity Bean Provider can use the `ejbRemove` method to implement any actions that must be done before the entity object's persistent representation is removed.

  The container synchronizes the instance's state before it invokes the `ejbRemove` method. This means that the state of the instance at the beginning of the `ejbRemove` method is the same as it would be at the beginning of a business method.

  This method and the database delete operation(s) execute in the transaction context determined by the transaction attribute of the `remove` method that triggered the `ejbRemove` method. The instance can still obtain the identity of the entity object via the `getPrimaryKey()` or `getEJBObject()` method of the `EntityContext` interface.

  After the entity Bean Provider's `ejbRemove` returns, the entity object's persistent representation is removed in the same transaction context.

  Since the instance will be entered into the pool, the state of the instance at the end of this method must be equivalent to the state of a passivated instance. This means that the instance must release any resource that it would normally release in the `ejbPassivate()` method. This means also that if the Bean Provider has assigned dependent objects to instance variables (i.e., to non-container-managed fields of the bean instance), the Bean Provider should discard the references by setting the instance variables to null.

- `public void ejbLoad();`

  When the container needs to synchronize the state of an enterprise bean instance with the entity object's persistent state, the container calls the `ejbLoad()` method.

  The entity Bean Provider can assume that the instance's persistent state has been loaded just before the `ejbLoad()` method is invoked. It is the responsibility of the Bean Provider to use the `ejbLoad()` method to recompute or initialize the values of any instance variables that depend on the entity bean's persistent state, such as references to dependent objects and collections. In general, any transient state that depends on the persistent state of an entity bean or its dependent objects should be recalculated using the `ejbLoad()` method. The entity bean can use the `ejbLoad()` method, for instance, to perform some computation on the values returned by the accessor methods (for example, uncompressing text fields).

  This method executes in the transaction context determined by the transaction attribute of the business method that triggered the `ejbLoad` method.

- `public void ejbStore();`

  When the container needs to synchronize the state of the entity object's persistent state with the state of the enterprise bean instance, the container first calls the `ejbStore()` method on the instance.

  The entity Bean Provider should use the `ejbStore()` method to update the instance using the accessor methods before its persistent state is synchronized. For example, the `ejbStore()` method may perform compression of text before the text is stored in the database.

The Bean Provider can assume that after the `ejbStore()` method returns, the persistent state of the instance is synchronized.

This method executes in the same transaction context as the previous `ejbLoad` or `ejbCreate` method invoked on the instance. All business methods invoked between the previous `ejbLoad` or `ejbCreate<METHOD>` method and this `ejbStore` method are also invoked in the same transaction context.

- `public` primary key type or collection `ejbFind<METHOD>(...)`;

  The Bean Provider of an entity bean with container managed persistence does not write the finder (`ejbFind<METHOD>(...)`) methods.

  The finder methods are generated at the entity bean deployment time using the Persistence Manager Provider's tools.

  The syntax for the Bean Provider's specification of finder methods is described in Chapter 10 "EJB QL: EJB Query Language for Container Managed Persistence Query Methods" .

- `public` type `ejbHome<METHOD>(...)`;

  The container invokes this method on the instance when the container selects the instance to execute a matching client-invoked `<METHOD>(...)` home method. The instance is in the pooled state (i.e., it is not assigned to any particular entity object identity) when the container selects the instance to execute the `ejbHome<METHOD>` method on it, and it is returned to the pooled state when the execution of the `ejbHome<METHOD>` method completes.

  The `ejbHome<METHOD>` method executes in the transaction context determined by the transaction attribute of the matching `<METHOD>(...)` home method, as described in Section 16.6.2.

  The entity bean provider provides the implementation of the `ejbHome<METHOD>(...)` method. The entity bean must not attempt to access its persistent state using the accessor methods or attempt to create instances of dependent object classes during this method because a home method is not specific to a particular bean instance.

- `public` type `ejbSelect<METHOD>(...)`;

  The Bean Provider may provide zero or more select methods. A select method is a special type of query method that is not exposed to the client in the home interface. The Bean Provider typically calls a select method within a business method.

  The Bean Provider defines the select methods as `abstract` methods.

  The select methods are generated at the entity bean deployment time using the Persistence Manager Provider's tools.

  The syntax for the specification of select methods is described in Chapter 10 "EJB QL: EJB Query Language for Container Managed Persistence Query Methods" .

  The `ejbSelect<METHOD>` method executes in the transaction context determined by the transaction attribute of the invoking business method.

9.6.3 Persistence Manager's view

The Persistence Manager is responsible for providing the concrete implementation of the entity bean class. The entity bean class provided by the Persistence Manager is the only entity bean class that is seen by the container. The Persistence Manager is free to use various techniques in its implementation of the concrete entity bean class, such as subclassing and delegation. If the Persistence Manager uses a delegation strategy rather than a simple subclassing of the abstract bean class provided by the Bean Provider, it must guarantee that the business methods defined by the abstract bean class are properly invoked.

The Persistence Manager must be aware of the lifecycle of an entity bean in order to manage the entity bean's persistent state and relationships.

The Persistence Manager is responsible for providing an implementation of a public constructor and the `ejbCreate<METHOD>`, `ejbRemove`, `ejbFind<METHOD>`, and `ejbSelect<METHOD>` methods. The Persistence Manager may additionally implement the following methods, depending on its caching strategies: `setEntityContext`, `unsetEntityContext`, `ejbPostCreate<METHOD>`, `ejbLoad`, `ejbStore`, `ejbActivate`, `ejbPassivate`. The Persistence Manager does not implement the `ejbHome<METHOD>` methods. The Persistence Manager must ensure that the methods that are provided by the Bean Provider are invoked according to this contract regardless which of these methods it implements.

The following describes the entity bean instance's view of the contract *as seen by the persistence manager's implementation of the concrete bean instance class*.

- A public constructor that takes no arguments.
- `public void setEntityContext(EntityContext ic);`

  A container uses this method to pass a reference to the `EntityContext` interface to the entity bean instance. If the entity bean instance needs to use the `EntityContext` interface during its lifetime, it must remember the `EntityContext` interface in an instance variable.

  This method executes with an unspecified transaction context (Refer to Section 16.6.5 for how the Container executes methods with an unspecified transaction context). An identity of an entity object is not available during this method.

  The instance can take advantage of the `setEntityContext()` method to allocate any resources that are to be held by the instance for its lifetime. Such resources cannot be specific to an entity object identity because the instance might be reused during its lifetime to serve multiple entity object identities.
- `public void unsetEntityContext();`

  A container invokes this method before terminating the life of the instance.

  This method executes with an unspecified transaction context. An identity of an entity object is not available during this method.

  The instance can take advantage of the `unsetEntityContext()` method to free any resources that are held by the instance. (These resources typically had been allocated by the `setEntityContext()` method.)
- `public` *`PrimaryKeyClass`* `ejbCreate<METHOD>(...);`

There are zero or more `ejbCreate<METHOD>(...)` methods, whose signatures match the signatures of the `create<METHOD>(...)` methods of the entity bean home interface. The container invokes an `ejbCreate<METHOD>(...)` method on an entity bean instance when a client invokes a matching `create<METHOD>(...)` method to create an entity object.

The Persistence Manager's `ejbCreate<METHOD>(...)` method must invoke the corresponding `ejbCreate<METHOD>(...)` method provided by the Bean Provider. Prior to invoking the `ejbCreate<METHOD>(...)` method provided by the Bean Provider, the Persistence Manager must ensure that the values that will be initially returned by the instance's get methods for container-managed fields will be the Java language defaults (e.g. 0 for integer, `null` for pointers), except for collection-valued cmr-fields, which must have the empty collection (or set) as their value.

The `ejbCreate<METHOD>(...)` methods must be defined to return the primary key class type. The entity object created by the `ejbCreate<METHOD>` method must have a unique primary key, which must be returned by the `ejbCreate<METHOD>(...)` method. This means that the primary key must be different from the primary keys of all the existing entity objects within the same home. However, it is legal to reuse the primary key of a previously removed entity object. The Persistence Manager's `ejbCreate<METHOD>(...)` method may, but is not required to, throw the `DuplicateKeyException` on the Bean Provider's attempt to create an entity object with a duplicate primary key.

The Persistence Manager may create the representation of the entity in the database immediately, or it can defer it to a later time (for example to the time after the matching `ejbPostCreate<METHOD>(...)` has been called, or to the end of the transaction), depending on the caching strategy that it uses.

An `ejbCreate<METHOD>(...)` method executes in the transaction context determined by the transaction attribute of the matching `create<METHOD>(...)` method, as described in subsection 16.6.2.

It is the responsibility of the Persistence Manager to create the representation of the persistent instance in the database in the same transaction context as the `ejbCreate<METHOD>(...)` method.

- `public void ejbPostCreate<METHOD>(...);`

  For each `ejbCreate<METHOD>(...)` method, there is a matching `ejbPostCreate<METHOD>(...)` method that has the same input parameters but the return value is `void`. The container invokes the matching `ejbPostCreate<METHOD>(...)` method on an instance after it invokes the `ejbCreate<METHOD>(...)` method with the same arguments. The instance can discover the primary key by calling `getPrimaryKey()` on its entity context object.

  The entity object identity is available during the `ejbPostCreate<METHOD>(...)` method.

  An `ejbPostCreate<METHOD>(...)` method executes in the same transaction context as the previous `ejbCreate<METHOD>(...)` method.

- `public void ejbActivate();`

  The container invokes this method on the instance when the container picks the instance from the pool and assigns it to a specific entity object identity. The `ejbActivate()` method gives the entity bean instance the chance to acquire additional resources that it needs while it is in the ready state.

This method executes with an unspecified transaction context. The instance can obtain the identity of the entity object via the `getPrimaryKey()` or `getEJBObject()` method on the entity context. The instance can rely on the fact that the primary key and entity object identity will remain associated with the instance until the completion of `ejbPassivate()` or `ejbRemove()`.

- `public void ejbPassivate();`
  The container invokes this method on an instance when the container decides to disassociate the instance from an entity object identity, and to put the instance back into the pool of available instances. The `ejbPassivate()` method gives the instance the chance to release any resources that should not be held while the instance is in the pool. (These resources typically had been allocated during the `ejbActivate()` method.)

  This method executes with an unspecified transaction context. The instance can still obtain the identity of the entity object via the `getPrimaryKey()` or `getEJBObject()` method of the `EntityContext` interface.

- `public void ejbRemove();`

  The container invokes the `ejbRemove()` method on an entity bean instance in response to a client-invoked `remove` operation on the entity bean's home or remote interface.The instance is in the ready state when `ejbRemove()` is invoked and it will be entered into the pool when the method completes.

  The container synchronizes the instance's state before it invokes the `ejbRemove` method. This means that the state of the instance at the beginning of the `ejbRemove` method is the same as it would be at the beginning of a business method.

  The entity Bean Provider can use the `ejbRemove` method to implement any actions that must be done before the entity object's representation is removed from the database. The Persistence Manager must therefore invoke the `ejbRemove` method provided by the Bean Provider and after that `ejbRemove` method returns, the Persistence Manager must remove the entity object's persistent representation.

  This method executes in the transaction context determined by the transaction attribute of the `remove` method that triggered the `ejbRemove` method. The instance can still obtain the identity of the entity object via the `getPrimaryKey()` or `getEJBObject()` method of the `EntityContext` interface.

  It is the responsibility of the Persistence Manager to remove the representation of the persistent object from the database in the same transaction context as the `ejbRemove` method.

  Since the instance will be entered into the pool, the state of the instance at the end of this method must be equivalent to the state of a passivated instance. This means that the instance must release any resource that it would normally release in the `ejbPassivate()` method.

- `public void ejbLoad();`

  When the container needs to synchronize the state of an enterprise bean instance with the entity object's persistent state, the container calls the `ejbLoad()` method. The Persistence Manager may first read the entity object's state from the database, depending on its caching strategy. It must then invoke the `ejbLoad()` method provided by the Bean Provider. Note that the Persistence Manager must call the `ejbLoad()` method provided by the Bean Provider regardless of the caching strategy that it uses.

This method executes in the transaction context determined by the transaction attribute of the business method that triggered the `ejbLoad` method.

- `public void ejbStore();`

  When the container needs to synchronize the state of the entity object in the database with the state of the enterprise bean instance, the container calls the `ejbStore()` method on the instance.

  The Persistence Manager must call the `ejbStore` method provided by the Bean Provider. After that `ejbStore()` method returns, the Persistence Manager may store the persistent state of the instance and its dependent objects to the database, depending on its caching strategy.

  This method executes in the same transaction context as the previous `ejbLoad` or `ejbCreate<METHOD>` method invoked on the instance. All business methods invoked between the previous `ejbLoad` or `ejbCreate <METHOD>`method and this `ejbStore` method are also invoked in the same transaction context.

  If the Persistence Manager uses a lazy storing caching strategy, it is the responsibility of the Persistence Manager to write the representation of the persistent object and all of its dependent objects to the database in the same transaction context as the `ejbStore` method.

- `public` primary key type or collection `ejbFind<METHOD>(...)`;

  The container invokes this method on the instance when the container selects the instance to execute a matching client-invoked `find<METHOD>(...)` method. The instance is in the pooled state (i.e., it is not assigned to any particular entity object identity) when the container selects the instance to execute the `ejbFind<METHOD>` method on it, and it is returned to the pooled state when the execution of the `ejbFind<METHOD>` method completes.

  The `ejbFind<METHOD>` method executes in the transaction context determined by the transaction attribute of the matching `find(...)` method, as described in subsection 16.6.2.

  The Persistence Manager is responsible for ensuring that updates to the states of all entity beans (and their dependent objects) in the same transaction context as the `ejbFind<METHOD> method` are visible in the results of the `ejbFind<METHOD> method`.

  The implementations of the finder methods are generated at the entity bean deployment time based on the declarative specification provided by the Bean Provider in the deployment descriptor using the Persistence Manager Provider's tools.

- `public` type `ejbSelect<METHOD>(...)`;

  A select method is a special type of query method that is not exposed to the client in the home interface.. The Bean Provider typically calls a select method within a business method.

  A select method executes in the transaction context determined by the transaction attribute of the invoking business method.

  The Persistence Manager is responsible for ensuring that all updates to the states of all entity beans (and their dependent objects) in the same transaction context as the `ejbSelect<METHOD> method` are visible in the results of the `ejbSelect<METHOD>` method.

  The implementations of the select methods are generated at the entity bean deployment time using the Persistence Manager provider's tools.

9.6.4 Container's view

This subsection describes the container's view of the state management contract. The container must call the following methods:

- `public void setEntityContext(ec);`

  The container invokes this method to pass a reference to the `EntityContext` interface to the entity bean instance. The container must invoke this method after it creates the instance, and before it puts the instance into the pool of available instances.

  The container invokes this method with an unspecified transaction context. At this point, the `EntityContext` is not associated with any entity object identity.

- `public void unsetEntityContext();`

  The container invokes this method when the container wants to reduce the number of instances in the pool. After this method completes, the container must not reuse this instance.

  The container invokes this method with an unspecified transaction context.

- `public` *`PrimaryKeyClass`* `ejbCreate<METHOD>(...);`
  `public void ejbPostCreate<METHOD>(...);`

  The container invokes these two methods during the creation of an entity object as a result of a client invoking a `create<METHOD>(...)` method on the entity bean's home interface.

  The container invokes the `ejbCreate<METHOD>(...)` method whose signature matches the `create<METHOD>(...)` method invoked by the client.

  The container is responsible for calling the `ejbCreate<METHOD>(...)` method, for obtaining from it the primary key fields of the newly created entity object persistent representation, and for creating an entity EJBObject reference for the newly created entity object. The Container must establish the primary key before it invokes the `ejbPostCreate<METHOD>(...)` method.

  The container then invokes the matching `ejbPostCreate<METHOD>(...)` method on the instance. The instance can discover the primary key by calling `getPrimaryKey()` on its entity context object.

  The container must invoke `ejbCreate<METHOD>` and `ejbPostCreate<METHOD>` in the transaction context determined by the transaction attribute of the matching `create<METHOD>(...)` method, as described in subsection 16.6.2.

- `public void ejbActivate();`

  The container invokes this method on an entity bean instance at activation time (i.e., when the instance is taken from the pool and assigned to an entity object identity). The container must ensure that the primary key of the associated entity object is available to the instance if the instance invokes the `getPrimaryKey()` or `getEJBObject()` method on its `EntityContext` interface.

  The container invokes this method with an unspecified transaction context.

  Note that instance is not yet ready for the delivery of a business method. The container must still invoke the `ejbLoad()` method prior to a business method.

- `public void ejbPassivate();`

The container invokes this method on an entity bean instance at passivation time (i.e., when the instance is being disassociated from an entity object identity and moved into the pool). The container must ensure that the identity of the associated entity object is still available to the instance if the instance invokes the `getPrimaryKey()` or `getEJBObject()` method on its entity context.

The container invokes this method with an unspecified transaction context.

Note that if the instance state has been updated by a transaction, the container must first invoke the `ejbStore()` method on the instance before it invokes `ejbPassivate()` on it.

- `public void ejbRemove();`

  The container invokes the `ejbRemove()` method in response to a client-invoked `remove` operation on the entity bean's home or remote interface.

  The container synchronizes the instance's state before it invokes the `ejbRemove` method. This means that the persistent state of the instance at the beginning of the `ejbRemove` method is the same as it would be at the beginning of a business method.

  The container must ensure that the identity of the associated entity object is still available to the instance in the `ejbRemove()` method (i.e., the instance can invoke the `getPrimaryKey()` or `getEJBObject()` method on its `EntityContext` in the `ejbRemove()` method).

  The container must ensure that the `ejbRemove` method is performed in the transaction context determined by the transaction attribute of the invoked `remove` method, as described in subsection 16.6.2.

- `public void ejbLoad();`

  When the container needs to synchronize the state of an enterprise bean instance with the entity object's state in the database, the container invokes the `ejbLoad()` method to allow the Persistence Manager to read the entity object's persistent state from the database. The exact times that the container invokes `ejbLoad` depend on the configuration of the component and the container, and are not defined by the EJB architecture. Typically, the container will call `ejbLoad` before the first business method within a transaction.

  The container must invoke this method in the transaction context determined by the transaction attribute of the business method that triggered the `ejbLoad` method.

- `public void ejbStore();`

  When the container needs to synchronize the state of the entity object in the database with the state of the enterprise bean instance, the container calls the `ejbStore()` method. This synchronization always happens at the end of a transaction. However, the container may also invoke this method when it passivates the instance in the middle of a transaction, or when it needs to transfer the most recent state of the entity object to another instance for the same entity object in the same transaction.

  The container must call the `ejbStore()` method before the container calls the persistence manager's `beforeCompletion()` method if the persistence manager has registered a synchronization object with the container, as described in Section 9.11.

  The container must invoke this method in the same transaction context as the previously invoked `ejbLoad` or `ejbCreate<METHOD>` method.

- `public` primary key type or collection `ejbFind<METHOD>(...);`

The container invokes the `ejbFind<METHOD>(...)` method on an instance when a client invokes a matching `find<METHOD>(...)` method on the entity bean's home interface. The container must pick an instance that is in the pooled state (i.e., the instance is not associated with any entity object identity) for the execution of the `ejbFind<METHOD>(...)` method. If there is no instance in the pooled state, the container creates one and calls the `setEntityContext(...)` method on the instance before dispatching the finder method.

Before invoking the `ejbFind<METHOD>(...)` method, the container must first synchronize the state of any entity bean instances that are participating in the same transaction context as is used to execute the `ejbFind<METHOD>(...)` by invoking the `ejbStore()` method on those entity bean instances.

After the `ejbFind<METHOD>(...)` method completes, the instance remains in the pooled state. The container may, but is not required to, activate the objects that were located by the finder using the transition through the `ejbActivate()` method.

The container must invoke the `ejbFind<METHOD>(...)` method in the transaction context determined by the transaction attribute of the matching `find(...)` method, as described in subsection 16.6.2.

If the `ejbFind<METHOD>` method is declared to return a single primary key, the container creates an entity EJBObject reference for the primary key and returns it to the client. If the `ejbFind<METHOD>` method is declared to return a collection of primary keys, the container creates a collection of entity EJBObject references for the primary keys returned from the `ejbFind<METHOD>`, and returns the collection to the client. (See Subsection 9.6.6 for information on collections.)

- `public` type `ejbHome<METHOD>(...)`;

  The container invokes the `ejbHome<METHOD>(...)` method on an instance when a client invokes a matching `<METHOD>(...)` home method on the entity bean's home interface. The container must pick an instance that is in the pooled state (i.e., the instance is not associated with any entity object identity) for the execution of the `ejbHome<METHOD>(...)` method. If there is no instance in the pooled state, the container creates one and calls the `setEntityContext(...)` method on the instance before dispatching the home method.

  After the `ejbHome<METHOD>(...)` method completes, the instance remains in the pooled state.

The container must invoke the `ejbHome<METHOD>(...)` method in the transaction context determined by the transaction attribute of the matching `<METHOD>(...)` home method, as described in subsection 16.6.2.

9.6.5 Operations allowed in the methods of the entity bean class

Table 4 defines the methods of an entity bean class in which the enterprise bean instances can access the methods of the `javax.ejb.EntityContext` interface, the `java:comp/env` environment naming context, resource managers, and other enterprise beans.

If an entity bean instance attempts to invoke a method of the `EntityContext` interface, and the access is not allowed in Table 4, the Container must throw the `java.lang.IllegalStateException.`

If an entity bean instance attempts to access a resource manager or an enterprise bean, and the access is not allowed in Table 4, the behavior is undefined by the EJB architecture.

Table 4 Operations allowed in the methods of an entity bean

Bean method	Bean method can perform the following operations
constructor	-
setEntityContext **unsetEntityContext**	**EntityContext methods: *getEJBHome*** **JNDI access to java:comp/env**
ejbCreate	**EntityContext methods: *getEJBHome, getCallerPrincipal, getRollbackOnly, isCallerInRole, setRollbackOnly*** **JNDI access to java:comp/env** **Resource manager access** **Enterprise bean access**
ejbPostCreate	**EntityContext methods: *getEJBHome, getCallerPrincipal, getRollbackOnly, isCallerInRole, setRollbackOnly, getEJBObject, getPrimaryKey*** **JNDI access to java:comp/env** **Resource manager access** **Enterprise bean access**
ejbRemove	**EntityContext methods: *getEJBHome, getCallerPrincipal, getRollbackOnly, isCallerInRole, setRollbackOnly, getEJBObject, getPrimaryKey*** **JNDI access to java:comp/env** **Resource manager access** **Enterprise bean access**
ejbFind* **ejbSelect*** **ejbHome**	**EntityContext methods: *getEJBHome, getCallerPrincipal, getRollbackOnly, isCallerInRole, setRollbackOnly*** **JNDI access to java:comp/env** **Resource manager access** **Enterprise bean access**

Table 4 Operations allowed in the methods of an entity bean

Bean method	Bean method can perform the following operations
ejbActivate **ejbPassivate**	**EntityContext methods:** ***getEJBHome, getEJBObject, getPrimaryKey*** **JNDI access to java:comp/env**
ejbLoad **ejbStore**	**EntityContext methods:** ***getEJBHome, getCallerPrincipal, getRollbackOnly, isCallerInRole, setRollbackOnly, getEJBObject, getPrimaryKey*** **JNDI access to java:comp/env** **Resource manager access** **Enterprise bean access**
business method from remote interface	**EntityContext methods:** ***getEJBHome, getCallerPrincipal, getRollbackOnly, isCallerInRole, setRollbackOnly, getEJBObject, getPrimaryKey*** **JNDI access to java:comp/env** **Resource manager access** **Enterprise bean access**

* Applies to methods implemented by the Persistence Manager only.

Additional restrictions:

- The `getRollbackOnly` and `setRollbackOnly` methods of the `EntityContext` interface should be used only in the enterprise bean methods that execute in the context of a transaction. The Container must throw the `java.lang.IllegalStateException` if the methods are invoked while the instance is not associated with a transaction.

Reasons for disallowing operations:

- Invoking the `getEJBObject` and `getPrimaryKey` methods is disallowed in the entity bean methods in which there is no entity object identity associated with the instance.
- Invoking the `getCallerPrincipal` and `isCallerInRole` methods is disallowed in the entity bean methods for which the Container does not have a client security context.
- Invoking the `getRollbackOnly` and `setRollbackOnly` methods is disallowed in the entity bean methods for which the Container does not have a meaningful transaction context.
- Accessing resource managers and enterprise beans, including accessing the persistent state of an entity bean instance, is disallowed in the entity bean methods for which the Container does not have a meaningful transaction context or client security context.

9.6.6 Finder methods

An entity bean's home interface defines one or more `finder` methods[14], one for each way to find an entity object or collection of entity objects within the home. The name of each finder method starts with the prefix "**find**", such as `findLargeAccounts(...)`. The arguments of a finder method are used by the entity bean implementation to locate the requested entity objects.

Every finder method except `ejbFindByPrimaryKey(key)` must be associated with a `query` element in the deployment descriptor. The entity Bean Provider declaratively specifies the EJB QL finder query and associates it with the finder method in the deployment descriptor. A finder method is normally characterized by an EJB QL query string specified in the `query` element. EJB QL is described in Chapter 10 "EJB QL: EJB Query Language for Container Managed Persistence Query Methods"

9.6.6.1 Single-object finder

Some finder methods (such as `findByPrimaryKey`) are designed to return at most one entity object. For these single-object finders, the result type of the `find<METHOD>(...)`method defined in the entity bean's home interface is the entity bean's remote interface. The result type of the corresponding `ejbFind<METHOD>(...)` method defined in the entity bean's implementation class is the entity bean's primary key type.

The following code illustrates the definition of a single-object finder.

```
// Entity's home interface
public AccountHome extends javax.ejb.EJBHome {
     ...
     Account findByPrimaryKey(AccountPrimaryKey primkey)
          throws FinderException, RemoteException;
     ...
}
```

In general, when defining a single-object finder method other than `findByPrimaryKey(...)`, the entity Bean Provider should be sure that the finder method will always return only a single entity object. This may occur, for example, if the EJB QL query string that is used to specify the finder query to the Persistence Manager includes an equality test on the entity bean's primary key fields. If the entity Bean Provider wishes to provide more flexibility to the Persistence Manager by using an unknown primary key class (see Section 9.10.1.3), the Bean Provider will typically define the finder method as a multi-object finder.

9.6.6.2 Multi-object finders

Some finder methods are designed to return multiple entity objects. For these multi-object finders, the result type of the `find<METHOD>(...)`method defined in the entity bean's home interface is a collection of objects implementing the entity bean's remote interface. The result type of the corresponding `ejbFind<METHOD>(...)` implementation method defined in the entity bean's implementation class is a collection of objects of the entity bean's primary key type.

[14] The *findByPrimaryKey(primaryKey)* method is mandatory for all Entity Beans.

The Bean Provider uses the Java™ 2 `java.util.Collection` interface to define a collection type for the result type of a finder method for an entity bean with container managed persistence.

The Persistence Manager must ensure that the objects in the `java.util.Collection` returned from the `ejbFind<METHOD>(...)` method are instances of the entity bean's primary key class.

A client program must use the `PortableRemoteObject.narrow(...)` method to convert the objects contained in the collections returned by the finder method to the entity bean's remote interface type.

The following is an example of a multi-object finder method definition:

```
// Entity's home interface
public AccountHome extends javax.ejb.EJBHome {
     ...
     java.util.Collection findLargeAccounts(double limit)
          throws FinderException, RemoteException;
     ...
}
```

9.6.7 Select methods

Select methods are special query methods for use by the Bean Provider within an entity bean instance or dependent object class instance. Unlike finder methods, select methods are not specified in the entity bean's home interface. A select method is an abstract method defined by the Bean Provider on an entity bean class or dependent object class. A select method must not be exposed through the home or remote interface of an entity bean.

The semantics of a select method, like those of a finder method, are defined by an EJB QL query string. A select method is similar to a finder method, but unlike a finder method, it can return values of any cmp- or cmr-field type. The EJB QL string specified for a select method must have a SELECT clause that designates the result type.

An `ejbSelect<METHOD>` is not based on the identity of the entity bean instance or dependent object class instance on which it is invoked. However, the Bean Provider can use the primary key of an entity bean or a dependent object class as an argument to an `ejbSelect<METHOD>` to define a query that is logically scoped to a particular entity bean or dependent object class instance.

The following table illustrates the semantics of finder and select methods.

Table 5 Comparison of finder and select methods

	Finder methods	**Select methods**
method	ejbFind<METHOD>	ejbSelect<METHOD>
visibility	exposed to client	internal only
EJB QL	SELECT clause optional	SELECT clause required
instance	arbitrary bean instance in pooled state	instance: current instance (could be bean instance in pooled state, bean instance in ready state, or dependent object class instance)
return value	entity objects of the same type as bean	any value that can be selected by EJB QL

9.6.7.1 Single-object select methods

Some select methods are designed to return at most one value. In general, when defining a single-object select method, the entity Bean Provider must be sure that the select method will always return only a single object or value. If the query specified by the select method returns multiple values of the designated type, the Persistence Manager must throw a `FinderException`.

The Bean Provider will typically define a select method as a multi-object select method.

9.6.7.2 Multi-object select methods

Some select methods are designed to return multiple values. For these multi-object select methods, the result type of the `ejbSelect<METHOD>(...)` method is a collection of objects.

The Bean Provider uses the Java™ 2 `java.util.Collection` interface or `java.util.Set` interface to define a collection type for the result type of a select method. The type of the elements of the collection is determined by the type of the SELECT clause of the corresponding EJB QL query. If the Bean Provider uses the `java.util.Collection` interface, the collection of values returned by the Persistence Manager may contain duplicates if the elements of the collection are not entity beans or dependent objects.

If the select method returns a collection of entity objects, the Bean Provider must use the `PortableRemoteObject.narrow(...)` method to convert the objects contained in the collection to the entity bean's remote interface type.

The following is an example of a multi-object select method definition in the `OrderBean` class:

```
// OrderBean implementation class
public abstract class OrderBean implements javax.ejb.EntityBean{
    ...
    public abstract java.util.Collection
                ejbSelectAllOrderedProducts(Date date)
        throws FinderException;
        // internal finder method to find all products ordered
```

9.6.8 Standard application exceptions for Entities

The EJB specification defines the following standard application exceptions:

- `javax.ejb.CreateException`
- `javax.ejb.DuplicateKeyException`
- `javax.ejb.FinderException`
- `javax.ejb.ObjectNotFoundException`
- `javax.ejb.RemoveException`

This section describes the use of these exceptions by entity beans with container managed persistence. The responsibilities for throwing the exceptions apply to the data access methods generated by the Persistence Manager Provider's tools.

9.6.8.1 CreateException

From the client's perspective, a `CreateException` (or a subclass of `CreateException`) indicates that an application level error occurred during the `create<METHOD>(...)` operation. If a client receives this exception, the client does not know, in general, whether the entity object was created but not fully initialized, or not created at all. Also, the client does not know whether or not the transaction has been marked for rollback. (However, the client may determine the transaction status using the `UserTransaction` interface.)

Both the Persistence Manager and the Bean Provider may throw the `CreateException` (or subclass of `CreateException`) from the `ejbCreate<METHOD>(...)` and `ejbPostCreate<METHOD>(...)` methods to indicate an application-level error from the create or initialization operation. Optionally, the Persistence Manager or Bean Provider may mark the transaction for rollback before throwing this exception.

The Persistence Manager or Bean Provider is encouraged to mark the transaction for rollback only if data integrity would be lost if the transaction were committed by the client. Typically, when a CreateException is thrown, it leaves the database in a consistent state, allowing the client to recover. For example, the ejbCreate<METHOD> method may throw the CreateException to indicate that the some of the arguments to the create<METHOD>(...) methods are invalid.

The Container treats the `CreateException` as any other application exception. See Section 17.3.

9.6.8.2 DuplicateKeyException

The `DuplicateKeyException` is a subclass of `CreateException`. It may be thrown by the Persistence Manager's `ejbCreate<METHOD>(...)` method to indicate to the client that the entity object (or dependent object class instance) cannot be created because an entity object (or dependent object class instance) with the same key already exists. The unique key causing the violation may be the primary key, or another key defined in the underlying database.

Normally, the Persistence Manager should not mark the transaction for rollback before throwing the exception.

When the client receives a `DuplicateKeyException`, the client knows that the entity was not created, and that the client's transaction has not typically been marked for rollback.

9.6.8.3 FinderException

From the client's perspective, a `FinderException` (or a subclass of `FinderException`) indicates that an application level error occurred during the `find(...)` operation. Typically, the client's transaction has not been marked for rollback because of the `FinderException`.

The Persistence Manager throws the `FinderException` (or subclass of `FinderException`) from an `ejbFind<METHOD>(...) or ejbSelect<METHOD>(..)` method to indicate an application-level error in the finder or select method. The Persistence Manager should not, typically, mark the transaction for rollback before throwing the `FinderException`.

The Container treats the `FinderException` as any other application exception. See Section 17.3.

9.6.8.4 ObjectNotFoundException

The `ObjectNotFoundException` is a subclass of `FinderException`. The Persistence Manager throws the `ObjectNotFoundException` from an `ejbFind<METHOD>(...)` or `ejbSelect<METHOD>(...)` method to indicate that the requested object does not exist.

Only single-object finder or select methods (see Subsections 9.6.6 and 9.6.7) should throw this exception. Multi-object finder or select methods must not throw this exception. Multi-object finder or select methods should return an empty collection as an indication that no matching objects were found.

9.6.8.5 RemoveException

From the client's perspective, a `RemoveException` (or a subclass of `RemoveException`) indicates that an application level error occurred during a `remove(...)` operation. If a client receives this exception, the client does not know, in general, whether the entity object was removed or not. The client also does not know if the transaction has been marked for rollback. (However, the client may determine the transaction status using the `UserTransaction` interface.)

The Persistence Manager throws the `RemoveException` (or subclass of `RemoveException`) from the `ejbRemove()` method to indicate an application-level error from the entity object removal operation. Optionally, the Persistence Manager may mark the transaction for rollback before throwing this exception.

The Persistence Manager is encouraged to mark the transaction for rollback only if data integrity would be lost if the transaction were committed by the client. Typically, when a RemoveException is thrown, it leaves the database in a consistent state, allowing the client to recover.

The Container treats the `RemoveException` as any other application exception. See Section 17.3.

9.6.9 Commit options

The Entity Bean protocol is designed to give the Container the flexibility to select the disposition of the instance state at transaction commit time. This flexibility allows the Container to optimally manage the association of an entity object identity with the enterprise bean instances.

The Container can select from the following commit-time options:

- **Option A**: The Container caches a "ready" instance between transactions. The Container ensures that the instance has exclusive access to the state of the object in the persistent storage. Therefore, the Container does not have to synchronize the instance's state from the persistent storage at the beginning of the next transaction.

- **Option B**: The Container caches a "ready" instance between transactions. In contrast to Option A, in this option the Container does not ensure that the instance has exclusive access to the state of the object in the persistent storage. Therefore, the Container must synchronize the instance's state from the persistent storage at the beginning of the next transaction.

- **Option C**: The Container does not cache a "ready" instance between transactions. The Container returns the instance to the pool of available instances after a transaction has completed.

The following table provides a summary of the commit-time options.

Table 6 Summary of commit-time options

	Write instance state to database	**Instance stays ready**	**Instance state remains valid**
Option A	Yes	Yes	Yes
Option B	Yes	Yes	No
Option C	Yes	No	No

Note that the container synchronizes the instance's state with the persistent storage at transaction commit for all three options.

The selection of the commit option is transparent to the entity bean implementation—the entity bean will work correctly regardless of the commit-time option chosen by the Container. The Bean Provider writes the entity bean in the same way.

The object interaction diagrams in Section 9.12.4 illustrate the three alternative commit options in detail.

Note: The Bean Provider relies on the `ejbLoad()` method to be invoked when commit options B and C are used in order to resynchronize the bean's transient state with its persistent state. It is the responsibility of the container to call the `ejbLoad()` method at the beginning of a new transaction when commit option B or C is used and the responsibility of the persistence manager to utilize this method according to its caching strategies. The Persistence Manager may also use knowledge of the which commit option is used by a container in managing its caching strategies and logical transaction isolation options.

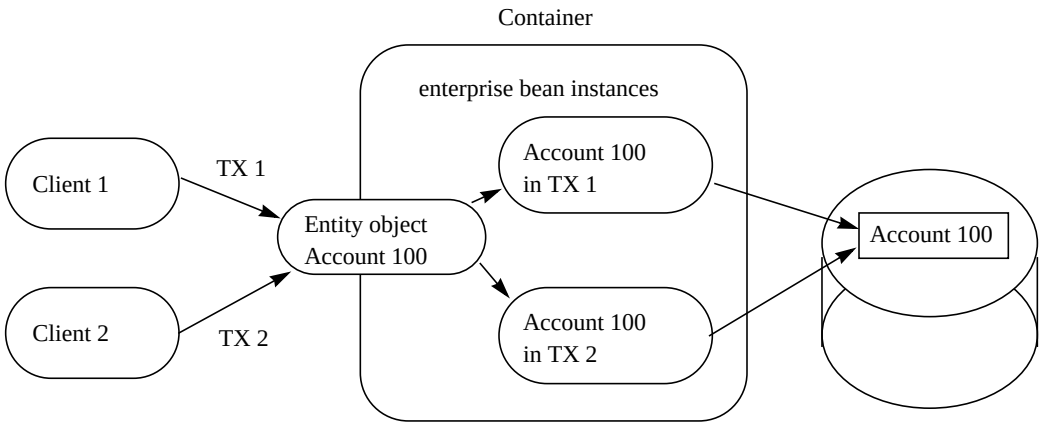
9.6.10 Concurrent access from multiple transactions

When writing the entity bean business methods, the Bean Provider does not have to worry about concurrent access from multiple transactions. The Bean Provider may assume that the container and persistence manager will ensure appropriate synchronization for entity objects that are accessed concurrently from multiple transactions.

The container typically uses one of the following implementation strategies to achieve proper synchronization. (These strategies are illustrative, not prescriptive.)

- The container activates multiple instances of the entity bean, one for each transaction in which the entity object is being accessed. The transaction synchronization is performed automatically by the underlying Persistence Manager during the accessor method calls performed by the business methods, and by the `ejbLoad`, `ejbCreate<METHOD>`, `ejbStore`, and `ejbRemove` methods. The Persistence Manager, together with the database system, provides all the necessary transaction synchronization; the container does not have to perform any synchronization logic. The commit-time options B and C in Subsection 9.12.4 apply to this type of container.

Figure 25 Multiple clients can access the same entity object using multiple instances

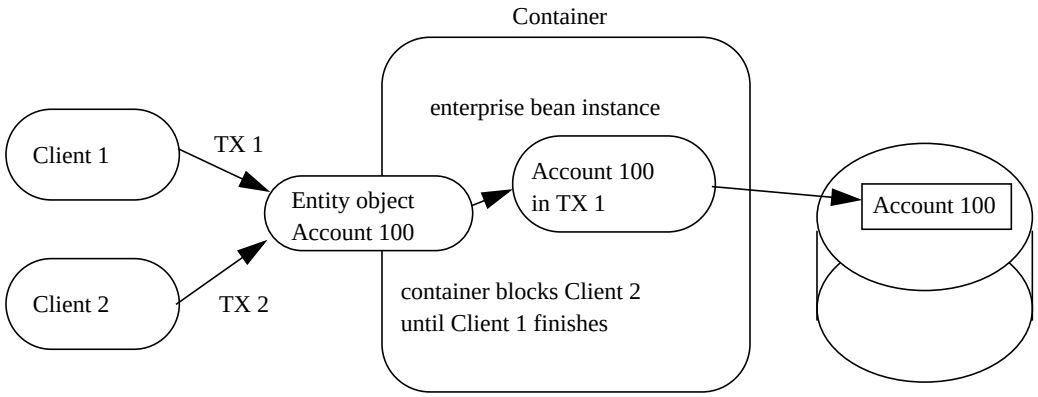


With this strategy, the type of lock acquired by ejbLoad or get accessor method (if a lazy loading cache management strategy is used) leads to a trade-off. If ejbLoad or the accessor method acquires an exclusive lock on the instance's state in the database, the throughput of read-only transactions could be impacted. If ejbLoad or the accessor method acquires a shared lock and the instance is updated, then either ejbStore or a set accessor method will need to promote the lock to an exclusive lock (which may cause a deadlock if it happens concurrently under multiple transactions), or, if the Persistence Manager uses an optimistic cache concurrency control strategy, the Persistence Manager will need to validate the state of the cache against the database at transaction commit (which may result in a rollback of the transaction).

It is expected that Persistence Managers will provide deployment-time configuration options that will allow control to be exercised over the logical transaction isolation levels that their caching strategies provide.

- The container acquires exclusive access to the entity object's state in the database. The container activates a single instance and serializes the access from multiple transactions to this instance. The commit-time option A in Subsection 9.12.4 applies to this type of container.

Figure 26 Multiple clients can access the same entity object using single instance



9.6.11 Non-reentrant and re-entrant instances

An entity Bean Provider can specify that an entity bean is non-reentrant. If an instance of a non-reentrant entity bean executes a client request in a given transaction context, and another request with the same transaction context arrives for the same entity object, the container will throw the `java.rmi.RemoteException` to the second request. This rule allows the Bean Provider to program the entity bean as single-threaded, non-reentrant code.

The functionality of some entity beans may require loopbacks in the same transaction context. An example of a loopback is when the client calls entity object A, A calls entity object B, and B calls back A in the same transaction context. The entity bean's method invoked by the loopback shares the current execution context (which includes the transaction and security contexts) with the Bean's method invoked by the client.

If the entity bean is specified as non-reentrant in the deployment descriptor, the Container must reject an attempt to re-enter the instance via the entity bean's remote interface while the instance is executing a business method. (This can happen, for example, if the instance has invoked another enterprise bean, and the other enterprise bean tries to make a loopback call.) The container must reject the loopback call and throw the `java.rmi.RemoteException` to the caller. The container must allow the call if the Bean's deployment descriptor specifies that the entity bean is re-entrant.

Re-entrant entity beans must be programmed and used with great caution. First, the Bean Provider must code the entity bean with the anticipation of a loopback call. Second, since the container cannot, in general, tell a loopback from a concurrent call from a different client, the client programmer must be careful to avoid code that could lead to a concurrent call in the same transaction context.

Concurrent calls in the same transaction context targeted at the same entity object are illegal and may lead to unpredictable results. Since the container cannot, in general, distinguish between an illegal concurrent call and a legal loopback, application programmers are encouraged to avoid using loopbacks. Entity beans that do not need callbacks should be marked as non-reentrant in the deployment descriptor, allowing the container to detect and prevent illegal concurrent calls from clients.

> *Note that an* `ejbSelect<METHOD>` *method that returns the same type as the entity bean on which it is defined may lead to a subsequent loopback call. Such methods should therefore be used with caution.*

9.7 Responsibilities of the Enterprise Bean Provider

This section describes the responsibilities of an entity Bean Provider to ensure that an entity bean with container managed persistence can be deployed in any EJB Container.

9.7.1 Classes and interfaces

The entity Bean Provider is responsible for providing the following class files:

- Entity bean class and any dependent classes
- Entity bean's remote interface
- Entity bean's home interface
- Primary key class

9.7.2 Enterprise bean class

The following are the requirements for an entity bean class:

The class must implement, directly or indirectly, the `javax.ejb.EntityBean` interface.

The class must be defined as `public` and must be `abstract`.

The class must define a public constructor that takes no arguments.

The class must not define the `finalize()` method.

The class may, but is not required to, implement the entity bean's remote interface[15]. If the class implements the entity bean's remote interface, the class must provide no-op implementations of the methods defined in the `javax.ejb.EJBObject` interface. The container will never invoke these methods on the bean instances at runtime.

The entity bean class must implement the business methods, and the `ejbCreate<METHOD>` and `ejbPostCreate<METHOD>` methods as described later in this section.

The entity bean class must implement the `ejbHome<METHOD>` methods that correspond to the home business methods specified in the bean's home interface. These methods are executed on an instance in the pooled state; hence they must not access state that is particular to a specific bean instance (e.g., the accessor methods of the bean's abstract persistence schema must not be used by these methods).

The entity bean class must implement the get and set accessor methods of the bean's abstract persistence schema as `abstract` methods.

The entity bean class may have superclasses and/or superinterfaces. If the entity bean has superclasses, the business methods, the `ejbCreate<METHOD>` and `ejbPostCreate<METHOD>` methods, and the methods of the `EntityBean` interface may be implemented in the enterprise bean class or in any of its superclasses.

The entity bean class is allowed to implement other methods (for example helper methods invoked internally by the business methods) in addition to the methods required by the EJB specification.

The entity bean class does not implement the `ejbFind<METHOD>` methods. The implementations of the `ejbFind<METHOD>` methods are generated at the entity bean deployment time using the Persistence Manager Provider's tools.

The entity bean class must implement any `ejbSelect<METHOD>` methods as abstract methods.

9.7.3 Dependent object classes

The following are the requirements for a dependent object class:

[15] If the entity bean class does implement the remote interface, care must be taken to avoid passing of `this` as a method argument or result. This potential error can be avoided by choosing not to implement the remote interface in the entity bean class.

The class must be defined as `public` and must be `abstract`.

The class must implement the get and set accessor methods of its abstract persistence schema as `abstract` methods.

The class must implement the `remove()` method as an abstract method.

The class must not define the `finalize()` method.

The dependent object class must define an `ejbCreate<METHOD>(...)` method and `ejbPostCreate<METHOD>(...)` method for every matching `create<METHOD>(...)` method whose result type is the dependent object class. The matching methods must have the same number and type of arguments, and the same return type (i.e., the dependent object class).

The dependent object class must implement any `ejbSelect<METHOD>` methods as abstract methods.

9.7.4 Dependent value classes

The following are the requirements for a dependent value class:

The class must be defined as `public` and must not be `abstract`.

The class must be serializable.

9.7.5 *ejbCreate<METHOD>* methods

The entity bean class may define zero or more `ejbCreate<METHOD>(...)` methods whose signatures must follow these rules:

The method name must have `ejbCreate` as its prefix.

The method must be declared as `public`.

The method must not be declared as `final` or `static`.

The return type must be the entity bean's primary key type.

The method arguments and return value types must be legal types for RMI-IIOP.

The throws clause must define the `javax.ejb.CreateException`. The throws clause may define arbitrary application specific exceptions.

Compatibility Note: EJB 1.0 allowed the ejbCreate method to throw the `java.rmi.RemoteException` to indicate a non-application exception. This practice was deprecated in EJB 1.1—an EJB 1.1 or EJB 2.0 compliant enterprise bean should throw the javax.ejb.EJBException or another java.lang.RuntimeException to indicate non-application exceptions to the Container (see Section 17.2.2).

9.7.6 *ejbPostCreate<METHOD>* methods

For each `ejbCreate<METHOD>(...)` method, the entity bean class must define a matching `ejbPostCreate<METHOD>(...)` method, using the following rules:

The method name must have `ejbPostCreate` as its prefix.

The method must be declared as `public`.

The method must not be declared as `final` or `static`.

The return type must be `void`.

The method arguments must be the same as the arguments of the matching `ejbCreate<METHOD>(...)` method.

The throws clause may define arbitrary application specific exceptions, including the `javax.ejb.CreateException`.

Compatibility Note: EJB 1.0 allowed the ejbPostCreate method to throw the `java.rmi.RemoteException` to indicate a non-application exception. This practice was deprecated in EJB 1.1—an EJB 1.1 or EJB 2.0 compliant enterprise bean should throw the javax.ejb.EJBException or another java.lang.RuntimeException to indicate non-application exceptions to the Container (see Section 17.2.2).

9.7.7 *ejbHome<METHOD>* methods

The entity bean class may define zero or more home methods whose signatures must follow the following rules:

The method name must have `ejbHome` as its prefix.

The method must be declared as `public`.

The method must not be declared as `static`.

The method argument and return value types must be legal types for RMI-IIOP.

The throws clause may define arbitrary application specific exceptions.

9.7.8 *ejbSelect<METHOD>* methods

The entity bean class or dependent object class may define one or more select methods whose signatures must follow the following rules:

The method name must have `ejbSelect` as its prefix.

The method must be declared as `public`.

The method must be declared as `abstract`.

The return value type of an `ejbSelect<METHOD>` method must not be an entity bean class type.

The throws clause must define the `javax.ejb.FinderException`. The throws clause may define arbitrary application specific exceptions.

Every select method must either be associated with an EJB QL query string in the deployment descriptor or a description must be given to indicate the semantics of the query to the user of the Persistence Manager Provider's tools. EJB QL is defined in Chapter 10. The Bean Provider must use the `description` element of the `query` element to specify a query that is not expressed in EJB QL. The Bean Provider must only use description-based queries if the query is not expressible in EJB QL.

The EJB QL string associated with a select method must include a SELECT clause.

9.7.9 Business methods

The entity bean class may define zero or more business methods whose signatures must follow these rules:

The method names can be arbitrary, but they must not start with 'ejb' to avoid conflicts with the callback methods used by the EJB architecture.

The business method must be declared as `public`.

The method must not be declared as `final` or `static`.

The method argument and return value types must be legal types for RMI-IIOP.

The throws clause may define arbitrary application specific exceptions.

Compatibility Note: EJB 1.0 allowed the business methods to throw the `java.rmi.RemoteException` to indicate a non-application exception. This practice was deprecated in EJB 1.1—an EJB 1.1 or EJB 2.0 compliant enterprise bean should throw the javax.ejb.EJBException or another java.lang.RuntimeException to indicate non-application exceptions to the Container (see Section 17.2.2).

9.7.10 Entity bean's remote interface

The following are the requirements for the entity bean's remote interface:

The interface must extend the `javax.ejb.EJBObject` interface.

The methods defined in the remote interface must follow the rules for RMI-IIOP. This means that their argument and return value types must be valid types for RMI-IIOP, and their throws clauses must include the `java.rmi.RemoteException`.

The remote interface is allowed to have superinterfaces. Use of interface inheritance is subject to the RMI-IIOP rules for the definition of remote interfaces.

For each method defined in the remote interface, there must be a matching method in the entity bean's class. The matching method must have:

- The same name.
- The same number and types of its arguments, and the same return type.
- All the exceptions defined in the throws clause of the matching method of the enterprise Bean class must be defined in the throws clause of the method of the remote interface.

The Bean Provider must not expose the dependent object classes that comprise the abstract persistent schema of an entity bean in the remote interface of the bean.

The get and set methods of the entity bean's abstract persistence schema must not be exposed through the remote interface except in the following cases:

- When the relationship is defined as a one-to-one or many-to-one relationship between two entity beans.
- When the get and set accessor methods are methods for a *cmp-field*.

The Bean Provider must not expose the persistent Collection classes that are used in container manager relationships.

9.7.11 Entity bean's home interface

The following are the requirements for the entity bean's home interface:

The interface must extend the `javax.ejb.EJBHome` interface.

The methods defined in this interface must follow the rules for RMI-IIOP. This means that their argument and return types must be of valid types for RMI-IIOP, and their throws clauses must include the `java.rmi.RemoteException`.

The home interface is allowed to have superinterfaces. Use of interface inheritance is subject to the RMI-IIOP rules for the definition of remote interfaces.

Each method defined in the home interface must be one of the following:

- A create method.
- A finder method.
- A home method.

Each `create` method must be named "**create**<METHOD>", e.g. `createLargeAccounts`. Each create method name must match one of the `ejbCreate<METHOD>` methods defined in the enterprise bean class. The matching `ejbCreate<METHOD>` method must have the same number and types of its arguments. (Note that the return type is different.)

The return type for a `create<METHOD>` method must be the entity bean's remote interface type.

All the exceptions defined in the throws clause of the matching `ejbCreate<METHOD>` and `ejbPostCreate<METHOD>` methods of the enterprise Bean class must be included in the throws clause of the matching `create` method of the home interface (i.e., the set of exceptions defined for the `create` method must be a superset of the union of exceptions defined for the `ejbCreate<METHOD>` and `ejbPostCreate<METHOD>` methods).

The throws clause of a `create<METHOD>` method must include the `javax.ejb.CreateException`.

Each `finder` method must be named "**`find`**`<METHOD>`" (e.g. `findLargeAccounts`).

The return type for a `find<METHOD>` method must be the entity bean's remote interface type (for a single-object finder), or a collection thereof (for a multi-object finder).

The home interface must always include the `findByPrimaryKey` method, which is always a single-object finder. The method must declare the primary key class as the method argument.

The throws clause of a `finder` method must include the `javax.ejb.FinderException`.

Home methods can have arbitrary names, but they must not start with "`create`", "`find`", or "`remove`". Their argument and return types must be of valid types for RMI-IIOP, and their throws clauses must include the `java.rmi.RemoteException`. The matching `ejbHome` method specified in the entity bean class must have the same number and types of arguments and must return the same type as the home method as specified in the home interface of the bean.

9.7.12 Entity bean's primary key class

The Bean Provider must specify a primary key class in the deployment descriptor.

The primary key type must be a legal Value Type in RMI-IIOP.

The class must provide suitable implementation of the `hashCode()` and `equals(Object other)` methods to simplify the management of the primary keys by the Persistence Manager.

9.7.13 Entity bean's deployment descriptor

The Bean Provider must specify the dependent objects classes in the `dependents` element.

The Bean Provider must specify the relationships in which the entity bean and dependent objects participate in the `relationships` element.

The Bean Provider must specify in the `ejb-entity-ref` element the remote entity beans that are used in relationships. Remote entity beans are those entity beans whose abstract persistence schemas are not available in the ejb-jar file and that participate in one way navigable relationships.

The Bean Provider must provide unique names to designate entity beans, remote entity beans, and dependent object classes as follows, and as described in Section 9.4.14.

- The Bean Provider must specify unique names for entity beans which are defined in the ejb-jar file by using the `ejb-name` element.
- The Bean Provider must specify a `remote-ejb-name` element in the `ejb-entity-ref` deployment descriptor element to provide a unique name for a remote entity bean that is used in a container managed relationship.
- The Bean Provider must specify a unique abstract schema name for an entity bean using the `abstract-schema-name` deployment descriptor element.
- The Bean Provider must specify a unique dependent name for a dependent object class using the `dependent-name` deployment descriptor element.

The Bean Provider should not use reserved identifiers as `ejb-names`, `remote-ejb-names`, `dependent-names`, or `abstract-schema-names`. Reserved identifiers are discussed in Section 10.2.4.1.

The Bean Provider must define a query for each finder or select method except `findByPrimaryKey(key)`. Typically this will be provided as the content of the `ejb-ql` element contained in the `query` element for the entity or dependent object class. The syntax of EJB QL is defined in Chapter 10.

Since EJB QL query strings are embedded in the deployment descriptor, which is an XML document, it may be necessary to encode the following characters in the query string: “>”, “<“.

9.8 The responsibilities of the Persistence Manager

This section describes the responsibilities of the Persistence Manager Provider to ensure that an entity bean with container managed persistence can be deployed in any EJB Container. The Persistence Manager is responsible for providing tools to prepare the entity bean for deployment and to generate the code to manage the persistent state and relationships of the entity bean instances at runtime.

9.8.1 Generation of implementation classes

The tools provided by the Persistence Manager Provider are responsible for the generation of additional classes when the entity bean is prepared for deployment in the operational environment. The tools obtain the information that they need for generation of additional classes by introspecting the classes and interfaces provided by the entity Bean Provider and by examining the entity bean’s deployment descriptor.

These tools must generate the following classes:

- A class that implements the entity bean class (i.e., a concrete class corresponding to the abstract entity bean class that was provided by the Bean Provider).

- Classes that implement the dependent object classes (i.e., concrete classes corresponding to the abstract dependent object classes that were provided by the Bean Provider).

- Classes that implement the `java.util.Collection` interfaces that are used for the relationships of the abstract persistence schema of an entity bean or dependent object class.

These classes are used by the persistence manager to manage the persistent state and relationships of the entity bean instances at runtime. Tools can use subclassing, delegation, and code generation.

9.8.2 Classes and interfaces

The Persistence Manager is responsible for providing the following class files:

- Implementation of the concrete entity bean class

- Implementation of the concrete dependent object classes

9.8.3 Enterprise bean class

The following are the requirements for a concrete entity bean class:

The class must implement, directly or indirectly, the `javax.ejb.EntityBean` interface.

The class must be defined as `public` and must not be `abstract`.

The class must not be defined as `final`.

The class must define a public constructor that takes no arguments.

The class must implement the get and set methods of the bean's abstract persistence schema.

The class must implement the `create<METHOD>(...)` methods for the dependent object classes that can be created by the class.

The class must not define the `finalize()` method.

The entity bean class must implement, directly or indirectly, the business methods, and the `ejbCreate<METHOD>` and `ejbPostCreate<METHOD>` methods as described later in this section.

The entity bean class must implement the `ejbFind<METHOD>(...)` and `ejbSelect<METHOD>(...)` methods.

The entity bean class may have superclasses and/or superinterfaces. If the entity bean has superclasses, the business methods, the `ejbCreate<METHOD>` and `ejbPostCreate<METHOD>` methods, and the methods of the `EntityBean` interface may be implemented in the enterprise bean class or in any of its superclasses.

The entity bean class is allowed to implement other methods in addition to the methods required by the EJB specification.

9.8.4 Dependent object classes

The following are the requirements for a concrete dependent object class:

The class must be defined as `public` and must not be `abstract`.

The class must not be defined as `final`.

The class must implement the get and set methods of the dependent object class's abstract persistence schema.

The class must implement the `remove()` method.

The class must implement the `ejbSelect<METHOD>(...)` methods.

The class must not define the `finalize()` method.

The class must implement the `create<METHOD>(...)` methods for the dependent object classes that can be created by the class.

The dependent object class is allowed to implement other methods in addition to the methods required by the EJB specification.

9.8.5 *ejbCreate<METHOD>* methods

The concrete entity bean class must define zero or more `ejbCreate<METHOD>(...)` methods whose signatures must follow these rules:

For each `ejbCreate<METHOD>(...)` method in the abstract entity bean class, there must be a method with the same argument and result types in the concrete entity bean class.

The method must be declared as `public`.

The method must not be declared as `final` or `static`.

The throws clause must define the `javax.ejb.CreateException`. The throws clause may define any application specific exceptions that are defined in the corresponding `ejbCreate<METHOD>(...)` method of the abstract entity bean class.

The entity object created by the `ejbCreate<METHOD>(...)` method must have a unique primary key. This means that the primary key must be different from the primary keys of all the existing entity objects within the same home. The `ejbCreate<METHOD>(...)` method may throw the `DuplicateKeyException` on an attempt to create an entity object with a duplicate primary key. However, it is legal to reuse the primary key of a previously removed entity object.

9.8.6 *ejbPostCreate<METHOD>* methods

For each `ejbPostCreate<METHOD>(...)` method in the abstract entity bean class, the concrete entity bean class may define a matching `ejbPostCreate<METHOD>(...)` method, using the following rules:

The method must be declared as `public`.

The method must not be declared as `final` or `static`.

The return type must be `void`.

The method arguments must be the same as the arguments of the matching `ejbPostCreate<METHOD>(...)` method of the abstract entity bean class.

The throws clause may define any application specific exceptions that are defined in the corresponding `ejbPostCreate<METHOD>(...)` method of the abstract entity bean class, including the `javax.ejb.CreateException`.

9.8.7 *ejbFind<METHOD>* methods

For each `find<METHOD>(...)` method in the home interface of the entity bean, there must be a corresponding `ejbFind<METHOD>(...)` method with the same argument types in the concrete entity bean class.

The method name must have `ejbFind` as its prefix.

The method must be declared as `public`.

The method argument and return value types must be legal types for RMI-IIOP.

The return type of a finder method must be the entity bean's primary key type, or a collection of primary keys.

The throws clause must define the `javax.ejb.FinderException`. The throws clause may define arbitrary application specific exceptions.

Every finder method except `ejbFindByPrimaryKey(key)` is specified in the `query` deployment descriptor element for the entity. The Persistence Manager must use the EJB QL query string that is the content of the `ejb-ql` element or the descriptive query specification contained in the `description` element as the definition of the query of the corresponding `ejbFind<METHOD>(...)` method.

9.8.8 *ejbSelect<METHOD>* methods

For each `ejbSelect<METHOD>(...)` method in the abstract entity bean class (or dependent object class), there must be a method with the same argument and result types in the concrete entity bean class (or dependent object class).

Every select method is specified in a `query` deployment descriptor element for the entity. The Persistence Manager must use the EJB QL query string that is the content of the `ejb-ql` element or the descriptive query specification that is contained in the `description` element as the definition of the query of the corresponding `ejbSelect<METHOD>(...)` method.

The Persistence Manager must throw a `FinderException` when a select method returns more than one value if it is defined as a single-object select method.

The Persistence Manager must use the corresponding EJB QL string and the type of the values selected as specified by the SELECT clause to determine the type of the values returned by a select method.

The Persistence Manager must ensure that there are no duplicates returned by a select method if the return type is `java.util.Set`.

9.9 The responsibilities of the Container Provider

This section describes the responsibilities of the Container Provider to support entity beans. The Container Provider is responsible for providing the deployment tools, and for managing the entity bean instances at runtime.

Because the EJB specification does not define the API between deployment tools and the container, we assume that the deployment tools described in this section are provided by the container provider. Alternatively, the deployment tools may be provided by a different vendor who uses the container vendor's specific API.

9.9.1 Generation of implementation classes

The deployment tools provided by the container provider are responsible for the generation of additional classes when the entity bean is deployed. The tools obtain the information that they need for generation of the additional classes by introspecting the classes and interfaces provided by the Persistence Manager Provider, and by examining the entity bean's deployment descriptor.

The deployment tools must generate the following classes:

- A class that implements the entity bean's home interface (i.e., the entity EJBHome class).
- A class that implements the entity bean's remote interface (i.e., the entity EJBObject class).

The deployment tools may also generate a class that mixes some container-specific code with the entity bean class. The code may, for example, help the container to manage the entity bean instances at runtime. Tools can use subclassing, delegation, and code generation.

The deployment tools may also allow generation of additional code that wraps the business methods and that is used to customize the business logic for an existing operational environment. For example, a wrapper for a `debit` function on the `Account` Bean may check that the debited amount does not exceed a certain limit, or perform security checking that is specific to the operational environment.

9.9.2 Entity EJBHome class

The entity EJBHome class, which is generated by deployment tools, implements the entity bean's home interface. This class implements the methods of the `javax.ejb.EJBHome` interface, and the type-specific `create` and `finder` methods specific to the entity bean.

The implementation of each `create<METHOD>(...)` method invokes a matching `ejbCreate<METHOD>(...)` method, followed by the matching `ejbPostCreate<METHOD>(...)` method, passing the `create<METHOD>(...)` parameters to these matching methods.

The implementation of the `remove(...)` methods defined in the `javax.ejb.EJBHome` interface must activate an instance (if an instance is not already in the ready state) and invoke the `ejbRemove` method on the instance.

The implementation of each `find<METHOD>(...)` method invokes a matching `ejbFind<METHOD>(...)` method. The implementation of the `find<METHOD>(...)` method must create an entity object reference for the primary key returned from the `ejbFind<METHOD>` and return the entity object reference to the client. If the `ejbFind<METHOD>` method returns a collection of primary keys, the implementation of the `find<METHOD>(...)` method must create a collection of entity object references for the primary keys and return the collection to the client.

Before invoking the `ejbFind<METHOD>(...)` method, the container must first synchronize the state of any entity bean instances that are participating in the same transaction context as the `ejbFind<METHOD>(...)` by invoking the `ejbStore()` method on those entity bean instances.

The implementation of each `<METHOD>(...)` home method invokes a matching `ejbHome<METHOD>(...)` method, passing the `<METHOD>(...)` parameters to the matching method.

9.9.3 Entity EJBObject class

The entity EJBObject class, which is generated by deployment tools, implements the entity bean's remote interface. It implements the methods of the `javax.ejb.EJBObject` interface and the business methods specific to the entity bean.

The implementation of the `remove(...)` method (defined in the `javax.ejb.EJBObject` interface) must activate an instance (if an instance is not already in the ready state) and invoke the `ejbRemove` method on the instance.

The implementation of each business method must activate an instance (if an instance is not already in the ready state) and invoke the matching business method on the instance.

9.9.4 Handle class

The deployment tools are responsible for implementing the handle class for the entity bean. The handle class must be serializable by the Java Serialization protocol.

As the handle class is not entity bean specific, the container may, but is not required to, use a single class for all deployed entity beans.

9.9.5 Home Handle class

The deployment tools responsible for implementing the home handle class for the entity bean. The handle class must be serializable by the Java Serialization protocol.

Because the home handle class is not entity bean specific, the container may, but is not required to, use a single class for the home handles of all deployed entity beans.

9.9.6 Meta-data class

The deployment tools are responsible for implementing the class that provides meta-data information to the client view contract. The class must be a valid RMI-IIOP Value Type, and must implement the `javax.ejb.EJBMetaData` interface.

Because the meta-data class is not entity bean specific, the container may, but is not required to, use a single class for all deployed enterprise beans.

9.9.7 Instance's re-entrance

The container runtime must enforce the rules defined in Section 9.6.11.

9.9.8 Transaction scoping, security, exceptions

The container runtime must follow the rules on transaction scoping, security checking, and exception handling described in Chapters 16, 20, and 17.

9.9.9 Implementation of object references

The container should implement the distribution protocol between the client and the container such that the object references of the home and remote interfaces used by entity bean clients are usable for a long period of time. Ideally, a client should be able to use an object reference across a server crash and restart. An object reference should become invalid only when the entity object has been removed, or after a reconfiguration of the server environment (for example, when the entity bean is moved to a different EJB server or container).

The motivation for this is to simplify the programming model for the entity bean client. While the client code needs to have a recovery handler for the system exceptions thrown from the individual method invocations on the home and remote interface, the client should not be forced to re-obtain the object references.

9.9.10 EntityContext

The container must implement the `EntityContext.getEJBObject()` method such that the bean instance can use the Java language cast to convert the returned value to the entity bean's remote interface type. Specifically, the bean instance does not have to use the `PortableRemoteObject.narrow(...)` method for the type conversion.

9.10 Primary Keys

9.10.1 Entity bean's primary key type

The container must be able to manipulate the primary key type of an entity bean. Therefore, the primary key type for an entity bean with container-managed persistence must follow the rules in this subsection, in addition to those specified in Subsection 9.7.12.

There are two ways to specify a primary key class for an entity bean with container-managed persistence:

- Primary key that maps to a single field in the entity bean class.
- Primary key that maps to multiple fields in the entity bean class.

The second method is necessary for implementing compound keys, and the first method is convenient for single-field keys. Without the first method, simple types such as String would have to be wrapped in a user-defined class.

9.10.1.1 Primary key that maps to a single field in the entity bean class

The Bean Provider uses the `primkey-field` element of the deployment descriptor to specify the container-managed field of the entity bean class that contains the primary key. The field's type must be the primary key type.

9.10.1.2 Primary key that maps to multiple fields in the entity bean class

The primary key class must be `public`, and must have a `public` constructor with no parameters.

All fields in the primary key class must be declared as public.

The names of the fields in the primary key class must be a subset of the names of the container-managed fields. (This allows the container to extract the primary key fields from an instance's container-managed fields, and vice versa.)

9.10.1.3 Special case: Unknown primary key class

In special situations, the entity Bean Provider may choose not to specify the primary key class for an entity bean with container-managed persistence. This case usually happens when the entity bean does not have a natural primary key, and/or the Bean Provider wants to allow the Deployer using the Persistence Manager Provider's tools to select the primary key fields at deployment time. The entity bean's primary key type will usually be derived from the primary key type used by the underlying database system that stores the entity objects. The primary key used by the database system may not be known to the Bean Provider.

When defining the primary key for the enterprise bean, the Deployer using the Persistence Manager Provider's tools may sometimes need to subclass the entity bean class to add additional container-managed fields (this typically happens for entity beans that do not have a natural primary key, and the primary keys are system-generated by the underlying database system that stores the entity objects).

In this special case, the type of the argument of the `findByPrimaryKey` method must be declared as `java.lang.Object`, and the return value of `ejbCreate<METHOD>()` must be declared as `java.lang.Object`. The Bean Provider must specify the primary key class in the deployment descriptor as of the type `java.lang.Object`.

The primary key class is specified at deployment time in the situations when the Bean Provider develops an entity bean that is intended to be used with multiple back-ends that provide persistence, and when these multiple back-ends require different primary key structures.

Use of entity beans with a deferred primary key type specification limits the client application programming model, because the clients written prior to deployment of the entity bean may not use, in general, the methods that rely on the knowledge of the primary key type.

The implementation of the enterprise bean class methods must be done carefully. For example, the methods should not depend on the type of the object returned from EntityContext.getPrimaryKey(), because the return type is determined by the Deployer after the EJB class has been written.

9.10.2 Dependent object's primary key type

The Persistence Manager uses the primary key of a dependent object to maintain the dependent object's persistent identity.

The Bean Provider can either specify the primary key of a dependent object class in terms of one or more persistent fields of the dependent object class or can defer the implementation of the primary key to the Persistence Manager.

9.10.2.1 Primary key that maps to one or more fields in the dependent object class

The Bean Provider uses the `pk-field` elements of the `dependent` deployment descriptor element to specify the container-managed fields of the dependent object class that contain the primary key. The names of the fields in the `pk-field` elements must be a subset of the names of the container-managed fields.

9.10.2.2 Unspecified dependent object primary key

The Bean Provider may choose not to specify the primary key for a dependent object class. This case usually happens when the dependent object does not have a natural primary key, and/or the Bean Provider wants to allow the Deployer using the Persistence Manager Provider's tools to select the primary key fields at deployment time. The dependent object's primary key type will usually be derived from the primary key type used by the underlying database system that stores the dependent objects. The primary key used by the database system may not be known to the Bean Provider.

When defining the primary key for a dependent object class, the Deployer using the Persistence Manager Provider's tools may sometimes need to subclass the dependent object class to add additional container-managed fields.

The primary key should be specified at deployment time in the situations when the Bean Provider develops a dependent object class that is intended to be used with multiple back-ends that provide persistence, and when these multiple back-ends require different primary key structures.

9.11 Other contracts between the Persistence Manager and Container

This section describes other contracts between the Persistence Manager and the Container.

9.11.1 Transaction context

In order to manage the access to the persistent state that it has cached on behalf of a bean instance, the persistence manager needs to keep track of when the transaction context for the bean instance has changed. There are various strategies that the Persistence Manager can use to detect when the transaction context for a bean instance has changed. The following are illustrative:

- The Persistence Manager can check which transaction context is in effect on each get or set accessor method access to the state of the bean instance.
- Because the transaction context of a bean instance can be changed by the container only on remote method call boundaries, the persistence manager can wrapper the remote methods of the entity bean to identify when a new remote method has been called, and hence detect when the transaction context may have changed.
- The Persistence Manager can use the `ejbLoad`, `ejbStore`, `ejbCreate`, and `ejbFind` method invocations to keep track of changes in transaction context. If Commit Option B or C is used, the Persistence Manager needs to check the transaction context in the `ejbLoad`, `ejbCreate`, and `ejbFind` methods only. If Commit Option A is used, however, the transaction context may have changed even though `ejbLoad` was not invoked. In this case, it is possible to use the `ejbStore` method to note that the immediately following accessor method invocation on the bean instance might occur in a different transaction context. The transaction context of the next accessor method invocation must be checked accordingly.

The Persistence Manager can use the `javax.transaction.TransactionManager.getTransaction()` method to identify the transaction context in effect for a given method invocation. The `getTransaction()` method returns the transaction object that represents the transaction context of the calling thread.

The Container provides an implementation of the TransactionManager interface to the Persistence Manager through JNDI. The Persistence Manager can locate the `TransactionManager` through the standard JNDI API as `java:pm/TransactionManager`.

9.11.2 Connection management

When there is a change in transaction context, the Persistence Manager may need to obtain a new connection from the container for use in accessing the persistent state of the bean.

The Persistence Manager may need to request from the Container a connection that is enlisted in the current transaction or a connection that is not enlisted.

The Container should typically provide to the Persistence Manager the ability to specify separate resource manager connection factories for these two types of connections, and make those resource manager connection factories available to the persistence manager through JNDI. As part of the persistence manager configuration process, the Persistence Manager Provider's tools will typically provide mechanisms that allow these resource manager connection factory dependencies to be declaratively expressed.

> *Note: This specification does not prescribe how the Persistence Manager obtains the resource manager connection factories, but recommends that the* `java:pm/env` *subcontext be used for this purpose. We expect to standardize this use in a later release of this specification.*

Only Persistence Managers that use an optimistic concurrency control strategy will typically need to obtain connections that have not been enlisted in the current transaction context.

- The Persistence Manager calls the `getConnection` method (e.g., in the case of JDBC, `javax.sql.DataSource.getConnection()`) on the resource manager connection factory that provides container management of the transactional enlistment of connections to obtain a connection that has been enlisted by the container in the transaction context of the calling thread. It is the container (not the persistence manager) that is responsible for the transaction management of the connection.

- The Persistence Manager calls the `getConnection` method on the resource manager connection factory that provides connections that have not been enlisted in the current transaction context to obtain a connection that has been not been enlisted in any transaction context. The persistence manager can manage transactions on the connection using the resource-adaptor-specific API. Persistence managers that use an optimistic concurrency control strategy are expected to make use of such non-enlisted connections for that portion of the transaction that precedes the commit phase. The transaction management on this connection is the responsibility of the persistence manager.

The container is responsible for providing the implementations of the resource manager connection factory methods that the persistence manager uses, and for making the resource manager connection factories for these methods available to the persistence manager through JNDI.

The persistence manager should assume that the container is doing pooling of connections. The persistence manager should therefore hold a connection no longer than necessary.

9.11.3 Connection management scenarios

There are a variety of connection management strategies that the Persistence Manager might use. The following scenarios are intended to be illustrative rather than prescriptive.

9.11.3.1 Scenario: Pessimistic concurrency control

When a persistence manager method (e.g., accessor method, `ejbStore()` method, etc.) is executing in a transaction, the connection that is used to access the database on behalf of that method must run under that same transaction. The Persistence Manager requests the Container to provide a connection that runs in the current transaction context, and caches the connection for use with that transaction.

9.11.3.2 Scenario: Optimistic concurrency control

In general, with an optimistic concurrency control caching strategy, the Persistence Manager needs to be able to use separate connections for the pre-commit phase of the transaction (i.e., the portion of the transaction that precedes the invocation of the `Synchronization.beforeCompletion()` method) and for the commit phase of the transaction, in order to avoid holding long-term read locks on data. During the transaction commit phase, the Persistence Manager needs to use a connection that has been enlisted by the container in the current transaction context.

The persistence manager obtains a connection that has not been enlisted by the container in the current transaction context for use during the pre-commit phase of the transaction, by calling the `getConnection` method on the resource manager connection factory that provides connections that have not been enlisted in the current transaction context.

When the persistence manager enters the commit phase of the transaction, the persistence manager obtains a connection that has been enlisted by the container in the transaction context of the calling thread by invoking the `getConnection` method on the resource manager connection factory that provides container management of the transactional enlistment of connections.

9.11.4 Synchronization notifications

If an optimistic concurrency control cache management strategy is used by the Persistence Manager, the Persistence Manager typically needs to be notified when it is necessary to flush its cached state to the database or other persistent store prior to a transaction commit. In order to do so, the Persistence Manager registers a `javax.transaction.Synchronization` object with the container by using the `javax.transaction.Transaction.registerSynchronization()` method.

If the Persistence Manager has registered a synchronization object, the container will invoke the `Synchronization.beforeCompletion()` method at the start of the transaction completion process. This will typically occur after the `beforeCompletion()` method is invoked on the `Synchronization` object registered by the container with the Transaction Manager in use by the container. (See Section 9.12.4.) This method executes in the same transaction context as the business method on behalf of which the commit is being executed. The Persistence Manager should use this notification to validate the cached persistent state of the transaction's entity beans and their associated dependent objects against the database state. If the respective states are not consistent, the Persistence Manager should mark the transaction for rollback using the `Transaction.setRollbackOnly()` method on the transaction object for the transaction. If the states are consistent, the Persistence Manager must flush the persistent state of the entity beans to the database or other persistent store and close the connections that it is using for the given transaction context.

The `Synchronization.afterCompletion()` method will be invoked by the container after the transaction is committed or rolled back. The `status` argument of the `afterCompletion()` method indicates the outcome of the transaction. The Persistence Manager can use this method to perform cleanup tasks or other management of its cached state.

9.11.5 Container responsibilities

The Container must provide an implementation of the `javax.transaction.TransactionManager` interface for use by the persistence manager. The Container must make the `TransactionManager` interface available to the persistence manager in the JNDI name space as `java:pm/TransactionManager`.

The container must ensure that the `getTransaction()` method returns a valid (non-null) `javax.transaction.Transaction` object that the Persistence Manager can use to identify the transaction context that is in effect, independent of whether the container is using distributed (JTA) transactions or a local transaction optimization. The use of a local transaction optimization strategy on the part of the container is not visible to the persistence manager.

The container is responsible for making the resource manager connection factories (e.g., `javax.sql.DataSource`) available to the persistence manager in the persistence manager's JNDI context.

The container must provide the implementation of the `getConnection` methods as part of the resource manager connection factory implementation that is provided to the persistence manager. In a typical implementation, the container's implementation of the resource manager connection factory interface should note the transactional context and identity of the persistence manager's calling thread and delegate to the resource-specific resource manager connection factory interface, described in [11] and [12].

It is the responsibility of the container (not the persistence manager) to manage transactions on all connections acquired from any resource manager connection factory that provides container management of the transactional enlistment of connections.

9.11.6 Persistence manager responsibilities

The persistence manager may register a `javax.transaction.Synchronization` object with the transaction manager.

The persistence manager may only use the `getTransaction()` and `getStatus()` methods of the `javax.transaction.TransactionManager` interface. If the persistence manager calls any other method of the `javax.transaction.TransactionManager` interface, the container must raise the `java.lang.IllegalStateException`.

The persistence manager may only use the `getStatus()`, `registerSynchronization()`, and `setRollbackOnly()` methods of the `javax.transaction.Transaction` interface. If the persistence manager calls any other method of the `javax.transaction.Transaction` interface, the container must raise the `java.lang.IllegalStateException`.

The persistence manager must not use any of the low-level XA and connection pooling interfaces on any resource manager connection factory or connection. These low-level interfaces are intended for the integration of a resource adaptor (e.g., JDBC driver) with the container and are not for use by the persistence manager.

The persistence manager is responsible for the transaction management of all connections that are obtained by the persistence manager from any resource manager connection factory that provides non-enlisted transactions.

9.11.7 Additional contracts between the Container and the Persistence Manager

As described in Section 9.6, the Persistence Manager interacts with the Container to receive notifications related to the lifecycle of the managed beans. The current EJB architecture, however, does not architect the full set of SPIs between the Container and the Persistence Manager: these interfaces are currently left to the Container Provider and Persistence Manager Provider.

The EJB 2.0 architecture, however, assumes that certain functionality is provided by the Container to the Persistence Manager through such SPIs.

For example, given a primary key for an entity whose abstract persistence schema is managed by the Persistence Manager, the Persistence Manager will typically need to request from the Container the EJBObject that corresponds to that primary key and entity bean instance in the given transaction context in order to implement the get accessor methods for cmr-fields and the `ejbSelect` methods that return EJBObjects. (Note that it is not sufficient for the Persistence Manager to invoke the `findByPrimaryKey` method in this case, since that method may run in a different transaction context.)

9.12 Object interaction diagrams

This section uses object interaction diagrams to illustrate the interactions between an entity bean instance, its persistence manager, and its container.

9.12.1 Notes

The object interaction diagrams illustrate a box labeled “container-provided classes.” These classes are either part of the container or are generated by the container tools. These classes communicate with each other through protocols that are container implementation specific. Therefore, the communication between these classes is not shown in the diagrams.

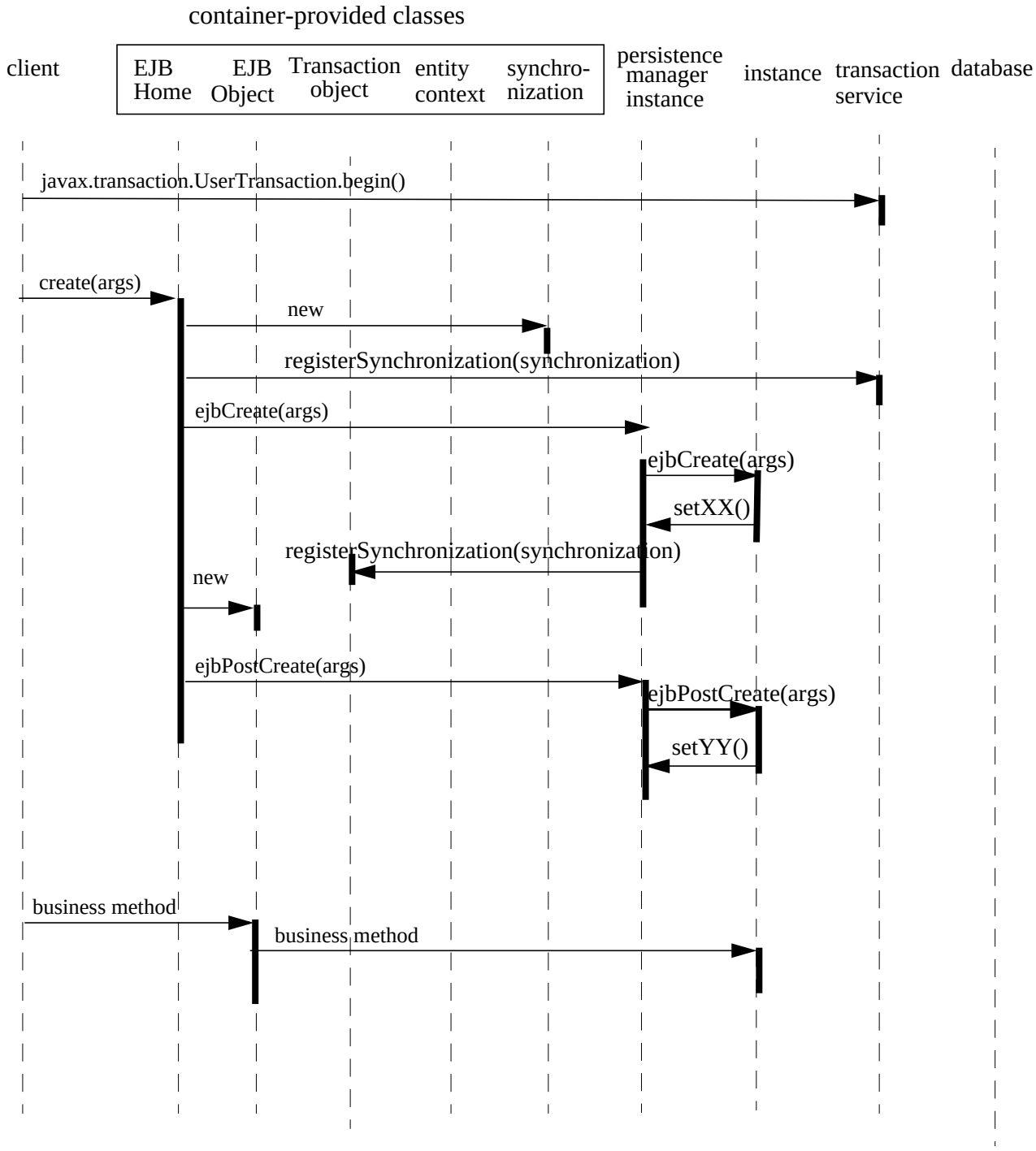
The class labeled “Transaction object” denotes the transaction object that the persistence manager obtains from the container by invoking the `getTransaction()` method on the transaction manager object provided to the persistence manager by the container.

The classes labeled “instance” and “persistence manager instance” denote those portions of the entity bean class as seen or provided by the Bean Provider and the persistence manager’s generated code respectively.

The classes shown in the diagrams should be considered as an illustrative implementation rather than as a prescriptive one.

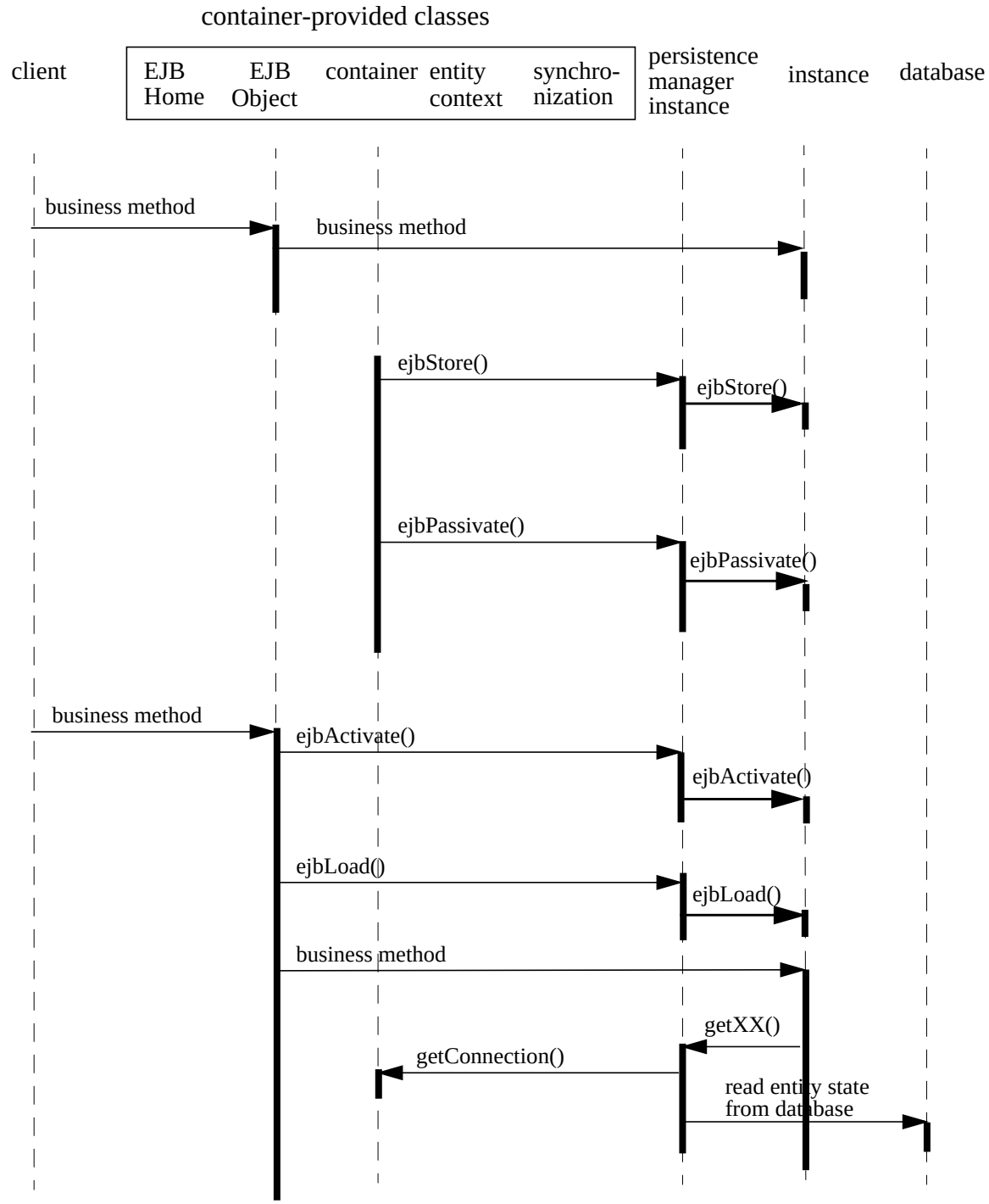
9.12.2 Creating an entity object

Figure 27 OID of creation of an entity object with container-managed persistence



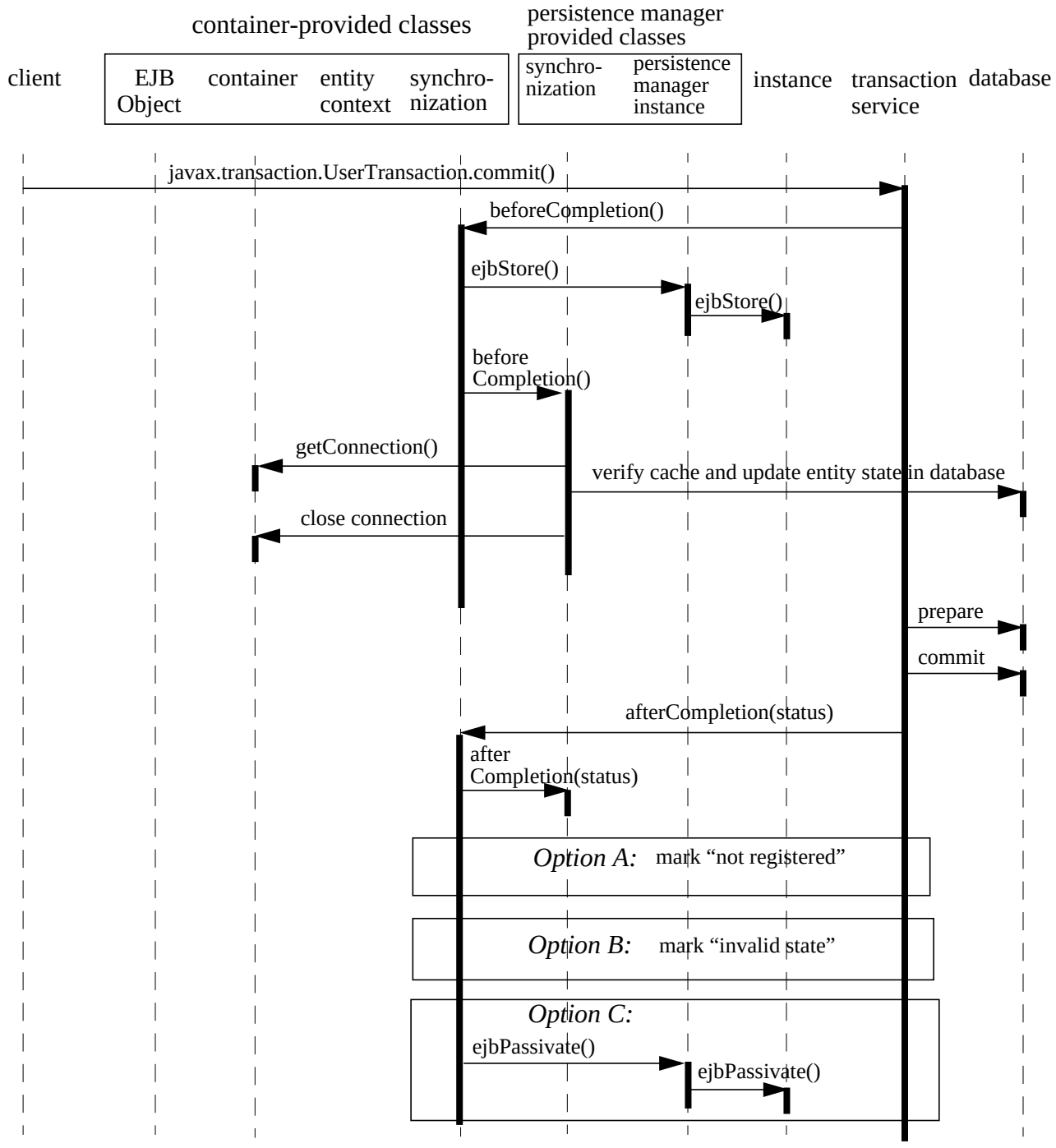
9.12.3 Passivating and activating an instance in a transaction

Figure 28 OID of passivation and reactivation of an entity bean instance with container managed persistence



9.12.4 Committing a transaction

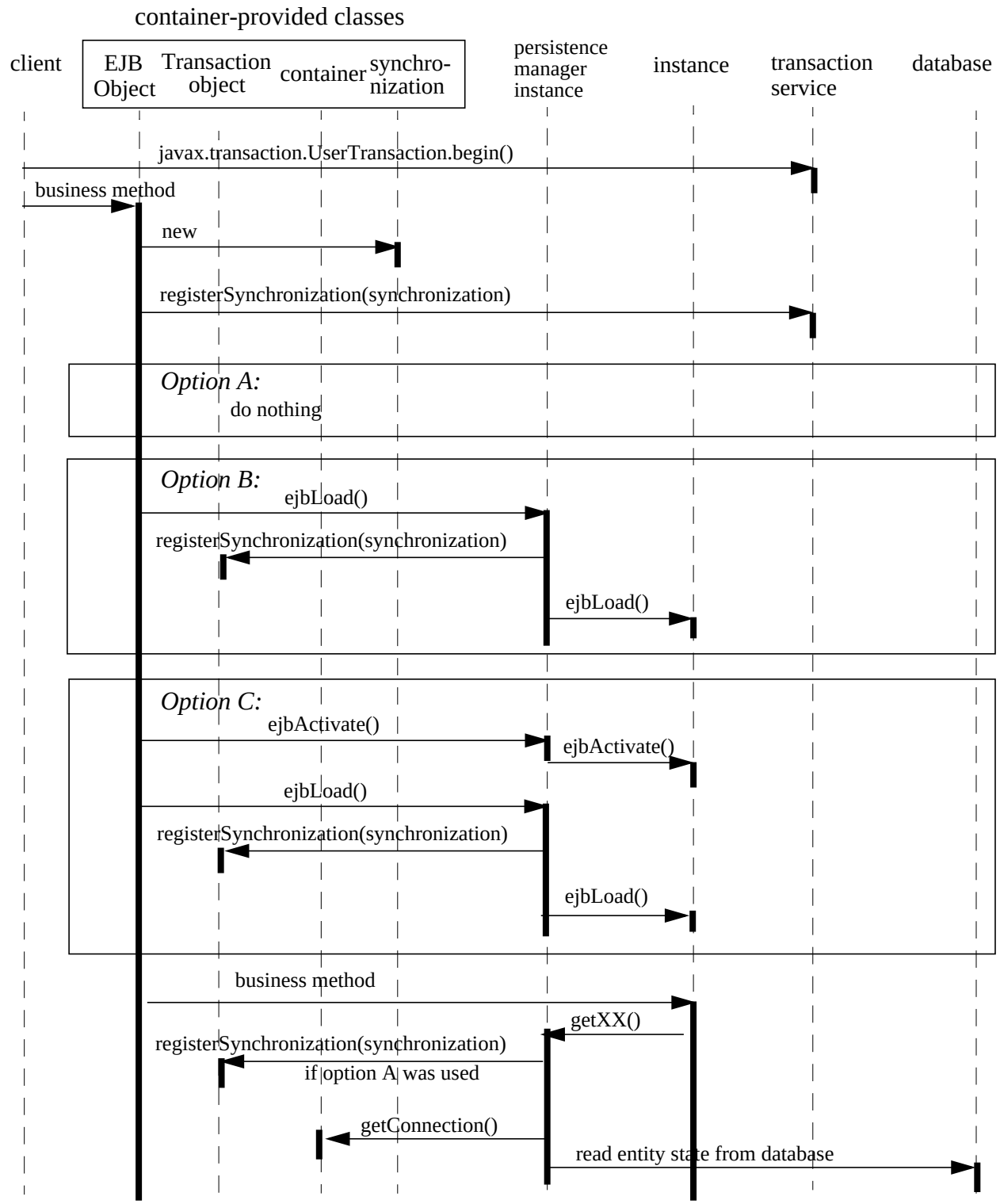
Figure 29 OID of transaction commit protocol for an entity bean instance with container-managed persistence



9.12.5 Starting the next transaction

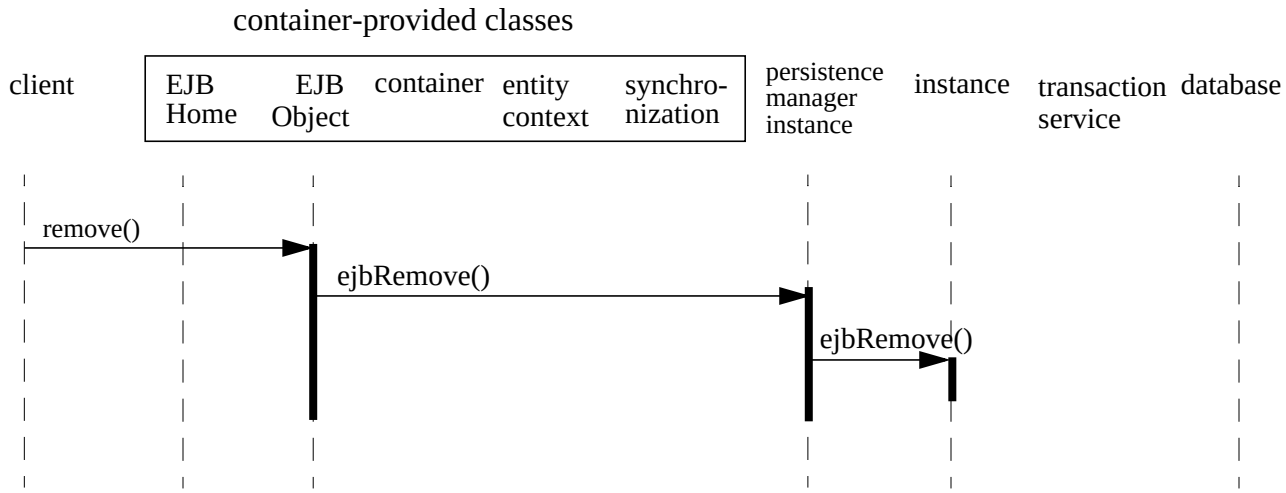
The following diagram illustrates the protocol performed for an entity bean instance with container-managed persistence at the beginning of a new transaction. The three options illustrated in the diagram correspond to the three commit options in the previous subsection.

Figure 30 OID of start of transaction for an entity bean instance with container-managed persistence



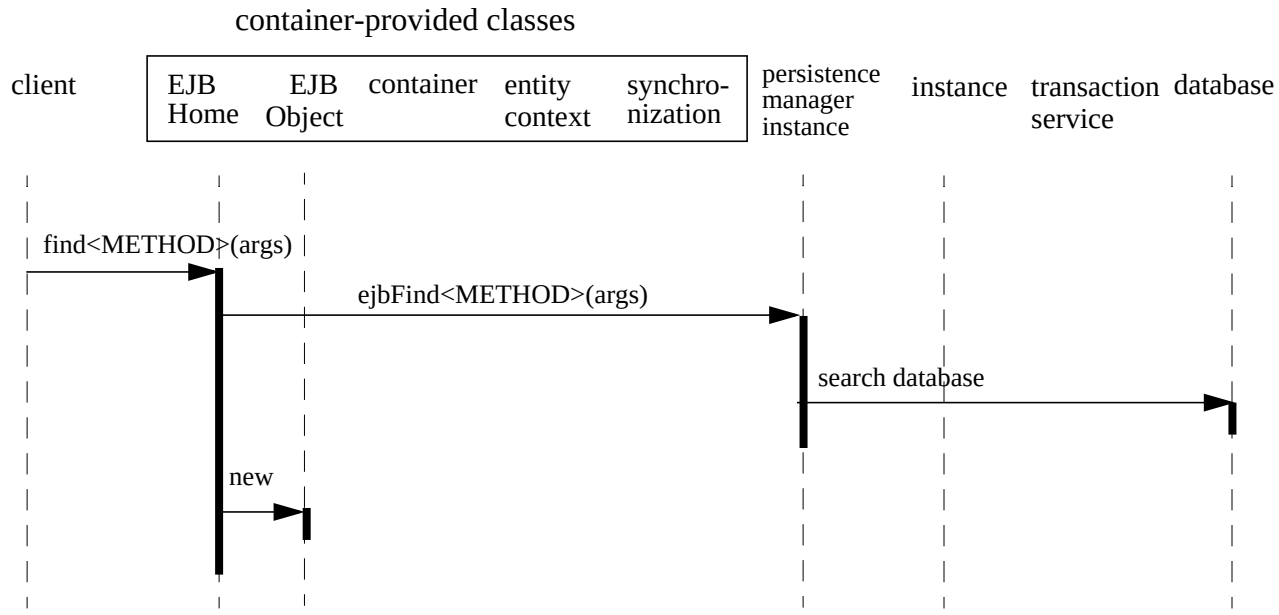
9.12.6 Removing an entity object

Figure 31 OID of removal of an entity bean object with container-managed persistence



9.12.7 Finding an entity object

Figure 32 OID of execution of a finder method on an entity bean instance with container-managed persistence



9.12.8 Adding and removing an instance from the pool

The diagrams in Subsections 9.12.2 through 9.12.7 did not show the sequences between the “does not exist” and “pooled” state (see the diagram in Section 9.6.1).

Figure 33 OID of a container adding an instance to the pool

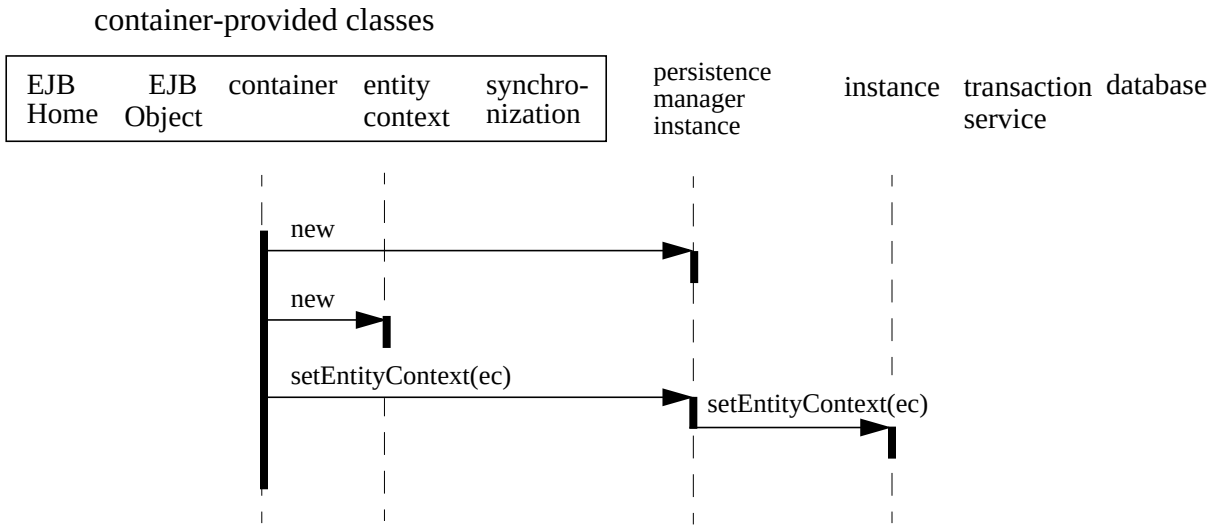
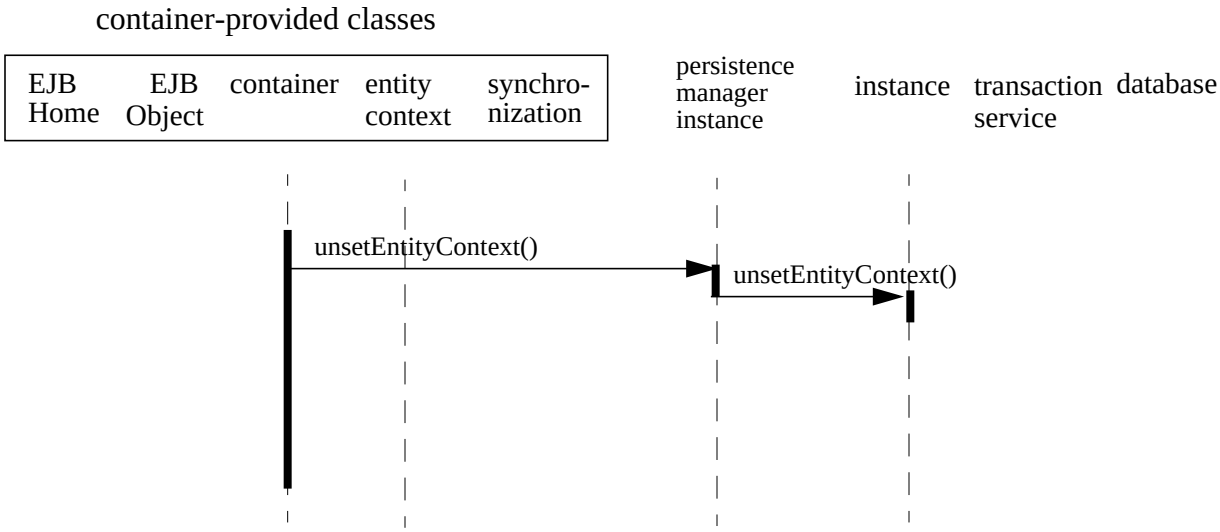


Figure 34 OID of a container removing an instance from the pool



Chapter 10

EJB QL: EJB Query Language for Container Managed Persistence Query Methods

The Enterprise Java Beans query language (EJB QL) defines query methods (finder and select methods) for entity beans with container managed persistence. EJB QL defines query methods so that they are portable across containers and persistence managers. EJB QL is a declarative, SQL-like language intended to be compiled to the target language of the persistent data store used by a persistence manager. It is based on a subset of SQL92 which is enhanced by path expressions that allow navigation over the relationships defined for entity beans and dependent object classes.

10.1 Overview

The Enterprise JavaBeans query language, EJB QL, is used to define queries for entity beans with container managed persistence. EJB QL lets the Bean Provider specify the semantics of query methods in a portable way.

EJB QL is a specification language that can be compiled to a target language, such as SQL, of a persistent store used by a persistence manager. This allows the responsibility for the execution of queries to be shifted to the native language facilities provided for the persistent store (e.g., RDBMS), instead of requiring queries to be executed directly on the persistent manager's representation of the entity beans' state. As a result, query methods are optimizable as well as portable.

The Enterprise JavaBeans query language uses the abstract persistence schemas of entity beans and dependent object classes, including their relationships, for its data model. It defines operators and expressions based on this data model.

The Bean Provider uses EJB QL to write queries based on the abstract persistence schemas and the relationships defined in the deployment descriptor. EJB QL depends on navigation and selection based on the cmp-fields and cmr-fields of the abstract schema types of related entity beans and dependent objects. The Bean Provider can navigate from an entity bean or dependent object to other dependent objects or beans by using the names of cmr-fields in EJB QL queries.

EJB QL allows the Bean Provider to use the abstract schema types of related entity beans in a query if the abstract persistence schemas of the related beans are defined in the same deployment descriptor as the query. The Bean Provider can navigate both to such locally-defined entity beans and to remote entity beans. (In this context, remote entity beans are entity beans with bean managed persistence, entity beans using EJB 1.1 container managed persistence, and beans whose abstract persistence schemas are defined in a different deployment descriptor.) Although the abstract persistence schemas of remote entity beans are not available to the Bean Provider, it is still possible to use EJB QL to navigate to such remote entity beans. In addition, special expressions in the language allow the Bean Provider to invoke the finder methods of remote entity beans in queries.

EJB QL queries can be used in two different ways:

- as queries for selecting existing entity objects through finder methods in the home interface. This use of EJB QL allows the results of a finder query to be usable by the clients of an entity bean.

- as queries for selecting objects or values derived from an entity bean or dependent object's abstract schema type. This use of EJB QL allows the Bean Provider to find objects or values related to the state of an entity bean or dependent object without exposing the results to the client.

An EJB QL query is a string which may contain a SELECT clause, a FROM clause, and a WHERE clause.

10.2 EJB QL Definition

EJB QL uses a SQL-like syntax to select objects or values based on the abstract schema types and relationships of entity beans and dependent objects[16]. The path expressions of EJB QL allow the Bean Provider to navigate over relationships defined by the cmr-fields of the abstract schema types of entity beans and dependent object classes.

This chapter provides the full definition of the language.

An EJB QL query is a string which may consist of the following three clauses:

- a SELECT clause, which indicates the types of the objects or values to be selected.
- a FROM clause, which provides navigation declarations that designate the domain to which the conditional expression specified in the WHERE clause of the query applies.
- a WHERE clause, which restricts the results that are returned by the query.

Of these three clauses, only the FROM clause is always required.

In BNF syntax, an EJB QL query is defined as:

EJB QL :: = [select_clause] from_clause [where_clause]

The clauses shown in the square brackets [] are optional. An EJB QL query must always have a FROM clause. There must be a SELECT clause when a query is defined for an `ejbSelect` method.

It is possible to parse and validate EJB QL queries before entity beans are deployed because EJB QL is based on the abstract schema types of entity beans and dependent object classes.

EJB QL is a typed expression language. Every expression in EJB QL has a type. The type of the expression is derived from the structure of the expression; the abstract schema types of the range variable declarations; the types to which the cmp-fields and cmr-fields evaluate; and the types of literals. The allowable types in EJB QL are the *abstract schema types* of entity beans and dependent objects, the defined types of cmp-fields, and the entity object types of remote entity beans.

The abstract schema type of an entity bean is derived from its entity bean type and the information provided in the deployment descriptor. Similarly, the abstract schema type of a dependent object is derived from its dependent object class and the deployment descriptor information.

Informally, the abstract schema type of an entity bean or dependent object can be characterized as follows:

[16] Arbitrary constructors which allow the Bean Provider to create new objects, flatten structures, map dependent object class values to dependent value class values, etc., are not within the current design goals of EJB QL.

- *For every get accessor method of the entity bean (dependent object class) that corresponds to a* `cmp-field` *element in the deployment descriptor, there is a field ("cmp-field") whose type is the same as that designated by the* `cmp-field` *element.*

- *For every get accessor method of the entity bean (dependent object class) that corresponds to a* `cmr-field` *element in the deployment descriptor, there is a field ("cmr-field") whose type is as follows:*
    - *if the* `role-source` *element specifies a* `dependent-name` *element, the abstract schema type of the dependent object class designated by the* `dependent-name` *element (or, if the role has a multiplicity of* `Many`*, a collection of such).*
    - *if the* `role-source` *element specifies an* `ejb-name` *element, the abstract schema type of the entity bean designated by the* `ejb-name` *element (or, if the role has a multiplicity of* `Many`*, a collection of such).*
    - *if the* `role-source` *element specifies a* `remote-ejb-name` *element, the remote interface type of the entity bean designated by the* `remote-ejb-name` *element (or, if the role has a multiplicity of* `Many`*, a collection of such).*

These types are specific to the EJB QL data model only. The Persistence Manager is not required to implement or otherwise materialize the abstract schema types.

As noted in Section 10.1, EJB QL is used for two types of query methods:

- Finder methods—Finder methods are defined in the home interface of an entity bean and return entity objects.

- Select methods—Select methods are a special type of query method not exposed to the client. The Bean Provider uses select methods to select the persistent state of an entity bean or dependent object class or to select entity objects or dependent objects that are related to the entity bean or dependent object for which the query is defined.

The syntax of an EJB QL query differs slightly depending on the type of query method for which it is defined. In particular,

- A finder method, which is defined in the home interface of an entity bean, must return either a remote object representing an entity object or a collection of remote objects representing a collection of entity objects. The result type of such a finder method is determined by the entity bean for which it is defined. The EJB QL query string for such a finder method may therefore not require a SELECT clause. (See Section 10.2.8.)

- A select method is defined on an entity bean class or dependent object class and is not exposed to the client of the entity bean. A query for a select method must have a SELECT clause which specifies the type of values to be selected.

For a more detailed discussion of the different types of query methods and restrictions on their result types, see Section 9.6.7.

An EJB QL query has parameters that correspond to the parameters of the finder or select method for which it is defined.

An EJB QL query is statically defined in the `ejb-ql` deployment descriptor element.

10.2.1 Abstract schema types and query domains

EJB QL is a typed expression language whose design is based on the type model of EJB 2.0 container managed persistence, with one difference. That difference is as follows: EJB QL queries use the abstract schema types of entity beans that are "local" and the remote types of the entity beans that are "remote." "Local" entity beans are those whose abstract persistence schemas are defined within the same deployment descriptor as the query. "Remote" entity beans are those whose abstract persistence schemas are not specified within the same deployment descriptor. Navigation within local beans (using cmr-fields) results in values of the related entity beans' abstract schema types rather than their remote types.

The domain of an EJB QL query is

- The abstract schema types of all entity beans and dependent object classes defined in the same deployment descriptor. These entity beans are considered to be local.
- All entity beans whose abstract persistence schemas are not defined within the same deployment descriptor as the query, but which participate in relationships defined within the deployment descriptor. These entity beans are considered to be *remote* because they are accessed remotely via finder methods on their home interface.

The Bean Provider creates an ejb-jar file which contains a deployment descriptor describing several entity beans, their relationships and their dependent objects. EJB QL assumes that a single deployment descriptor in an ejb-jar file constitutes a nondecomposable unit for the persistence manager responsible for implementing the abstract persistence schemas of the entity beans, their dependent objects, and the relationships defined in the deployment descriptor and the ejb-jar file. Therefore, queries can be written by utilizing navigation over the cmr-fields of related beans (and dependent object classes) supplied in the same ejb-jar by the Bean Provider because they are implemented and managed by the same persistence manager.

The domain of a query may be restricted by the *navigability* of the relationships of the entity bean or dependent object class on which it is based. The cmr-fields of an entity bean or dependent object's abstract schema type determine navigability. Using the cmr-fields and their values, a query can select related dependent objects and entity beans and use their abstract schema types in the query. The queries can be specified over the navigable relationships described in the deployment descriptor.

10.2.2 Naming

Entity beans and dependent object classes are designated in EJB QL query strings as follows:

- The abstract schema type of an entity bean that has an abstract persistence schema defined within the same deployment descriptor is designated by the bean's `abstract-schema-name`, such as `OrderBean`.
- The abstract schema type of a dependent object class is designated by its `dependent-name`.
- A remote entity bean that is referenced by a finder expression is designated by its `remote-ejb-name` (as contained in the `ejb-entity-ref` deployment descriptor element).
- A local entity bean that is referenced by a finder expression is designated by its `ejb-name`.

Table 7 illustrates the naming convention recommended for an entity bean and used in the examples that follow.

Table 7 Entity Bean Naming Conventions

Entity bean for an Order	As used in EJB QL
Entity bean's name (`ejb-name`)	OrderEJB
Abstract schema name (`abstract-schema-name`)	OrderBean
Implementation class	com.acme.ecommerce.OrderBean (not used)
entity bean's remote type	com.acme.ecommerce.Order (implicitly inferred but not explicitly named in EJB QL)

The development process includes the naming of local and remote entity beans. The Bean Provider assigns unique names to entity beans and to dependent object classes so that they can be used within queries. These unique names are scoped within the deployment descriptor file.

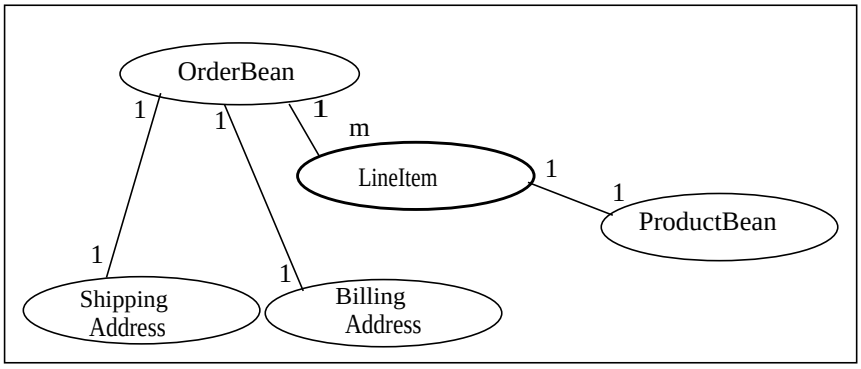
To enable queries to be written, the EJB 2.0 specification introduces a naming approach that differs slightly from the EJB 1.1 naming approach. The application assembler must not change the values assigned to any of the following deployment descriptor elements when resolving `ejb-links`: `abstract-schema-name`, `dependent-name`, `ejb-name`, and `remote-ejb-name`. The linking convention for resolving ejb-names is discussed in Section 19.3.2.

10.2.3 Examples

The following naming convention refers to entity beans in subsequent examples: An entity bean as a whole is designated by `<name>EJB` and its implementation class and abstract schema type are designated by `<name>Bean`.

The first example, case (1), assumes that the Bean Provider provides two entity beans, `OrderEJB` and `ProductEJB`. The same deployment descriptor defines the entity beans `OrderEJB` and `ProductEJB`; their abstract schema types, `OrderBean` and `ProductBean`, respectively; and the dependent object classes `LineItem`, `ShippingAddress`, and `BillingAddress` (with abstract schema types `LineItem`, `ShippingAddress`, and `BillingAddress` respectively). These beans are logically in the same ejb-jar file, as shown in Figure 35.

Figure 35 Two beans, OrderEJB and ProductEJB, with abstract persistence schemas in the same ejb-jar file.



In this example, `ProductEJB` is co-located with `OrderEJB`. The abstract schema types of both beans are available to the Bean Provider and the Bean Provider can use the cmp-fields and cmr-fields of the abstract schema types of the related beans in queries. The dependent object classes `ShippingAddress` and `BillingAddress` are used in two different one-to-one relationships by `OrderBean`. There is also a one-to-many relationship between `OrderBean` and `Lineitem`. The dependent object class `LineItem` is related to `ProductEJB` in a one-to-one relationship.

EJB QL allows the Bean Provider to specify finder queries for `OrderEJB` by navigating over the cmr-fields defined in `OrderBean` and `LineItem`. A finder method query to find all orders with pending line items might be written as follows:

```
FROM OrderBean o, l IN o.lineItems
WHERE l.shipped = FALSE
```

This query navigates over the cmr-field `lineItems` of the abstract schema type `OrderBean` to find line items, and uses the cmp-field `shipped` of `LineItem` to select those orders that have at least one line item that has not yet shipped. (Note that this query does not select orders that have no line items.)

Note that although predefined reserved identifiers, such as FALSE, FROM, WHERE and IN appear in upper case in this example, they are case insensitive.

This example does not use a SELECT clause because the query is defined for a finder method on the entity bean's home interface and it contains only one reference to the entity bean's abstract schema type using a range variable. Such queries must always return entity objects of the bean type for which the query is defined.

Because the same deployment descriptor defines the abstract persistence schemas of both beans, the Bean Provider can also specify a query for `OrderEJB` that utilizes the abstract schema type of `ProductEJB`, and hence the cmp-fields and cmr-fields of the abstract schema types `OrderBean` and `ProductBean`. This can be done because the abstract schema types `OrderBean` and `ProductBean` are related to each other by means of their relationships with `LineItem`. For example, if the abstract schema type `ProductBean` has a cmp-field named `product_type`, a finder query for `OrderEJB` can be specified using this cmp-field. Such a finder query might be: "Find all orders for products which have the product type *office supplies*". An EJB QL query string for this might be:

```
FROM OrderBean o, l IN o.lineItems
WHERE l.product.product_type = 'office_supplies'
```

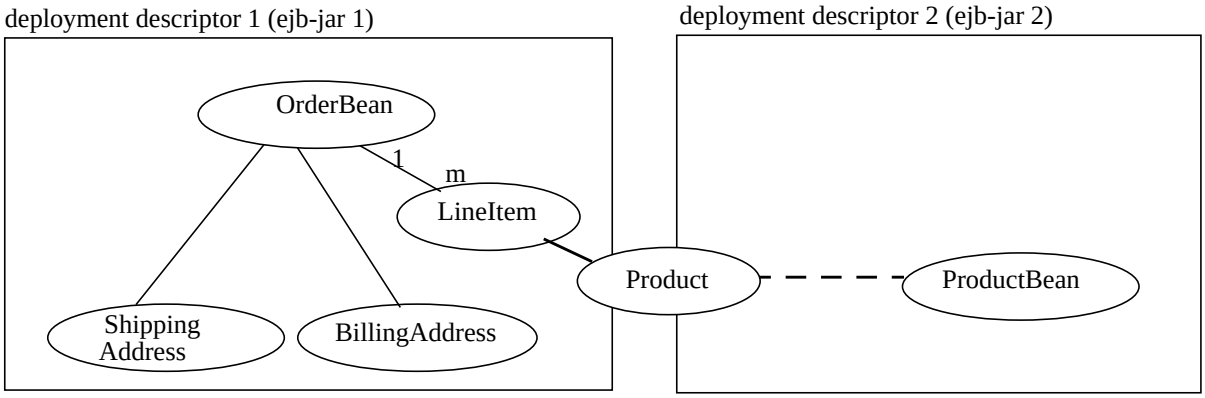
This query is specified by using the `abstract-schema-name` for `OrderEJB`, namely `OrderBean`, which designates the abstract schema type over which the query ranges. The basis for the navigation is provided by the cmr-fields `lineItems` and `product` of the abstract schema types `OrderBean` and `LineItem` respectively.

> *Note that the deployment descriptor describes the relationships between entity beans and dependent objects using the name of the entity bean, namely the* `ejb-name` *element defined for an entity bean. In this example,* `OrderEJB` *and* `ProductEJB` *are the ejb-names of the entity beans. Container managed persistence allows one-way navigable relationships from dependent objects and entity beans to remote entity beans (entity beans with bean managed persistence, entity beans using EJB 1.1 container managed persistence, and beans whose abstract persistence schemas are defined in a different deployment descriptor). The Bean Provider designates a unique name for such a remote bean by using the* `remote-ejb-name` *deployment descriptor element. See Section 9.4.14 for a discussion of the use of the* `ejb-name` *and* `remote-ejb-name` *deployment descriptor elements in defining relationships.*

EJB QL also allows navigation to entity beans whose abstract persistence schemas are not defined in the same deployment descriptor. The Bean Provider can write queries that utilize relationships to such remote entity beans. The Bean Provider, however, cannot access the abstract schema types of the remote beans because the abstract schema types are not available, but he or she can define queries referencing remote entity beans using special navigation expressions within EJB QL.

Figure 36 illustrates referencing remote entity beans. This example, case (2), uses the same entity beans as in the previous example. However, in this case a different deployment descriptor defines the abstract persistence schema of `ProductEJB`. The relationship to `ProductEJB` is defined using a `remote-ejb-name`. The Bean Provider can utilize the abstract schema type `OrderBean` and the dependent object classes `LineItem`, `ShippingAddress`, and `BillingAddress` in queries, but cannot use the abstract schema type `ProductBean` because it is not known by the Bean Provider. However, the Bean Provider can find related remote objects of `ProductEJB` and can also utilize finder methods defined on `ProductEJB` to locate remote entity objects.

Figure 36 The abstract persistence schemas of OrderEJB and ProductEJB are in different deployment descriptors, and hence two different ejb-jar files.



The remainder of this chapter illustrates the capabilities of EJB QL using sample queries based on the extended examples in case (1) and case (2). By using `ProductEJB` in different ways, the sample queries illustrate the EJB QL usage differences between local beans and remote beans. The following sections cover the use of abstract schema types and remote types within EJB QL in detail.

10.2.4 The FROM clause and navigational declarations

In an EJB QL query, the FROM clause defines the scope of the query by declaring identification variables. The FROM clause designates the domain of the query, which may be constrained by path expressions.

The identification variables declared in the FROM clause designate instances of a particular type. The FROM clause can contain multiple identification variable declarations separated by a comma (`,`).

*from_clause::=***FROM** *identification_variable_declaration*
*[, identification_variable_declaration]**
identification_variable_declaration ::= collection_member_declaration |
range_variable_declaration
collection_member_declaration ::= identifier **IN** *collection_valued_path_expression |*
identifier **IN** *collection_valued_reference_expression*
*range_variable_declaration :: { abstract_schema_name | dependent_name} [**AS**] identifier*

We discuss identifiers and identification variables first because they are key constructs in the FROM clause. Following this discussion, we turn to collection member declarations and range variable declarations.

10.2.4.1 Identifiers

An identifier is a character sequence of unlimited length. The character sequence must begin with a Java identifier start character and all other characters must be Java identifier part characters. An identifier start character is any character for which the method `Character.isJavaIdentifierStart` returns true. This includes the underscore (_) character and the dollar sign ($) character. An identifier part character is any character for which the method `Character.isJavaIdentifierPart` returns true. The question mark (?) character is reserved for use by EJB QL.

The following are the reserved identifiers in EJB QL: *SELECT*, *FROM*, *WHERE*, *NULL*, *TRUE*, *FALSE*, *NOT*, *AND*, *OR*, *BETWEEN*, *LIKE*, *IN*, *AS*, *FROM*, *WHERE*, *UNKNOWN*, *EMPTY*, and *IS*.

10.2.4.2 Identification variables

An identification variable is a valid identifier declared in the FROM clause of an EJB QL query. An identification variable may be declared using the special operators IN and, optionally, AS.

All identification variables must be declared in the FROM clause. They cannot be declared in other clauses.

Identification variables are identifiers. An identification variable must not be a reserved identifier or have the same name as any of the following:

- ejb-name
- remote-ejb-name
- abstract-schema-name
- dependent-name

Like other identifiers, identification variables are case insensitive.

An identification variable evaluates to a value of the type of the expression used in declaring the variable. For example, recall the FROM clause of the example EJB QL finder query for `OrderEJB`:

```
FROM OrderBean o, l IN o.lineItems
```

In the declaration `l IN o.lineItems`, the identification variable `l` evaluates to any `LineItem` value directly reachable from `OrderBean`. The cmr-field `lineItems` is a reference to the collection of `LineItem` dependent objects and the identification variable `l` refers to an element of this collection. The type of `l` is the abstract schema type of `LineItem`.

Identification variables designate a member of an abstract schema type of an entity bean or dependent object class or an element of a collection. Identification variables are existentially quantified in an EJB QL query.

An identification variable thus always designates a reference to a single value. It is declared in one of two ways:

1. A range variable is declared using the abstract schema name of an entity bean or the dependent name of a dependent object class, where abstract schema name and dependent name are the values of the `abstract-schema-name` and `dependent-name` deployment descriptor elements of the entity bean or dependent object class respectively.

2. A collection member identification variable is declared using a collection-valued path expression or collection-valued reference expression.

The identification variable declarations are evaluated from left to right in the FROM clause, and an identification variable declaration can use the result of a preceding identification variable declaration of the query string.

10.2.4.3 Range variable declarations

An identification variable can range over the abstract schema type of an entity bean or a dependent object class. The syntax for declaring an identification variable as a range variable is similar to SQL; optionally, it uses the AS operator.

Objects or values that are related to an entity bean or a dependent object are typically obtained by navigation using path expressions. However, navigation does not reach all objects. Dependent objects may be "detached," that is, not related to other dependent objects or entity beans. The Bean Provider can utilize a range variable (i.e., an identification variable declared by a *range_variable_declaration*) to range over all instances of a dependent object abstract schema type to select dependent objects regardless of whether they can be reached using navigation. Range variable declarations thus allow the Bean Provider to designate a "root" for objects which may not be reachable by navigation.

Multiple range variable declarations are useful for queries where the Bean Provider needs to compare multiple values ranging over the same abstract schema type. See Section 10.2.6.

10.2.4.4 Collection member declarations

An identification variable, when declared by a *collection_member_declaration*, ranges over values of a collection obtained by navigation using a path expression. A path expression represents a navigation involving the cmr-fields of the abstract schema type of a local entity bean or a dependent object class. Because a path expression can be based on another path expression, the navigation can use the cmr-fields of related entity beans and dependent object classes. Path expressions are covered in Section 10.2.4.6.

An identification variable of a collection member declaration is declared using a special operator, the reserved identifier `IN`, followed by a collection-valued path expression or a collection-valued reference expression. The path expression evaluates to a collection type specified as a result of navigation to a collection-valued cmr-field in the abstract schema type of an entity bean or dependent object class.

For example, the FROM clause for a query defined for `OrderEJB` might contain the following identification variable declaration clause:

```
l IN o.lineItems
```

In this example, `lineItems` is the name of a cmr-field whose value is a collection of instances of the abstract schema type of the `LineItem` dependent object class. The identification variable `l` designates a member of this collection, a *single* `LineItem` instance. In this example, `o` is an identification variable of the abstract schema type of OrderBean.

Note, however, that the identification variable declaration `p IN lineItems.product` is illegal because `lineItems` evaluates to a collection and path expressions cannot be further defined using collection-valued path expressions.

10.2.4.5 Example

The following FROM clause contains two identification variable declaration clauses. The identification variable declared in the first clause is used in the second clause. The clauses declare the variables `o` and `l` respectively. The range variable declaration `OrderBean AS o` designates the identification variable `o` as a range variable whose type is the abstract schema type of `OrderBean`. The identification variable `l` has the abstract schema type of `LineItem`. Because the clauses are evaluated from left to right, the identification variable `l` can utilize the results of the navigation on `o`.

```
FROM OrderBean AS o, l IN o.lineItems
```

10.2.4.6 Path expressions

An identification variable followed by a navigation operator and a cmp-field or cmr-field is a path expression.

EJB QL has two navigation operators used in constructing a path expression:

- The dot operator (`.`) designates navigation to cmr-fields and cmp-fields. The dot operator can only be used for navigation within the abstract schema type of an entity bean or dependent object class. It cannot be used for navigation to entity objects; hence, it cannot be used for navigation to remote entity beans.

- The `=>` operator designates navigation to remote beans (entity objects).

Depending on navigability, a path expression that leads to a cmr-field may be further composed using the navigation operators (`.` and `=>`). Path expressions can be composed from other path expressions only if the original path expression evaluates to a single valued type (not a collection) corresponding to a cmr-field. The type of the path expression is the type computed as the result of navigation; that is, the type of a cmp-field or a cmr-field to which the expression navigates. A path expression that ends in a cmp-field is terminal and cannot be further composed.

Path expressions comprise not only single-valued path expressions and collection-valued path expressions, as described below, but also single-valued reference expressions and collection-valued reference expressions. These latter path expressions allow navigation to remote types and are described in Section10.2.4.7 “Path expressions that reference remote interface types” .

The syntax for single valued path expressions and collection valued path expressions is defined as follows:

single_valued_path_expression ::=
{single_valued_navigation|identification_variable}.cmp_field |
single_valued_navigation
single_valued_navigation ::=
identification_variable.[single_valued_cmr_field.] single_valued_cmr_field*
collection_valued_path_expression ::=
*identification_variable.[single_valued_cmr_field.]*collection_valued_cmr_field)*

A *single_valued_cmr_field* is designated by a cmr-field name in a one-to-one or many-to-one relationship. This type of expression evaluates to a single value of the abstract schema type of the related entity bean or dependent object class. The type of the expression is the abstract schema type of the related entity bean or dependent object.

A *collection_valued_cmr_field* is designated by a cmr-field in a one-to-many or a many-to-many relationship. The type of the expression is the abstract schema type of the related entity bean or dependent object. The type of a *collection_valued_cmr_field* is a collection of values of the designated type.

Navigation to related entity beans whose abstract persistence schemas are defined in the same deployment descriptor always results in a value of the related entity bean's abstract schema type. Navigation using the dot (`.`) operator in a path expression can be used to a cmr-field that refers to an entity bean only if that entity bean's abstract persistence schema is defined within the same deployment descriptor. Navigation using the dot (`.`) operator to cmr-fields that refer to entity bean remote interface types is not allowed. This is because the abstract persistence schema of an entity bean that is not co-located in the same deployment descriptor is unavailable. Navigation to a remote entity bean is possible however using the `=>` operator, which provides navigation to the remote interface type of a related entity bean. See Section 10.2.4.7.

In the example, where the abstract persistence schemas of `ProductEJB` and `OrderEJB` are defined in the same deployment descriptor, if `l` is an identification variable representing an instance of type `LineItem`, the path expression `l.product` has the abstract schema type `ProductBean`. If the abstract persistence schemas of the two beans are not defined in the same deployment descriptor, as in case (2) of Section 10.2.1, the path expression `l.product` is invalid because the dot navigation operator (`.`) cannot be used to navigate to the remote interface type of an entity bean.

The evaluation of a path expression to a cmp-field results in the Java type designated by the cmp-field. The expression `l.product.name` in the example thus has the type `java.lang.String` as a result of navigational composition to the cmp-field `name`.

It is syntactically illegal to compose a path expression from a path expression that evaluates to a collection. For example, if `o` designates `OrderBean,` the path expression `o.lineItems.product` is illegal since navigation to `lineItems` results in a collection. This case should produce an error when the EJB QL query string is verified. To handle such a navigation, an identification variable must be declared to range over the elements of the `lineItems` collection in the FROM clause. Another path expression must be designated to navigate over each such element in the WHERE clause of the query, as follows:

```
FROM OrderBean AS o, l in o.lineItems
WHERE l.product.name = 'widget'
```

10.2.4.7 Path expressions that reference remote interface types

The EJB QL navigation operator `=>` is used to navigate to instances of an entity bean's remote type. The Bean Provider uses this operator in an expression to obtain the remote interface of a related entity bean (that is, the entity object). The `=>` operator is used to navigate to remote entity beans or to local entity beans that are to be treated as remote. References to remote interfaces can only be handled by using the `=>` operator.

In the example, when the identification variable `l` designates an instance of type `LineItem`, the expression `l=>product` evaluates to the type of the remote interface for `ProductEJB`, namely `Product`. Its value is an instance of `Product`; that is, an entity object.

The Bean Provider can use path expressions that reference remote interface types to access entity objects *remotely*, even if the abstract persistence schema of the entity bean is defined in the same deployment descriptor as the query. Therefore, the expression `l=>product` is valid for both case (1) and case (2) discussed in Section 10.2.2.

EJB QL does not allow further navigation from remote entity beans. Therefore, path expressions that reference remote interface types are terminal.

Note that only collection-valued reference expressions can be used in declaring identification variables. Single-valued reference expressions can be used in the SELECT clause and the WHERE clause of queries.

Path expressions that reference remote interface types have the following form:

single_valued_reference_expression ::=
{single_valued_navigation | identification_variable} **=>** *single_valued_cmr_field*
collection_valued_reference_expression ::=
{single_valued_navigation | identification_variable} **=>** *collection_valued_cmr_field*

10.2.5 WHERE clause and conditional cxpressions

The WHERE clause of a query consists of a conditional expression used to select objects or values that satisfy the expression. Thus, the WHERE clause restricts the result of a query.

A WHERE clause is defined as follows:

where_clause ::= **WHERE** *conditional_expression*

The following sections describe the language constructs used in the conditional expressions of an EJB QL query.

10.2.5.1 Literals

A string literal is enclosed in single quotes—for example: 'literal'. A string literal that includes a single quote is represented by two single quotes—for example: 'literal''s'. EJB QL string literals are like Java `String` literals in that they use unicode character encoding.

An exact numeric literal is a numeric value without a decimal point, such as 57, -957, +62. Exact numeric literals support numbers in the range of Java `long`. Exact numeric literals use the Java integer literal syntax.

An approximate numeric literal is a numeric value in scientific notation, such as 7E3, -57.9E2, or a numeric value with a decimal, such as 7., -95.7, +6.2. Approximate numeric literals support numbers in the range of Java `double`. Approximate literals use the Java floating point literal syntax.

The Bean Provider may utilize appropriate suffixes to indicate the specific type of the literal in accordance with the Java Language Specification.

The boolean literals are TRUE and FALSE.

Although predefined reserved literals appear in upper case, they are *case insensitive*.

10.2.5.2 Identification variables

All identification variables used in the WHERE clause of an EJB QL query must be declared in the FROM clause, as described in Section 10.2.4.2.

Identification variables are existentially quantified in the WHERE clause. This means that an identification variable represents a member of a collection or an instance of an entity bean or dependent object's abstract schema type. An identification variable never designates a collection in its entirety.

10.2.5.3 Path expressions

It is illegal to use a *collection_valued_path_expression* within a WHERE clause as part of a conditional expression except in an *empty_collection_comparison_expression*.

It is illegal to use a *collection_valued_reference_expression* within a WHERE clause as part of a conditional expression except in an *empty_collection_comparison_expression*.

10.2.5.4 Input parameters

The following rules apply to input parameters. Input parameters can only be used in the WHERE clause of a query.

- Input parameters are designated by the question mark (`?`) prefix followed by an integer. For example: `?1`.
- The number of distinct input parameters in an EJB QL query must not exceed the number of input parameters for the finder or select method. It is not required that the EJB QL query use all input parameters for the finder or select method.
- Input parameters must be numbered starting from 1.
- An input parameter evaluates to the type of the corresponding parameter defined in the signature of the finder or select method with which the query is associated.
- Input parameters can only be used in conditional expressions involving single-valued path expressions or single-valued reference expressions.

10.2.5.5 Conditional expression composition

Conditional expressions are composed of themselves, comparison operations, logical operations, path expressions that evaluate to boolean values, and boolean literals.

Arithmetic expressions can be used in comparison expressions and are composed of themselves, arithmetic operations, path expressions that evaluate to numeric values, and numeric literals.

A finder expression can be used only within a conditional expression.

Standard bracketing ( ) for ordering expression evaluation is supported.

Conditional expressions are defined as follows:

conditional_expression ::= conditional_term | conditional_exp ***OR*** *conditional_term*
conditional_term ::= conditional_factor | conditional_term ***AND*** *conditional_factor*
conditional_factor ::= [***NOT*** *] conditional-test*
*conditional_test :: = conditional_primary [****IS*** *[* ***NOT*** *] {****TRUE*** *|* ***FALSE*** *|* ***UNKNOWN****}]*
conditional_primary ::= simple_cond_expression | (conditional_expression)
simple_cond_expression ::= comparison_expression | between_expression | like_expression |
in_expression | null_comparison_expression |
empty_collection_comparison_expression

10.2.5.6 Operators and operator precedence

The operators are listed below in order of increasing precedence.

- Logical operators in precedence order are: `NOT`, `AND`, `OR`
- Comparison operators are : `=`, `>`, `>=`, `<`, `<=`, `<>` (not equal)
- Arithmetic operators in precedence order:
  +, - unary
  *, / multiplication and division
  +, - addition and subtraction
  Arithmetic operations must use Java numeric promotion.
- Navigation operator (`.`)
- Remote interface reference operator `=>`

The following sections describe other operators used in specific expressions.

10.2.5.7 Between expressions

The syntax for the use of the comparison operator [NOT] BETWEEN in an arithmetic expression is as follows:

*arithmetic_expression [**NOT**] **BETWEEN** arithmetic-expr **AND** arithmetic-expr*

Examples are:

`p.age BETWEEN 15 and 19` is equivalent to `p.age >= 15 AND p.age <= 19`

`p.age NOT BETWEEN 15 and 19` is equivalent to `p.age < 15 OR p.age > 19`

If the value of an arithmetic expression used in a between expression is NULL, the value of the BETWEEN expression is unknown.

10.2.5.8 In expressions

The syntax for the use of the comparison operator [NOT] IN in a comparison expression is as follows:

*single_valued_path_expression [**NOT**] **IN** (string-literal1, string-literal2,...) |*
*single_valued_reference_expression [**NOT**] **IN** (finder_expression)*

The *single_valued_path_expression* must have a *String* value.

The *single_valued_reference_expression* must have a remote interface object type value.

Examples are:

`o.country IN ('UK', 'US', 'France')` is true for `UK` and false for `Peru`, and is equivalent to the expression `(o.country = 'UK') OR (o.country = 'US') OR (o.country = 'France')`.

`o.country NOT IN ('UK', 'US', 'France')` is false for `UK` and true for `Peru`, and is equivalent to the expression `NOT ((o.country = 'UK') OR (o.country = 'US') OR (o.country = 'France'))`.

There must be at least one string-literal in the comma separated string literal list that defines the set of values for the `IN` expression.

If the value of a single-valued path expression or single-valued reference expression in an IN or NOT IN expression is `NULL`, the value of the expression is unknown.

See Section 10.2.5.11 for information on finder expressions. See Section 10.3.3 for an example of the use of IN expressions with finder expressions.

10.2.5.9 Like expressions

The syntax for the use of the comparison operator [NOT] LIKE in a conditional expression is as follows:

single_valued_path_expression ***[NOT] LIKE*** *pattern-value* ***[ESCAPE*** *escape-character]*

The *single_valued_path_expression* must have a *String* value. The *pattern-value* is a string literal 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. The optional *escape-character* is a single character string literal and is used to escape the special meaning of the underscore and percent characters in *pattern-value*.

Examples are:

- *address.phone LIKE '12%3'* is true for '123' '12993' and false for '1234'
- *asentence.word LIKE 'l_se'* is true for 'lose' and false for 'loose'
- *aword.underscored LIKE '_%' ESCAPE '\'* is true for '_foo' and false for 'bar'
- *address.phone NOT LIKE '12%3'* is false for '123' and '12993' and true for '1234'

If the value of the *single_valued_path_expression* is `NULL`, the value of the LIKE expression is unknown.

10.2.5.10 Null comparison expressions

The syntax for the use of the comparison operator IS NULL in a conditional expression is as follows:

single_valued_path_expression **IS NULL | *single_valued_path_expression* IS NOT NULL**

A null comparison expression tests whether or not the single valued path expression is a `NULL` value.

Path expressions containing `NULL` values during evaluation return `NULL` values.

10.2.5.11 Empty collection comparison expressions

The syntax for the use of the comparison operator IS EMPTY in an *empty_collection_comparison_expression* is as follows:

collection_valued_path_expression ***IS [NOT] EMPTY|***
collection_valued_reference_expression ***IS [NOT] EMPTY***

This expression tests whether or not the collection designated by the collection-valued path expression or collection-valued reference expression is empty (that is, it has no elements).

This is the only type of expression where a collection-valued path or reference expression can be used in the WHERE clause.

The collection designated by the collection-valued path expression used in an empty collection comparison expression must not be used in the FROM clause for the declaration of an identification variable. An identification variable declared as a member of a collection implicitly designates the existence of a non-empty relationship; testing whether the same collection is empty is contradictory. Therefore, the following query is invalid.

```
FROM OrderBean o, l in o.lineItems
WHERE o.lineItems IS EMPTY
```

If the value of the collection-valued path expression or collection-valued reference expression in an empty collection comparison expression is `NULL`, the value of the empty comparison expression is unknown.

10.2.5.12 Finder expressions

A finder expression is used to evaluate an entity bean's finder method. Finder expressions allow queries to invoke the finder methods of an entity bean's home interface. A finder expression, however, *cannot* be used to invoke the select method of an entity bean or dependent object class.

A finder expression has the following syntax:

*EjbName****>>****finder_method_name(*arg_1*, ..., *arg_n*).*

EjbName designates an entity bean. It must correspond to either an `ejb-name` that specifies an entity bean whose abstract persistence schema is defined in the deployment descriptor or a `remote-ejb-name` that specifies an entity bean in an `ejb-entity-ref` element. The operator >> locates the home interface of the bean and calls the finder method on the designated entity bean.

The finder method of the entity bean is designated by *finder_method_name*. The arguments must match those of the signature of the finder method of the entity bean designated by *EjbName*.

Finder method arguments must be literals representing numeric values, string values, the boolean values `TRUE` and `FALSE`, constructor expressions, or input variables. The constructors used in constructor expressions are restricted to the following: `Boolean`, `Byte`, `Integer`, `Long`, `Short`, `Float`, and `Double`. These constructors in finder expressions take only literals as arguments. (Note that this is more restrictive than their Java definitions.)

For example, the following is a valid finder expression: the argument to the finder is an object of type `Integer`.

```
ProductEJB >> findByQuantity(new Integer(13000))
```

The Bean Provider must not use other types of arguments, such as path expressions, in finder expressions. For example, the following query is illegal.

```
SELECT o
FROM OrderBean AS o, l IN o.lineItems
WHERE l=>product IN
     (ProductEJB >> findByMakerAndCity(?1,o.shipping_address.city))
```

The Bean Provider must not write a finder query that includes a finder expression using the bean's own finder methods.

10.2.5.13 Functional expressions

EJB QL includes the following built-in functions[17].

String Functions:

- `CONCAT(String, String)` returns a `String`
- `SUBSTRING(String, start, length)` returns a `String`
- `LOCATE(String, String [, start])` returns an `int`
- `LENGTH(String)` returns an `int`

Note that `start` and `length` designate the positions in a string defined by an `int`.

Arithmetic Functions:

- `ABS(number)` returns a number (`int`, `float`, or `double`)
- `SQRT(double)` returns a `double`

[17] These functions are a subset of the functions defined for JDBC 2.0 drivers, as described in Appendix A in JDBC[API] API tutorial and Reference, Second Edition.

10.2.6 SELECT clause

An EJB QL query string used for a select method requires a SELECT clause. The SELECT clause is optional for a finder query if the FROM clause contains a single range variable that ranges over the abstract schema type of the entity bean for which the finder method is defined. Otherwise, a SELECT clause is required for a finder query.

The SELECT clause defines the types of values to be returned by the query. The valid return type for a SELECT clause is determined by the `ejbSelect<METHOD>` method, as described in Section 9.6.7, or by the entity bean with which the finder method is associated.

The SELECT clause has the following syntax:

select_clause ::=
SELECT {single_valued_path_expression |
single_valued_reference_expression |
identification_variable |
@@identification_variable}

The SELECT clause determines the type of the values returned by a query. The type of a single-valued path expression or identification variable specified in the body of a SELECT clause cannot be an entity bean's abstract schema type. For example, the following query is illegal:

```
SELECT l.product FROM OrderBean AS o, l IN o.lineItems
```

The Bean Provider, however, can return *remote* objects (that is, entity object types), as in the following query:

```
SELECT l=>product FROM OrderBean AS o, l IN o.lineItems
```

The `ejbSelect<METHOD>` method for which this query is specified returns a collection of remote objects of type `Product`.

Note that the SELECT clause must be specified to return a single-valued expression. The following query returns a collection of all line items that are related to some order.

```
SELECT l FROM OrderBean AS o, l IN o.lineItems
```

The query below, however, is not valid:

```
SELECT o.lineItems FROM OrderBean AS o
```

It is the responsibility of the Persistence Manager to map the types returned by the query to the Java types that are returned by the `ejbFind` or `ejbSelect` method with which the query is associated.

Unlike select methods, finder methods always return entity objects. If the FROM clause contains one identification variable designating the entity bean's abstract schema type, the selected entity object values are implicit in the query definition and the finder query does not require a SELECT clause.

However, if the Bean Provider wants to select values by comparing more than one instance of an entity bean abstract schema type, the query string requires a SELECT clause to designate the result. This case is identified by the use of more than one range variable declaration in the FROM clause of the entity bean's abstract schema type. When there is more than one range variable designating the entity bean's abstract schema type in the FROM clause, the finder query must have a SELECT clause.

An EJB QL query cannot return an entity bean's abstract schema type in the SELECT clause. This is because finder and select methods cannot return values (or collections of values) that correspond to entity bean abstract schema types in the programming model. To ensure proper typing of returned values, EJB QL uses the `@@` operator to designate casting of an identification variable that corresponds to an entity bean's abstract schema type to an entity object.

The following finder method query returns orders whose quantity is greater than the order quantity for John Smith. This example illustrates the use of two different range variables, both of the abstract schema type `OrderBean`, and the use of the @@ operator.

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

In EJB QL, this type of casting is used only to designate the appropriate return type for an entity bean.

Because finder methods cannot return arbitrary types, the SELECT clause of an EJB QL query defined for a finder method must always have the same type as that of the remote interface type of the entity bean for which the finder method is defined. In contrast, select methods can return dependent object types or other values, in addition to entity bean remote types.

For example, the following EJB QL returns all line items regardless of whether a line item is related to an order or a product:

```
SELECT l
FROM LineItems AS l
```

The following example returns all line items related to an order:

```
SELECT l
FROM OrderBean o, l IN o.lineItems
```

10.2.7 Null values

When the target of a reference does not exist in the persistent store, its value is regarded as NULL. SQL 92 NULL semantics [21] defines the evaluation of conditional expressions containing NULL values. The following is a brief description of these semantics:

- Comparison or arithmetic operations with an unknown value always yield an unknown value.
- Path expressions that contain NULL values during evaluation return NULL values.
- The IS NULL and IS NOT NULL operators convert a NULL cmp-field or single-valued cmr-field value into the respective TRUE or FALSE value.
- Boolean operators use three valued logic, defined by Table 8, Table 9, and Table 10.
- Conditional tests use three valued logic, defined by Table 11.

Table 8 Definition of the AND operator

AND	T	F	U
T	T	F	U
F	F	F	F
U	U	F	U

Table 9 Definition of the OR operator

OR	T	F	U
T	T	T	T
F	T	F	U
U	T	U	U

Table 10 Definition of the NOT operator

NOT	
T	F
F	T
U	U

Table 11 Definition of the conditional test

conditional test	expression value		
	T	**F**	**U**
expression IS TRUE	T	F	F
expression IS FALSE	F	T	F
expression IS UNKNOWN	F	F	T

10.2.8 Equality semantics

EJB QL only permits *like* type values to be compared. There is one exception to this rule. It is valid to compare exact numeric values and approximate numeric values (the rules of Java numeric promotion define the required type conversion). The conditional expression is disallowed when attempting the comparison of non-like type values except for this numeric case.

According to the semantics of value comparison, values are compared with respect to their representation in Java, not according to their representation in the persistent data source. For example, numeric primitive types cannot be assumed to have `NULL` values. If the Bean Provider wishes to allow null values for cmp-fields, he or she should specify those cmp-fields to have the equivalent Java object types instead of primitive types; for example, `Integer` rather than `int`.

`String` and `Boolean` comparison is restricted to = and <>. Two strings are equal if and only if they contain the same sequence of characters. This is different from SQL.

Two entity beans of the same abstract schema type are equal if and only if they have the same primary key value.

Two dependent objects of the same abstract schema type are equal if and only if they have the same primary key value.

Two remote objects of the same type are considered equal if and only if the entity objects have the same primary key value.

10.2.9 Restrictions

Date and time values should use the standard Java `long` millisecond value. A date or time literal in an EJB QL query should be an integer literal for a millisecond value. The standard way to produce millisecond values is to use `java.util.Calendar`.

Although SQL supports fixed decimal comparison in arithmetic expressions, EJB QL does not. For this reason EJB QL restricts exact numeric literals to those without a decimal point (and numerics with a decimal point as an alternate representation for approximate numeric values).

EJB QL does not support the use of comments.

The container managed persistence data model does not currently support inheritance. Therefore, entity beans, dependent objects, or value classes of different types cannot be compared. EJB QL queries that contain such comparisons are invalid.

10.3 Examples

The following examples illustrate the syntax and semantics of EJB QL. These examples are based on the example presented in Section 10.2.3.

10.3.1 Simple queries

Find all orders:

```
FROM OrderBean o
```

Find all orders that need to be shipped to California:

```
FROM OrderBean o
WHERE o.shipping_address.state = 'CA'
```

10.3.2 Queries with dependent object classes

Find all orders which have line items:

```
FROM OrderBean o, l in o.lineItems
```

Note that the result of this query does not include orders with no associated line items.

The above query can also be written as:

```
FROM OrderBean o
WHERE o.lineItems IS NOT EMPTY
```

Find all orders that have no line items:

```
FROM OrderBean o
WHERE o.lineItems IS EMPTY
```

Find all pending orders:

```
FROM OrderBean o, l in o.lineItems
WHERE l.shipped = FALSE
```

Find all orders in which the shipping address differs from the billing address. This example assumes that the Bean Provider uses two distinct dependent object classes to designate shipping and billing addresses.

```
FROM OrderBean o
WHERE
NOT (o.shipping_address.state = o.billing_address.state AND
     o.shipping_address.city = o.billing_address.city AND
     o.shipping_address.street = o.billing_address.street)
```

If the Bean Provider uses a single class `Address` in two different relationships for both the shipping address and the billing address, the above expression can be simplified based on the equality rules defined in Section 10.2.8. The query can then be written as:

```
FROM OrderBean o
WHERE o.shipping_address <> o.billing_address
```

The query checks whether the same dependent object (identified by its primary key) is related to an order through two distinct relationships.

10.3.3 Queries that refer to other entity beans

Consider the query: “Find all orders for a book titled ‘Applying Enterprise JavaBeans: Component-Based Development for the J2EE Platform’”.

The following example illustrates the case where the same deployment descriptor defines the query and the abstract persistence schemas of OrderEJB and ProductEJB. The Bean Provider can use the abstract schema types of both entity beans. This is similar to Case (1) in 10.2.2.

```
FROM OrderBean o, l IN o.lineItems
WHERE l.product.type = ‘book’ AND
     l.product.name = ‘Applying Enterprise JavaBeans:
     Component-Based Development for the J2EE Platform’
```

The query below is written using a finder method of ProductEJB. This query finds a book by its name and its author. The ProductEJB bean may or may not be co-located in the same deployment desriptor. The finder expression locates the required entity object in either case.

```
FROM OrderBean o, l IN o.lineItems
WHERE l=>product IN (ProductEJB >> findByNameAndType(‘Applying Enter-
prise
     JavaBeans: Component-Based Development for the J2EE Platform’,
‘book’))
```

The expression `l=>product` evaluates to the remote interface type of ProductEJB, which is Product. The finder expression evaluates to a collection of instances of the remote interface of the entity bean ProductEJB. The comparison operator `IN` is evaluated by testing equality based on the primary keys of the respective entity objects.

10.3.4 Queries using input parameters

A query similar to the previous one may be written using the input parameters of the finder method, as follows:

```
FROM OrderBean o, l IN o.lineItems
WHERE l=>product IN (ProductEJB>>findByNameAndType(?1, ?2))
```

The following query finds the orders for a product designated by an input parameter:

```
FROM OrderBean o, l IN lineItems
WHERE l=>product = ?1
```

For this query, the input parameter must be the remote type of the `ProductEJB` entity bean.

10.3.5 Queries for select methods

The following select queries illustrate the selection of values other than entity objects. Select methods are internal to an entity bean or dependent object's implementation class.

The following EJB QL query selects all products that have been ordered.

```
SELECT l=>product
FROM OrderBean o, l IN o.lineItems
```

To restrict this query, the Bean Provider may provide parameters to the select method, and hence to the query. The following query finds all products in the order specified by a particular order number. The order number is specified by a parameter that corresponds to the primary key of OrderBean.

```
SELECT l=>product
FROM OrderBean o, l IN o.lineItems
WHERE o.ordernumber = ?1
```

Depending on the return type of the select method, the returned collection may contain duplicates. The collection returned by a finder or select method should not contain duplicates when the type returned by the query corresponds to an entity object or a dependent object class, since entity objects and dependent object classes are identified by a primary key. However, collections of other types may contain duplicates. It is the responsibility of the Persistence Manager to ensure that duplicates are removed when necessary (that is, when the result type of the method is `java.util.Set`).

Consider the following example:

```
SELECT o.shipping_address.city
FROM OrderBean o
```

This query returns the names of all the cities of the shipping addresses of all orders. The result type of the `ejbSelect<METHOD>` method, which is either `java.util.Collection` or `java.util.Set`, determines whether the query may return duplicate city names.

10.3.6 EJB QL and SQL

EJB QL, like SQL, treats the FROM clause as a cartesian product. The Bean Provider must use caution in defining identification variables and restricting the domain of the query in the FROM clause. The FROM clause is similar to that of SQL in that the declared identification variables affect the results of the query even if they are not used in the WHERE clause. Even if the Bean Provider defines range variables that have types other than the entity beans' abstract schema types and the Bean Provider does not use identification variables in the WHERE clause, the domain of the query still depends on whether there are any values of the declared type.

The FROM clause in the following example defines a query over all OrderBeans that have line items and existing products. If there is no ProductBean instance in the persistent store, the domain of the query is empty and no OrderBean is selected.

```
FROM OrderBean AS o, l in o.lineItems, ProductBean p
```

The Persistence Manager can represent the abstract schemas of a set of entity beans and dependent object classes in an application using a relational database. There are multiple ways to define a mapping to RDBMS tables. Although this area is beyond the scope of this specification, a sample mapping and translation of EJB QL to SQL is provided to clarify the semantics of EJB QL.

A typical mapping strategy from a set of entity beans and dependent object classes to a RDBMS might be to map each entity bean and dependent object class to a separate table. One-to-many relationships may be represented by foreign keys in the related table from the many side and many-to-many relationships may be represented by using an auxiliary table that contains the primary keys of the related objects.

Because the FROM clause represents a cartesian product, the SQL result may contain duplicates. However, since every entity bean and dependent object has a primary key and is unique, it is possible to eliminate duplicates from the result. Therefore, the Persistence Manager would typically utilize a SELECT DISTINCT clause in translating an EJB QL query to SQL when selecting entity beans or dependent objects. Collections of other values, such as primitive Java types, can contain duplicates if the return type of the `ejbSelect<METHOD>` is `java.util.Collection`.

The following translation example illustrates the mapping of entity beans and dependent objects to RDBMS tables. The entity bean OrderEJB is represented by the table ORDER and the dependent object class LineItem is represented by the table LINEITEM. The column OKEY represents the primary key for OrderBean, FKEY represents the foreign key column of LINEITEM that holds the values of the ORDER primary keys. FKEY is defined in the LINEITEM table to model the one-to-many relationship.

Using this mapping, the following EJB QL finder query

```
FROM OrderBean o, l in o.lineItems
WHERE l.quantity > 5
```

may be represented in SQL as

```
SELECT DISTINCT o.OKEY
FROM ORDERBEAN o, LINEITEM l
WHERE o.OKEY = l.FKEY AND l.QUANTITY > 5
```

10.4 EJB QL BNF

EJB QL BNF notation summary:

- { ... } grouping
- [...] optional constructs
- **boldface** keywords

The following is the complete BNK notation for EJB QL:

EJB QL :: = [select_clause] from_clause [where_clause]
*from_clause::=***FROM** *identification_variable_declaration*
*[**,** identification_variable_declaration]**
identification_variable_declaration ::= collection_member_declaration |
range_variable_declaration
collection_member_declaration ::= identifier **IN** *collection_valued_path_expression |*
identifier **IN** *collection_valued_reference_expression*
*range_variable_declaration ::{abstract_schema_name | dependent_name } [***AS***] identifier*
single_valued_path_expression ::=
{single_valued_navigation|identification_variable}.cmp_field |
single_valued_navigation
single_valued_navigation ::=
identification_variable.[single_valued_cmr_field.] single_valued_cmr_field*
collection_valued_path_expression ::=
*identification_variable**.**[single_valued_cmr_field.]*collection_valued_cmr_field*
single_valued_reference_expression::=
{single_valued_navigation | identification_variable} **=>** *single_valued_cmr_field*
collection_valued_reference_expression::=
{single_valued_navigation | identification_variable} **=>** *collection_valued_cmr_field*
select_clause ::= **SELECT** *{single_valued_path_expression |*
single_valued_reference_expression |
identification_variable |
@@identification_variable }
where_clause ::= **WHERE** *conditional_expression*
conditional_expression ::= conditional_term | conditional_exp **OR** *conditional_term*
conditional_term ::= conditional_factor | conditional_term **AND** *conditional_factor*
conditional_factor ::= [**NOT** *] conditional-test*
*conditional_test :: = conditional_primary [***IS** *[* **NOT** *] {***TRUE** *|* **FALSE** *|* **UNKNOWN***}]*
conditional_primary ::= simple_cond_expression |(conditional_expression)
simple_cond_expression ::= comparison_expression | between_expression | like_expression |
in_expression | null_comparison_expression |
empty_collection_comparison_expression
between_expression ::=
*arithmetic-expression [***NOT***]* **BETWEEN**
arithmetic-expression **AND** *arithmetic-expression*
in_expression ::=
*single_valued_path_expression [***NOT***]* **IN** *(string-literal1, string-literal2,...) |*

*single_valued_reference_expression [***NOT***] IN (finder_expression)*

like_expression ::=

*single_valued_path_expression [***NOT***] LIKE pattern-value [***ESCAPE** *escape-character]*

null_comparison_expression ::=

single_valued_path_expression **IS NULL** *|*

single_valued_path_expression **IS NOT NULL**

*finder_expression ::= EjbName**>>**finder_method_name(*arg_1*, ..., *arg_n*)*

arg_i = { input_parameter | literal | constructor_expression}

constructor_expression = **new** *object_constructor(Literal)*

*object_constructor = {***Boolean** *|* **Byte** *|* **Integer** *|* **Long** *|* **Short** *|* **Float** *|* **Double***}*

empty_collection_comparison_expression ::=

{collection_valued_path_expression | collection_valued_reference_expression}

IS [**NOT**] **EMPTY**

comparison_expression ::=

string_expression {=|<>} string_expression |

boolean_expression {=|<>} boolean_expression |

datetime_expression { = | <> | > | <} datetime_expression |

reference_expression { = | <> } reference_expression |

single_value_designator comparison-operator single_value_designator

single_value_designator :: = scalar_expression

comparison_operator ::=

= | > | >= | < | <= | <>

scalar_expression ::= arithmetic_expression

arithmetic_expression ::= arithmetic_term | arithmetic_expression { + | -} arithmetic_term

*arithmetic_term :: = arithmetic_factor | arithmetic-term { * | /} arithmetic_factor*

arithmetic_factor :: = {+ |-} arithmetic_primary

arithmetic_primary ::= single_valued_path_expression | literal | (arithmetic_expression) | input_parameter | functions_returning_numerics

string_expression ::= string_primary

string_primary ::= single_valued_path_expression | literal | (string_expression) | functions_returning_strings | input_parameter

datetime_expression ::= single_valued_path_expression | input_parameter

boolean_expression ::= single_valued_path_expression | input_parameter | literal

reference_expression ::= single_valued_reference_expression | input_parameter

functions_returning_strings ::= **CONCAT***(string_expression, string_expression) |*

SUBSTRING*(string_expression,arithmetic_expression,arithmetic_expression)|*

functions_returning_numerics::=

LENGTH*(string_expression) |*

LOCATE*(string_expression, string_expression[, arithmetic_expression])*

ABS*(arithmetic_expression) |*

SQRT*(arithmetic_expression)*

Chapter 11 Entity Bean Component Contract for Bean Managed Persistence

The entity bean component contract for bean managed persistence is the contract between an entity bean and its container. It defines the life cycle of the entity bean instances and the model for method delegation of the client-invoked business methods. The main goal of this contract is to ensure that a component using bean managed persistence is portable across all compliant EJB Containers.

This chapter defines the enterprise Bean Provider's view of this contract and the Container Provider's responsibility for managing the life cycle of the enterprise bean instances. It also describes the Bean Provider's responsibilities when persistence is provided by the Bean Provider.

11.1 Overview of Bean Managed Entity Persistence

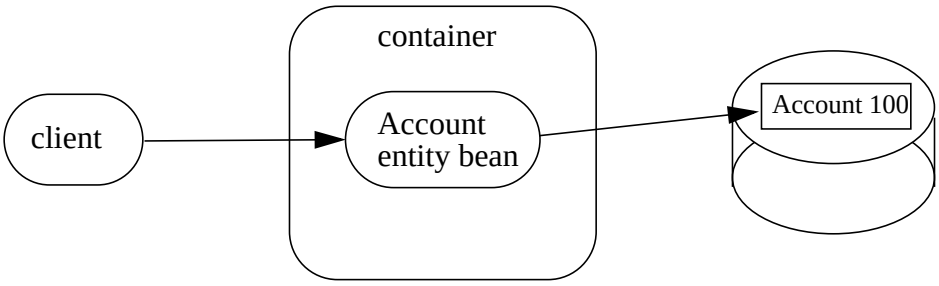
An entity bean implements an object view of an entity stored in an underlying database, or an entity implemented by an existing enterprise application (for example, by a mainframe program or by an ERP application). The data access protocol for transferring the state of the entity between the entity bean instances and the underlying database is referred to as object **persistence**.

The entity bean component protocol for bean managed persistence allows the entity Bean Provider to implement the entity bean's persistence directly in the entity bean class or in one or more helper classes provided with the entity bean class. This chapter describes the contracts for bean managed persistence. Container managed persistence, which allows the entity Bean Provider to delegate the management of the entity bean's persistence to the Container Provider and Persistence Manager, is discussed in Chapter 9.

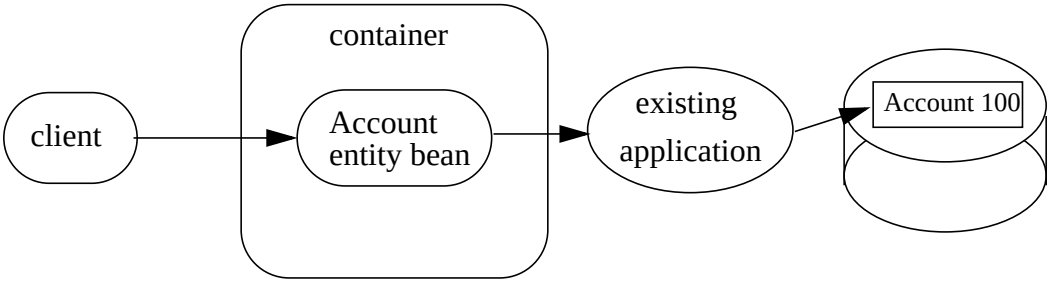
In many cases, the underlying data source may be an existing application rather than a database.

Figure 37 Client view of underlying data sources accessed through entity bean

(a) Entity bean is an object view of a record in the database



(b) Entity bean is an object view of an existing application



11.1.1 Granularity of entity beans

This section provides guidelines to the Bean Providers for modeling of business objects as entity beans.

In general, an entity bean should represent an independent business object that has an independent identity and lifecycle, and is referenced by multiple enterprise beans and/or clients.

A *dependent object* should not be implemented as an entity bean. Instead, a dependent object is better implemented as a Java class (or several classes) and included as part of the entity bean on which it depends.

A dependent object can be characterized as follows. An object B is a dependent object of an object A, if B is created by A, accessed only by A, and removed by A. This implies, for example, that if B exists when A is being removed, B is automatically removed as well. It also implies that other programs can access the object B only indirectly through object A. In other words, the object A fully manages the lifecycle of the object B.

For example, a purchase order might be implemented as an entity bean, but the individual line items on the purchase order should be implemented as helper classes, not as entity beans. An employee record might be implemented as an entity bean, but the employee address and phone number should be implemented as helper classes, not as entity beans.

The state of an entity object that has dependent objects is often stored in multiple records in multiple database tables.

In addition, the Bean Provider must take into consideration the following when making a decision on the granularity of an entity object:

> Every method call to an entity object via the remote and home interface is potentially a remote call. Even if the calling and called entity bean are collocated in the same JVM, the call must go through the container, which must create copies of all the parameters that are passed through the interface by value (i.e., all parameters that do not extend the `java.rmi.Remote` interface). The container is also required to check security and apply the declarative transaction attribute on the inter-component calls. The overhead of an inter-component call will likely be prohibitive for object interactions that are too fine-grained.

11.1.2 Entity Bean Provider's view of persistence and relationships

Using bean-managed persistence, the entity Bean Provider writes database access calls (e.g. using JDBC™ or SQLJ) directly in the entity bean component. The data access calls are performed in the `ejbCreate<METHOD>(...)`, `ejbRemove()`, `ejbFind<METHOD>()`, `ejbLoad()`, and `ejbStore()` methods; and/or in the business methods.

The data access calls can be coded directly into the entity bean class, or they can be encapsulated in a data access component that is part of the entity bean. Directly coding data access calls in the entity bean class may make it more difficult to adapt the entity bean to work with a database that has a different schema, or with a different type of database.

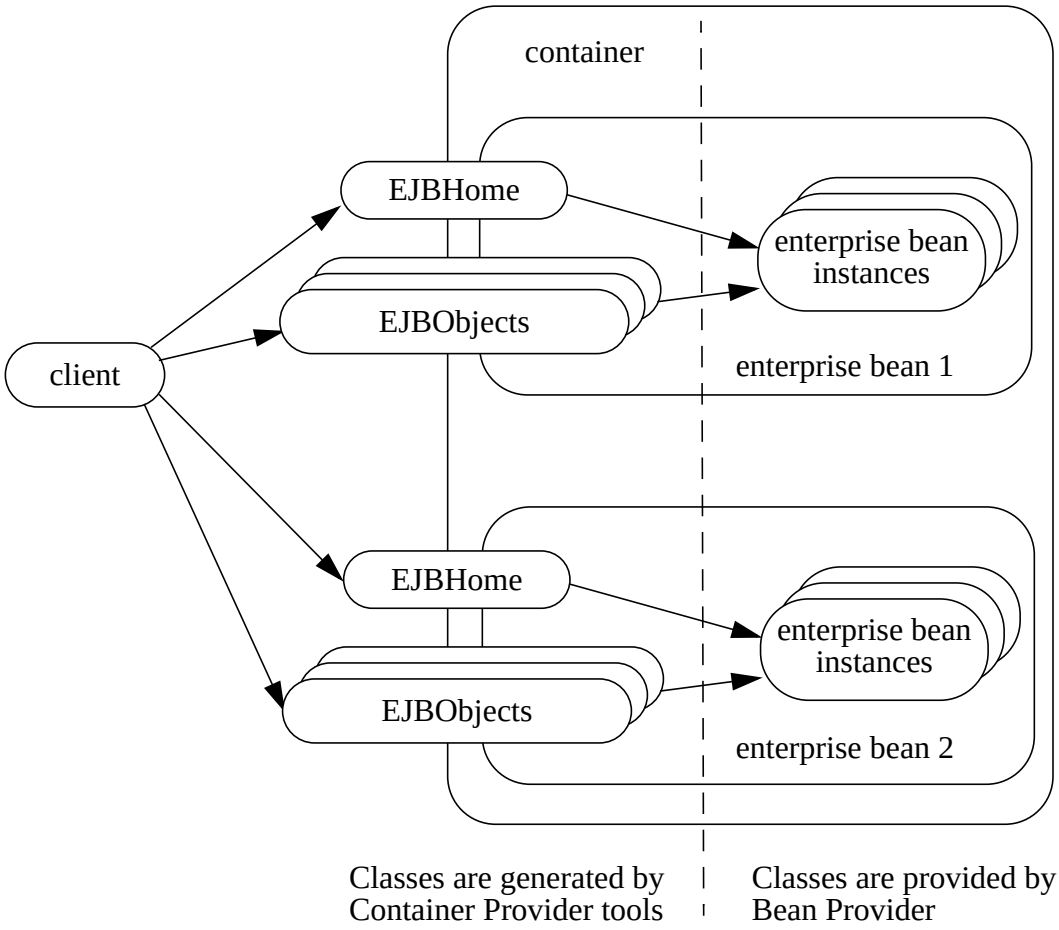
We expect that most enterprise beans with bean managed persistence will be created by application development tools which will encapsulate data access in components. These data access components will probably not be the same for all tools. Further, if the data access calls are encapsulated in data access components, the data access components may require deployment interfaces to allow adapting data access to different schemas or even to a different database type. This EJB specification does not define the architecture for data access objects, strategies for tailoring and deploying data access components or ensuring portability of these components for bean managed persistence.

In contrast to container managed persistence, the entity bean provider does not provide a description of the relationships and dependent classes in the deployment descriptor. With bean managed persistence, it is the responsibility of the bean provider to maintain relationships within the code that he or she supplies and to locate related beans by utilizing JNDI lookup.

11.1.3 Runtime execution model

This section describes the runtime model and the classes used in the description of the contract between an entity bean with bean managed persistence and its container.

Figure 38 Overview of the entity bean runtime execution model



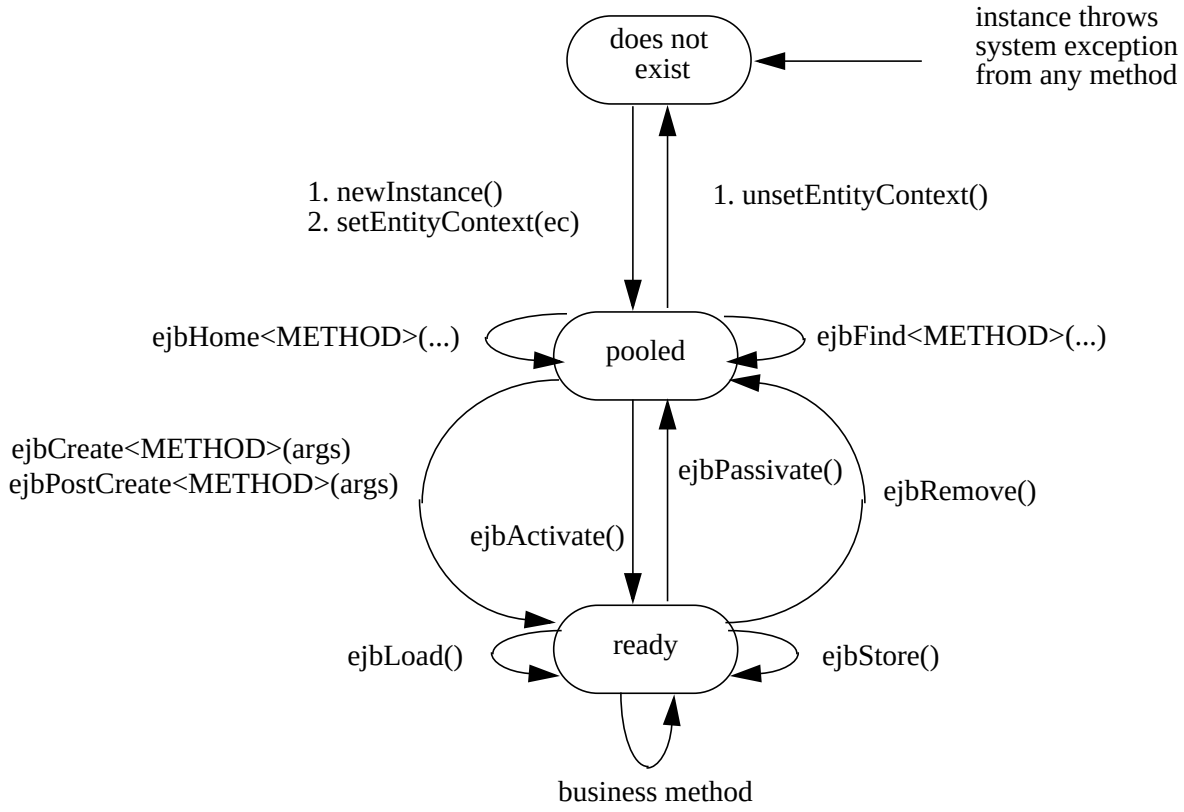
An **enterprise bean instance** is an object whose class was provided by the Bean Provider.

An entity **EJBObject** is an object whose class was generated at deployment time by the Container Provider's tools. The entity EJBObject class implements the entity bean's remote interface. A client never references an entity bean instance directly—a client always references an entity EJBObject whose class is generated by the Container Provider's tools.

An entity **EJBHome** object provides the life cycle operations (create, remove, find) for its entity objects as well as home business methods, which are not specific to an entity bean instance. The class for the entity EJBHome object is generated by the Container Provider's tools at deployment time. The entity EJBHome object implements the entity bean's home interface that was defined by the Bean Provider.

11.1.4 Instance life cycle

Figure 39 Life cycle of an entity bean instance.



An entity bean instance is in one of the following three states:

- It does not exist.
- Pooled state. An instance in the pooled state is not associated with any particular entity object identity.
- Ready state. An instance in the ready state is assigned an entity object identity.

The following steps describe the life cycle of an entity bean instance:

- An entity bean instance's life starts when the container creates the instance using `newInstance()`. The container then invokes the `setEntityContext()` method to pass the instance a reference to the `EntityContext` interface. The `EntityContext` interface

allows the instance to invoke services provided by the container and to obtain the information about the caller of a client-invoked method.

- The instance enters the pool of available instances. Each entity bean has its own pool. While the instance is in the available pool, the instance is not associated with any particular entity object identity. All instances in the pool are considered equivalent, and therefore any instance can be assigned by the container to any entity object identity at the transition to the ready state. While the instance is in the pooled state, the container may use the instance to execute any of the entity bean's finder methods (shown as `ejbFind<METHOD>(...)` in the diagram or home methods (shown as `ejbHome<METHOD>(...)` in the diagram). The instance does **not** move to the ready state during the execution of a finder or a home method.

- An instance transitions from the pooled state to the ready state when the container selects that instance to service a client call to an entity object. There are two possible transitions from the pooled to the ready state: through the `ejbCreate<METHOD>(...)` and `ejbPostCreate<METHOD>(...)` methods, or through the `ejbActivate()` method. The container invokes the `ejbCreate<METHOD>(...)` and `ejbPostCreate<METHOD>(...)` methods when the instance is assigned to an entity object during entity object creation (i.e., when the client invokes a `create<METHOD>` method on the entity bean's home object). The container invokes the `ejbActivate()` method on an instance when an instance needs to be activated to service an invocation on an existing entity object—this occurs because there is no suitable instance in the ready state to service the client's call.

- When an entity bean instance is in the ready state, the instance is associated with a specific entity object identity. While the instance is in the ready state, the container can invoke the `ejbLoad()` and `ejbStore()` methods zero or more times. A business method can be invoked on the instance zero or more times. Invocations of the `ejbLoad()` and `ejbStore()` methods can be arbitrarily mixed with invocations of business methods. The purpose of the `ejbLoad` and `ejbStore` methods is to synchronize the state of the instance with the state of the entity in the underlying data source—the container can invoke these methods whenever it determines a need to synchronize the instance's state.

- The container can choose to passivate an entity bean instance within a transaction. To passivate an instance, the container first invokes the `ejbStore` method to allow the instance to synchronize the database state with the instance's state, and then the container invokes the `ejbPassivate` method to return the instance to the pooled state.

- Eventually, the container will transition the instance to the pooled state. There are three possible transitions from the ready to the pooled state: through the `ejbPassivate()` method, through the `ejbRemove()` method, and because of a transaction rollback for `ejbCreate()`, `ejbPostCreate()`, or `ejbRemove()` (not shown in Figure 39). The container invokes the `ejbPassivate()` method when the container wants to disassociate the instance from the entity object identity without removing the entity object. The container invokes the `ejbRemove()` method when the container is removing the entity object (i.e., when the client invoked the `remove()` method on the entity object's remote interface, or one of the `remove()` methods on the entity bean's home interface). If `ejbCreate()`, `ejbPostCreate()`, or `ejbRemove()` is called and the transaction rolls back, the container will transition the bean instance to the pooled state.

- When the instance is put back into the pool, it is no longer associated with an entity object identity. The container can assign the instance to any entity object within the same entity bean home.

- An instance in the pool can be removed by calling the `unsetEntityContext()` method on the instance.

Notes:

1. The `EntityContext` interface passed by the container to the instance in the `setEntityContext` method is an interface, not a class that contains static information. For example, the result of the `EntityContext.getPrimaryKey()` method might be different each time an instance moves from the pooled state to the ready state, and the result of the `getCallerPrincipal()` and `isCallerInRole(...)` methods may be different in each business method.

2. A `RuntimeException` thrown from any method of the entity bean class (including the business methods and the callbacks invoked by the container) results in the transition to the "does not exist" state. The container must not invoke any method on the instance after a `RuntimeException` has been caught. From the client perspective, the corresponding entity object continues to exist. The client can continue accessing the entity object through its remote interface because the container can use a different entity bean instance to delegate the client's requests. Exception handling is described further in Chapter 17.

3. The container is not required to maintain a pool of instances in the pooled state. The pooling approach is an example of a possible implementation, but it is not the required implementation. Whether the container uses a pool or not has no bearing on the entity bean coding style.

11.1.5 The entity bean component contract

This section specifies the contract between an entity bean with bean managed persistence and its container.

11.1.5.1 Entity bean instance's view

The following describes the entity bean instance's view of the contract:

The entity Bean Provider is responsible for implementing the following methods in the entity bean class:

- A public constructor that takes no arguments. The Container uses this constructor to create instances of the entity bean class.

- `public void setEntityContext(EntityContext ic);`

  A container uses this method to pass a reference to the `EntityContext` interface to the entity bean instance. If the entity bean instance needs to use the `EntityContext` interface during its lifetime, it must remember the `EntityContext` interface in an instance variable.

This method executes with an unspecified transaction context (Refer to Subsection 16.6.5 for how the Container executes methods with an unspecified transaction context). An identity of an entity object is not available during this method.

The instance can take advantage of the `setEntityContext()` method to allocate any resources that are to be held by the instance for its lifetime. Such resources cannot be specific to an entity object identity because the instance might be reused during its lifetime to serve multiple entity object identities.

- `public void unsetEntityContext();`

  A container invokes this method before terminating the life of the instance.

  This method executes with an unspecified transaction context. An identity of an entity object is not available during this method.

  The instance can take advantage of the `unsetEntityContext()` method to free any resources that are held by the instance. (These resources typically had been allocated by the `setEntityContext()` method.)

- `public` *`PrimaryKeyClass`* `ejbCreate<METHOD>(...);`

  There are zero[18] or more `ejbCreate<METHOD>(...)` methods, whose signatures match the signatures of the `create<METHOD>(...)` methods of the entity bean home interface. The container invokes an `ejbCreate<METHOD>(...)` method on an entity bean instance when a client invokes a matching `create<METHOD>(...)` method to create an entity object.

  The implementation of the `ejbCreate<METHOD>(...)` method typically validates the client-supplied arguments, and inserts a record representing the entity object into the database. The method also initializes the instance's variables. The `ejbCreate<METHOD>(...)` method must return the primary key for the created entity object.

  An `ejbCreate<METHOD>(...)` method executes in the transaction context determined by the transaction attribute of the matching `create<METHOD>(...)` method, as described in subsection 16.6.2.

- `public void ejbPostCreate<METHOD>(...);`

  For each `ejbCreate<METHOD>(...)` method, there is a matching `ejbPostCreate<METHOD>(...)` method that has the same input parameters but the return value is `void`. The container invokes the matching `ejbPostCreate<METHOD>(...)` method on an instance after it invokes the `ejbCreate<METHOD>(...)` method with the same arguments. The entity object identity is available during the `ejbPostCreate<METHOD>(...)` method. The instance may, for example, obtain the remote interface of the associated entity object and pass it to another enterprise bean as a method argument.

  An `ejbPostCreate<METHOD>(...)` method executes in the same transaction context as the previous `ejbCreate<METHOD>(...)` method.

- `public void ejbActivate();`

[18] An entity enterprise Bean has no *ejbCreate*`<METHOD>`*(...)* and *ejbPostCreate*`<METHOD>`*(...)* methods if it does not define any create methods in its home interface. Such an entity enterprise Bean does not allow the clients to create new EJB objects. The enterprise Bean restricts the clients to accessing entities that were created through direct database inserts.

The container invokes this method on the instance when the container picks the instance from the pool and assigns it to a specific entity object identity. The `ejbActivate()` method gives the entity bean instance the chance to acquire additional resources that it needs while it is in the ready state.

This method executes with an unspecified transaction context. The instance can obtain the identity of the entity object via the `getPrimaryKey()` or `getEJBObject()` method on the entity context. The instance can rely on the fact that the primary key and entity object identity will remain associated with the instance until the completion of `ejbPassivate()` or `ejbRemove()`.

Note that the instance should not use the `ejbActivate()` method to read the state of the entity from the database; the instance should load its state only in the `ejbLoad()` method.

- `public void ejbPassivate();`
  The container invokes this method on an instance when the container decides to disassociate the instance from an entity object identity, and to put the instance back into the pool of available instances. The `ejbPassivate()` method gives the instance the chance to release any resources that should not be held while the instance is in the pool. (These resources typically had been allocated during the `ejbActivate()` method.)

  This method executes with an unspecified transaction context. The instance can still obtain the identity of the entity object via the `getPrimaryKey()` or `getEJBObject()` method of the `EntityContext` interface.

  Note that an instance should not use the `ejbPassivate()` method to write its state to the database; an instance should store its state only in the `ejbStore()` method.

- `public void ejbRemove();`

  The container invokes this method on an instance as a result of a client's invoking a `remove` method. The instance is in the ready state when `ejbRemove()` is invoked and it will be entered into the pool when the method completes.

  This method executes in the transaction context determined by the transaction attribute of the `remove` method that triggered the `ejbRemove` method. The instance can still obtain the identity of the entity object via the `getPrimaryKey()` or `getEJBObject()` method of the `EntityContext` interface.

  The container synchronizes the instance's state before it invokes the `ejbRemove` method. This means that the state of the instance variables at the beginning of the `ejbRemove` method is the same as it would be at the beginning of a business method.

  An entity bean instance should use this method to remove the entity object's representation in the database.

  Since the instance will be entered into the pool, the state of the instance at the end of this method must be equivalent to the state of a passivated instance. This means that the instance must release any resource that it would normally release in the `ejbPassivate()` method.

- `public void ejbLoad();`

  The container invokes this method on an instance in the ready state to inform the instance that it must synchronize the entity state cached in its instance variables from the entity state in the database. The instance must be prepared for the container to invoke this method at any time that the instance is in the ready state.

If the instance is caching the entity state (or parts of the entity state), the instance must not use the previously cached state in the subsequent business method. The instance may take advantage of the `ejbLoad` method, for example, to refresh the cached state by reading it from the database.

This method executes in the transaction context determined by the transaction attribute of the business method that triggered the `ejbLoad` method.

- `public void ejbStore();`

  The container invokes this method on an instance to inform the instance that the instance must synchronize the entity state in the database with the entity state cached in its instance variables. The instance must be prepared for the container to invoke this method at any time that the instance is in the ready state.

  An instance must write any updates cached in the instance variables to the database in the `ejbStore()` method.

  This method executes in the same transaction context as the previous `ejbLoad` or `ejbCreate<METHOD>` method invoked on the instance. All business methods invoked between the previous `ejbLoad` or `ejbCreate<METHOD>` method and this `ejbStore` method are also invoked in the same transaction context.

- `public` primary key type or collection `ejbFind<METHOD>(...)`;

  The container invokes this method on the instance when the container selects the instance to execute a matching client-invoked `find<METHOD>(...)` method. The instance is in the pooled state (i.e., it is not assigned to any particular entity object identity) when the container selects the instance to execute the `ejbFind<METHOD>` method on it, and it is returned to the pooled state when the execution of the `ejbFind<METHOD>` method completes.

  The `ejbFind<METHOD>` method executes in the transaction context determined by the transaction attribute of the matching `find(...)` method, as described in subsection 16.6.2.

  The implementation of an `ejbFind<METHOD>` method typically uses the method's arguments to locate the requested entity object or a collection of entity objects in the database. The method must return a primary key or a collection of primary keys to the container (see Subsection 11.1.8).

- `public` type `ejbHome<METHOD>(...)`;

  The container invokes this method on any instance when the container selects the instance to execute a matching client-invoked `<METHOD>(...)` home method. The instance is in the pooled state (i.e., it is not assigned to any particular entity object identity) when the container selects the instance to execute the `ejbHome<METHOD>` method on it, and it is returned to the pooled state when the execution of the `ejbHome<METHOD>` method completes.

  The `ejbHome<METHOD>` method executes in the transaction context determined by the transaction attribute of the matching `<METHOD>(...)` home method, as described in subsection 16.6.2.

  The entity bean provider provides the implementation of the `ejbHome<METHOD>(...)`.

11.1.5.2 Container's view:

This subsection describes the container's view of the state management contract. The container must call the following methods:

- `public void setEntityContext(ec);`

  The container invokes this method to pass a reference to the `EntityContext` interface to the entity bean instance. The container must invoke this method after it creates the instance, and before it puts the instance into the pool of available instances.

  The container invokes this method with an unspecified transaction context. At this point, the `EntityContext` is not associated with any entity object identity.

- `public void unsetEntityContext();`

  The container invokes this method when the container wants to reduce the number of instances in the pool. After this method completes, the container must not reuse this instance.

  The container invokes this method with an unspecified transaction context.

- `public` *`PrimaryKeyClass`* `ejbCreate<METHOD>(...);`
  `public void ejbPostCreate<METHOD>(...);`

  The container invokes these two methods during the creation of an entity object as a result of a client invoking a `create<METHOD>(...)` method on the entity bean's home interface.

  The container first invokes the `ejbCreate<METHOD>(...)` method whose signature matches the `create<METHOD>(...)` method invoked by the client. The `ejbCreate<METHOD>(...)` method returns a primary key for the created entity object. The container creates an entity EJBObject reference for the primary key. The container then invokes a matching `ejbPostCreate<METHOD>(...)` method to allow the instance to fully initialize itself. Finally, the container returns the entity object's remote interface (i.e., a reference to the entity EJBObject) to the client.

  The container must invoke the `ejbCreate<METHOD>(...)` and `ejbPostCreate<METHOD>(...)` methods in the transaction context determined by the transaction attribute of the matching `create<METHOD>(...)` method, as described in subsection 16.6.2.

- `public void ejbActivate();`

  The container invokes this method on an entity bean instance at activation time (i.e., when the instance is taken from the pool and assigned to an entity object identity). The container must ensure that the primary key of the associated entity object is available to the instance if the instance invokes the `getPrimaryKey()` or `getEJBObject()` method on its `EntityContext` interface.

  The container invokes this method with an unspecified transaction context.

  Note that instance is not yet ready for the delivery of a business method. The container must still invoke the `ejbLoad()` method prior to a business method.

- `public void ejbPassivate();`

  The container invokes this method on an entity bean instance at passivation time (i.e., when the instance is being disassociated from an entity object identity and moved into the pool). The container must ensure that the identity of the associated entity object is still available to the

instance if the instance invokes the `getPrimaryKey()` or `getEJBObject()` method on its entity context.

The container invokes this method with an unspecified transaction context.

Note that if the instance state has been updated by a transaction, the container must first invoke the `ejbStore()` method on the instance before it invokes `ejbPassivate()` on it.

- `public void ejbRemove();`

  The container invokes this method before it ends the life of an entity object as a result of a client invoking a `remove` operation.

  The container invokes this method in the transaction context determined by the transaction attribute of the invoked `remove` method.
  The container must ensure that the identity of the associated entity object is still available to the instance in the `ejbRemove()` method (i.e., the instance can invoke the `getPrimaryKey()` or `getEJBObject()` method on its `EntityContext` in the `ejbRemove()` method).

  The container must ensure that the instance's state is synchronized from the state in the database before invoking the `ejbRemove()` method (i.e., if the instance is not already synchronized from the state in the database, the container must invoke `ejbLoad` before it invokes `ejbRemove`).

- `public void ejbLoad();`

  The container must invoke this method on the instance whenever it becomes necessary for the instance to synchronize its instance state from its state in the database. The exact times that the container invokes `ejbLoad` depend on the configuration of the component and the container, and are not defined by the EJB architecture. Typically, the container will call `ejbLoad` before the first business method within a transaction to ensure that the instance can refresh its cached state of the entity object from the database. After the first `ejbLoad` within a transaction, the container is not required to recognize that the state of the entity object in the database has been changed by another transaction, and it is not required to notify the instance of this change via another `ejbLoad` call.

  The container must invoke this method in the transaction context determined by the transaction attribute of the business method that triggered the `ejbLoad` method.

- `public void ejbStore();`

  The container must invoke this method on the instance whenever it becomes necessary for the instance to synchronize its state in the database with the state of the instance's fields. This synchronization always happens at the end of a transaction. However, the container may also invoke this method when it passivates the instance in the middle of a transaction, or when it needs to transfer the most recent state of the entity object to another instance for the same entity object in the same transaction (see Subsection 16.7).

  The container must invoke this method in the same transaction context as the previously invoked `ejbLoad` or `ejbCreate<METHOD>` method.

- `public` primary key type or collection `ejbFind<METHOD>(...)`;

  The container invokes the `ejbFind<METHOD>(...)` method on an instance when a client invokes a matching `find<METHOD>(...)` method on the entity bean's home interface. The container must pick an instance that is in the pooled state (i.e., the instance is not associated

with any entity object identity) for the execution of the `ejbFind<METHOD>(...)` method. If there is no instance in the pooled state, the container creates one and calls the `setEntityContext(...)` method on the instance before dispatching the finder method.

Before invoking the `ejbFind<METHOD>(...)` method, the container must first synchronize the state of any entity bean instances that are participating in the same transaction context as is used to execute the `ejbFind<METHOD>(...)` by invoking the `ejbStore()` method on those entity bean instances.

After the `ejbFind<METHOD>(...)` method completes, the instance remains in the pooled state. The container may, but is not required to, activate the objects that were located by the finder using the transition through the `ejbActivate()` method.

The container must invoke the `ejbFind<METHOD>(...)` method in the transaction context determined by the transaction attribute of the matching `find(...)` method, as described in subsection 16.6.2.

If the `ejbFind<METHOD>` method is declared to return a single primary key, the container creates an entity EJBObject reference for the primary key and returns it to the client. If the `ejbFind<METHOD>` method is declared to return a collection of primary keys, the container creates a collection of entity EJBObject references for the primary keys returned from `ejbFind<METHOD>`, and returns the collection to the client. (See Subsection 11.1.8 for information on collections.)

- `public` type `ejbHome<METHOD>(...)`;

  The container invokes the `ejbHome<METHOD>(...)` method on an instance when a client invokes a matching `<METHOD>(...)` home method on the entity bean's home interface. The container must pick an instance that is in the pooled state (i.e., the instance is not associated with any entity object identity) for the execution of the `ejbHome<METHOD>(...)` method. If there is no instance in the pooled state, the container creates one and calls the `setEntityContext(...)` method on the instance before dispatching the home method.

  After the `ejbHome<METHOD>(...)` method completes, the instance remains in the pooled state.

  The container must invoke the `ejbHome<METHOD>(...)` method in the transaction context determined by the transaction attribute of the matching `<METHOD>(...)` home method, as described in subsection 16.6.2.

11.1.6 Operations allowed in the methods of the entity bean class

Table 12 defines the methods of an entity bean class in which the enterprise bean instances can access the methods of the `javax.ejb.EntityContext` interface, the `java:comp/env` environment naming context, resource managers, and other enterprise beans.

If an entity bean instance attempts to invoke a method of the `EntityContext` interface, and the access is not allowed in Table 12, the Container must throw the `java.lang.IllegalStateException`.

If an entity bean instance attempts to access a resource manager or an enterprise bean, and the access is not allowed in Table 12, the behavior is undefined by the EJB architecture.

Table 12 Operations allowed in the methods of an entity bean

Bean method	Bean method can perform the following operations
constructor	-
setEntityContext unsetEntityContext	EntityContext methods: *getEJBHome* JNDI access to java:comp/env
ejbCreate	EntityContext methods: *getEJBHome*, *getCallerPrincipal*, *getRollbackOnly*, *isCallerInRole*, *setRollbackOnly* JNDI access to java:comp/env Resource manager access Enterprise bean access
ejbPostCreate	EntityContext methods: *getEJBHome*, *getCallerPrincipal*, *getRollbackOnly*, *isCallerInRole*, *setRollbackOnly*, *getEJBObject*, *getPrimaryKey* JNDI access to java:comp/env Resource manager access Enterprise bean access
ejbRemove	EntityContext methods: *getEJBHome*, *getCallerPrincipal*, *getRollbackOnly*, *isCallerInRole*, *setRollbackOnly*, *getEJBObject*, *getPrimaryKey* JNDI access to java:comp/env Resource manager access Enterprise bean access
ejbFind ejbHome	EntityContext methods: *getEJBHome*, *getCallerPrincipal*, *getRollbackOnly*, *isCallerInRole*, *setRollbackOnly* JNDI access to java:comp/env Resource manager access Enterprise bean access
ejbActivate ejbPassivate	EntityContext methods: *getEJBHome*, *getEJBObject*, *getPrimaryKey* JNDI access to java:comp/env
ejbLoad ejbStore	EntityContext methods: *getEJBHome*, *getCallerPrincipal*, *getRollbackOnly*, *isCallerInRole*, *setRollbackOnly*, *getEJBObject*, *getPrimaryKey* JNDI access to java:comp/env Resource manager access Enterprise bean access
business method from remote interface	EntityContext methods: *getEJBHome*, *getCallerPrincipal*, *getRollbackOnly*, *isCallerInRole*, *setRollbackOnly*, *getEJBObject*, *getPrimaryKey* JNDI access to java:comp/env Resource manager access Enterprise bean access

Additional restrictions:

- The `getRollbackOnly` and `setRollbackOnly` methods of the `EntityContext` interface should be used only in the enterprise bean methods that execute in the context of a transaction. The Container must throw the `java.lang.IllegalStateException` if the methods are invoked while the instance is not associated with a transaction.

Reasons for disallowing operations:

- Invoking the `getEJBObject` and `getPrimaryKey` methods is disallowed in the entity bean methods in which there is no entity object identity associated with the instance.

- Invoking the `getCallerPrincipal` and `isCallerInRole` methods is disallowed in the entity bean methods for which the Container does not have a client security context.

- Invoking the `getRollbackOnly` and `setRollbackOnly` methods is disallowed in the entity bean methods for which the Container does not have a meaningful transaction context. These are the methods that have the `NotSupported`, `Never`, or `Supports` transaction attribute.

- Accessing resource managers and enterprise beans is disallowed in the entity bean methods for which the Container does not have a meaningful transaction context or client security context.

11.1.7 Caching of entity state and the `ejbLoad` and `ejbStore` methods

An instance of an entity bean with bean-managed persistence can cache the entity object's state between business method invocations. An instance may choose to cache the entire entity object's state, part of the state, or no state at all.

The container-invoked `ejbLoad` and `ejbStore` methods assist the instance with the management of the cached entity object's state. The instance must handle the `ejbLoad` and `ejbStore` methods as follows:

- When the container invokes the `ejbStore` method on the instance, the instance must push all cached updates of the entity object's state to the underlying database. The container invokes the `ejbStore` method at the end of a transaction, and may also invoke it at other times when the instance is in the ready state. (For example the container may invoke `ejbStore` when passivating an instance in the middle of a transaction, or when transferring the instance's state to another instance to support distributed transactions in a multi-process server.)

- When the container invokes the `ejbLoad` method on the instance, the instance must discard any cached entity object's state. The instance may, but is not required to, refresh the cached state by reloading it from the underlying database.

The following examples, which are illustrative but not prescriptive, show how an instance may cache the entity object's state:

- An instance loads the entire entity object's state in the `ejbLoad` method and caches it until the container invokes the `ejbStore` method. The business methods read and write the cached

entity state. The `ejbStore` method writes the updated parts of the entity object's state to the database.

- An instance loads the most frequently used part of the entity object's state in the `ejbLoad` method and caches it until the container invokes the `ejbStore` method. Additional parts of the entity object's state are loaded as needed by the business methods. The `ejbStore` method writes the updated parts of the entity object's state to the database.

- An instance does not cache any entity object's state between business methods. The business methods access and modify the entity object's state directly in the database. The `ejbLoad` and `ejbStore` methods have an empty implementation.

We expect that most entity developers will not manually code the cache management and data access calls in the entity bean class. We expect that they will rely on application development tools to provide various data access components that encapsulate data access and provide state caching.

11.1.7.1 `ejbLoad` and `ejbStore` with the `NotSupported` transaction attribute

The use of the `ejbLoad` and `ejbStore` methods for caching an entity object's state in the instance works well only if the Container can use transaction boundaries to drive the `ejbLoad` and `ejbStore` methods. When the `NotSupported`[19] transaction attribute is assigned to a remote interface method, the corresponding enterprise bean class method executes with an unspecified transaction context (See Subsection 16.6.5). This means that the Container does not have any well-defined transaction boundaries to drive the `ejbLoad` and `ejbStore` methods on the instance.

Therefore, the `ejbLoad` and `ejbStore` methods are "unreliable" for the instances that the Container uses to dispatch the methods with an unspecified transaction context. The following are the only guarantees that the Container provides for the instances that execute the methods with an unspecified transaction context:

- The Container invokes at least one `ejbLoad` between `ejbActivate` and the first business method in the instance.

- The Container invokes at least one `ejbStore` between the last business method on the instance and the `ejbPassivate` method.

Because the entity object's state accessed between the `ejbLoad` and `ejbStore` method pair is not protected by a transaction boundary for the methods that execute with an unspecified transaction context, the Bean Provider should not attempt to use the `ejbLoad` and `ejbStore` methods to control caching of the entity object's state in the instance. Typically, the implementation of the `ejbLoad` and `ejbStore` methods should be a no-op (i.e., an empty method), and each business method should access the entity object's state directly in the database.

[19] This applies also to the `Never` and `Supports` attribute.

11.1.8 Finder method return type

11.1.8.1 Single-object finder

Some finder methods (such as `ejbFindByPrimaryKey`) are designed to return at most one entity object. For these single-object finders, the result type of the `find<METHOD>(...)`method defined in the entity bean's home interface is the entity bean's remote interface. The result type of the corresponding `ejbFind<METHOD>(...)` method defined in the entity's implementation class is the entity bean's primary key type.

The following code illustrates the definition of a single-object finder.

```
// Entity's home interface
public AccountHome extends javax.ejb.EJBHome {
     ...
     Account findByPrimaryKey(AccountPrimaryKey primkey)
          throws FinderException, RemoteException;
     ...
}

// Entity's implementation class
public AccountBean implements javax.ejb.EntityBean {
     ...
     public AccountPrimaryKey ejbFindByPrimaryKey(
               AccountPrimaryKey primkey)
          throws FinderException
     {
          ...
     }
     ...
}
```

11.1.8.2 Multi-object finders

Some finder methods are designed to return multiple entity objects. For these multi-object finders, the result type of the `find<METHOD>(...)`method defined in the entity bean's home interface is a *collection* of objects implementing the entity bean's remote interface. The result type of the corresponding `ejbFind<METHOD>(...)` implementation method defined in the entity bean's implementation class is a collection of objects of the entity bean's primary key type.

The Bean Provider can choose two types to define a collection type for a finder:

- the Java™ 2 `java.util.Collection` interface
- the JDK™ 1.1 `java.util.Enumeration` interface

A Bean Provider targeting containers and clients based on Java 2 should use the `java.util.Collection` interface for the finder's result type.

A Bean Provider who wants to ensure that the entity bean is compatible with containers and clients based on JDK 1.1 must use the `java.util.Enumeration` interface for the finder's result type[20].

The Bean Provider must ensure that the objects in the `java.util.Enumeration` or `java.util.Collection` returned from the `ejbFind<METHOD>(...)` method are instances of the entity bean's primary key class.

A client program must use the `PortableRemoteObject.narrow(...)` method to convert the objects contained in the collections returned by the finder method to the entity bean's remote interface type.

The following is an example of a multi-object finder method definition that is compatible with containers and clients based on Java 2:

```
// Entity's home interface
public AccountHome extends javax.ejb.EJBHome {
     ...
     java.util.Collection findLargeAccounts(double limit)
          throws FinderException, RemoteException;
     ...
}


// Entity's implementation class
public AccountBean implements javax.ejb.EntityBean {
     ...
     public java.util.Collection ejbFindLargeAccounts(
               double limit) throws FinderException
     {
          ...
     }
     ...
}
```

The following is an example of a multi-object finder method definition compatible with containers and clients that are based on both JDK 1.1 and Java 2:

```
// Entity's home interface
public AccountHome extends javax.ejb.EJBHome {
     ...
     java.util.Enumeration findLargeAccounts(double limit)
          throws FinderException, RemoteException;
     ...
}


// Entity's implementation class
public AccountBean implements javax.ejb.EntityBean {
     ...
     public java.util.Enumeration ejbFindLargeAccounts(
               double limit) throws FinderException
     {
          ...
     }
     ...
}
```

[20] The finder will be also compatible with Java 2-based Containers and Clients.

11.1.9 Standard application exceptions for Entities

The EJB specification defines the following standard application exceptions:

- `javax.ejb.CreateException`
- `javax.ejb.DuplicateKeyException`
- `javax.ejb.FinderException`
- `javax.ejb.ObjectNotFoundException`
- `javax.ejb.RemoveException`

11.1.9.1 CreateException

From the client's perspective, a `CreateException` (or a subclass of `CreateException`) indicates that an application level error occurred during the `create<METHOD>(...)` operation. If a client receives this exception, the client does not know, in general, whether the entity object was created but not fully initialized, or not created at all. Also, the client does not know whether or not the transaction has been marked for rollback. (However, the client may determine the transaction status using the `UserTransaction` interface.)

The Bean Provider throws the `CreateException` (or subclass of `CreateException`) from the `ejbCreate<METHOD>(...)` and `ejbPostCreate<METHOD>(...)` methods to indicate an application-level error from the create or initialization operation. Optionally, the Bean Provider may mark the transaction for rollback before throwing this exception.

The Bean Provider is encouraged to mark the transaction for rollback only if data integrity would be lost if the transaction were committed by the client. Typically, when a CreateException is thrown, it leaves the database in a consistent state, allowing the client to recover. For example, ejbCreate may throw the CreateException to indicate that the some of the arguments to the create<METHOD>(...) methods are invalid.

The Container treats the `CreateException` as any other application exception. See Section 17.3.

11.1.9.2 DuplicateKeyException

The `DuplicateKeyException` is a subclass of `CreateException`. It is thrown by the `ejbCreate<METHOD>(...)` methods to indicate to the client that the entity object cannot be created because an entity object with the same key already exists. The unique key causing the violation may be the primary key, or another key defined in the underlying database.

Normally, the Bean Provider should not mark the transaction for rollback before throwing the exception.

When the client receives the `DuplicateKeyException`, the client knows that the entity was not created, and that the client's transaction has not typically been marked for rollback.

11.1.9.3 FinderException

From the client's perspective, a `FinderException` (or a subclass of `FinderException`) indicates that an application level error occurred during the `find(...)` operation. Typically, the client's transaction has not been marked for rollback because of the `FinderException`.

The Bean Provider throws the `FinderException` (or subclass of `FinderException`) from the `ejbFind<METHOD>(...)` methods to indicate an application-level error in the finder method. The Bean Provider should not, typically, mark the transaction for rollback before throwing the `FinderException`.

The Container treats the `FinderException` as any other application exception. See Section 17.3.

11.1.9.4 ObjectNotFoundException

The `ObjectNotFoundException` is a subclass of `FinderException`. It is thrown by the `ejbFind<METHOD>(...)` methods to indicate that the requested entity object does not exist.

Only single-object finders (see Subsection 11.1.8) should throw this exception. Multi-object finders must not throw this exception. Multi-object finders should return an empty collection as an indication that no matching objects were found.

11.1.9.5 RemoveException

From the client's perspective, a `RemoveException` (or a subclass of `RemoveException`) indicates that an application level error occurred during a `remove(...)` operation. If a client receives this exception, the client does not know, in general, whether the entity object was removed or not. The client also does not know if the transaction has been marked for rollback. (However, the client may determine the transaction status using the `UserTransaction` interface.)

The Bean Provider throws the `RemoveException` (or subclass of `RemoveException`) from the `ejbRemove()` method to indicate an application-level error from the entity object removal operation. Optionally, the Bean Provider may mark the transaction for rollback before throwing this exception.

The Bean Provider is encouraged to mark the transaction for rollback only if data integrity would be lost if the transaction were committed by the client. Typically, when a RemoveException is thrown, it leaves the database in a consistent state, allowing the client to recover.

The Container treats the `RemoveException` as any other application exception. See Section 17.3.

11.1.10 Commit options

The Entity Bean protocol is designed to give the Container the flexibility to select the disposition of the instance state at transaction commit time. This flexibility allows the Container to optimally manage the caching of entity object's state and the association of an entity object identity with the enterprise bean instances.

The Container can select from the following commit-time options:

- **Option A**: The Container caches a "ready" instance between transactions. The Container ensures that the instance has exclusive access to the state of the object in the persistent storage. Therefore, the Container does not have to synchronize the instance's state from the persistent storage at the beginning of the next transaction.

- **Option B**: The Container caches a "ready" instance between transactions. In contrast to Option A, in this option the Container does not ensure that the instance has exclusive access to the state of the object in the persistent storage. Therefore, the Container must synchronize the instance's state from the persistent storage at the beginning of the next transaction.

- **Option C**: The Container does not cache a "ready" instance between transactions. The Container returns the instance to the pool of available instances after a transaction has completed.

The following table provides a summary of the commit-time options.

Table 13 Summary of commit-time options

	Write instance state to database	**Instance stays ready**	**Instance state remains valid**
Option A	Yes	Yes	Yes
Option B	Yes	Yes	No
Option C	Yes	No	No

Note that the container synchronizes the instance's state with the persistent storage at transaction commit for all three options.

The selection of the commit option is transparent to the entity bean implementation—the entity bean will work correctly regardless of the commit-time option chosen by the Container. The Bean Provider writes the entity bean in the same way.

The object interaction diagrams in subsection 11.4.4 illustrate the three alternative commit options in detail.

11.1.11 Concurrent access from multiple transactions

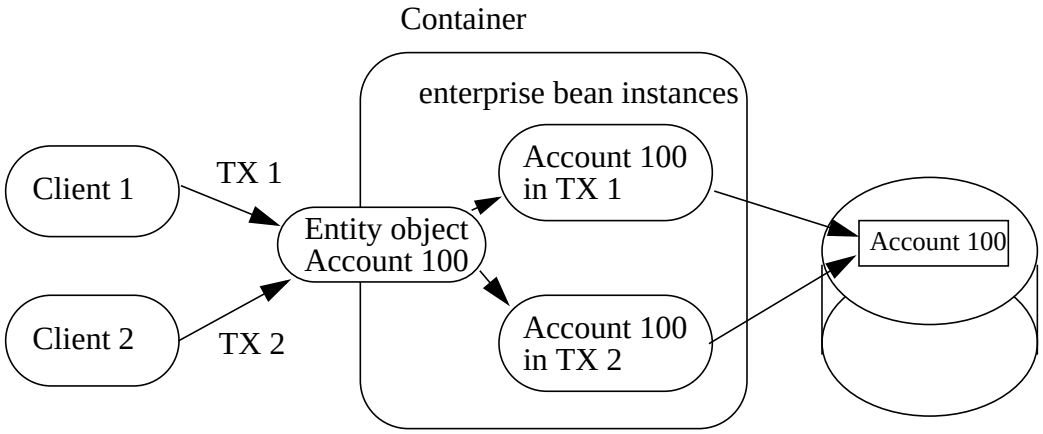
When writing the entity bean business methods, the Bean Provider does not have to worry about concurrent access from multiple transactions. The Bean Provider may assume that the container will ensure appropriate synchronization for entity objects that are accessed concurrently from multiple transactions.

The container typically uses one of the following implementation strategies to achieve proper synchronization. (These strategies are illustrative, not prescriptive.)

- The container activates multiple instances of the entity bean, one for each transaction in which the entity object is being accessed. The transaction synchronization is performed automatically by the underlying database during the database access calls performed by the business meth-

ods; and by the `ejbLoad`, `ejbCreate<METHOD>`, `ejbStore`, and `ejbRemove` methods. The database system provides all the necessary transaction synchronization; the container does not have to perform any synchronization logic. The commit-time options B and C in Subsection 11.4.4 apply to this type of container.

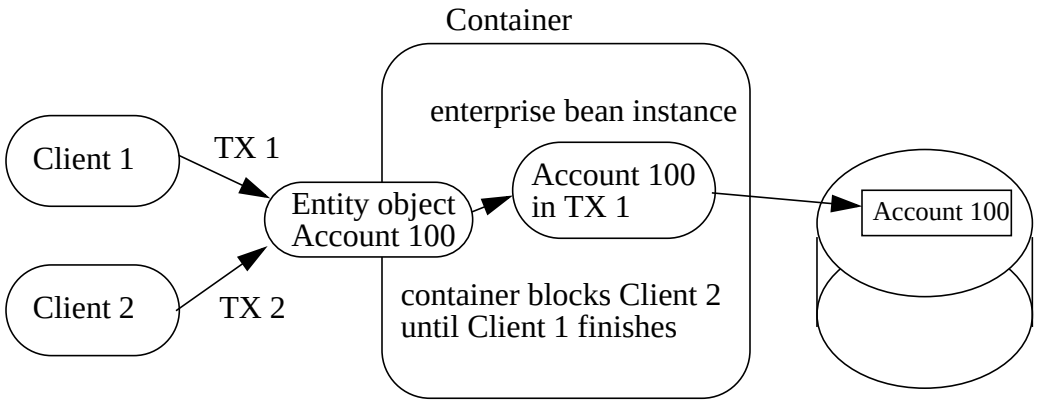
Figure 40 Multiple clients can access the same entity object using multiple instances



With this strategy, the type of lock acquired by ejbLoad leads to a trade-off. If ejbLoad acquires an exclusive lock on the instance's state in the database, then throughput of read-only transactions could be impacted. If ejbLoad acquires a shared lock and the instance is updated, then ejbStore will need to promote the lock to an exclusive lock. This may cause a deadlock if it happens concurrently under multiple transactions.

- The container acquires exclusive access to the entity object's state in the database. The container activates a single instance and serializes the access from multiple transactions to this instance. The commit-time option A in Subsection 11.4.4 applies to this type of container.

Figure 41 Multiple clients can access the same entity object using single instance



11.1.12 Non-reentrant and re-entrant instances

An entity Bean Provider entity can specify that an entity bean is non-reentrant. If an instance of a non-reentrant entity bean executes a client request in a given transaction context, and another request with the same transaction context arrives for the same entity object, the container will throw the `java.rmi.RemoteException` to the second request. This rule allows the Bean Provider to program the entity bean as single-threaded, non-reentrant code.

The functionality of some entity beans may require loopbacks in the same transaction context. An example of a loopback is when the client calls entity object A, A calls entity object B, and B calls back A in the same transaction context. The entity bean's method invoked by the loopback shares the current execution context (which includes the transaction and security contexts) with the Bean's method invoked by the client.

If the entity bean is specified as non-reentrant in the deployment descriptor, the Container must reject an attempt to re-enter the instance via the entity bean's remote interface while the instance is executing a business method. (This can happen, for example, if the instance has invoked another enterprise bean, and the other enterprise bean tries to make a loopback call.) The container must reject the loopback call and throw the `java.rmi.RemoteException` to the caller. The container must allow the call if the Bean's deployment descriptor specifies that the entity bean is re-entrant.

Re-entrant entity beans must be programmed and used with great caution. First, the Bean Provider must code the entity bean with the anticipation of a loopback call. Second, since the container cannot, in general, tell a loopback from a concurrent call from a different client, the client programmer must be careful to avoid code that could lead to a concurrent call in the same transaction context.

Concurrent calls in the same transaction context targeted at the same entity object are illegal and may lead to unpredictable results. Since the container cannot, in general, distinguish between an illegal concurrent call and a legal loopback, application programmers are encouraged to avoid using loopbacks. Entity beans that do not need callbacks should be marked as non-reentrant in the deployment descriptor, allowing the container to detect and prevent illegal concurrent calls from clients.

11.2 Responsibilities of the Enterprise Bean Provider

This section describes the responsibilities of a bean managed persistence entity Bean Provider to ensure that the entity bean can be deployed in any EJB Container.

11.2.1 Classes and interfaces

The entity Bean Provider is responsible for providing the following class files:

- Entity bean class and any dependent classes.
- Entity bean's remote interface
- Entity bean's home interface
- Primary key class

11.2.2 Enterprise bean class

The following are the requirements for an entity bean class:

The class must implement, directly or indirectly, the `javax.ejb.EntityBean` interface.

The class must be defined as `public` and must not be `abstract`.

The class must not be defined as `final`.

The class must define a public constructor that takes no arguments.

The class must not define the `finalize()` method.

The class may, but is not required to, implement the entity bean's remote interface[21]. If the class implements the entity bean's remote interface, the class must provide no-op implementations of the methods defined in the `javax.ejb.EJBObject` interface. The container will never invoke these methods on the bean instances at runtime.

[21] If the entity bean class does implement the remote interface, care must be taken to avoid passing of `this` as a method argument or result. This potential error can be avoided by choosing not to implement the remote interface in the entity bean class.

A no-op implementation of these methods is required to avoid defining the entity bean class as abstract.

The entity bean class must implement the business methods, and the `ejbCreate<METHOD>, ejbPostCreate<METHOD>, ejbFind<METHOD>` and `ejbHome<METHOD>` methods as described later in this section.

The entity bean class must implement the `ejbHome<METHOD>` methods that correspond to the home business methods specified in the bean's home interface. These methods are executed on an instance in the pooled state; hence they must not access state that is particular to a specific bean instance.

The entity bean class may have superclasses and/or superinterfaces. If the entity bean has superclasses, the business methods, the `ejbCreate and ejbPostCreate` methods, the finder methods, and the methods of the `EntityBean` interface may be implemented in the enterprise bean class or in any of its superclasses.

The entity bean class is allowed to implement other methods (for example helper methods invoked internally by the business methods) in addition to the methods required by the EJB specification.

11.2.3 *ejbCreate<METHOD>* methods

The entity bean class may define zero or more `ejbCreate<METHOD>(...)` methods whose signatures must follow these rules:

The method name must have `ejbCreate` as its prefix.

The method must be declared as `public`.

The method must not be declared as `final` or `static`.

The return type must be the entity bean's primary key type.

The method argument and return value types must be legal types for RMI-IIOP.

The throws clause may define arbitrary application specific exceptions, including the `javax.ejb.CreateException`.

Compatibility Note: EJB 1.0 allowed the ejbCreate method to throw the `java.rmi.RemoteException` to indicate a non-application exception. This practice was deprecated in EJB 1.1—an EJB 1.1 or EJB 2.0 compliant enterprise bean should throw the javax.ejb.EJBException or another java.lang.RuntimeException to indicate non-application exceptions to the Container (see Section 17.2.2).

The entity object created by the `ejbCreate<METHOD>` method must have a unique primary key. This means that the primary key must be different from the primary keys of all the existing entity objects within the same home. The `ejbCreate<METHOD>` method should throw the `DuplicateKeyException` on an attempt to create an entity object with a duplicate primary key. However, it is legal to reuse the primary key of a previously removed entity object.

11.2.4 *ejbPostCreate<METHOD>* methods

For each `ejbCreate<METHOD>(...)` method, the entity bean class must define a matching `ejbPostCreate<METHOD>(...)` method, using the following rules:

The method name must have `ejbPostCreate` as its prefix.

The method must be declared as `public`.

The method must not be declared as `final` or `static`.

The return type must be `void`.

The method arguments must be the same as the arguments of the matching `ejbCreate<METHOD>(...)` method.

The throws clause may define arbitrary application specific exceptions, including the `javax.ejb.CreateException`.

Compatibility Note: EJB 1.0 allowed the ejbPostCreate method to throw the `java.rmi.RemoteException` to indicate a non-application exception. This practice was deprecated in EJB 1.1—an EJB 1.1 or EJB 2.0 compliant enterprise bean should throw the javax.ejb.EJBException or another java.lang.RuntimeException to indicate non-application exceptions to the Container (see Section 17.2.2).

11.2.5 *ejbFind* methods

The entity bean class may also define additional `ejbFind<METHOD>(...)` finder methods.

The signatures of the finder methods must follow the following rules:

A finder method name must start with the prefix "**ejbFind**" (e.g. `ejbFindByPrimaryKey`, `ejbFindLargeAccounts`, `ejbFindLateShipments`).

A finder method must be declared as `public`.

The method must not be declared as `final` or `static`.

The method argument types must be legal types for RMI-IIOP.

The return type of a finder method must be the entity bean's primary key type, or a collection of primary keys (see Subsection 11.1.8).

The throws clause may define arbitrary application specific exceptions, including the `javax.ejb.FinderException`.

Compatibility Note: EJB 1.0 allowed the finder methods to throw the `java.rmi.RemoteException` to indicate a non-application exception. This practice was deprecated in EJB 1.1—an EJB 1.1 or EJB 2.0 compliant enterprise bean should throw the javax.ejb.EJBException or another java.lang.RuntimeException to indicate non-application exceptions to the Container (see Section 17.2.2).

Every entity bean must define the `ejbFindByPrimaryKey` method. The result type for this method must be the primary key type (i.e., the `ejbFindByPrimaryKey` method must be a single-object finder).

11.2.6 *ejbHome<METHOD>* methods

The entity bean class may define zero or more home methods whose signatures must follow the following rules:

The method name must have `ejbHome` as its prefix.

The method must be declared as `public`.

The method must not be declared as `static`.

The method argument and return value types must be legal types for RMI-IIOP.

The throws clause may define arbitrary application specific exceptions.

11.2.7 Business methods

The entity bean class may define zero or more business methods whose signatures must follow these rules:

The method names can be arbitrary, but they must not start with ‘ejb’ to avoid conflicts with the callback methods used by the EJB architecture.

The business method must be declared as `public`.

The method must not be declared as `final` or `static`.

The method argument and return value types must be legal types for RMI-IIOP.

The throws clause may define arbitrary application specific exceptions.

Compatibility Note: EJB 1.0 allowed the business methods to throw the `java.rmi.RemoteException` to indicate a non-application exception. This practice was deprecated in EJB 1.1—an EJB 1.1 or EJB 2.0 compliant enterprise bean should throw the javax.ejb.EJBException or another java.lang.RuntimeException to indicate non-application exceptions to the Container (see Section 17.2.2).

11.2.8 Entity bean's remote interface

The following are the requirements for the entity bean's remote interface:

The interface must extend the `javax.ejb.EJBObject` interface.

The methods defined in the remote interface must follow the rules for RMI-IIOP. This means that their argument and return value types must be valid types for RMI-IIOP, and their throws clauses must include the `java.rmi.RemoteException`.

The remote interface is allowed to have superinterfaces. Use of interface inheritance is subject to the RMI-IIOP rules for the definition of remote interfaces.

For each method defined in the remote interface, there must be a matching method in the entity bean's class. The matching method must have:

- The same name.
- The same number and types of its arguments, and the same return type.
- All the exceptions defined in the throws clause of the matching method of the enterprise Bean class must be defined in the throws clause of the method of the remote interface.

11.2.9 Entity bean's home interface

The following are the requirements for the entity bean's home interface:

The interface must extend the `javax.ejb.EJBHome` interface.

The methods defined in this interface must follow the rules for RMI-IIOP. This means that their argument and return types must be of valid types for RMI-IIOP, and that their throws clauses must include the `java.rmi.RemoteException`.

The home interface is allowed to have superinterfaces. Use of interface inheritance is subject to the RMI-IIOP rules for the definition of remote interfaces.

Each method defined in the home interface must be one of the following:

- A create method.
- A finder method.
- A home method.

Each `create` method must be the named "**create**<**METHOD**>", and it must match one of the `ejbCreate<METHOD>` methods defined in the enterprise Bean class. The matching `ejbCreate<METHOD>` method must have the same number and types of its arguments. (Note that the return type is different.)

The return type for a `create<METHOD>` method must be the entity bean's remote interface type.

All the exceptions defined in the throws clause of the matching `ejbCreate<METHOD>` and `ejbPostCreate<METHOD>` methods of the enterprise Bean class must be included in the throws clause of the matching `create` method of the home interface (i.e., the set of exceptions defined for the `create<METHOD>` method must be a superset of the union of exceptions defined for the `ejbCreate<METHOD>` and `ejbPostCreate<METHOD>` methods).

The throws clause of a `create<METHOD>` method must include the `javax.ejb.CreateException`.

Each `finder` method must be named "**`find`**`<METHOD>`" (e.g. `findLargeAccounts`), and it must match one of the `ejbFind<METHOD>` methods defined in the entity bean class (e.g. `ejbFindLargeAccounts`). The matching `ejbFind<METHOD>` method must have the same number and types of arguments. (Note that the return type may be different.)

The return type for a `find<METHOD>` method must be the entity bean's remote interface type (for a single-object finder), or a collection thereof (for a multi-object finder).

The home interface must always include the `findByPrimaryKey` method, which is always a single-object finder. The method must declare the primary key class as the method argument.

All the exceptions defined in the throws clause of an `ejbFind` method of the entity bean class must be included in the throws clause of the matching `find` method of the home interface.

The throws clause of a `finder` method must include the `javax.ejb.FinderException`.

Home methods can have arbitrary names, provided that they do not clash with `create`, `find` and `remove` method names. Their argument and return types must be of valid types for RMI-IIOP, and their throws clauses must include the `java.rmi.RemoteException`. The matching `ejbHome` method specified in the entity bean class must have the same number and types of arguments and must return the same type as the home method as specified in the home interface of the bean.

11.2.10 Entity bean's primary key class

The Bean Provider must specify a primary key class in the deployment descriptor.

The primary key type must be a legal Value Type in RMI-IIOP.

The class must provide suitable implementation of the `hashCode()` and `equals(Object other)` methods to simplify the management of the primary keys by client code.

11.3 The responsibilities of the Container Provider

This section describes the responsibilities of the Container Provider to support bean managed persistence entity beans. The Container Provider is responsible for providing the deployment tools, and for managing entity bean instances at runtime.

Because the EJB specification does not define the API between deployment tools and the container, we assume that the deployment tools are provided by the container provider. Alternatively, the deployment tools may be provided by a different vendor who uses the container vendor's specific API.

11.3.1 Generation of implementation classes

The deployment tools provided by the container provider are responsible for the generation of additional classes when the entity bean is deployed. The tools obtain the information that they need for generation of the additional classes by introspecting the classes and interfaces provided by the entity Bean Provider and by examining the entity bean's deployment descriptor.

The deployment tools must generate the following classes:

- A class that implements the entity bean's home interface (i.e., the entity EJBHome class).
- A class that implements the entity bean's remote interface (i.e., the entity EJBObject class).

The deployment tools may also generate a class that mixes some container-specific code with the entity bean class. The code may, for example, help the container to manage the entity bean instances at runtime. Tools can use subclassing, delegation, and code generation.

The deployment tools may also allow generation of additional code that wraps the business methods and that is used to customize the business logic for an existing operational environment. For example, a wrapper for a `debit` function on the `Account` Bean may check that the debited amount does not exceed a certain limit, or perform security checking that is specific to the operational environment.

11.3.2 Entity EJBHome class

The entity EJBHome class, which is generated by deployment tools, implements the entity bean's home interface. This class implements the methods of the `javax.ejb.EJBHome` interface, and the type-specific `create` and `finder` methods specific to the entity bean.

The implementation of each `create<METHOD>(...)` method invokes a matching `ejbCreate<METHOD>(...)` method, followed by the matching `ejbPostCreate<METHOD>(...)` method, passing the `create<METHOD>(...)` parameters to these matching methods.

The implementation of the `remove(...)` methods defined in the `javax.ejb.EJBHome` interface must activate an instance (if an instance is not already in the ready state) and invoke the `ejbRemove` method on the instance.

The implementation of each `find<METHOD>(...)` method invokes a matching `ejbFind<METHOD>(...)` method. The implementation of the `find<METHOD>(...)` method must create an entity object reference for the primary key returned from the `ejbFind<METHOD>` and return the entity object reference to the client. If the `ejbFind<METHOD>` method returns a collection of primary keys, the implementation of the `find<METHOD>(...)` method must create a collection of entity object references for the primary keys and return the collection to the client.

Before invoking the `ejbFind<METHOD>(...)` method, the container must first synchronize the state of any entity bean instances that are participating in the same transaction context as is used to execute the `ejbFind<METHOD>(...)` by invoking the `ejbStore()` method on those entity bean instances.

The implementation of each `<METHOD>(...)` home method invokes a matching `ejbHome<METHOD>(...)` method defined in the entity bean's class.

The implementation of the `ejbHome<METHOD>(...)` methods are provided by the bean provider.

11.3.3 Entity EJBObject class

The entity EJBObject class, which is generated by deployment tools, implements the entity bean's remote interface. It implements the methods of the `javax.ejb.EJBObject` interface and the business methods specific to the entity bean.

The implementation of the `remove(...)` method (defined in the `javax.ejb.EJBObject` interface) must activate an instance (if an instance is not already in the ready state) and invoke the `ejbRemove` method on the instance.

The implementation of each business method must activate an instance (if an instance is not already in the ready state) and invoke the matching business method on the instance.

11.3.4 Handle class

The deployment tools are responsible for implementing the handle class for the entity bean. The handle class must be serializable by the Java Serialization protocol.

As the handle class is not entity bean specific, the container may, but is not required to, use a single class for all deployed entity beans.

11.3.5 Home Handle class

The deployment tools responsible for implementing the home handle class for the entity bean. The handle class must be serializable by the Java Serialization protocol.

Because the home handle class is not entity bean specific, the container may, but is not required to, use a single class for the home handles of all deployed entity beans.

11.3.6 Meta-data class

The deployment tools are responsible for implementing the class that provides meta-data information to the client view contract. The class must be a valid RMI-IIOP Value Type, and must implement the `javax.ejb.EJBMetaData` interface.

Because the meta-data class is not entity bean specific, the container may, but is not required to, use a single class for all deployed enterprise beans.

11.3.7 Instance's re-entrance

The container runtime must enforce the rules defined in Section 11.1.12.

11.3.8 Transaction scoping, security, exceptions

The container runtime must follow the rules on transaction scoping, security checking, and exception handling described in Chapters 16, 20, and 17.

11.3.9 Implementation of object references

The container should implement the distribution protocol between the client and the container such that the object references of the home and remote interfaces used by entity bean clients are usable for a long period of time. Ideally, a client should be able to use an object reference across a server crash and restart. An object reference should become invalid only when the entity object has been removed, or after a reconfiguration of the server environment (for example, when the entity bean is moved to a different EJB server or container).

The motivation for this is to simplify the programming model for the entity bean client. While the client code needs to have a recovery handler for the system exceptions thrown from the individual method invocations on the home and remote interface, the client should not be forced to re-obtain the object references.

11.3.10 EntityContext

The container must implement the `EntityContext.getEJBContext()` method such that the bean instance can use the Java language cast to convert the returned value to the entity bean's remote interface type. Specifically, the bean instance does not have to use the `PortableRemoteObject.narrow(...)` method for the type conversion.

11.4 Object interaction diagrams

This section uses object interaction diagrams to illustrate the interactions between a bean managed persistence entity bean instance and its container.

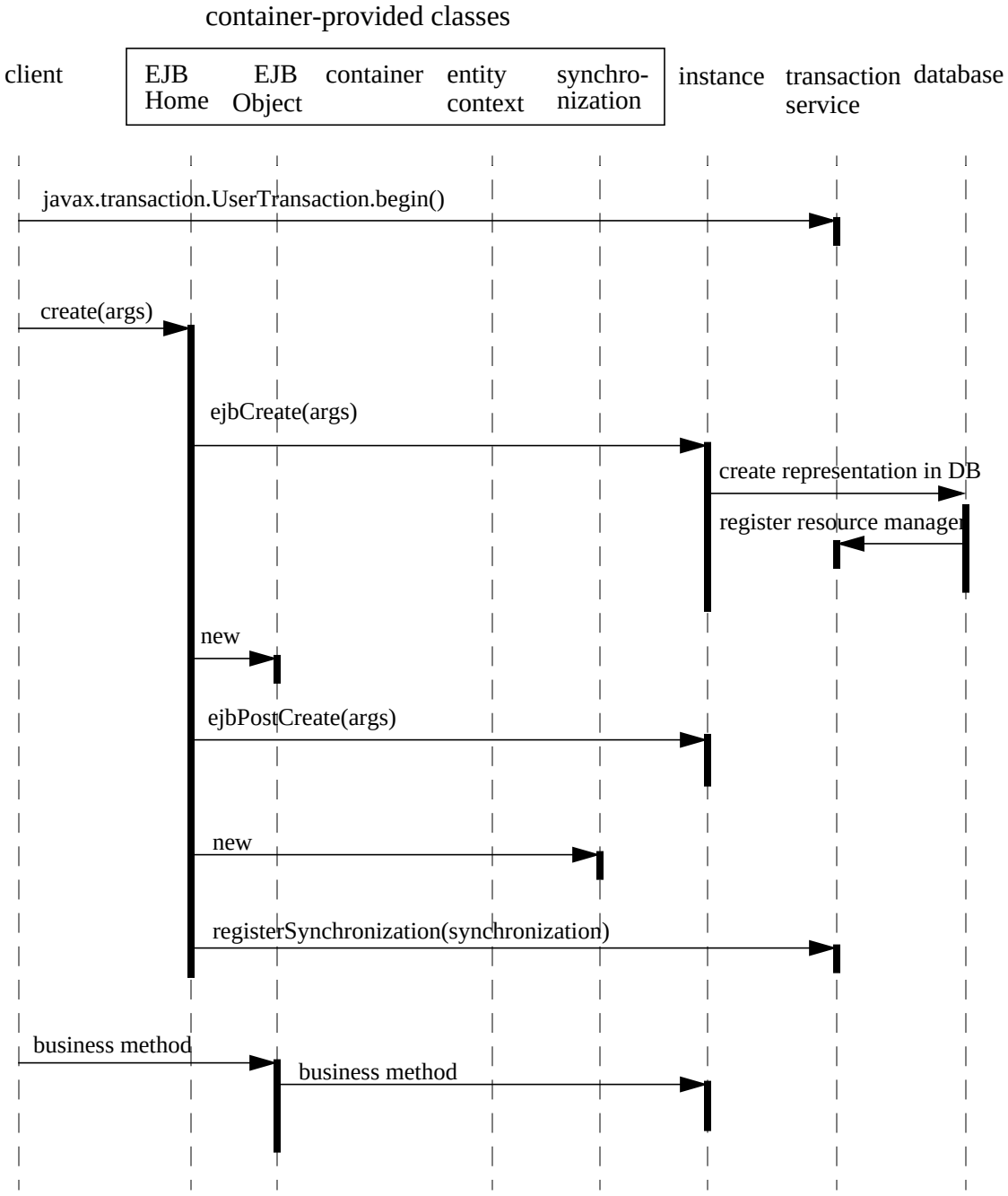
11.4.1 Notes

The object interaction diagrams illustrate a box labeled “container-provided classes.” These classes are either part of the container or are generated by the container tools. These classes communicate with each other through protocols that are container implementation specific. Therefore, the communication between these classes is not shown in the diagrams.

The classes shown in the diagrams should be considered as an illustrative implementation rather than as a prescriptive one.

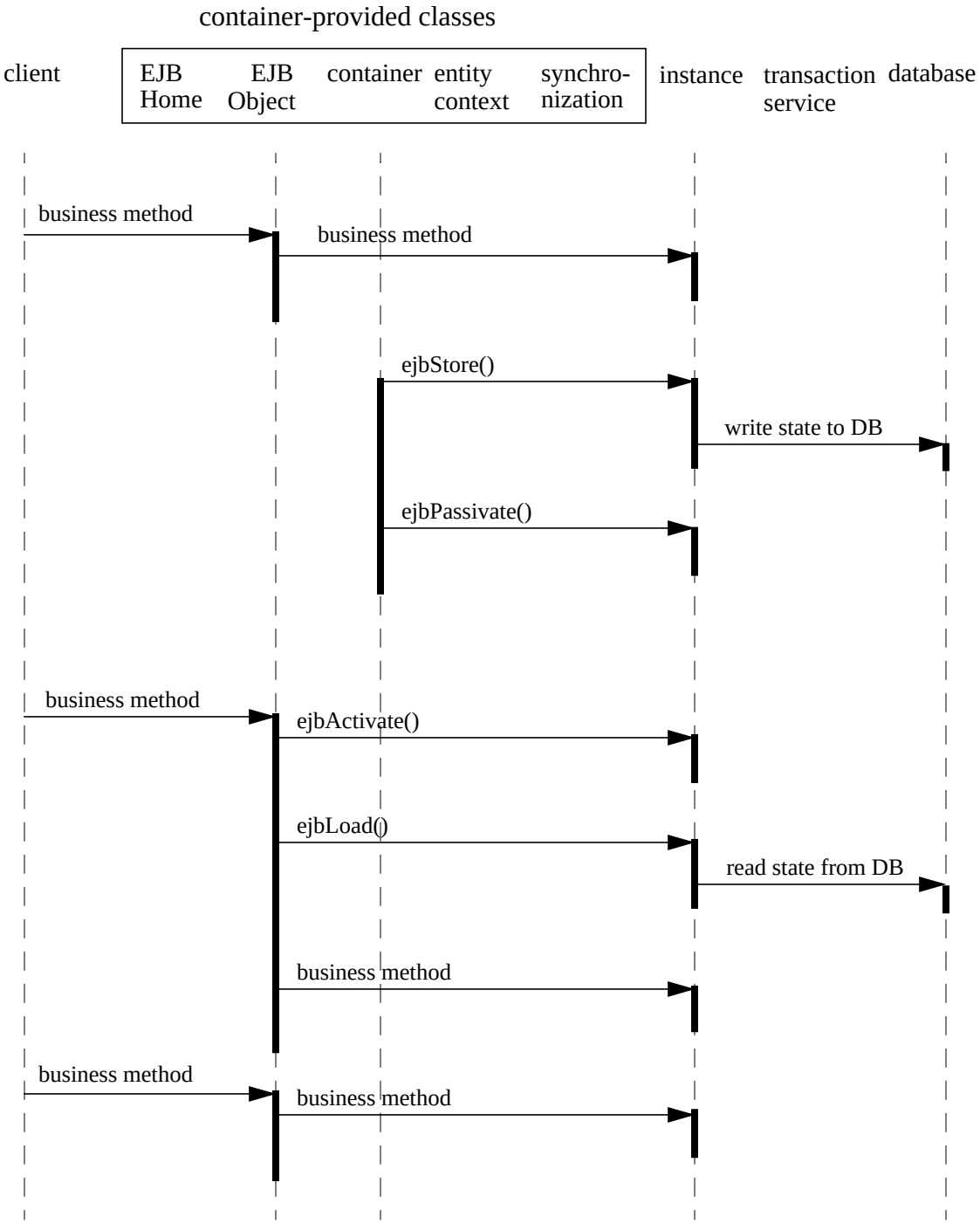
11.4.2 Creating an entity object

Figure 42 OID of Creation of an entity object with bean-managed persistence



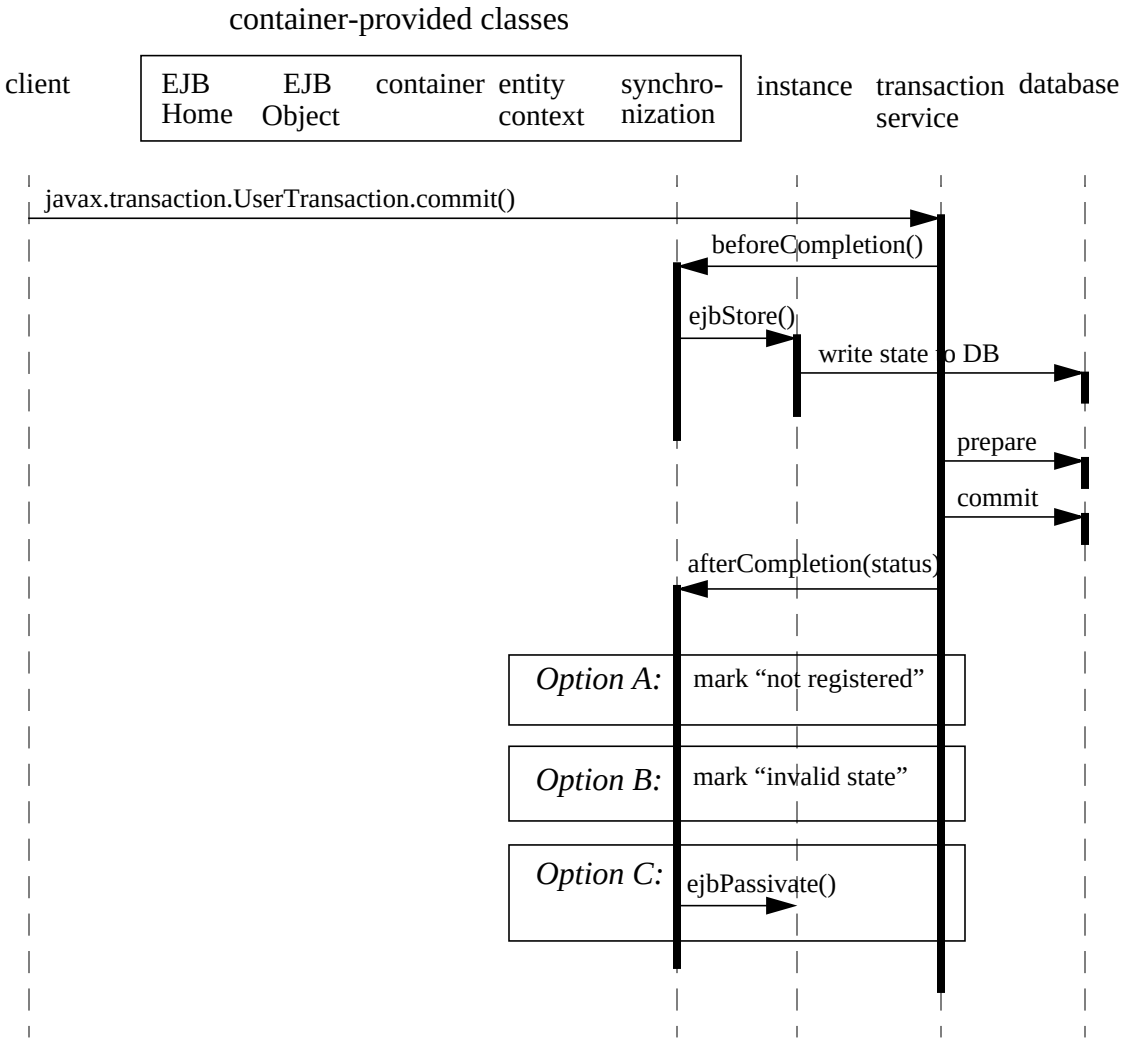
11.4.3 Passivating and activating an instance in a transaction

Figure 43 OID of passivation and reactivation of an entity bean instance with bean-managed persistence



11.4.4 Committing a transaction

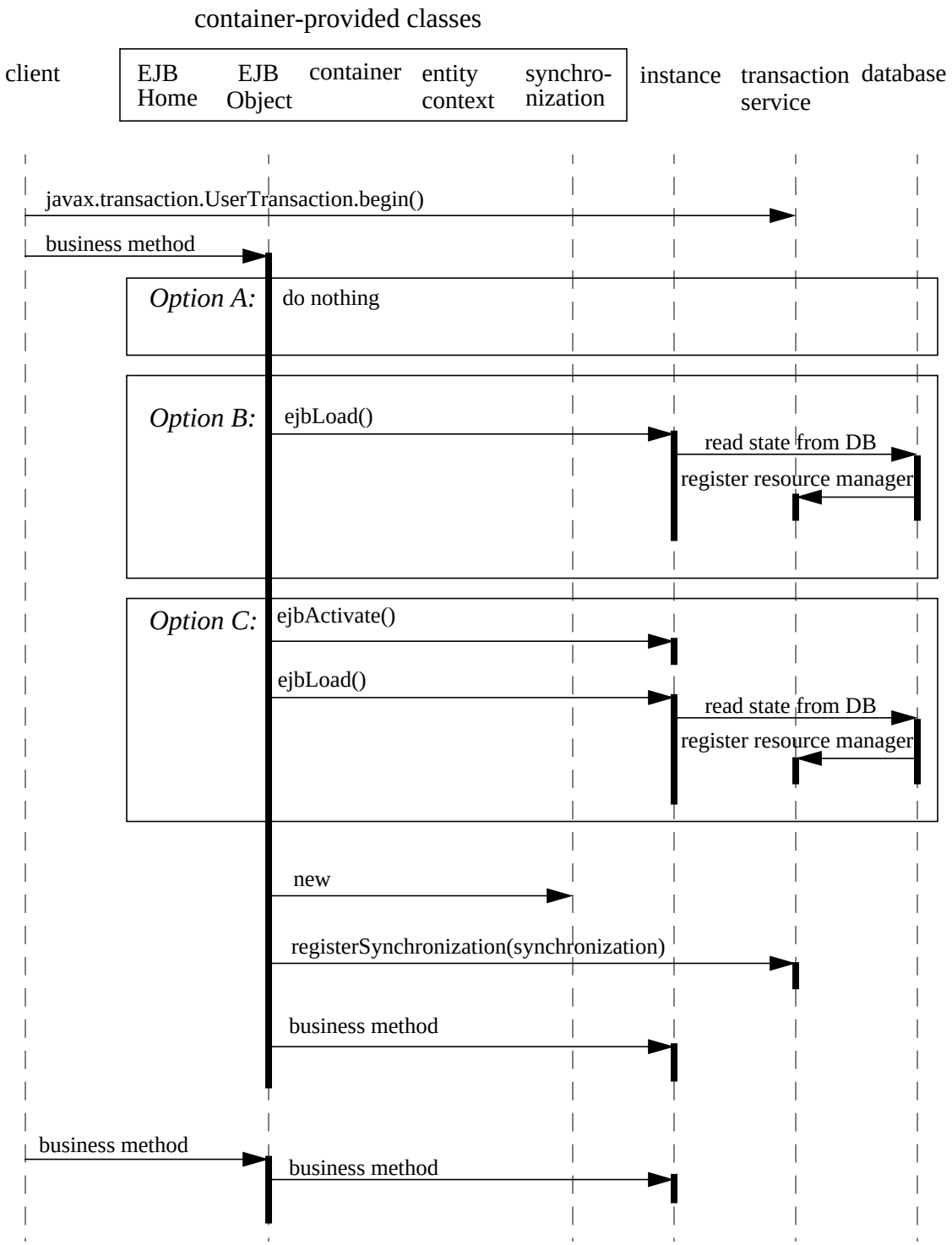
Figure 44 OID of transaction commit protocol for an entity bean instance with bean-managed persistence



11.4.5 Starting the next transaction

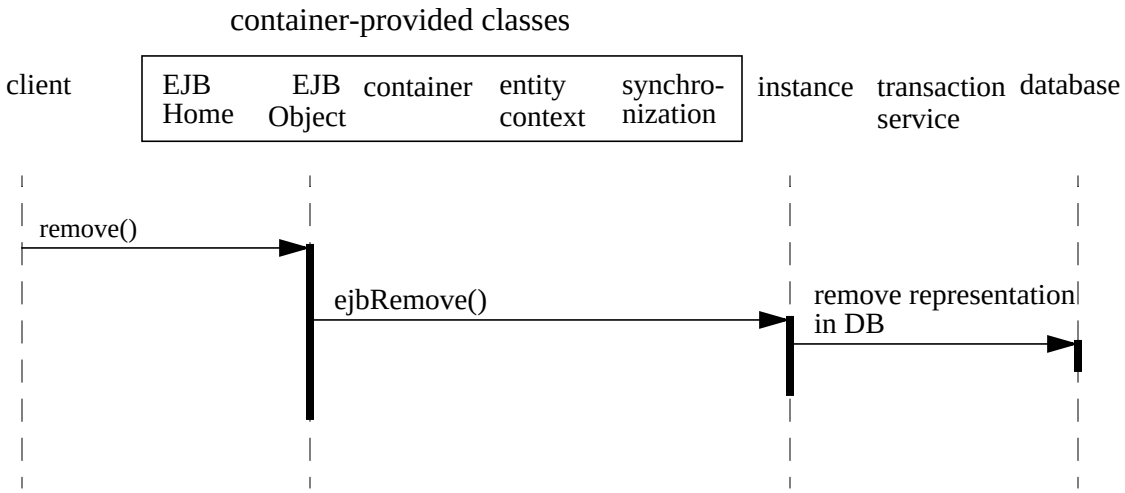
The following diagram illustrates the protocol performed for an entity bean instance with bean-managed persistence at the beginning of a new transaction. The three options illustrated in the diagram correspond to the three commit options in the previous subsection.

Figure 45 OID of start of transaction for an entity bean instance with bean-managed persistence



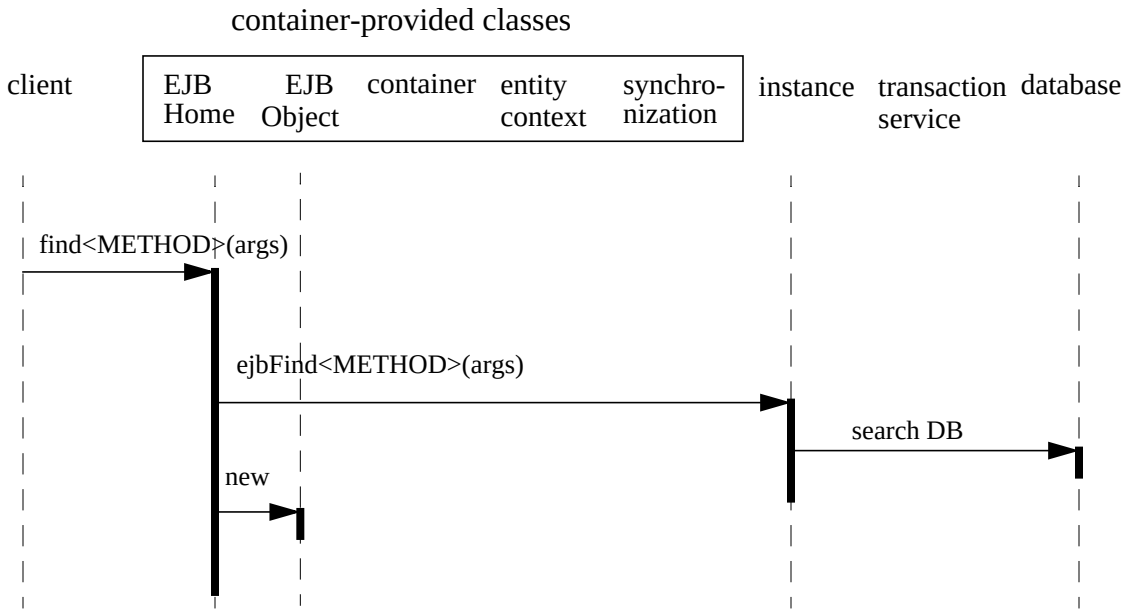
11.4.6 Removing an entity object

Figure 46 OID of removal of an entity bean object with bean-managed persistence



11.4.7 Finding an entity object

Figure 47 OID of execution of a finder method on an entity bean instance with bean-managed persistence



11.4.8 Adding and removing an instance from the pool

The diagrams in Subsections 11.4.2 through 11.4.7 did not show the sequences between the “does not exist” and “pooled” state (see the diagram in Section 11.1.4).

Figure 48 OID of a container adding an instance to the pool

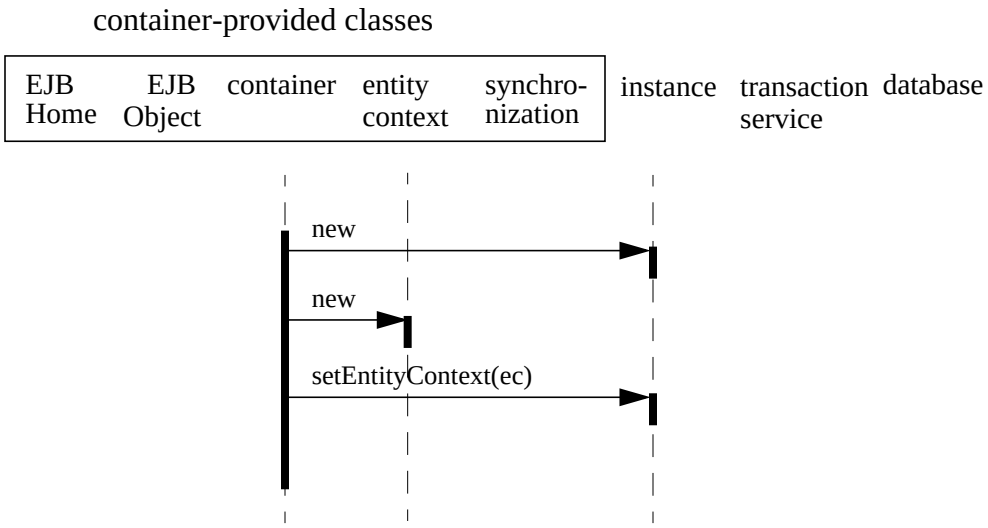
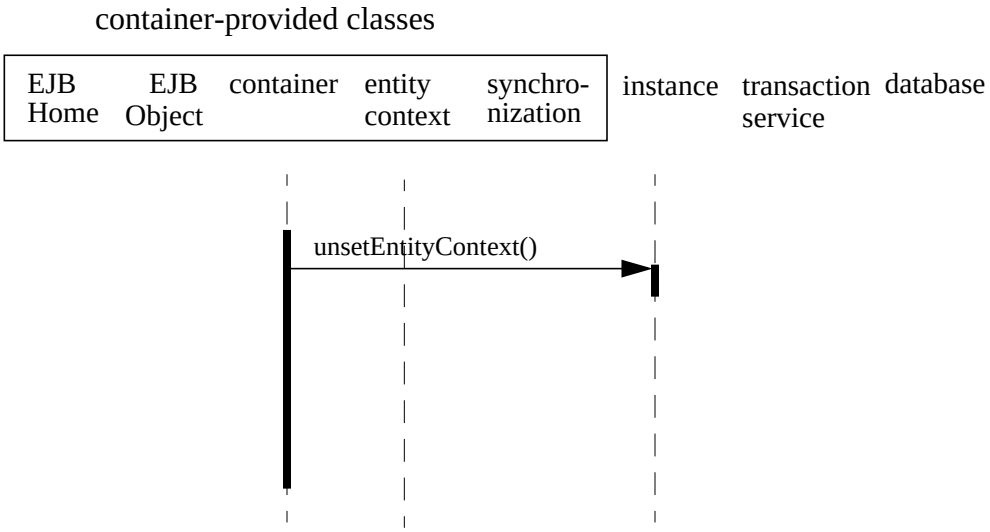


Figure 49 OID of a container removing an instance from the pool



Chapter 12

Example bean managed persistence entity scenario

This chapter describes an example development and deployment scenario for an entity bean using bean managed persistence. We use the scenario to explain the responsibilities of the entity Bean Provider and those of the container provider.

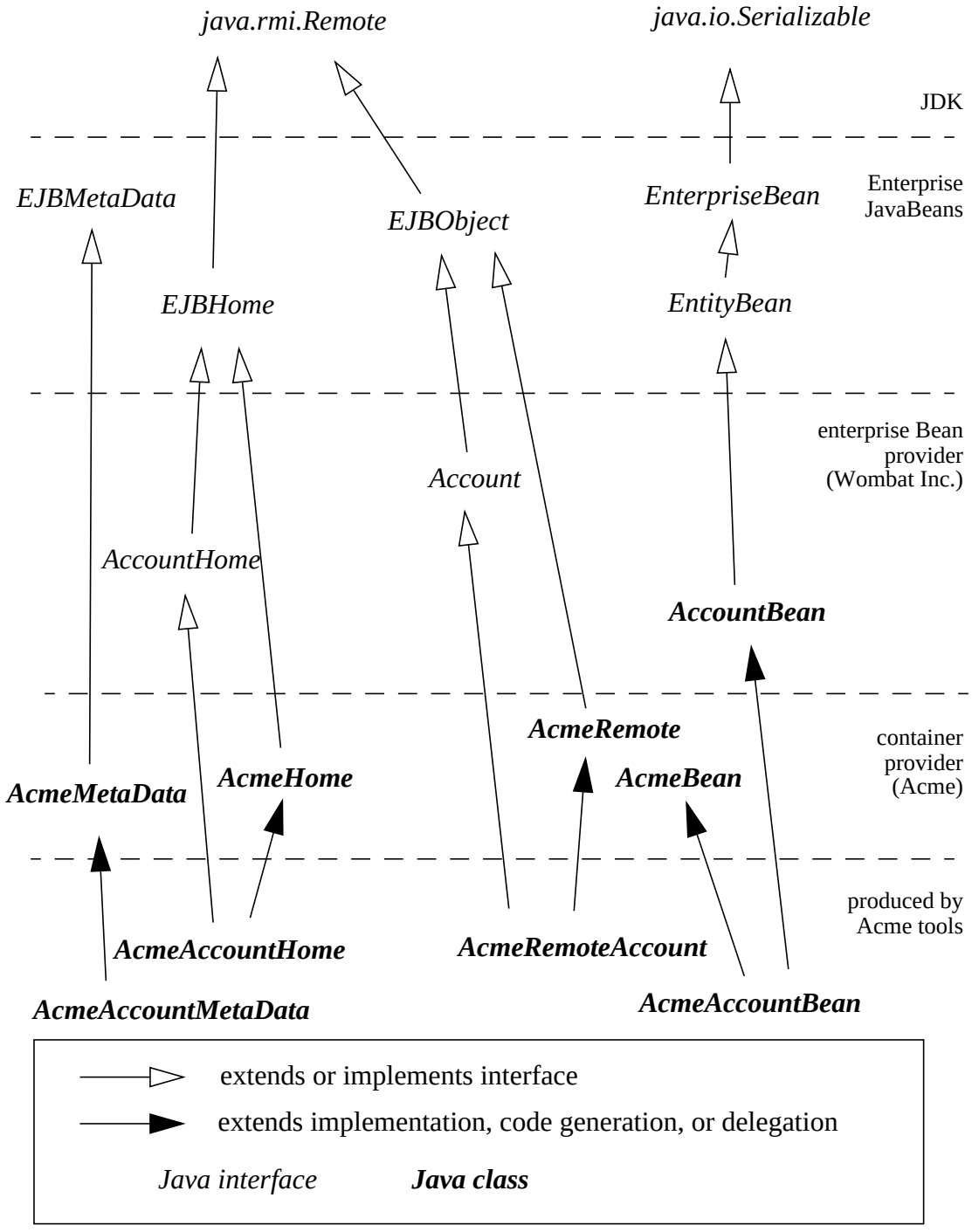
The classes generated by the container provider's tools in this scenario should be considered illustrative rather than prescriptive. Container providers are free to implement the contract between an entity bean and its container in a different way that achieves an equivalent effect (from the perspectives of the entity Bean Provider and the client-side programmer).

12.1 Overview

Wombat Inc. has developed the `AccountBean` *entity bean. The AccountBean entity bean is deployed in a container provided by the Acme Corporation.*

12.2 Inheritance relationship

Figure 50 Example of the inheritance relationship between the interfaces and classes:



12.2.1 What the entity Bean Provider is responsible for

Wombat Inc. is responsible for providing the following:

- *Define the entity bean's remote interface (Account). The remote interface defines the business methods callable by a client. The remote interface must extend the javax.ejb.EJBObject interface, and follow the standard rules for a RMI-IIOP remote interface. The remote interface must be defined as public.*

- *Write the business logic in the entity bean class (AccountBean). The entity bean class may, but is not required to, implement the entity bean's remote interface (Account). The entity bean must implement the methods of the javax.ejb.EntityBean interface, the ejbCreate*<METHOD>*(...) and ejbPostCreate*<METHOD>*(...) methods invoked at entity object creation, and the finder methods.*

- *Define a home interface (AccountHome) for the entity bean. The home interface defines the entity bean's specific create and finder methods. The home interface must be defined as public, extend the javax.ejb.EJBHome interface, and follow the standard rules for RMI-IIOP remote interfaces.*

- *Define a deployment descriptor that specifies any declarative information that the entity bean provider wishes to pass with the entity bean to the next stage of the development/deployment workflow.*

12.2.2 Classes supplied by Container Provider

The following classes are supplied by the container provider, Acme Corp:

- *The AcmeHome class provides the Acme implementation of the javax.ejb.EJBHome methods.*

- *The AcmeRemote class provides the Acme implementation of the javax.ejb.EJBObject methods.*

- *The AcmeBean class provides additional state and methods to allow Acme's container to manage its entity bean instances. For example, if Acme's container uses an LRU algorithm, then AcmeBean may include the clock count and methods to use it.*

- *The AcmeMetaData class provides the Acme implementation of the javax.ejb.EJBMetaData methods.*

12.2.3 What the container provider is responsible for

The tools provided by Acme Corporation are responsible for the following:

- *Generate the entity EJBOBject class (AcmeRemoteAccount) that implements the entity bean's remote interface. The tools also generate the classes that implement the communication protocol specific artifacts for the remote interface.*

- *Generate the implementation of the entity bean class suitable for the Acme container (AcmeAccountBean). AcmeAccountBean includes the business logic from the AccountBean class mixed with the services defined in the AcmeBean class. Acme tools can use inheritance, delegation, and code generation to achieve mix-in of the two classes.*

- *Generate the entity EJBHome class (AcmeAccountHome) for the entity bean. that implements the entity bean's home interface (AccountHome). The tools also generate the classes that implement the communication protocol specific artifacts for the home interface.*

- *Generate a class (AcmeAccountMetaData) that implements the javax.ejb.EJBMetaData interface for the Account Bean.*

The above classes and tools are container-specific (i.e., they reflect the way Acme Corp implemented them). Other container providers may use different mechanisms to produce their runtime classes, and the generated classes most likely will be different from those generated by Acme's tools.

Chapter 13

EJB 1.1 Entity Bean Component Contract for Container Managed Persistence

This chapter specifies the EJB 1.1 entity bean component contract for container managed persistence.

While we require container providers to support backward compatibility for EJB 1.1 entity beans with container managed persistence by the implementation of this contract, we highly recommend that Bean Providers use the Entity Bean Component Contract for Container Managed Persistence specified in Chapter 9 for the development of new entity beans because of the more complete functionality that it provides.

13.1 EJB 1.1 Entity beans with container-managed persistence

Chapter 11 “Entity Bean Component Contract for Bean Managed Persistence” describes the component contract for entity beans with bean-managed persistence. The contract for an EJB 1.1 entity bean with container-managed persistence is the same as the contract for an entity bean with bean-managed persistence as described in Chapter 11, except for the differences described in this chapter.

13.1.1 Container-managed fields

An EJB 1.1 entity bean with container-managed persistence relies on the Container Provider's tools to generate methods that perform data access on behalf of the entity bean instances. The generated methods transfer data between the entity bean instance's variables and the underlying resource manager at the times defined by the EJB specification. The generated methods also implement the creation, removal, and lookup of the entity object in the underlying database.

An entity bean with container-manager persistence must not code explicit data access—all data access must be deferred to the Container.

The EJB 1.1 entity Bean Provider is responsible for using the `cmp-field` elements of the deployment descriptor to declare the instance's fields that the Container must load and store at the defined times. The fields must be defined in the entity bean class as `public`, and must not be defined as `transient`.

The container is responsible for transferring data between the entity bean's instance variables and the underlying data source before or after the execution of the `ejbCreate`, `ejbRemove`, `ejbLoad`, and `ejbStore` methods, as described in the following subsections. The container is also responsible for the implementation of the finder methods.

The EJB 2.0 deployment descriptor for an EJB 1.1 entity bean with container managed persistence indicates that the entity bean uses container-managed persistence and that the value of its `cmp-version` element is `1.x`.

The EJB 1.1 component contract does not architect support for relationships for entity beans with container managed persistence. EJB 2.0 does not support the use of the `cmr-field`, `dependents`, `ejb-relation`, or `query` deployment descriptor elements or their subelements for EJB 1.1 entity beans.

The following requirements ensure that an EJB 1.1 entity bean with container managed persistence can be deployed in any compliant container.

- The Bean Provider must ensure that the Java types assigned to the container-managed fields are restricted to the following: Java primitive types, Java serializable types, and references of enterprise beans' remote or home interfaces.

- The Container Provider may, but is not required to, use Java Serialization to store the container-managed fields in the database. If the container chooses a different approach, the effect should be equivalent to that of Java Serialization. The Container must also be capable of persisting references to enterprise beans' remote and home interfaces (for example, by storing their handle or primary key).

Although the above requirements allow the Bean Provider to specify almost any arbitrary type for the container-managed fields, we expect that in practice the Bean Provider of EJB 1.1 entity beans with container managed persistence will use relatively simple Java types, and that most Containers will be able to map these simple Java types to columns in a database schema to externalize the entity state in the database, rather than use Java serialization.

If the Bean Provider expects that the container-managed fields will be mapped to database fields, he or she should provide mapping instructions to the Deployer. The mapping between the instance's container-managed fields and the schema of the underlying database manager will be then realized by the data access classes generated by the container provider's tools. Because entity beans are typically coarse-grained objects, the content of the container-managed fields may be stored in multiple rows, possibly spread across multiple database tables. These mapping techniques are beyond the scope of the EJB specification, and do not have to be supported by an EJB compliant container. (The container may simply use the Java serialization protocol in all cases).

Because a compliant EJB Container is not required to provide any support for mapping the container-managed fields to a database schema, a Bean Provider of entity beans that need a particular mapping to an underlying database schema should use bean managed persistence or the container managed persistence contract specified in Chapter 9 of this specification instead.

The provider of EJB 1.1 entity beans with container-managed persistence must take into account the following limitations of the EJB 1.1 container-managed persistence protocol:

- Data aliasing problems. If container-managed fields of multiple entity beans map to the same data item in the underlying database, the entity beans may see an inconsistent view of the data item if the multiple entity beans are invoked in the same transaction. (That is, an update of the data item done through a container-managed field of one entity bean may not be visible to another entity bean in the same transaction if the other entity bean maps to the same data item.)

- Eager loading of state. The Container loads the entire entity object state into the container-managed fields before invoking the `ejbLoad` method. This approach may not be optimal for entity objects with large state if most business methods require access to only parts of the state.

An entity bean designer who runs into the limitations of EJB 1.1 container-managed persistence should use the container managed persistence contracts specified in Chapter 9 of this specification instead.

13.1.2 ejbCreate, ejbPostCreate

With bean-managed persistence, the entity Bean Provider is responsible for writing the code that inserts a record into the database in the `ejbCreate(...)` methods. However, with container-managed persistence, the container performs the database insert after the `ejbCreate(...)` method completes.

The Container must ensure that the values of the container-managed fields are set to the Java language defaults (e.g. 0 for integer, `null` for pointers) prior to invoking an `ejbCreate(...)` method on an instance.

The EJB 1.1 entity Bean Provider's responsibility is to initialize the container-managed fields in the `ejbCreate(...)` methods from the input arguments such that when an `ejbCreate(...)` method returns, the container can extract the container-managed fields from the instance and insert them into the database.

The `ejbCreate(...)` methods must be defined to return the primary key class type. The implementation of the `ejbCreate(...)` methods should be coded to return a `null`. The returned value is ignored by the Container.

Note: The above requirement is to allow the creation of an entity bean with bean-managed persistence by subclassing an EJB 1.1 entity bean with container-managed persistence. The Java language rules for overriding methods in subclasses requires the signatures of the ejbCreate(...) methods in the subclass and the superclass be the same.

The container is responsible for creating the entity object's representation in the underlying database, extracting the primary key fields of the newly created entity object representation in the database, and for creating an entity EJBObject reference for the newly created entity object. The Container must establish the primary key before it invokes the `ejbPostCreate(...)` method. The container may create the representation of the entity in the database immediately after `ejbCreate(...)` returns, or it can defer it to a later time (for example to the time after the matching `ejbPostCreate(...)` has been called, or to the end of the transaction).

The container then invokes the matching `ejbPostCreate(...)` method on the instance. The instance can discover the primary key by calling `getPrimaryKey()` on its entity context object.

The container must invoke `ejbCreate`, perform the database insert operation, and invoke `ejbPostCreate` in the transaction context determined by the transaction attribute of the matching `create(...)` method, as described in subsection 16.6.2.

The Container throws the `DuplicateKeyException` if the newly created entity object would have the same primary key as one of the existing entity objects within the same home.

13.1.3 ejbRemove

The container invokes the `ejbRemove()` method on an entity bean instance with container-managed persistence in response to a client-invoked `remove` operation on the entity bean's home or remote interface.

The entity Bean Provider can use the `ejbRemove` method to implement any actions that must be done before the entity object's representation is removed from the database.

The container synchronizes the instance's state before it invokes the `ejbRemove` method. This means that the state of the instance variables at the beginning of the `ejbRemove` method is the same as it would be at the beginning of a business method.

After `ejbRemove` returns, the container removes the entity object's representation from the database.

The container must perform `ejbRemove` and the database delete operation in the transaction context determined by the transaction attribute of the invoked `remove` method, as described in subsection 16.6.2.

13.1.4 ejbLoad

When the container needs to synchronize the state of an enterprise bean instance with the entity object's state in the database, the container reads the entity object's state from the database into the container-managed fields and then it invokes the `ejbLoad()` method on the instance.

The entity Bean Provider can rely on the container's having loaded the container-managed fields from the database just before the container invokes the `ejbLoad()` method. The entity bean can use the `ejbLoad()` method, for instance, to perform some computation on the values of the fields that were read by the container (for example, uncompressing text fields).

13.1.5 ejbStore

When the container needs to synchronize the state of the entity object in the database with the state of the enterprise bean instance, the container first calls the `ejbStore()` method on the instance, and then it extracts the container-managed fields and writes them to the database.

The entity Bean Provider should use the `ejbStore()` method to set up the values of the container-managed fields just before the container writes them to the database. For example, the `ejbStore()` method may perform compression of text before the text is stored in the database.

13.1.6 finder methods

The entity Bean Provider does not write the finder (`ejbFind<METHOD>(...)`) methods.

The finder methods are generated at the entity bean deployment time using the container provider's tools. The tools can, for example, create a subclass of the entity bean class that implements the `ejbFind<METHOD>()` methods, or the tools can generate the implementation of the finder methods directly in the class that implements the entity bean's home interface.

Note that the `ejbFind<METHOD>` names and parameter signatures of EJB 1.1 entity beans do not provide the container tools with sufficient information for automatically generating the implementation of the finder methods for methods other than `ejbFindByPrimaryKey`. Therefore, the bean provider is responsible for providing a description of each finder method. The entity bean Deployer uses container tools to generate the implementation of the finder methods based in the description supplied by the bean provider.

The EJB1.1 component contract for container managed persistence does not specify the format of the finder method description. A Bean Provider of entity beans that needs this functionality should use the container managed persistence contract specified in Chapter 9 of this specification instead.

13.1.7 home methods

The EJB1.1 entity bean contract does not support `ejbHome` methods. A Bean Provider of entity beans that need the home method functionality should use the container managed persistence contracts specified in Chapter 9 of this specification instead.

13.1.8 create methods

The EJB1.1 entity bean contract does not support `create<METHOD>` methods. A Bean Provider of entity beans that needs the flexibility in method naming that `create<METHOD>` methods provide should use the container managed persistence contracts specified in Chapter 9 of this specification instead.

13.1.9 primary key type

The container must be able to manipulate the primary key type. Therefore, the primary key type for an entity bean with container-managed persistence must follow the rules in this subsection, in addition to those specified in Subsection 11.2.10.

There are two ways to specify a primary key class for an entity bean with container-managed persistence:

- Primary key that maps to a single field in the entity bean class.
- Primary key that maps to multiple fields in the entity bean class.

The second method is necessary for implementing compound keys, and the first method is convenient for single-field keys. Without the first method, simple types such as String would have to be wrapped in a user-defined class.

13.1.9.1 Primary key that maps to a single field in the entity bean class

The Bean Provider uses the `primkey-field` element of the deployment descriptor to specify the container-managed field of the entity bean class that contains the primary key. The field's type must be the primary key type.

13.1.9.2 Primary key that maps to multiple fields in the entity bean class

The primary key class must be `public`, and must have a `public` constructor with no parameters.

All fields in the primary key class must be declared as public.

The names of the fields in the primary key class must be a subset of the names of the container-managed fields. (This allows the container to extract the primary key fields from an instance's container-managed fields, and vice versa.)

13.1.9.3 Special case: Unknown primary key class

In special situations, the entity Bean Provider may choose not to specify the primary key class for an entity bean with container-managed persistence. This case usually happens when the entity bean does not have a natural primary key, and the Bean Provider wants to allow the Deployer to select the primary key fields at deployment time. The entity bean's primary key type will usually be derived from the primary key type used by the underlying database system that stores the entity objects. The primary key used by the database system may not be known to the Bean Provider.

When defining the primary key for the enterprise bean, the Deployer may sometimes need to subclass the entity bean class to add additional container-managed fields (this typically happens for entity beans that do not have a natural primary key, and the primary keys are system-generated by the underlying database system that stores the entity objects).

In this special case, the type of the argument of the `findByPrimaryKey` method must be declared as `java.lang.Object`, and the return value of `ejbCreate()` must be declared as `java.lang.Object`. The Bean Provider must specify the primary key class in the deployment descriptor as of the type `java.lang.Object`.

The primary key class is specified at deployment time in the situations when the Bean Provider develops an entity bean that is intended to be used with multiple back-ends that provide persistence, and when these multiple back-ends require different primary key structures.

Use of entity beans with a deferred primary key type specification limits the client application programming model, because the clients written prior to deployment of the entity bean may not use, in general, the methods that rely on the knowledge of the primary key type.

The implementation of the enterprise bean class methods must be done carefully. For example, the methods should not depend on the type of the object returned from EntityContext.getPrimaryKey(), because the return type is determined by the Deployer after the EJB class has been written.

13.2 Object interaction diagrams

This section uses object interaction diagrams to illustrate the interactions between an EJB 1.1 entity bean instance and its container.

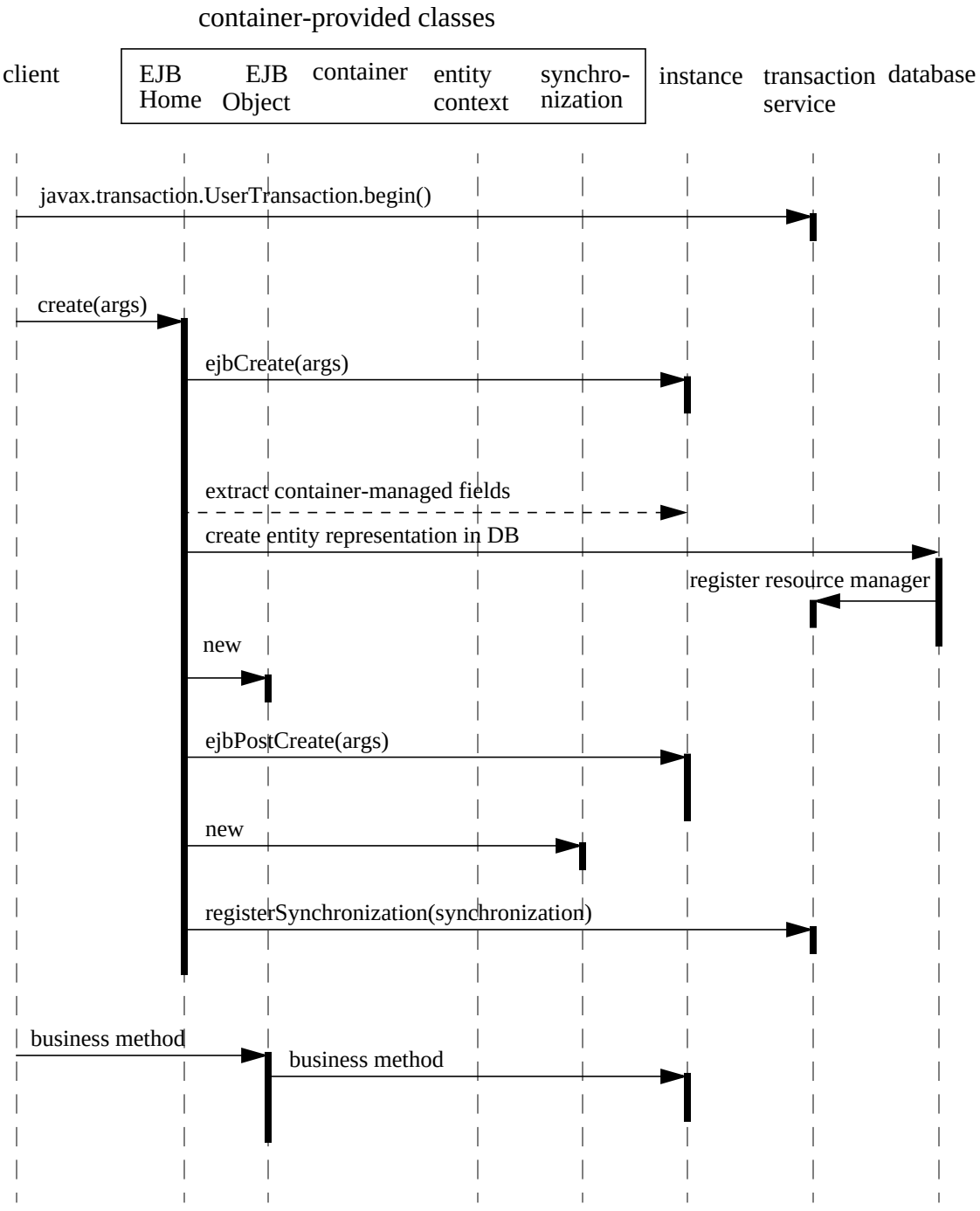
13.2.1 Notes

The object interaction diagrams illustrate a box labeled “container-provided classes.” These classes are either part of the container or are generated by the container tools. These classes communicate with each other through protocols that are container implementation specific. Therefore, the communication between these classes is not shown in the diagrams.

The classes shown in the diagrams should be considered as an illustrative implementation rather than as a prescriptive one

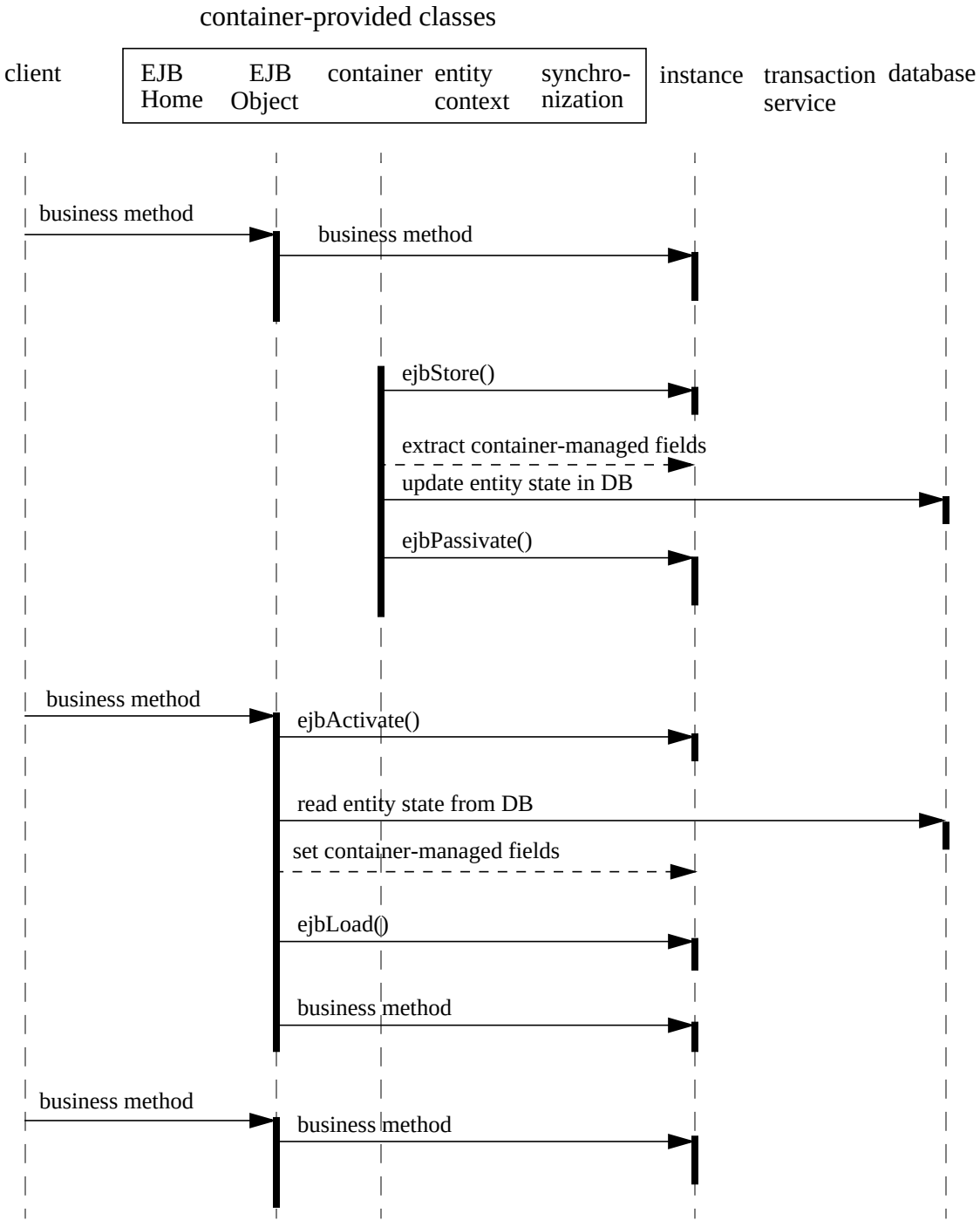
13.2.2 Creating an entity object

Figure 51 OID of creation of an entity object with EJB 1.1 container-managed persistence



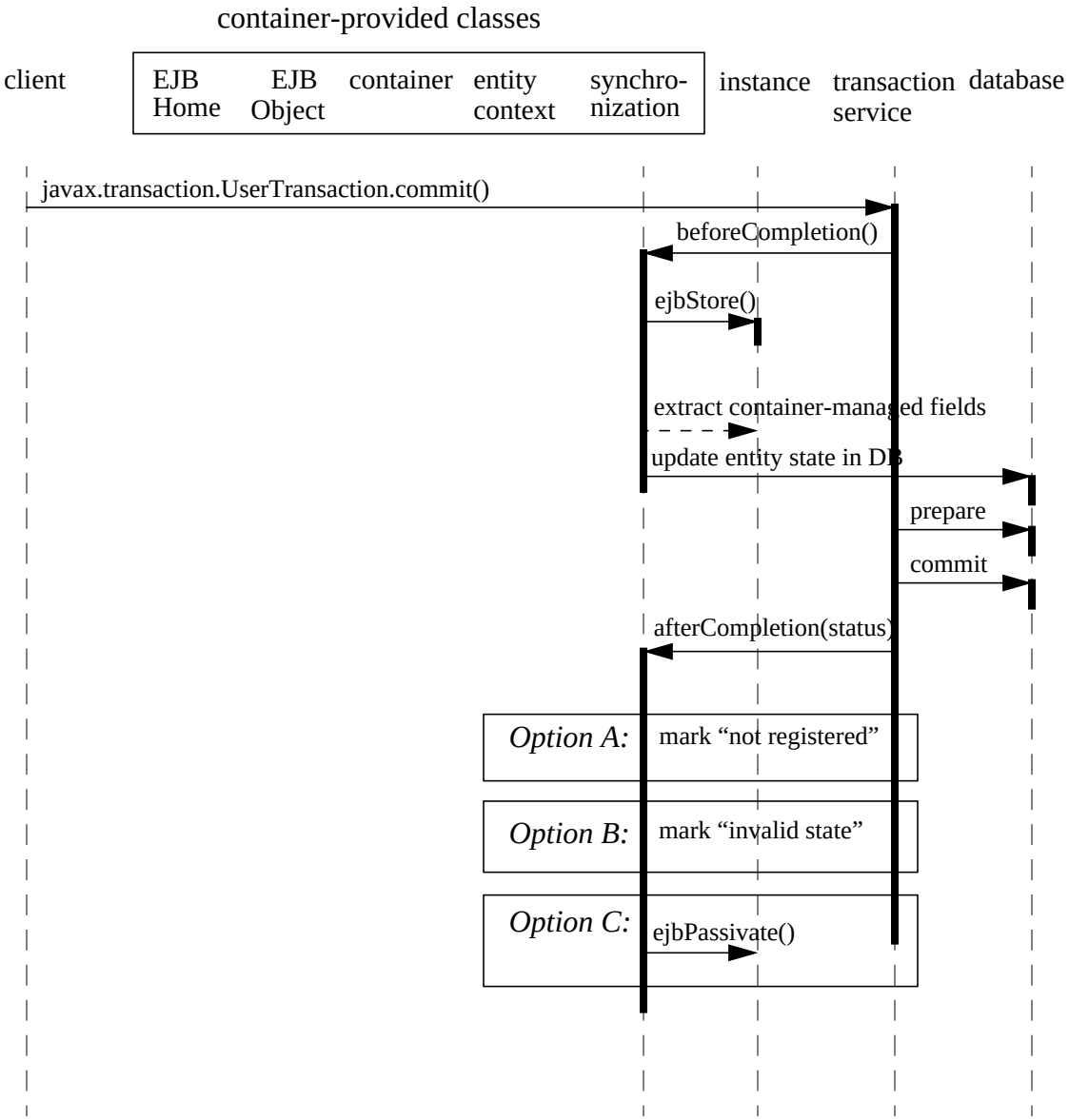
13.2.3 Passivating and activating an instance in a transaction

Figure 52 OID of passivation and reactivation of an entity bean instance with EJB 1.1 CMP



13.2.4 Committing a transaction

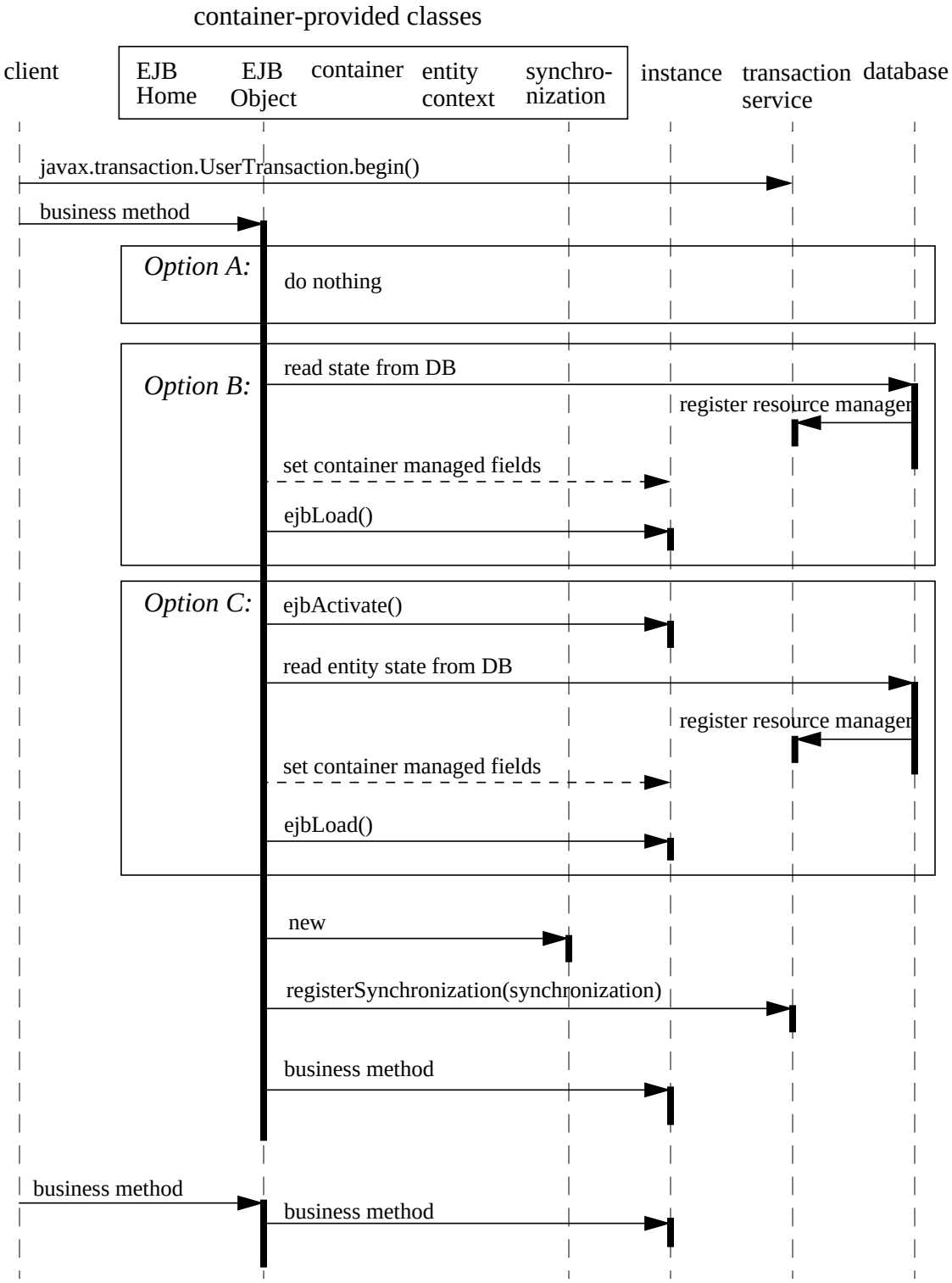
Figure 53 OID of transaction commit protocol for an entity bean instance with EJB 1.1 container-managed persistence



13.2.5 Starting the next transaction

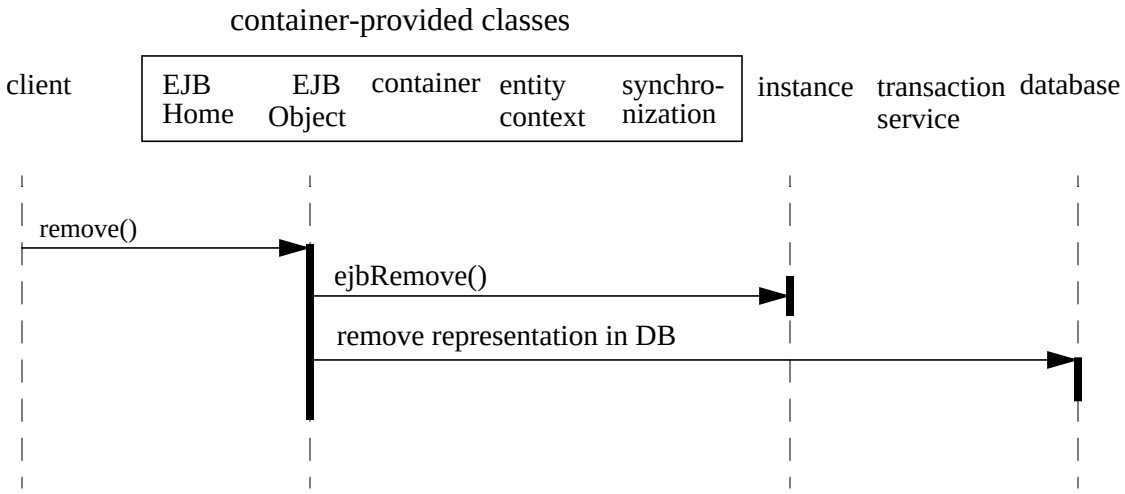
The following diagram illustrates the protocol performed for an entity bean instance with EJB 1.1 container-managed persistence at the beginning of a new transaction. The three options illustrated in the diagram correspond to the three commit options in the previous subsection.

Figure 54 OID of start of transaction for an entity bean instance with EJB 1.1 container-managed persistence



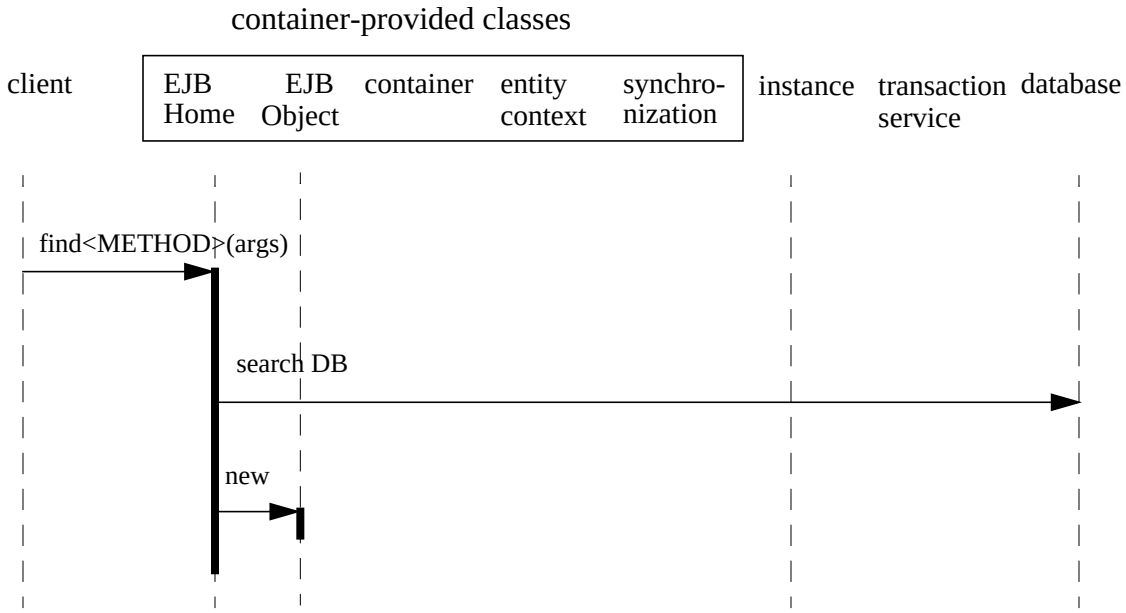
13.2.6 Removing an entity object

Figure 55 OID of removal of an entity bean object with EJB 1.1 container-managed persistence



13.2.7 Finding an entity object

Figure 56 OID of execution of a finder method on an entity bean instance with EJB 1.1 container-managed persistence



13.2.8 Adding and removing an instance from the pool

The diagrams in Subsections 13.2.7 through 13.2.7 did not show the sequences between the “does not exist” and “pooled” state (see the diagram in Section 11.1.4).

Figure 57 OID of a container adding an instance to the pool

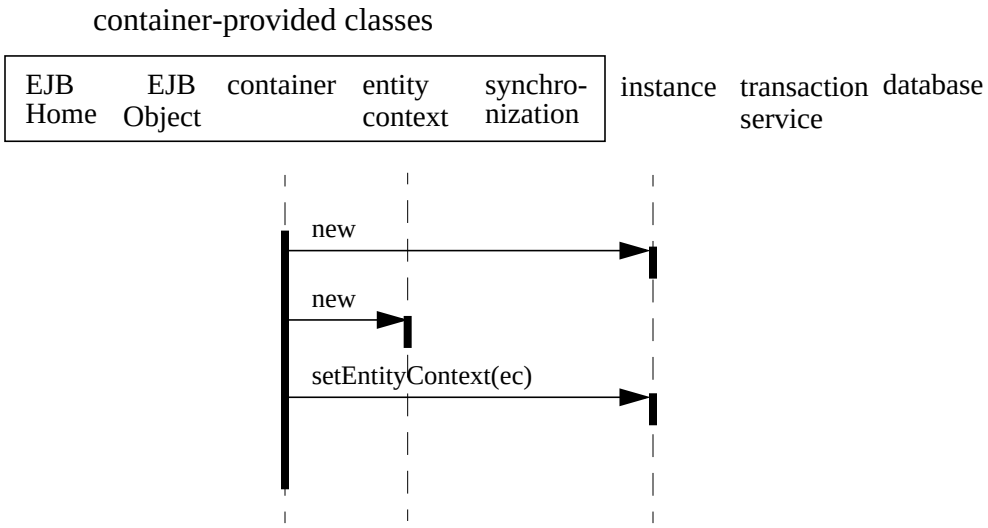
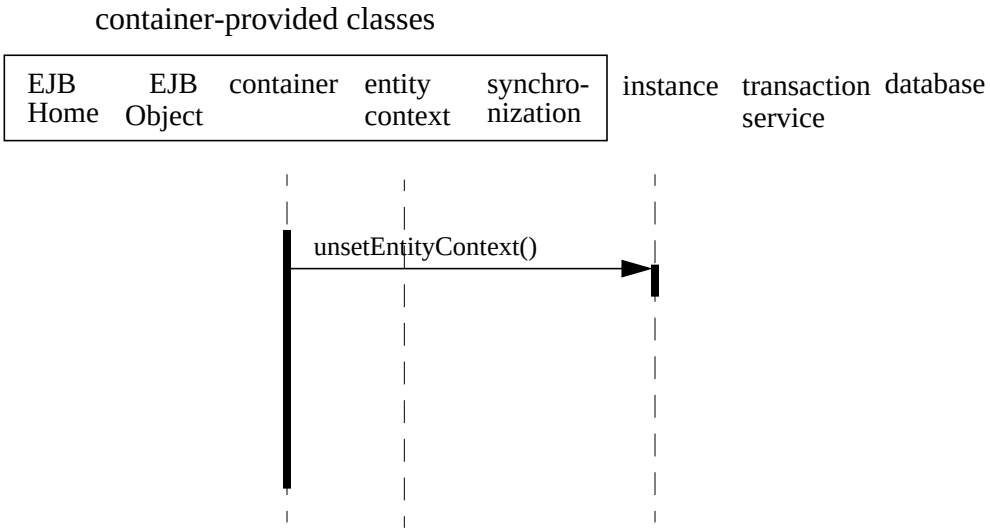


Figure 58 OID of a container removing an instance from the pool



Chapter 14 Message-driven Bean Component Contract

This chapter specifies the contract between a message-driven bean and its container. It defines the life cycle of the message-driven bean instances.

This chapter defines the developer's view of message-driven bean state management and the container's responsibility for managing it.

14.1 Overview

A message-driven bean is an asynchronous message consumer. A message-driven bean is invoked by the container as a result of the arrival of a JMS message. A message-driven bean has neither a home nor a remote interface. A message-driven bean instance is an instance of a message-driven bean class.

To a client, a message-driven bean is a JMS message consumer that implements some business logic running on the server. A client accesses a message-driven bean through JMS by sending messages to the JMS Destination (Queue or Topic) for which the message-driven bean class is the MessageListener.

Message-driven bean instances have no conversational state. This means that all bean instances are equivalent when they are not involved in servicing a client message.

Message-driven beans are anonymous. They have no client-visible identity.

A message-driven bean instance is created by the container to handle the processing of the messages for which the message-driven bean is the consumer. Its lifetime is controlled by the container.

A message-driven bean instance has no state for a specific client. However, the instance variables of the message-driven bean instance can contain state across the handling of client messages. Examples of such state include an open database connection and an object reference to an EJB object.

14.2 Goals

The goal of the message-driven bean model is to make developing an enterprise bean that is asynchronously invoked to handle the processing of incoming JMS messages as simple as developing the same functionality in any other JMS MessageListener.

A further goal of the message-driven bean model is to allow for the concurrent processing of a stream of messages by means of container-provided pooling of message-driven bean instances.

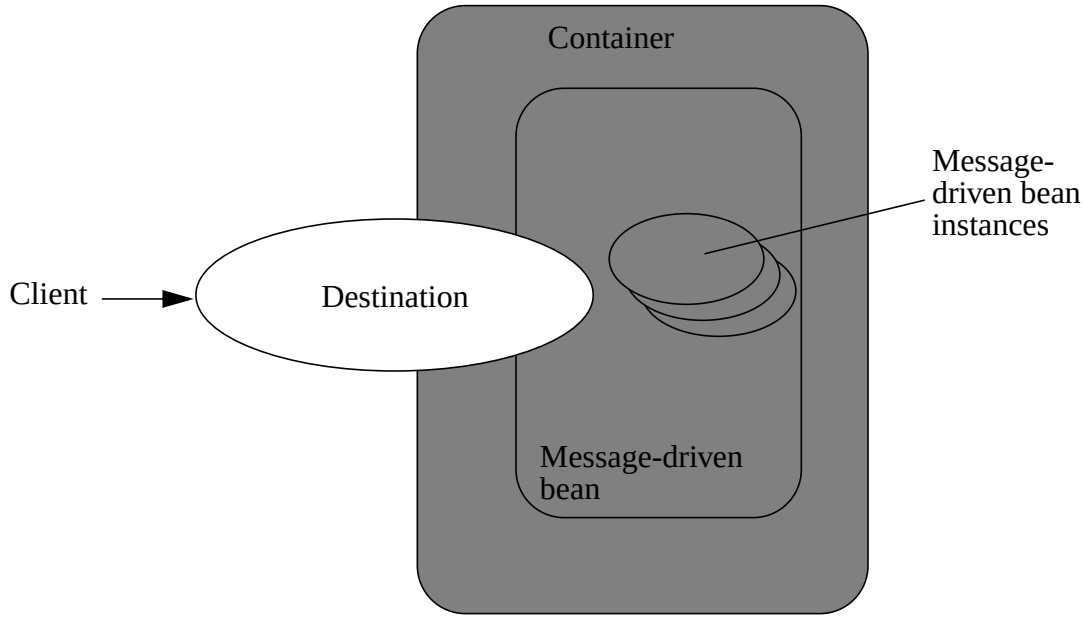
> *While the EJB 2.0 specification requires support for only JMS-based messaging, a future goal of the message-driven bean model is to provide support for other types of messaging in addition to JMS, and to allow for message-driven beans that are written to their APIs.*

14.3 Client view of a message-driven bean

To a client, a message-driven bean is simply a JMS message consumer. The client sends messages to the Destination (Queue or Topic) for which the message-driven bean is the MessageListener just as it would to any other Destination. The message-driven bean, like any other JMS message consumer, handles the processing of the messages.

From the perspective of the client, the existence of a message-driven bean is completely hidden behind the JMS destination for which the message-driven bean is the message listener. The following diagram illustrates the view that is provided to a message-driven bean’s clients.

Figure 59 Client view of message-driven beans deployed in a container



A client locates the JMS Destination associated with a message-driven bean by using JNDI. For example, the Queue for the `StockInfo` message-driven bean can be located using the following code segment:

```
Context initialContext = new InitialContext();
Queue stockInfoQueue = (javax.jms.Queue)initialContext.lookup
    (“java:comp/env/jms/stockInfoQueue”);
```

A client's JNDI name space may be configured to include the JMS Destinations of message-driven beans installed in multiple EJB Containers located on multiple machines on a network. The actual locations of an enterprise bean and EJB Container are, in general, transparent to the client using the enterprise bean.

The remainder of this section describes the message-driven bean life cycle in detail and the protocol between the message-driven bean and its container.

14.4 Protocol between a message-driven bean instance and its container

From its creation until destruction, a message-driven bean instance lives in a container. The container provides security, concurrency, transactions, and other services for the message-driven bean. The container manages the life cycle of the message-driven bean instances, notifying the instances when bean action may be necessary, and providing a full range of services to ensure that the message-driven bean implementation is scalable and can support the concurrent processing of a large number of messages.

From the Bean Provider's point of view, a message-driven bean exists as long as its container does. It is the container's responsibility to ensure that the message-driven bean comes into existence when the container is started up and that instances of the bean are ready to receive an asynchronous message delivery before the delivery of messages is started.

The Bean Provider can use the deployment descriptor to indicate whether a message-driven bean is intended for use with a topic or queue, and, if the former, whether or not topic subscriptions are to be durable.

Durable topic subscriptions, as well as queues, ensure that messages are not missed even if the EJB server is not running. Reliable applications will typically make use of queues or durable topic subscriptions rather than non-durable topic subscriptions.

If a non-durable topic subscription is used, it is the container's responsibility to make sure that the message driven bean subscription is active (i.e., that there is a message driven bean available to service the message) in order to ensure that messages are not missed as long as the EJB server is running. Messages may be missed, however, when a bean is not available to service them. This will occur, for example, if the EJB server goes down for any period of time.

Containers themselves make no actual service demands on the message-driven bean instances. The calls a container makes on a bean instance provide it with access to container services and deliver notifications issued by the container.

Since all instances of a message-driven bean are equivalent, a client message can be delivered to any available instance.

14.4.1 The required *MessageDrivenBean* interface

All message-driven beans must implement the `MessageDrivenBean` interface.

The `setMessageDrivenContext` method is called by the bean's container to associate a message-driven bean instance with its context maintained by the **container**. Typically a message-driven bean instance retains its message-driven context as part of its state.

The `ejbRemove` notification signals that the instance is in the process of being removed by the container. In the `ejbRemove` method, the instance releases the resources that it is holding.

14.4.2 The required *javax.jms.MessageListener* interface

All message-driven beans must implement the `javax.jms.MessageListener` interface.

The `onMessage` method is called by the bean's container when a message has arrived for the bean to service. The `onMessage` method contains the business logic that handles the processing of the message. The `onMessage` method has a single argument, the incoming message.

Only message-driven beans can asynchronously receive messages. Session and entity beans are not permitted to be JMS MessageListeners.

14.4.3 The *MessageDrivenContext* interface

The container provides the message-driven bean instance with a `MessageDrivenContext`. This gives the message-driven bean instance access to the instance's context maintained by the container. The `MessageDrivenContext` interface has the following methods:

- The `setRollbackOnly` method allows the instance to mark the current transaction such that the only outcome of the transaction is a rollback. Only instances of a message-driven bean with container-managed transaction demarcation can use this method.
- The `getRollbackOnly` method allows the instance to test if the current transaction has been marked for rollback. Only instances of a message-driven bean with container-managed transaction demarcation can use this method.
- The `getUserTransaction` method returns the `javax.transaction.UserTransaction` interface that the instance can use to demarcate transactions, and to obtain transaction status. Only instances of a message-driven bean with bean-managed transaction demarcation can use this method.
- The `getCallerPrincipal` method is inherited from the EJBContext interface. Message-driven bean instances must not call this method.
- The `isCallerInRole` method is inherited from the EJBContext interface. Message-driven bean instances must not call this method.
- The `getEJBHome` method is inherited from the EJBContext interface. Message-driven bean instances must not call this method.

14.4.4 Message-driven bean's *ejbCreate()* method

The container creates an instance of a message-driven bean in three steps. First, the container calls the bean class' `newInstance` method to create a new message-driven bean instance. Second, the container calls the `setMessageDrivenContext` method to pass the context object to the instance. Third, the container calls the instance's `ejbCreate` method.

Each message-driven bean class must have one `ejbCreate` method, with no arguments.

14.4.5 Serializing message-driven bean methods

A container serializes calls to each message-driven bean instance. Most containers will support many instances of a message-driven bean executing concurrently; however, each instance sees only a serialized sequence of method calls. Therefore, a message-driven bean does not have to be coded as reentrant.

The container must serialize all the container-invoked callbacks (i.e., `ejbRemove` methods), and it must serialize these callbacks with the `onMessage` method calls.

14.4.6 Concurrency of message processing

A container allows many instances of a message-driven bean class to be executing concurrently, thus allowing for the concurrent processing of a stream of messages. No guarantees are made as to the exact order in which messages are delivered to the instances of the message-driven bean class, although the container should attempt to deliver messages in order when it does not impair the concurrency of message processing. Message-driven beans should therefore be prepared to handle messages that are out of sequence: for example, the message to cancel a reservation may be delivered before the message to make the reservation.

14.4.7 Transaction context of message-driven bean methods

The `onMessage` method is invoked in the scope of a transaction determined by the transaction attribute specified in the deployment descriptor. If the bean is specified as using container-managed transaction demarcation, either the `Required` or the `NotSupported` transaction attribute must be used.[22]

When a message-driven bean using bean-managed transaction demarcation uses the `javax.transaction.UserTransaction` interface to demarcate transactions, the message receipt that causes the bean to be invoked is not part of the transaction. If the message receipt is to be part of the transaction, container-managed transaction demarcation with the `Required` transaction attribute must be used.

[22] Use of the other transaction attributes is not meaningful for message-driven beans, because there can be no pre-existing transaction context and no client to handle exceptions.

A message-driven bean's `newInstance`,`setMessageDrivenContext`, `ejbCreate`, and `ejbRemove` methods are called with an unspecified transaction context. Refer to Subsection 16.6.5 for how the Container executes methods with an unspecified transaction context.

There is never a client transaction context available when a message-driven bean is invoked because a transaction context does not flow with a JMS message.

14.4.8 Message acknowledgment

Message-driven beans should not attempt to use the JMS API for message acknowledgment. Message acknowledgment is automatically handled by the container. If the message-driven bean uses container managed transaction demarcation, message acknowledgment is handled automatically as a part of the transaction commit. If bean managed transaction demarcation is used, the message receipt cannot be part of the bean-managed transaction, and, in this case, the receipt is acknowledged by the container. If bean managed transaction demarcation is used, the Bean Provider can indicate in the `acknowedge-mode` deployment descriptor element whether JMS AUTO_ACKNOWLEDGE semantics or DUPS_OK_ACKNOWLEDGE semantics should apply. If the `acknowledge-mode` deployment descriptor element is not specified, JMS AUTO_ACKNOWLEDGE semantics are assumed.

14.4.9 Association of a message-driven bean with a destination

A message-driven bean is associated with a JMS Destination (Queue or Topic) when the bean is deployed in the container. It is the responsibility of the Deployer to associate the message-driven bean with a Queue or Topic.

The Deployer should avoid associating more than one message-driven bean with the same JMS Queue. If there are multiple JMS consumers for a queue, JMS does not define how messages are distribued between the queue receivers.

The Bean Provider may provide advice to the Deployer as to whether a message-driven bean is intended to be associated with a queue or a topic by using the `message-driven-destination` deployment descriptor element.

If the message-driven bean is intended to be used with a topic, the Bean Provider may further indicate whether a durable or non-durable subscription should be used by specifying the `subscription-durability` element. If a topic subscription is specified and the `subscription-durability` element is not specified, a non-durable subscription is assumed.

14.4.10 Dealing with exceptions

The `onMessage` method of a message-driven bean must not throw application exceptions or the `java.rmi.RemoteException`.

Message-driven beans, like other well-behaved JMS MessageListeners, should not, in general, throw `RuntimeExceptions`.

A `RuntimeException` thrown from any method of the message-driven bean class (including the `onMessage` method and the callbacks invoked by the Container) results in the transition to the "does not exist" state. Exception handling is described in detail in Chapter 17.

From the client perspective, the message consumer continues to exist. If the client continues sending messages to the Destination associated with the bean, the Container can delegate the client's messages to another instance.

14.4.11 Missed ejbRemove() calls

The Bean Provider cannot assume that the Container will always invoke the `ejbRemove()` method on a message-driven bean instance. The following scenarios result in `ejbRemove()` not being called on an instance:

- A crash of the EJB Container.
- A system exception thrown from the instance's method to the Container.

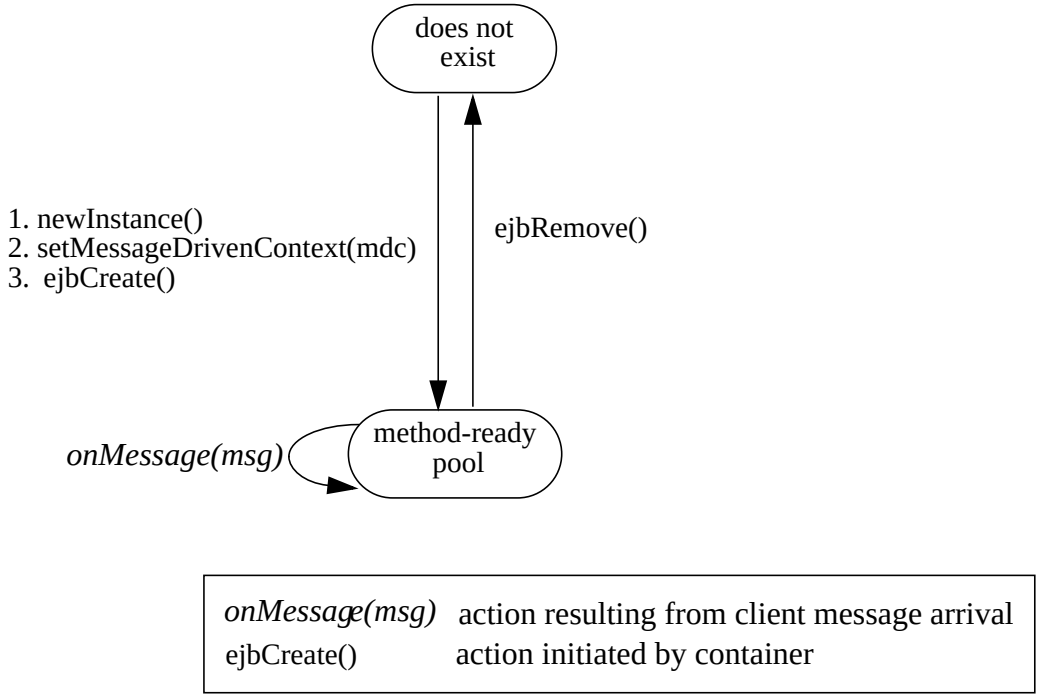
If the message-driven bean instance allocates resources in the `ejbCreate()` method and/or in the `onMessage` method, and releases normally the resources in the `ejbRemove()` method, these resources will not be automatically released in the above scenarios. The application using the message-driven bean should provide some clean up mechanism to periodically clean up the unreleased resources.

14.5 Message-driven bean state diagram

When a client sends a message to a Destination for which a message-driven bean is the consumer, the container selects one of its **method-ready** instances and invokes the instance's `onMessage` method.

The following figure illustrates the life cycle of a MESSAGE-DRIVEN bean instance.

Figure 60 Lifecycle of a MESSAGE-DRIVEN bean.



The following steps describe the lifecyle of a message-driven bean instance:

- A message-driven bean instance's life starts when the container invokes `newInstance` on the message-driven bean class to create a new instance. Next, the container calls `setMessageDrivenContext` followed by `ejbCreate` on the instance.
- The message-driven bean instance is now ready to be delivered a message sent to its Destination by any client.
- When the container no longer needs the instance (which usually happens when the container wants to reduce the number of instances in the method-ready pool), the container invokes `ejbRemove` on it. This ends the life of the message-driven bean instance.

14.5.1 Operations allowed in the methods of a message-driven bean class

Table 14 defines the methods of a message-driven bean class in which the message-driven bean instances can access the methods of the `javax.ejb.MessageDrivenContext` interface, the `java:comp/env` environment naming context, resource managers, and other enterprise beans.

If a message-driven bean instance attempts to invoke a method of the `MessageDrivenContext` interface, and the access is not allowed in Table 14, the Container must throw and log the `java.lang.IllegalStateException.`

If a bean instance attempts to access a resource manager or an enterprise bean and the access is not allowed in Table 14, the behavior is undefined by the EJB architecture.

Table 14 Operations allowed in the methods of a message-driven bean

Bean method	**Bean method can perform the following operations**	
	Container-managed transaction demarcation	**Bean-managed transaction demarcation**
constructor	-	-
setMessageDriven-Context	JNDI access to java:comp/env	JNDI access to java:comp/env
ejbCreate ejbRemove	JNDI access to java:comp/env	MessageDrivenContext methods: *getUserTransaction* JNDI access to java:comp/env
onMessage	MessageDrivenContext methods: *getRollbackOnly, setRollbackOnly* JNDI access to java:comp/env Resource manager access Enterprise bean access	MessageDrivenContext methods: *getUserTransaction* UserTransaction methods JNDI access to java:comp/env Resource manager access Enterprise bean access

Additional restrictions:

- The `getRollbackOnly` and `setRollbackOnly` methods of the `MessageDriven-Context` interface should be used only in the message-driven bean methods that execute in the context of a transaction. The Container must throw the `java.lang.IllegalState-Exception` if the methods are invoked while the instance is not associated with a transaction.

The reasons for disallowing operations in Table 14:

- Invoking the `getCallerPrincipal` and `isCallerInRole` methods is disallowed in the message-driven bean methods because the Container does not have a client security context. The Container must throw and log the `java.lang.IllegalStateException` if either of these methods is invoked.

- Invoking the `getRollbackOnly` and `setRollbackOnly` methods is disallowed in the message-driven bean methods for which the Container does not have a meaningful transaction context, and for all message-driven beans with bean-managed transaction demarcation.

- The `UserTransaction` interface is unavailable to message-driven beans with container-managed transaction demarcation.
- Invoking `getEJBHome` is disallowed in message-driven bean methods because there are no EJBHome objects for message-driven beans. The Container must throw and log the `java.lang.IllegalStateException` if this method is invoked.

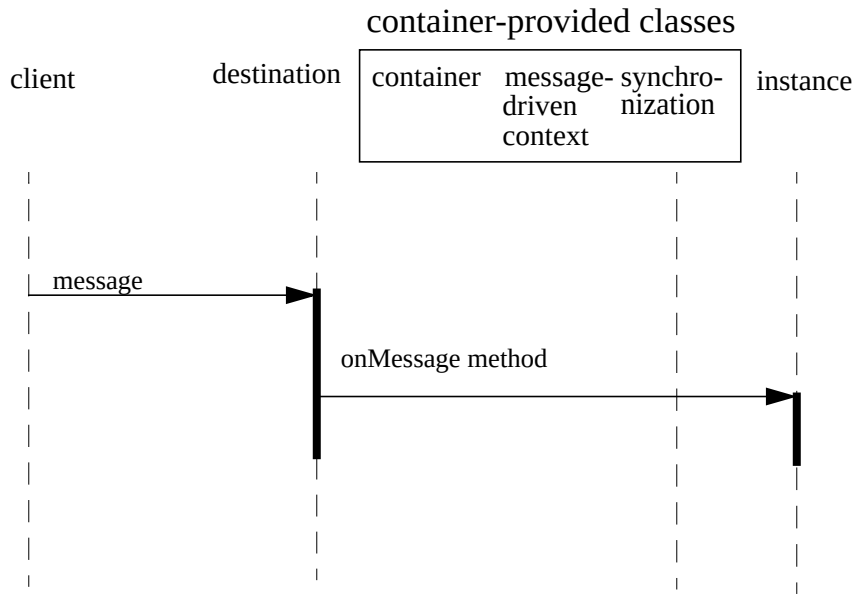
14.6 Object interaction diagrams for a MESSAGE-DRIVEN bean

This section contains object interaction diagrams that illustrate the interaction of the classes.

14.6.1 Message receipt: *onMessage* method invocation

The following diagram illustrates the invocation of the `onMessage` method.

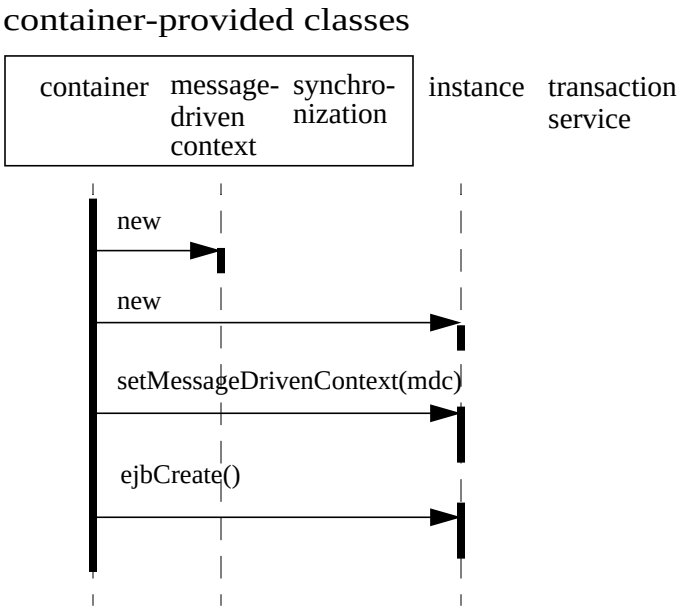
Figure 61 OID for invocation of onMessage method on MESSAGE-DRIVEN bean instance



14.6.2 Adding instance to the pool

The following diagram illustrates the sequence for a container adding an instance to the method-ready pool.

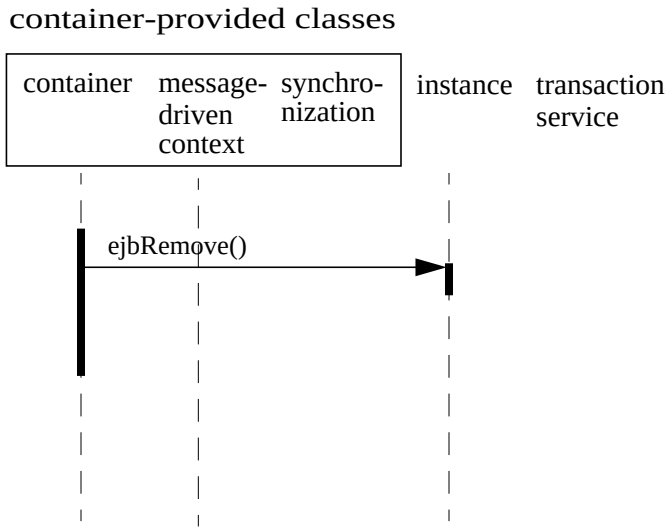
Figure 62 OID for container adding instance of a MESSAGE-DRIVEN bean to a method-ready pool



14.6.3 Removing instance from the pool

The following diagram illustrates the sequence for a container removing an instance from the method-ready pool.

Figure 63 OID for a container removing an instance of MESSAGE-DRIVEN bean from ready pool



14.7 The responsibilities of the bean provider

This section describes the responsibilities of the message-driven bean provider to ensure that a message-driven bean can be deployed in any EJB Container.

14.7.1 Classes and interfaces

The message-driven bean provider is responsible for providing the following class files:

- Message-driven bean class.

14.7.2 Message-driven bean class

The following are the requirements for the message-driven bean class:

The class must implement, directly or indirectly, the `javax.ejb.MessageDrivenBean` interface.

The class must implement, directly or indirectly, the `javax.jms.MessageListener` interface.

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 arguments. The Container uses this constructor to create instances of the message-driven bean class.

The class must not define the `finalize()` method.

The class must implement the `ejbCreate()` method.

The message-driven bean class may have superclasses and/or superinterfaces. If the message-driven bean has superclasses, the `ejbCreate` method, and the methods of the `MessageDrivenBean` and `MessageListener` interfaces may be defined in the message-driven bean class or in any of its superclasses.

The message-driven bean class is allowed to implement other methods (for example, helper methods invoked internally by the `onMessage` method) in addition to the methods required by the EJB specification.

14.7.3 *ejbCreate* method

The message-driven bean class must define one `ejbCreate()` method whose signature must follow these rules:

The method name must be `ejbCreate`.

The method must be declared as `public`.

The method must not be declared as `final` or `static`.

The return type must be `void`.

The method must have no arguments.

The throws clause must not define any application exceptions.

14.7.4 *onMessage* method

The message-driven bean class must define one `onMessage` method whose signature must follow these rules:

The method must be declared as `public`.

The method must not be declared as `final` or `static`.

The return type must be `void`.

The method must have a single argument of type `javax.jms.Message`.

The throws clause must not define any application exceptions.

14.7.5 *ejbRemove* method

The message-driven bean class must define one `ejbRemove()` method whose signature must follow these rules:

The method name must be `ejbRemove`.

The method must be declared as `public`.

The method must not be declared as `final` or `static`.

The return type must be `void`.

The method must have no arguments.

The throws clause must not define any application exceptions.

14.8 The responsibilities of the container provider

This section describes the responsibilities of the container provider to support a message-driven bean. The container provider is responsible for providing the deployment tools, and for managing the message-driven bean instances at runtime.

Because the EJB specification does not define the API between deployment tools and the container, we assume that the deployment tools are provided by the container provider. Alternatively, the deployment tools may be provided by a different vendor who uses the container vendor's specific API.

14.8.1 Generation of implementation classes

The deployment tools provided by the container are responsible for the generation of additional classes when the message-driven bean is deployed. The tools obtain the information that they need for generation of the additional classes by introspecting the classes and interfaces provided by the enterprise bean provider and by examining the message-driven bean's deployment descriptor.

The deployment tools may generate a class that mixes some container-specific code with the message-driven bean class. This code may, for example, help the container to manage the bean instances at runtime. Subclassing, delegation, and code generation can be used by the tools.

14.8.2 Deployment of message-driven beans.

The container provider must support the deployment of a message-driven bean as the consumer of a JMS queue or a durable subscription.

14.8.3 Non-reentrant instances

The container must ensure that only one thread can be executing an instance at any time.

14.8.4 Transaction scoping, security, exceptions

The container must follow the rules with respect to transaction scoping, security checking, and exception handling, as described in Chapters 16, 20, and 17.

Chapter 15 Example Message-driven Bean Scenario

This chapter describes an example development and deployment scenario of a message-driven bean. We use the scenario to explain the responsibilities of the bean provider and those of the container provider.

The classes generated by the container provider's tools in this scenario should be considered illustrative rather than prescriptive. Container providers are free to implement the contract between a message-driven bean and its container in a different way, provided that it achieves an equivalent effect (from the perspectives of the bean provider and the client-side programmer).

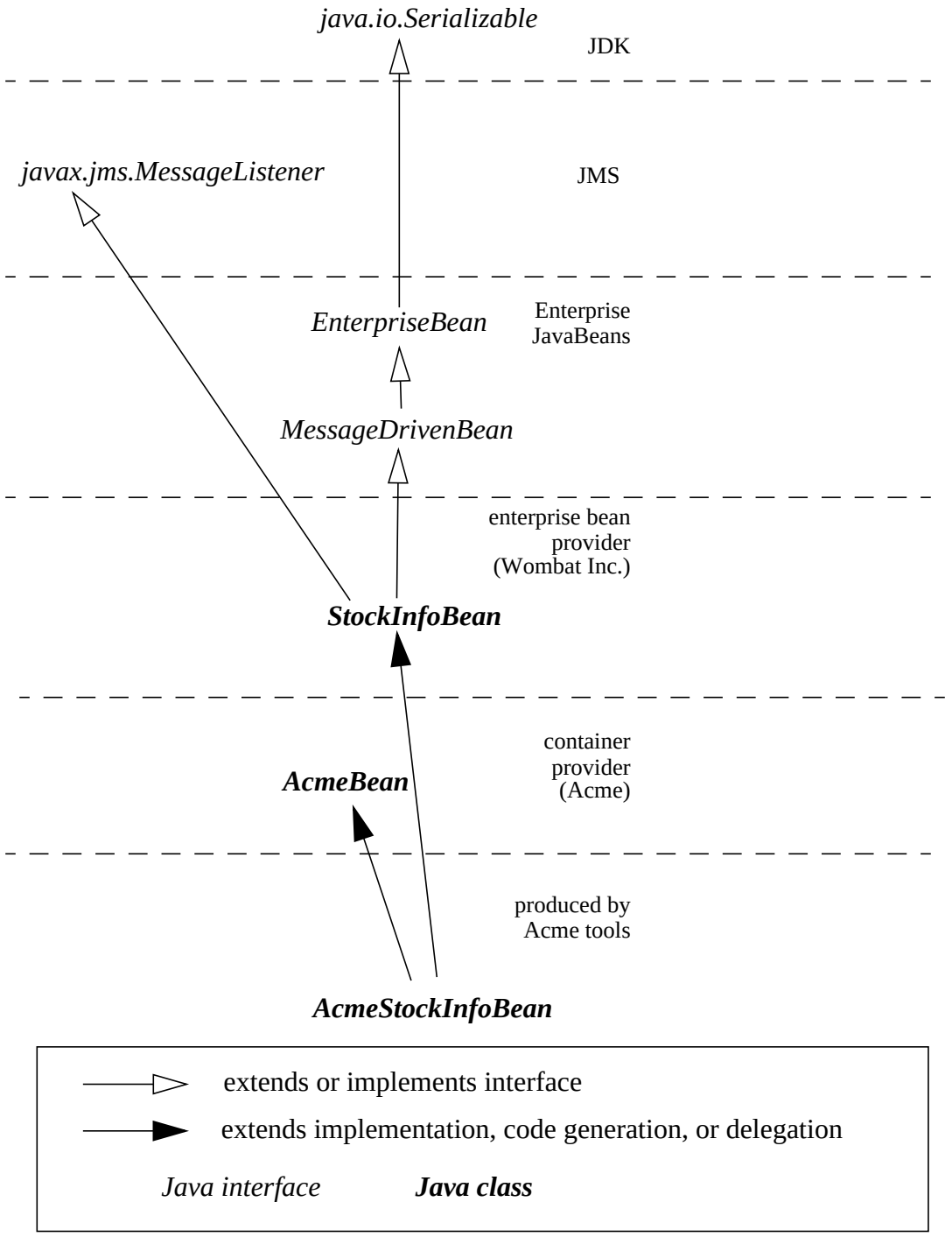
15.1 Overview

Wombat Inc. has developed the `StockInfoBean` *message-driven Bean. The StockInfoBean is deployed in a container provided by the Acme Corporation.*

15.2 Inheritance relationship

An example of the inheritance relationship between the interfaces and classes is illustrated in the following diagram:

Figure 64 Example of Inheritance Relationships Between EJB Classes



15.2.1 What the message-driven Bean provider is responsible for

Wombat Inc. is responsible for providing the following:

- *Write the business logic in the message-driven Bean class (StockInfoBean), defining the onMessage method that is invoked when the bean is to service a JMS message. The message-driven Bean must implement the javax.ejb.MessageDrivenBean and javax.jms.MessageListener interfaces, and define the ejbCreate() method invoked at an EJB object creation.*

- *Define a deployment descriptor that specifies any declarative metadata that the message-driven Bean provider wishes to pass with the Bean to the next stage of the development/deployment workflow.*

15.2.2 Classes supplied by container provider

The following classes are supplied by the container provider Acme Corp:

- *The AcmeBean class provides additional state and methods to allow Acme's container to manage its message-driven Bean instances.*

15.2.3 What the container provider is responsible for

The tools provided by Acme Corporation are responsible for the following:

- *Generate the implementation of the message-driven Bean class suitable for the Acme container (AcmeStockInfoBean). AcmeStockInfoBean includes the business logic from the StockInfoBean class mixed with the services defined in the AcmeBean class. Acme tools can use inheritance, delegation, and code generation to achieve a mix-in of the two classes.*

Many of the above classes and tools are container-specific (i.e., they reflect the way Acme Corp implemented them). Other container providers may use different mechanisms to produce their runtime classes, which will likely be different from those generated by Acme's tools.

Chapter 16 Support for Transactions

One of the key features of the Enterprise JavaBeans™ architecture is support for distributed transactions. The Enterprise JavaBeans architecture allows an application developer to write an application that atomically updates data in multiple databases which may be distributed across multiple sites. The sites may use EJB Servers from different vendors.

16.1 Overview

This section provides a brief overview of transactions and illustrates a number of transaction scenarios in EJB.

16.1.1 Transactions

Transactions are a proven technique for simplifying application programming. Transactions free the application programmer from dealing with the complex issues of failure recovery and multi-user programming. If the application programmer uses transactions, the programmer divides the application's work into units called transactions. The transactional system ensures that a unit of work either fully completes, or the work is fully rolled back. Furthermore, transactions make it possible for the programmer to design the application as if it ran in an environment that executes units of work serially.

Support for transactions is an essential component of the Enterprise JavaBeans architecture. The enterprise Bean Provider and the client application programmer are not exposed to the complexity of distributed transactions. The Bean Provider can choose between using programmatic transaction demarcation in the enterprise bean code (this style is called *bean-managed transaction demarcation*) or declarative transaction demarcation performed automatically by the EJB Container (this style is called *container-managed transaction demarcation*).

With bean-managed transaction demarcation, the enterprise bean code demarcates transactions using the `javax.transaction.UserTransaction` interface. All resource manager[23] accesses between the `UserTransaction.begin` and `UserTransaction.commit` calls are part of a transaction.

With container-managed transaction demarcation, the Container demarcates transactions per instructions provided by the Application Assembler in the deployment descriptor. These instructions, called *transaction attributes*, tell the container whether it should include the work performed by an enterprise bean method in a client's transaction, run the enterprise bean method in a new transaction started by the Container, or run the method with "no transaction" (Refer to Subsection 16.6.5 for the description of the "no transaction" case).

Regardless whether an enterprise bean uses bean-managed or container-managed transaction demarcation, the burden of implementing transaction management is on the EJB Container and Server Provider. The EJB Container and Server implement the necessary low-level transaction protocols, such as the two-phase commit protocol between a transaction manager and a database system or JMS provider, transaction context propagation, and distributed two-phase commit.

Many applications will consist of one or several enterprise beans that all use a single resource manager (typically a relational database management system). The EJB Container can make use of resource manager local transactions as an optimization technique for enterprise beans for which distributed transactions are not needed. A resource manager local transaction does not involve control or coordination by an external transaction manager. The container's use of local transactions as an optimization technique for enterprise beans with either container managed transaction demarcation or bean managed transaction demarcation is not visible to the enterprise beans. For a discussion of the use of resource manager local transactions as a container optimization strategy, refer to [9] and [12].

16.1.2 Transaction model

The Enterprise JavaBeans architecture supports flat transactions. A flat transaction cannot have any child (nested) transactions.

> *Note: The decision not to support nested transactions allows vendors of existing transaction processing and database management systems to incorporate support for Enterprise JavaBeans. If these vendors provide support for nested transactions in the future, Enterprise JavaBeans may be enhanced to take advantage of nested transactions.*

[23] The terms *resource* and *resource manager* used in this chapter refer to the resources declared in the enterprise bean's deployment descriptor using the `resource-ref` element. This includes not only database resources, but also JMS Connections. These resources are considered to be "managed" by the Container.

16.1.3 Relationship to JTA and JTS

The Java™ Transaction API (JTA) [5] is a specification of the interfaces between a transaction manager and the other parties involved in a distributed transaction processing system: the application programs, the resource managers, and the application server.

The Java Transaction Service (JTS) [6] API is a Java binding of the CORBA Object Transaction Service (OTS) 1.1 specification. JTS provides transaction interoperability using the standard IIOP protocol for transaction propagation between servers. The JTS API is intended for vendors who implement transaction processing infrastructure for enterprise middleware. For example, an EJB Server vendor may use a JTS implementation as the underlying transaction manager.

The EJB architecture does not require the EJB Container to support the JTS interfaces. The EJB architecture requires that the EJB Container support the JTA API defined in [5] and the Connector APIs defined in [12].

16.2 Sample scenarios

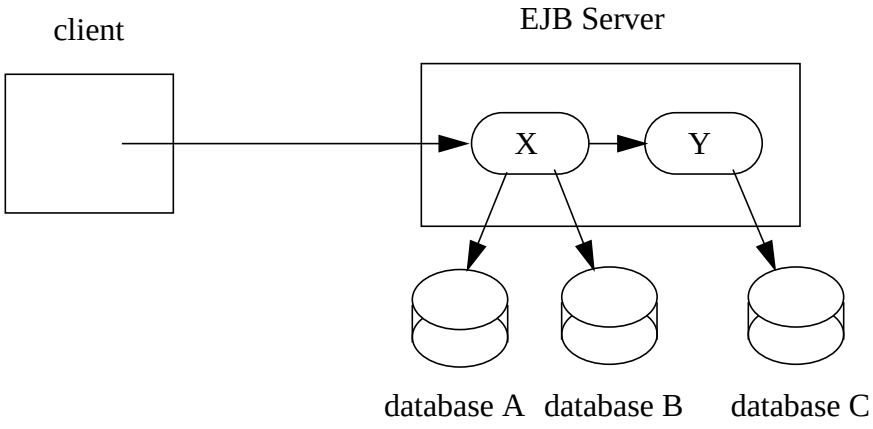
This section describes several scenarios that illustrate the distributed transaction capabilities of the Enterprise JavaBeans architecture.

16.2.1 Update of multiple databases

The Enterprise JavaBeans architecture makes it possible for an application program to update data in multiple databases in a single transaction.

In the following figure, a client invokes the enterprise Bean X. Bean X updates data using two database connections that the Deployer configured to connect with two different databases, A and B. Then X calls another enterprise Bean Y. Bean Y updates data in database C. The EJB Server ensures that the updates to databases A, B, and C are either all committed or all rolled back.

Figure 65 Updates to Simultaneous Databases



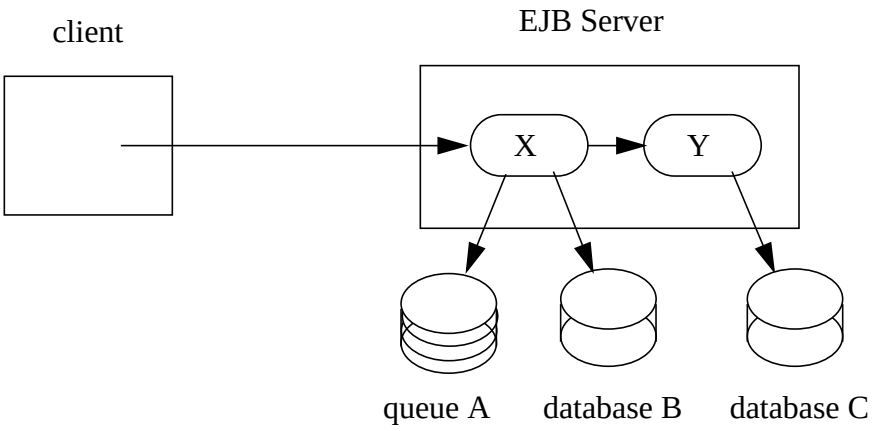
The application programmer does not have to do anything to ensure transactional semantics. The enterprise Beans X and Y perform the database updates using the standard JDBC™ API. Behind the scenes, the EJB Server enlists the database connections as part of the transaction. When the transaction commits, the EJB Server and the database systems perform a two-phase commit protocol to ensure atomic updates across all three databases.

16.2.2 Messages sent or received over JMS sessions and update of multiple databases

The Enterprise JavaBeans architecture makes it possible for an application program to send messages to or receive messages from one or more JMS Destinations and/or to update data in one or more databases in a single transaction.

In the following figure, a client invokes the enterprise Bean X. Bean X sends a message to a JMS queue A and updates data in a database B using connections that the Deployer configured to connect with a JMS provider and a database. Then X calls another enterprise Bean Y. Bean Y updates data in database C. The EJB Server ensures that the operations on A, B, and C are either all committed, or all rolled back.

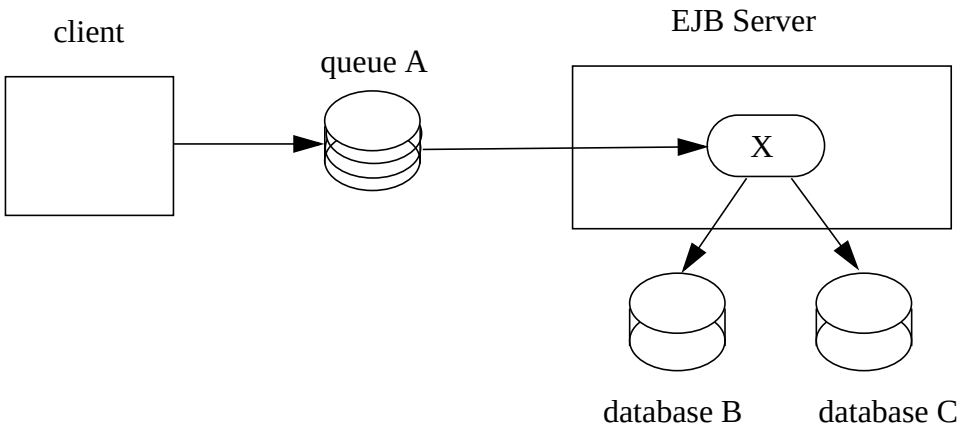
Figure 66 Message sent to JMS queue and updates to multiple databases



The application programmer does not have to do anything to ensure transactional semantics. The enterprise Beans X and Y perform the message send and database updates using the standard JMS and JDBC™ APIs. Behind the scenes, the EJB Server enlists the session on the connection to the JMS provider and the database connections as part of the transaction. When the transaction commits, the EJB Server and the messaging and database systems perform a two-phase commit protocol to ensure atomic updates across all the three resources.

In the following figure, a client sends a message to the JMS queue A serviced by the message-driven Bean X. Bean X updates data using two database connections that the Deployer configured to connect with two different databases, B and C. The EJB Server ensures that the dequeueing of the JMS message, its receipt by Bean X, and the updates to databases B and C are either all committed or all rolled back.

Figure 67 Message sent to JMS queue serviced by message-driven bean and updates to multiple databases

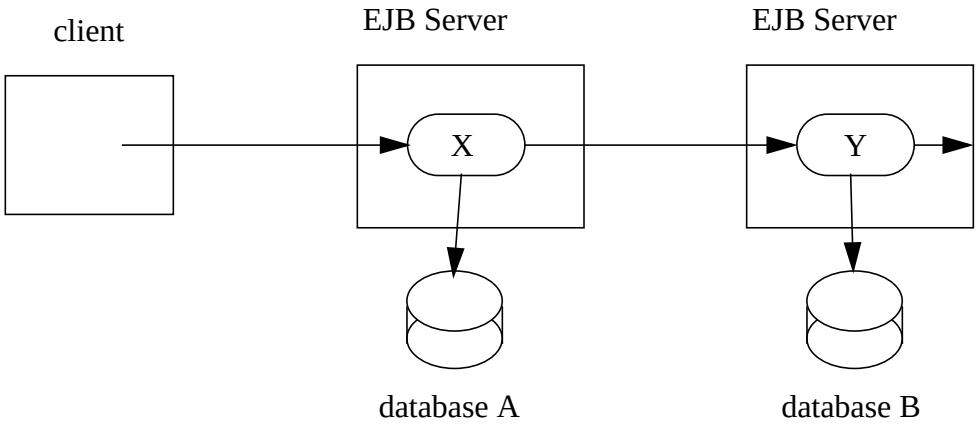


16.2.3 Update of databases via multiple EJB Servers

The Enterprise JavaBeans architecture allows updates of data at multiple sites to be performed in a single transaction.

In the following figure, a client invokes the enterprise Bean X. Bean X updates data in database A, and then calls another enterprise Bean Y that is installed in a remote EJB Server. Bean Y updates data in database B. The Enterprise JavaBeans architecture makes it possible to perform the updates to databases A and B in a single transaction.

Figure 68 Updates to Multiple Databases in Same Transaction



When X invokes Y, the two EJB Servers cooperate to propagate the transaction context from X to Y. This transaction context propagation is transparent to the application-level code.

At transaction commit time, the two EJB Servers use a distributed two-phase commit protocol (if the capability exists) to ensure the atomicity of the database updates.

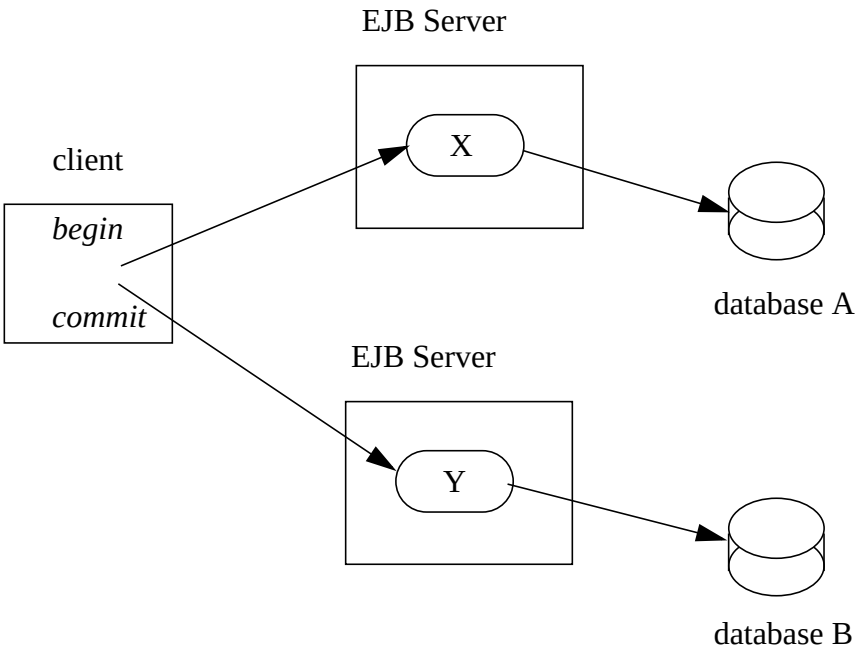
16.2.4 Client-managed demarcation

A Java client can use the `javax.transaction.UserTransaction` interface to explicitly demarcate transaction boundaries. The client program obtains the `javax.transaction.UserTransaction` interface using JNDI as defined in the Java 2, Enterprise Edition specification [9].

The EJB specification does not imply that the `javax.transaction.UserTransaction` is available to all Java clients. The Java 2, Enterprise Edition specification specifies the client environments in which the `javax.transaction.UserTransaction` interface is available.

A client program using explicit transaction demarcation may perform, via enterprise beans, atomic updates across multiple databases residing at multiple EJB Servers, as illustrated in the following figure.

Figure 69 Updates on Multiple Databases on Multiple Servers



The application programmer demarcates the transaction with `begin` *and* `commit` *calls. If the enterprise beans X and Y are configured to use a client transaction (i.e., their methods have either the Required, Mandatory, or Supports transaction attribute), the EJB Server ensures that the updates to databases A and B are made as part of the client's transaction.*

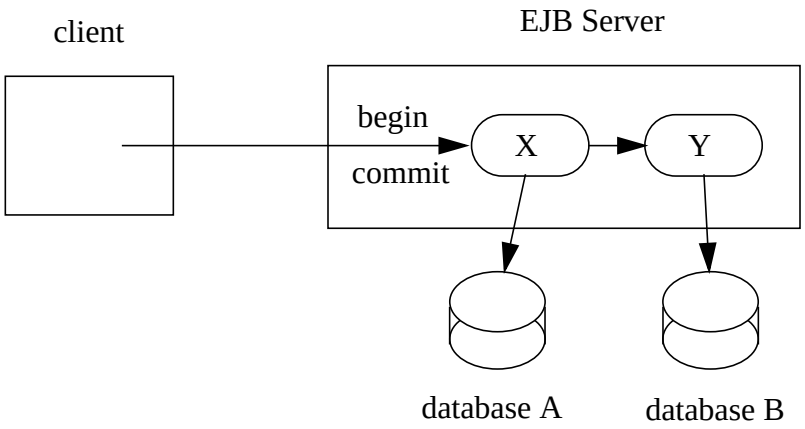
16.2.5 Container-managed demarcation

Whenever a client invokes an enterprise Bean, the container interposes on the method invocation. The interposition allows the container to control transaction demarcation declaratively through the **transaction attribute** set by the Application Assembler. (See [16.4.1] for a description of transaction attributes.)

For example, if an enterprise Bean method is configured with the `Required` transaction attribute, the container behaves as follows: If the client request is not associated with a transaction context, the Container automatically initiates a transaction whenever a client invokes an enterprise bean method that requires a transaction context. If the client request contains a transaction context, the container includes the enterprise bean method in the client transaction.

The following figure illustrates such a scenario. A non-transactional client invokes the enterprise Bean X, and the invoked method has the Required *transaction attribute. Because the message from the client does not include a transaction context, the container starts a new transaction before dispatching the remote method on X. Bean X's work is performed in the context of the transaction. When X calls other enterprise Beans (Y in our example), the work performed by the other enterprise Beans is also automatically included in the transaction (subject to the transaction attribute of the other enterprise Bean).*

Figure 70 Update of Multiple Databases from Non-Transactional Client



The container automatically commits the transaction at the time X returns a reply to the client.

If a message-driven bean is configured with the `Required` transaction attribute, the container behaves as follows: Because there is never a client transaction context available for a message-driven bean, the container automatically starts a new transaction before the dequeuing of the JMS message and, hence, before the invocation of the message-driven bean's `onMessage` method. The Container automatically enlists the resource manager associated with the arriving message and all the resource managers accessed by the `onMessage` method with the transaction.

16.3 Bean Provider's responsibilities

This section describes the Bean Provider's view of transactions and defines the Bean Provider's responsibilities.

16.3.1 Bean-managed versus container-managed transaction demarcation

When designing an enterprise bean, the Bean Provider must decide whether the enterprise bean will demarcate transactions programmatically in the business methods (bean-managed transaction demarcation), or whether the transaction demarcation is to be performed by the Container based on the *transaction attributes* in the deployment descriptor (container-managed transaction demarcation).

A Session Bean or a Message-driven Bean can be designed with bean-managed transaction demarcation or with container-managed transaction demarcation. (But it cannot be both at the same time.)

An Entity Bean must always be designed with container-managed transaction demarcation.

An enterprise bean instance can access resource managers in a transaction only in the enterprise bean's methods in which there is a transaction context available. An entity bean with container managed persistence can access its persistent state in a transaction only in the enterprise bean's methods in which there is a transaction context available. Refer to Table 2 on page 70, Table 3 on page 80, Table 4 on page 175, Table 12 on page 259, and Table 14 on page 318.

16.3.1.1 Non-transactional execution

Some enterprise beans may need to access resource managers that do not support an external transaction coordinator. The Container cannot manage the transactions for such enterprise beans in the same way that it can for the enterprise beans that access resource managers that support an external transaction coordinator.

If an enterprise bean needs to access a resource manager that does not support an external transaction coordinator, the Bean Provider should design the enterprise bean with container-managed transaction demarcation and assign the `NotSupported` transaction attribute to all the bean's methods. The EJB architecture does not specify the transactional semantics of the enterprise bean methods. See Subsection 16.6.5 for how the Container implements this case.

16.3.2 Isolation levels

Transactions not only make completion of a unit of work atomic, but they also isolate the units of work from each other, provided that the system allows concurrent execution of multiple units of work.

The *isolation level* describes the degree to which the access to a resource manager by a transaction is isolated from the access to the resource manager by other concurrently executing transactions.

The following are guidelines for managing isolation levels in enterprise beans.

- The API for managing an isolation level is resource-manager specific. (Therefore, the EJB architecture does not define an API for managing isolation level.)
- If an enterprise bean uses multiple resource managers, the Bean Provider may specify the same or different isolation level for each resource manager. This means, for example, that if an enterprise bean accesses multiple resource managers in a transaction, access to each resource manager may be associated with a different isolation level.
- The Bean Provider must take care when setting an isolation level. Most resource managers require that all accesses to the resource manager within a transaction are done with the same isolation level. An attempt to change the isolation level in the middle of a transaction may cause undesirable behavior, such as an implicit sync point (a commit of the changes done so far).
- For session beans and message-driven beans with bean-managed transaction demarcation, the Bean Provider can specify the desirable isolation level programmatically in the enterprise bean's methods, using the resource-manager specific API. For example, the Bean Provider can use the `java.sql.Connection.setTransactionIsolation(...)` method to set the appropriate isolation level for database access.
- For entity beans with container-managed persistence, transaction isolation is managed by the data access classes that are generated by the persistence manager provider's tools. The tools must ensure that the management of the isolation levels performed by the data access classes will not result in conflicting isolation level requests for a resource manager within a transaction.
- Additional care must be taken if multiple enterprise beans access the same resource manager in the same transaction. Conflicts in the requested isolation levels must be avoided.

16.3.3 Enterprise beans using bean-managed transaction demarcation

This subsection describes the requirements for the Bean Provider of an enterprise bean with bean-managed transaction demarcation.

The enterprise bean with bean-managed transaction demarcation must be a Session bean or a Message-driven bean.

An instance that starts a transaction must complete the transaction before it starts a new transaction.

The Bean Provider uses the `UserTransaction` interface to demarcate transactions. All updates to the resource managers between the `UserTransaction.begin()` and `UserTransaction.commit()` methods are performed in a transaction. While an instance is in a transaction, the instance must not attempt to use the resource-manager specific transaction demarcation API (e.g. it must not invoke the `commit()` or `rollback()` method on the `java.sql.Connection` interface or on the `javax.jms.Session` interface).

A stateful Session Bean instance may, but is not required to, commit a started transaction before a business method returns. If a transaction has not been completed by the end of a business method, the Container retains the association between the transaction and the instance across multiple client calls until the instance eventually completes the transaction.

The bean-managed transaction demarcation programming model presented to the programmer of a stateful Session Bean is natural because it is the same as that used by a stand-alone Java application.

A stateless session bean instance must commit a transaction before a business method returns.

A message-driven bean instance must commit a transaction before the `onMessage` method returns.

The following example illustrates a business method that performs a transaction involving two database connections.

```
public class MySessionEJB implements SessionBean {
     EJBContext ejbContext;

     public void someMethod(...) {
          javax.transaction.UserTransaction ut;
          javax.sql.DataSource ds1;
          javax.sql.DataSource ds2;
          java.sql.Connection con1;
          java.sql.Connection con2;
          java.sql.Statement stmt1;
          java.sql.Statement stmt2;

          InitialContext initCtx = new InitialContext();

          // obtain con1 object and set it up for transactions
          ds1 = (javax.sql.DataSource)
                    initCtx.lookup("java:comp/env/jdbc/Database1");
          con1 = ds1.getConnection();

          stmt1 = con1.createStatement();

          // obtain con2 object and set it up for transactions
          ds2 = (javax.sql.DataSource)
                    initCtx.lookup("java:comp/env/jdbc/Database2");
          con2 = ds2.getConnection();

          stmt2 = con2.createStatement();

          //
          // Now do a transaction that involves con1 and con2.
          //
          ut = ejbContext.getUserTransaction();

          // start the transaction
          ut.begin();

          // Do some updates to both con1 and con2. The Container
          // automatically enlists con1 and con2 with the transaction.
          stmt1.executeQuery(...);
          stmt1.executeUpdate(...);
          stmt2.executeQuery(...);
          stmt2.executeUpdate(...);
          stmt1.executeUpdate(...);
          stmt2.executeUpdate(...);

          // commit the transaction
          ut.commit();

          // release connections
          stmt1.close();
          stmt2.close();
          con1.close();
          con2.close();
     }
     ...
}
```

The following example illustrates a business method that performs a transaction involving both a database connection and a JMS connection.

```
public class MySessionEJB implements SessionBean {
    EJBContext ejbContext;

    public void someMethod(...) {
        javax.transaction.UserTransaction ut;
        javax.sql.DataSource ds;
        java.sql.Connection dcon;
        java.sql.Statement stmt;
        javax.jms.QueueConnectionFactory qcf;
        javax.jms.QueueConnection qcon;
        javax.jms.Queue q;
        javax.jms.QueueSession qsession;
        javax.jms.QueueSender qsender;
        javax.jms.Message message;

        InitialContext initCtx = new InitialContext();

        // obtain db conn object and set it up for transactions

        ds = (javax.sql.DataSource)
                 initCtx.lookup("java:comp/env/jdbc/Database");
        dcon = ds.getConnection();

        stmt = dcon.createStatement();

        // obtain jms conn object and set up session for transactions
        qcf = (javax.jms.QueueConnectionFactory)
                 initCtx.lookup("java:comp/env/jms/qConnFactory");
        qcon = qcf.createQueueConnection();
        qsession = qcon.createQueueSession(true,0);
        q = (javax.jms.Queue)
                 initCtx.lookup("java:comp/env/jms/jmsQueue");
        qsender = qsession.createSender(q);
        message = qsession.createTextMessage();
        message.setText("some message");

        //
        // Now do a transaction that involves the two connections.
        //
        ut = ejbContext.getUserTransaction();

        // start the transaction
        ut.begin();

        // Do database updates and send message. The Container
        // automatically enlists dcon and qsession with the
        // transaction.
        stmt.executeQuery(...);
        stmt.executeUpdate(...);
        stmt.executeUpdate(...);
        qsender.send(message);

        // commit the transaction
        ut.commit();
```

```
            // release connections
            stmt.close();
            qsender.close();
            qsession.close();
            dcon.close();
            qcon.close();
        }
        ...
    }
```

The following example illustrates a stateful Session Bean that retains a transaction across three client calls, invoked in the following order: *method1*, *method2*, and *method3*.

```
public class MySessionEJB implements SessionBean {
     EJBContext ejbContext;
     javax.sql.DataSource ds1;
     javax.sql.DataSource ds2;
     java.sql.Connection con1;
     java.sql.Connection con2;

     public void method1(...) {
          java.sql.Statement stmt;

          InitialContext initCtx = new InitialContext();

          // obtain user transaction interface
          ut = ejbContext.getUserTransaction();

          // start a transaction
          ut.begin();

          // make some updates on con1
          ds1 = (javax.sql.DataSource)
                    initCtx.lookup("java:comp/env/jdbc/Database1");
          con1 = ds1.getConnection();
          stmt = con1.createStatement();
          stmt.executeUpdate(...);
          stmt.executeUpdate(...);

          //
          // The Container retains the transaction associated with the
          // instance to the next client call (which is method2(...)).
     }

     public void method2(...) {
          java.sql.Statement stmt;

          InitialContext initCtx = new InitialContext();

          // make some updates on con2
          ds2 = (javax.sql.DataSource)
                    initCtx.lookup("java:comp/env/jdbc/Database2");
          con2 = ds2.getConnection();
          stmt = con2.createStatement();
          stmt.executeUpdate(...);
          stmt.executeUpdate(...);

          // The Container retains the transaction associated with the
          // instance to the next client call (which is method3(...)).
     }

     public void method3(...) {
          java.sql.Statement stmt;

          // obtain user transaction interface
          ut = ejbContext.getUserTransaction();

          // make some more updates on con1 and con2
          stmt = con1.createStatement();
```

```
            stmt.executeUpdate(...);
            stmt = con2.createStatement();
            stmt.executeUpdate(...);

            // commit the transaction
            ut.commit();

            // release connections
            stmt.close();
            con1.close();
            con2.close();
        }
        ...
    }
```

It is possible for an enterprise bean to open and close a database connection in each business method (rather than hold the connection open until the end of transaction). In the following example, if the client executes the sequence of methods (*method1*, *method2*, *method2*, *method2*, and *method3)*, all the database updates done by the multiple invocations of *method2* are performed in the scope of the same transaction, which is the transaction started in *method1* and committed in *method3*.

```
public class MySessionEJB implements SessionBean {
     EJBContext ejbContext;
     InitialContext initCtx;

     public void method1(...) {
          java.sql.Statement stmt;

          // obtain user transaction interface
          ut = ejbContext.getUserTransaction();

          // start a transaction
          ut.begin();
     }

     public void method2(...) {
          javax.sql.DataSource ds;
          java.sql.Connection con;
          java.sql.Statement stmt;

          // open connection
          ds = (javax.sql.DataSource)
                    initCtx.lookup("java:comp/env/jdbc/Database");
          con = ds.getConnection();

          // make some updates on con
          stmt = con.createStatement();
          stmt.executeUpdate(...);
          stmt.executeUpdate(...);

          // close the connection
          stmt.close();
          con.close();
     }

     public void method3(...) {
          // obtain user transaction interface
          ut = ejbContext.getUserTransaction();

          // commit the transaction
          ut.commit();
     }
     ...
}
```

16.3.3.1 `getRollbackOnly()` and `setRollbackOnly()` method

An enterprise bean with bean-managed transaction demarcation must not use the `getRollbackOnly()` and `setRollbackOnly()` methods of the `EJBContext` interface.

An enterprise bean with bean-managed transaction demarcation has no need to use these methods, because of the following reasons:

- An enterprise bean with bean-managed transaction demarcation can obtain the status of a transaction by using the `getStatus()` method of the `javax.transaction.UserTransaction` interface.
- An enterprise bean with bean-managed transaction demarcation can rollback a transaction using the `rollback()` method of the `javax.transaction.UserTransaction` interface.

16.3.4 Enterprise beans using container-managed transaction demarcation

This subsection describes the requirements for the Bean Provider of an enterprise bean using container-managed transaction demarcation.

The enterprise bean's business methods or `onMessage` method must not use any resource-manager specific transaction management methods that would interfere with the Container's demarcation of transaction boundaries. For example, the enterprise bean methods must not use the following methods of the `java.sql.Connection` interface: `commit()`, `setAutoCommit(...)`, and `rollback()` or the following methods of the `javax.jms.Session` interface: `commit()` and `rollback()`.

The enterprise bean's business methods or `onMessage` method must not attempt to obtain or use the `javax.transaction.UserTransaction` interface.

The following is an example of a business method in an enterprise bean with container-managed transaction demarcation. The business method updates two databases using JDBC™ connections. The Container provides transaction demarcation per the Application Assembler's instructions.

```
public class MySessionEJB implements SessionBean {
    EJBContext ejbContext;

    public void someMethod(...) {
        java.sql.Connection con1;
        java.sql.Connection con2;
        java.sql.Statement stmt1;
        java.sql.Statement stmt2;

        // obtain con1 and con2 connection objects
        con1 = ...;
        con2 = ...;

        stmt1 = con1.createStatement();
        stmt2 = con2.createStatement();

        //
        // Perform some updates on con1 and con2. The Container
        // automatically enlists con1 and con2 with the container-
        // managed transaction.
        //
        stmt1.executeQuery(...);
        stmt1.executeUpdate(...);

        stmt2.executeQuery(...);
        stmt2.executeUpdate(...);

        stmt1.executeUpdate(...);
        stmt2.executeUpdate(...);

        // release connections
        con1.close();
        con2.close();
    }
    ...
}
```

16.3.4.1 `javax.ejb.SessionSynchronization` interface

A stateful Session Bean with container-managed transaction demarcation can optionally implement the `javax.ejb.SessionSynchronization` interface. The use of the `SessionSynchronization` interface is described in Subsection 6.5.3.

16.3.4.2 `javax.ejb.EJBContext.setRollbackOnly()` method

An enterprise bean with container-managed transaction demarcation can use the `setRollbackOnly()` method of its `EJBContext` object to mark the transaction such that the transaction can never commit. Typically, an enterprise bean marks a transaction for rollback to protect data integrity before throwing an application exception, because application exceptions do not automatically cause the Container to rollback the transaction.

For example, an AccountTransfer bean which debits one account and credits another account could mark a transaction for rollback if it successfully performs the debit operation, but encounters a failure during the credit operation.

16.3.4.3 `javax.ejb.EJBContext.getRollbackOnly()` method

An enterprise bean with container-managed transaction demarcation can use the `getRollbackOnly()` method of its `EJBContext` object to test if the current transaction has been marked for rollback. The transaction might have been marked for rollback by the enterprise bean itself, by other enterprise beans, or by other components (outside of the EJB specification scope) of the transaction processing infrastructure.

16.3.5 Use of JMS APIs in transactions

The Bean Provider must not make use of the JMS request/reply paradigm (sending of a JMS message, followed by the synchronous receipt of a reply to that message) within a single transaction. Because a JMS message is not delivered to its final destination until the transaction commits, the receipt of the reply within the same transaction will never take place.

Because the container manages the transactional enlistment of JMS sessions on behalf of a bean, the parameters of the `createQueueSession(boolean transacted, int acknowledgeMode)` and `createTopicSession(boolean transacted, int acknowledgeMode)` methods are ignored. It is recommended that the Bean Provider specify that a session is transacted, but provide `0` for the value of the acknowledgment mode.

The Bean Provider should not use the JMS `acknowledge()` method either within a transaction or within an unspecified transaction context. Message acknowledgment in an unspecified transaction context is handled by the container. Section 16.6.5 describes some of the techniques that the container can use for the implementation of a method invocation with an unspecified transaction context.

16.3.6 Declaration in deployment descriptor

The Bean Provider of a Session Bean or a Message-driven Bean must use the `transaction-type` element to declare whether the Session Bean or Message-driven Bean is of the bean-managed or container-managed transaction demarcation type. (See Chapter 21 for information about the deployment descriptor.)

The transaction-type element is not supported for Entity beans because all Entity beans must use container-managed transaction demarcation.

The Bean Provider of an enterprise bean with container-managed transaction demarcation may optionally specify the transaction attributes for the enterprise bean's methods. See Subsection 16.4.1.

16.4 Application Assembler's responsibilities

This section describes the view and responsibilities of the Application Assembler.

There is no mechanism for an Application Assembler to affect enterprise beans with bean-managed transaction demarcation. The Application Assembler must not define transaction attributes for an enterprise bean with bean-managed transaction demarcation.

The Application Assembler can use the *transaction attribute* mechanism described below to manage transaction demarcation for enterprise beans using container-managed transaction demarcation.

16.4.1 Transaction attributes

Note: The transaction attributes may be specified either by the Bean Provider or by the Application Assembler.

A transaction attribute is a value associated with a method of a session or entity bean's remote or home interface or with the `onMessage` method of a message-driven bean. The transaction attribute specifies how the Container must manage transactions for a method when a client invokes the method via the enterprise bean's home or remote interface or when the method is invoked as the result of the arrival of a JMS message.

The transaction attribute must be specified for the following methods:

- For a session bean, the transaction attributes must be specified for the methods defined in the bean's remote interface and all the direct and indirect superinterfaces of the remote interface, excluding the methods of the `javax.ejb.EJBObject` interface. Transaction attributes must not be specified for the methods of a session bean's home interface.

- For an entity bean, the transaction attributes must be specified for the methods defined in the bean's remote interface and all the direct and indirect superinterfaces of the remote interface, excluding the `getEJBHome`, `getHandle`, `getPrimaryKey`, and `isIdentical` methods; and for the methods defined in the bean's home interface and all the direct and indirect superinterfaces of the home interface, excluding the `getEJBMetaData` and `getHomeHandle` methods.

- For a message-driven bean, the transaction attribute must be specified for the bean's `onMessage` method.

Providing the transaction attributes for an enterprise bean is an optional requirement for the Application Assembler, because, for a given enterprise bean, the Application Assembler must either specify a value of the transaction attribute for **all** the methods for which a transaction attribute must be specified, or the Assembler must specify **none**. If the transaction attributes are not specified for the methods of an enterprise bean, the Deployer will have to specify them.

Enterprise JavaBeans defines the following values for the transaction attribute:

- `NotSupported`
- `Required`
- `Supports`

- `RequiresNew`
- `Mandatory`
- `Never`

Refer to Subsection 16.6.2 for the specification of how the value of the transaction attribute affects the transaction management performed by the Container.

For message-driven beans, only the `Required` and `NotSupported` transaction attributes may be used.

For entity beans that use EJB 2.0 container managed persistence, only the `Required`, `RequiresNew`, or `Mandatory` transaction attributes may be used for the methods defined in the bean's remote interface and all the direct and indirect superinterfaces of the remote interface, excluding the `getEJBHome`, `getHandle`, `getPrimaryKey`, and `isIdentical` methods; and for the methods defined in the bean's home interface and all the direct and indirect superinterfaces of the home interface, excluding the `getEJBMetaData` and `getHomeHandle` methods.

If an enterprise bean implements the `javax.ejb.SessionSynchronization` interface, the Application Assembler can specify only the following values for the transaction attributes of the bean's methods: `Required`, `RequiresNew`, or `Mandatory`.

The above restriction is necessary to ensure that the enterprise bean is invoked only in a transaction. If the bean were invoked without a transaction, the Container would not be able to send the transaction synchronization calls.

The tools used by the Application Assembler can determine if the bean implements the `javax.ejb.SessionSynchronization` *interface, for example, by using the Java reflection API on the enterprise bean's class.*

The following is the description of the deployment descriptor rules that the Application Assembler uses to specify transaction attributes for the methods of the session and entity beans' remote and home interfaces and message-driven beans' `onMessage` methods. (See Section 21.5 for the complete syntax of the deployment descriptor.)

The Application Assembler uses the `container-transaction` elements to define the transaction attributes for the methods of session and entity bean remote and home interfaces and for the `onMessage` methods of message-driven beans. Each `container-transaction` element consists of a list of one or more `method` elements, and the `trans-attribute` element. The `container-transaction` element specifies that all the listed methods are assigned the specified transaction attribute value. It is required that all the methods specified in a single `container-transaction` element be methods of the same enterprise bean.

The `method` element uses the `ejb-name`, `method-name`, and `method-params` elements to denote one or more methods of an enterprise bean's home and remote interfaces. There are three legal styles of composing the `method` element:

Style 1:

```
<method>
      <ejb-name>EJBNAME</ejb-name>
      <method-name>*</method-name>
</method>
```

This style is used to specify a default value of the transaction attribute for the methods for which there is no Style 2 or Style 3 element specified. There must be at most one `container-transaction` element that uses the Style 1 `method` element for a given enterprise bean.

Style 2:

```
<method>
      <ejb-name>EJBNAME</ejb-name>
      <method-name>METHOD</method-name>
</method>
```

This style is used for referring to a specified method of the remote or home interface of the specified enterprise bean. If there are multiple methods with the same overloaded name, this style refers to all the methods with the same name. There must be at most one `container-transaction` element that uses the Style 2 `method` element for a given method name. If there is also a `container-transaction` element that uses Style 1 element for the same bean, the value specified by the Style 2 element takes precedence.

Style 3:

```
<method>
     <ejb-name>EJBNAME</ejb-name>
     <method-name>METHOD</method-name>
     <method-params>
          <method-param>PARAMETER_1</method-param>
          ...
          <method-param>PARAMETER_N</method-param>
     </method-params>
</method>
```

This style is used to refer to a single method within a set of methods with an overloaded name. The method must be one defined in the remote or home interface of the specified enterprise bean. If there is also a `container-transaction` element that uses the Style 2 element for the method name, or the Style 1 element for the bean, the value specified by the Style 3 element takes precedence.

The optional `method-intf` element can be used to differentiate between methods with the same name and signature that are defined in both the remote and home interfaces.

The following is an example of the specification of the transaction attributes in the deployment descriptor. The `updatePhoneNumber` method of the `EmployeeRecord` enterprise bean is assigned the transaction attribute `Mandatory`; all other methods of the `EmployeeRecord` bean are assigned the attribute `Required`. All the methods of the enterprise bean `AardvarkPayroll` are assigned the attribute `RequiresNew`.

```
<ejb-jar>
     ...
     <assembly-descriptor>
          ...
          <container-transaction>
               <method>
                    <ejb-name>EmployeeRecord</ejb-name>
                    <method-name>*</method-name>
               </method>
               <trans-attribute>Required</trans-attribute>
          </container-transaction>

          <container-transaction>
               <method>
                    <ejb-name>EmployeeRecord</ejb-name>
                    <method-name>updatePhoneNumber</method-name>
               </method>
               <trans-attribute>Mandatory</trans-attribute>
          </container-transaction>

          <container-transaction>
               <method>
                    <ejb-name>AardvarkPayroll</ejb-name>
                    <method-name>*</method-name>
               </method>
               <trans-attribute>RequiresNew</trans-attribute>
          </container-transaction>
     </assembly-descriptor>
</ejb-jar>
```

16.5 Deployer's responsibilities

The Deployer is responsible for ensuring that the methods of the deployed enterprise beans with container-managed transaction demarcation have been assigned a transaction attribute. If the transaction attributes have not been assigned previously by the Assembler, they must be assigned by the Deployer.

16.6 Container Provider responsibilities

This section defines the responsibilities of the Container Provider.

Every client method invocation on a session or entity bean object via the bean's remote and home interface and every invocation of the `onMessage` method on a message-driven bean is interposed by the Container, and every connection to a resource manager used by an enterprise bean is obtained via the Container. This managed execution environment allows the Container to affect the enterprise bean's transaction management.

This does not imply that the Container must interpose on every resource manager access performed by the enterprise bean. Typically, the Container interposes only on the resource manager connection factory (e.g. a JDBC data source) JNDI look up by registering the container-specific implementation of the resource manager connection factory object. The resource manager connection factory object allows the Container to obtain the javax.transaction.xa.XAResource interface as described in the JTA specification and pass it to the transaction manager. After the set up is done, the enterprise bean communicates with the resource manager without going through the Container.

16.6.1 Bean-managed transaction demarcation

This subsection defines the Container's responsibilities for the transaction management of enterprise beans with bean-managed transaction demarcation.

Note that only Session and Message-driven beans can be used with bean-managed transaction demarcation. A Bean Provider is not allowed to provide an Entity bean with bean-managed transaction demarcation.

The Container must manage client invocations to an enterprise bean instance with bean-managed transaction demarcation as follows. When a client invokes a business method via the enterprise bean's remote or home interface, the Container suspends any transaction that may be associated with the client request. If there is a transaction associated with the instance (this would happen if the instance started the transaction in some previous business method), the Container associates the method execution with this transaction.

The Container must make the `javax.transaction.UserTransaction` interface available to the enterprise bean's business method or `onMessage` method via the `javax.ejb.EJBContext` interface and under the environment entry `java:comp/UserTransaction`. When an instance uses the `javax.transaction.UserTransaction` interface to demarcate a transaction, the Container must enlist all the resource managers used by the instance between the `begin()` and `commit()`—or `rollback()`— methods with the transaction. When the instance attempts to commit the transaction, the Container is responsible for the global coordination of the transaction commit[24].

In the case of a *stateful* session bean, it is possible that the business method that started a transaction completes without committing or rolling back the transaction. In such a case, the Container must retain the association between the transaction and the instance across multiple client calls until the instance commits or rolls back the transaction. When the client invokes the next business method, the Container must invoke the business method in this transaction context.

[24] The Container typically relies on a transaction manager that is part of the EJB Server to perform the two-phase commit across all the enlisted resource managers. If only a single resource manager is involved in the transaction and the deployment descriptor indicates that connection sharing may be used, the Container may use the local transaction optimization. See [9] and [12] for further discussion.

If a *stateless* session bean instance starts a transaction in a business method, it must commit the transaction before the business method returns. The Container must detect the case in which a transaction was started, but not completed, in the business method, and handle it as follows:

- Log this as an application error to alert the system administrator.
- Roll back the started transaction.
- Discard the instance of the session bean.
- Throw the `java.rmi.RemoteException` to the client.

If a message-driven bean instance starts a transaction in the `onMessage` method, it must commit the transaction before the `onMessage` method returns. The Container must detect the case in which a transaction was started, but not completed, in the `onMessage` method, and handle it as follows:

- Log this as an application error to alert the system administrator.
- Roll back the started transaction.
- Discard the instance of the message-driven bean.

The actions performed by the Container for an instance with bean-managed transaction are summarized by the following table. T1 is a transaction associated with a client request, T2 is a transaction that is currently associated with the instance (i.e. a transaction that was started but not completed by a previous business method).

Table 15 Container's actions for methods of beans with bean-managed transaction

Client's transaction	Transaction currently associated with instance	Transaction associated with the method
none	none	none
T1	none	none
none	T2	T2
T1	T2	T2

The following items describe each entry in the table:

- If the client request is not associated with a transaction and the instance is not associated with a transaction, or if the bean is a message-driven bean, the container invokes the instance with an unspecified transaction context.
- If the client is associated with a transaction T1, and the instance is not associated with a transaction, the container suspends the client's transaction association and invokes the method with

an unspecified transaction context. The container resumes the client's transaction association (T1) when the method completes. This case can never happen for a Message-driven Bean.

- If the client request is not associated with a transaction and the instance is already associated with a transaction T2, the container invokes the instance with the transaction that is associated with the instance (T2). This case can never happen for a stateless Session Bean or a Message-driven Bean.

- If the client is associated with a transaction T1, and the instance is already associated with a transaction T2, the container suspends the client's transaction association and invokes the method with the transaction context that is associated with the instance (T2). The container resumes the client's transaction association (T1) when the method completes. This case can never happen for a stateless Session Bean or a Message-driven Bean.

The Container must allow the enterprise bean instance to serially perform several transactions in a method.

When an instance attempts to start a transaction using the `begin()` method of the `javax.transaction.UserTransaction` interface while the instance has not committed the previous transaction, the Container must throw the `javax.transaction.NotSupportedException` in the `begin()` method.

The Container must throw the `java.lang.IllegalStateException` if an instance of a bean with bean-managed transaction demarcation attempts to invoke the `setRollbackOnly()` or `getRollbackOnly()` method of the `javax.ejb.EJBContext` interface.

16.6.2 Container-managed transaction demarcation for Session and Entity Beans

The Container is responsible for providing the transaction demarcation for the session and entity beans that the Bean Provider declared with container-managed transaction demarcation. For these enterprise beans, the Container must demarcate transactions as specified in the deployment descriptor by the Application Assembler. (See Chapter 21 for more information about the deployment descriptor.)

The following subsections define the responsibilities of the Container for managing the invocation of an enterprise bean business method when the method is invoked via the enterprise bean's home or remote interface. The Container's responsibilities depend on the value of the transaction attribute.

16.6.2.1 `NotSupported`

The Container invokes an enterprise Bean method whose transaction attribute is set to `NotSupported` with an unspecified transaction context.

If a client calls with a transaction context, the container suspends the association of the transaction context with the current thread before invoking the enterprise bean's business method. The container resumes the suspended association when the business method has completed. The suspended transaction context of the client is not passed to the resource managers or other enterprise Bean objects that are invoked from the business method.

If the business method invokes other enterprise beans, the Container passes no transaction context with the invocation.

Refer to Subsection 16.6.5 for more details of how the Container can implement this case.

16.6.2.2 Required

The Container must invoke an enterprise Bean method whose transaction attribute is set to `Required` with a valid transaction context.

If a client invokes the enterprise Bean's method while the client is associated with a transaction context, the container invokes the enterprise Bean's method in the client's transaction context.

If the client invokes the enterprise Bean's method while the client is not associated with a transaction context, the container automatically starts a new transaction before delegating a method call to the enterprise Bean business method. The Container automatically enlists all the resource managers accessed by the business method with the transaction. If the business method invokes other enterprise beans, the Container passes the transaction context with the invocation. The Container attempts to commit the transaction when the business method has completed. The container performs the commit protocol before the method result is sent to the client.

16.6.2.3 Supports

The Container invokes an enterprise Bean method whose transaction attribute is set to `Supports` as follows.

- If the client calls with a transaction context, the Container performs the same steps as described in the `Required` case.
- If the client calls without a transaction context, the Container performs the same steps as described in the `NotSupported` case.

The Supports *transaction attribute must be used with caution. This is because of the different transactional semantics provided by the two possible modes of execution. Only the enterprise beans that will execute correctly in both modes should use the* Supports *transaction attribute.*

16.6.2.4 RequiresNew

The Container must invoke an enterprise Bean method whose transaction attribute is set to `RequiresNew` with a new transaction context.

If the client invokes the enterprise Bean's method while the client is not associated with a transaction context, the container automatically starts a new transaction before delegating a method call to the enterprise Bean business method. The Container automatically enlists all the resource managers accessed by the business method with the transaction. If the business method invokes other enterprise beans, the Container passes the transaction context with the invocation. The Container attempts to commit the transaction when the business method has completed. The container performs the commit protocol before the method result is sent to the client.

If a client calls with a transaction context, the container suspends the association of the transaction context with the current thread before starting the new transaction and invoking the business method. The container resumes the suspended transaction association after the business method and the new transaction have been completed.

16.6.2.5 `Mandatory`

The Container must invoke an enterprise Bean method whose transaction attribute is set to `Mandatory` in a client's transaction context. The client is required to call with a transaction context.

- If the client calls with a transaction context, the Container performs the same steps as described in the `Required` case.
- If the client calls without a transaction context, the Container throws the `javax.transaction.TransactionRequiredException` exception.

16.6.2.6 `Never`

The Container invokes an enterprise Bean method whose transaction attribute is set to `Never` without a transaction context defined by the EJB specification. The client is required to call without a transaction context.

- If the client calls with a transaction context, the Container throws the `java.rmi.RemoteException` exception.
- If the client calls without a transaction context, the Container performs the same steps as described in the `NotSupported` case.

16.6.2.7 Transaction attribute summary

The following table provides a summary of the transaction context that the Container passes to the business method and resource managers used by the business method, as a function of the transaction attribute and the client's transaction context. T1 is a transaction passed with the client request, while T2 is a transaction initiated by the Container.

Table 16 Transaction attribute summary

Transaction attribute	Client's transaction	Transaction associated with business method	Transaction associated with resource managers
NotSupported	none	none	none
	T1	none	none
Required	none	T2	T2
	T1	T1	T1

Table 16 Transaction attribute summary

Transaction attribute	Client's transaction	Transaction associated with business method	Transaction associated with resource managers
Supports	none	none	none
	T1	T1	T1
RequiresNew	none	T2	T2
	T1	T2	T2
Mandatory	none	**error**	N/A
	T1	T1	T1
Never	none	none	none
	T1	**error**	N/A

If the enterprise bean's business method invokes other enterprise beans via their home and remote interfaces, the transaction indicated in the column "Transaction associated with business method" will be passed as part of the client context to the target enterprise bean.

See Subsection 16.6.5 for how the Container handles the "none" case in Table 16.

16.6.2.8 Handling of `setRollbackOnly()` method

The Container must handle the `EJBContext.setRollbackOnly()` method invoked from a business method executing with the `Required`, `RequiresNew`, or `Mandatory` transaction attribute as follows:

- The Container must ensure that the transaction will never commit. Typically, the Container instructs the transaction manager to mark the transaction for rollback.
- If the Container initiated the transaction immediately before dispatching the business method to the instance (as opposed to the transaction being inherited from the caller), the Container must note that the instance has invoked the `setRollbackOnly()` method. When the business method invocation completes, the Container must roll back rather than commit the transaction. If the business method has returned normally or with an application exception, the Container must pass the method result or the application exception to the client after the Container performed the rollback.

The Container must throw the `java.lang.IllegalStateException` if the `EJBContext.setRollbackOnly()` method is invoked from a business method executing with the `Supports`, `NotSupported`, or `Never` transaction attribute.

16.6.2.9 Handling of `getRollbackOnly()` method

The Container must handle the `EJBContext.getRollbackOnly()` method invoked from a business method executing with the `Required`, `RequiresNew`, or `Mandatory` transaction attribute.

The Container must throw the `java.lang.IllegalStateException` if the `EJBContext.getRollbackOnly()` method is invoked from a business method executing with the `Supports`, `NotSupported`, or `Never` transaction attribute.

16.6.2.10 Handling of `getUserTransaction()` method

If an instance of an enterprise bean with container-managed transaction demarcation attempts to invoke the `getUserTransaction()` method of the `EJBContext` interface, the Container must throw the `java.lang.IllegalStateException`.

16.6.2.11 `javax.ejb.SessionSynchronization` callbacks

If a Session Bean class implements the `javax.ejb.SessionSynchronization` interface, the Container must invoke the `afterBegin()`, `beforeCompletion()`, and `afterCompletion(...)` callbacks on the instance as part of the transaction commit protocol.

The Container invokes the `afterBegin()` method on an instance before it invokes the first business method in a transaction.

The Container invokes the `beforeCompletion()` method to give the enterprise bean instance the last chance to cause the transaction to rollback. The instance may cause the transaction to roll back by invoking the `EJBContext.setRollbackOnly()` method.

The Container invokes the `afterCompletion(Boolean committed)` method after the completion of the transaction commit protocol to notify the enterprise bean instance of the transaction outcome.

16.6.3 Container-managed transaction demarcation for Message-driven Beans

The Container is responsible for providing the transaction demarcation for the message-driven beans that the Bean Provider declared as with container-managed transaction demarcation. For these enterprise beans, the Container must demarcate transactions as specified in the deployment descriptor by the Application Assembler. (See Chapter 21 for more information about the deployment descriptor.)

The following subsections define the responsibilities of the Container for managing the invocation of a message-driven bean's `onMessage` method. The Container's responsibilities depend on the value of the transaction attribute.

Only the `NotSupported` and `Required` transaction attributes may be used for message-driven beans. The use of the other transaction attributes is not meanigful for message-driven beans because there can be no pre-existing transaction context (`RequiresNew`, `Supports`) and no client to handle exceptions (`Mandatory`, `Never`).

16.6.3.1 NotSupported

The Container invokes a message-driven Bean method whose transaction attribute is set to `NotSupported` with an unspecified transaction context.

If the `onMessage` method invokes other enterprise beans, the Container passes no transaction context with the invocation.

16.6.3.2 Required

The Container must invoke a message-driven Bean method whose transaction attribute is set to `Required` with a valid transaction context. Because there is never a client transaction context available for a message-driven bean, the container automatically starts a new transaction before the dequeuing of the JMS message and, hence, before the invocation of the message-driven bean's `onMessage` method. The Container automatically enlists the resource manager associated with the arriving message and all the resource managers accessed by the `onMessage` method with the transaction. If the `onMessage` method invokes other enterprise beans, the Container passes the transaction context with the invocation. The Container attempts to commit the transaction when the `onMessage` method has completed. If the `onMessage` method does not successfully complete or the transaction is rolled back by the Container, JMS message redelivery semantics apply.

16.6.3.3 Handling of `setRollbackOnly()` method

The Container must handle the `EJBContext.setRollbackOnly()` method invoked from a `onMessage` method executing with the `Required` transaction attribute as follows:

- The Container must ensure that the transaction will never commit. Typically, the Container instructs the transaction manager to mark the transaction for rollback.
- The Container must note that the instance has invoked the `setRollbackOnly()` method. When the method invocation completes, the Container must roll back rather than commit the transaction.

The Container must throw and log the `java.lang.IllegalStateException` if the `EJBContext.setRollbackOnly()` method is invoked from an `onMessage` method executing with the `NotSupported` transaction attribute

16.6.3.4 Handling of `getRollbackOnly()` method

The Container must handle the `EJBContext.getRollbackOnly()` method invoked from an `onMessage` method executing with the `Required` transaction attribute.

The Container must throw and log the `java.lang.IllegalStateException` if the `EJBContext.getRollbackOnly()` method is invoked from an `onMessage` method executing with the `NotSupported` transaction attribute.

16.6.3.5 Handling of `getUserTransaction()` method

If an instance of a message-driven bean with container-managed transaction demarcation attempts to invoke the `getUserTransaction()` method of the `EJBContext` interface, the Container must throw and log the `java.lang.IllegalStateException`.

16.6.4 Local transaction optimization

The container may use a local transaction optimization for enterprise beans whose deployment descriptor indicates that connections to a resource manager are shareable (see Section 19.4.1.2 “Declaration of resource manager connection factory references in deployment descriptor”). The container manages the use of the local transaction optimization transparent to the application.

The container may use the optimization for transactions initiated by the container for a bean with container managed transaction demarcation and for transactions initiated by a bean with bean managed transaction demarcation with the `UserTransaction` interface. The container cannot apply the optimization for transactions imported from a different container.

The use of local transaction optimization approach is discussed in [9] and [12].

16.6.5 Handling of methods that run with “an unspecified transaction context”

The term “an unspecified transaction context” is used in the EJB specification to refer to the cases in which the EJB architecture does not fully define the transaction semantics of an enterprise bean method execution.

This includes the following cases:

- The execution of a method of an enterprise bean with container-managed transaction demarcation for which the value of the transaction attribute is `NotSupported`, `Never`, or `Supports`[25].
- The execution of the `ejbCreate<METHOD>`, `ejbRemove`, `ejbPassivate`, and `ejbActivate` methods of a session bean with container-managed transaction demarcation.
- The execution of the `ejbCreate<METHOD>` and `ejbRemove` methods of a message-driven bean with container-managed transaction demarcation.

The EJB specification does not prescribe how the Container should manage the execution of a method with an unspecified transaction context—the transaction semantics are left to the Container implementation. Some techniques for how the Container may choose to implement the execution of a method with an unspecified transaction context are as follows (the list is not inclusive of all possible strategies):

[25] For the `Supports` attribute, the handling described in this section applies only to the case when the client calls without a transaction context.

- The Container may execute the method and access the underlying resource managers without a transaction context.
- The Container may treat each call of an instance to a resource manager as a single transaction (e.g. the Container may set the auto-commit option on a JDBC connection).
- The Container may merge multiple calls of an instance to a resource manager into a single transaction.
- The Container may merge multiple calls of an instance to multiple resource managers into a single transaction.
- If an instance invokes methods on other enterprise beans, and the invoked methods are also designated to run with an unspecified transaction context, the Container may merge the resource manager calls from the multiple instances into a single transaction.
- Any combination of the above.

Since the enterprise bean does not know which technique the Container implements, the enterprise bean must be written conservatively not to rely on any particular Container behavior.

A failure that occurs in the middle of the execution of a method that runs with an unspecified transaction context may leave the resource managers accessed from the method in an unpredictable state. The EJB architecture does not define how the application should recover the resource managers' state after such a failure.

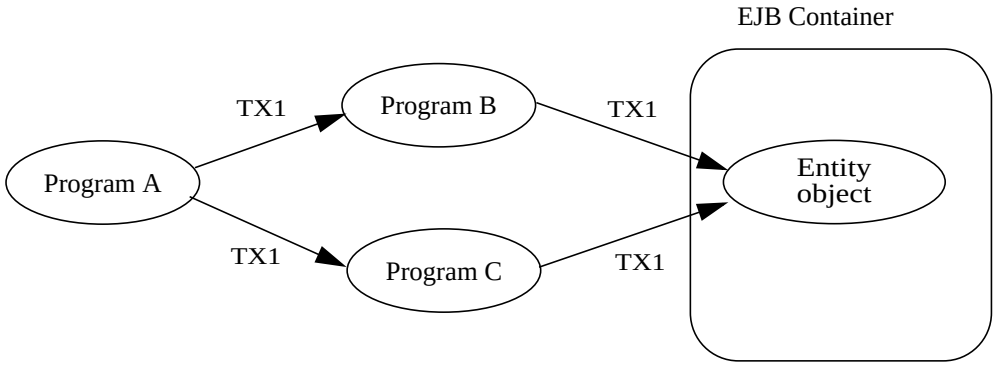
16.7 Access from multiple clients in the same transaction context

This section describes a more complex distributed transaction scenario, and specifies the Container's behavior required for this scenario.

16.7.1 Transaction "diamond" scenario with an entity object

An entity object may be accessed by multiple clients in the same transaction. For example, program A may start a transaction, call program B and program C in the transaction context, and then commit the transaction. If programs B and C access the same entity object, the topology of the transaction creates a diamond.

Figure 71 Transaction diamond scenario with entity object



An example (not realistic in practice) is a client program that tries to perform two purchases at two different stores within the same transaction. At each store, the program that is processing the client's purchase request debits the client's bank account.

It is difficult to implement an EJB server that handles the case in which programs B and C access an entity object through different network paths. This case is challenging because many EJB servers implement the EJB Container as a collection of multiple processes, running on the same or multiple machines. Each client is typically connected to a single process. If clients B and C connect to different EJB Container processes, and both B and C need to access the same entity object in the same transaction, the issue is how the Container can make it possible for B and C to see a consistent state of the entity object within the same transaction[26].

The above example illustrates a simple diamond. We use the term diamond to refer to any distributed transaction scenario in which an entity object is accessed in the same transaction through multiple network paths.

Note that in the diamond scenario the clients B and C access the entity object serially. Concurrent access to an entity object in the same transaction context would be considered an application programming error, and it would be handled in a Container-specific way.

Note that the issue of handling diamonds is not unique to the EJB architecture. This issue exists in all distributed transaction processing systems.

The following subsections define the responsibilities of the EJB Roles when handling distributed transaction topologies that may lead to a diamond involving an entity object.

[26] This diamond problem applies only to the case when B and C are in the same transaction.

16.7.2 Container Provider's responsibilities

This Subsection specifies the EJB Container's responsibilities with respect to the diamond case involving an entity object.

The EJB specification requires that the Container provide support for local diamonds. In a local diamond, components A, B, C, and D are deployed in the same EJB Container.

The EJB specification does not require an EJB Container to support distributed diamonds. In a distributed diamond, a target entity object is accessed from multiple clients in the same transaction through multiple network paths, and the clients (programs B and C) are not enterprise beans deployed in the same EJB Container as the target entity object.

If the Container Provider chooses not to support distributed diamonds, and if the Container can detect that a client invocation would lead to a diamond, the Container should throw the `java.rmi.RemoteException` to the client.

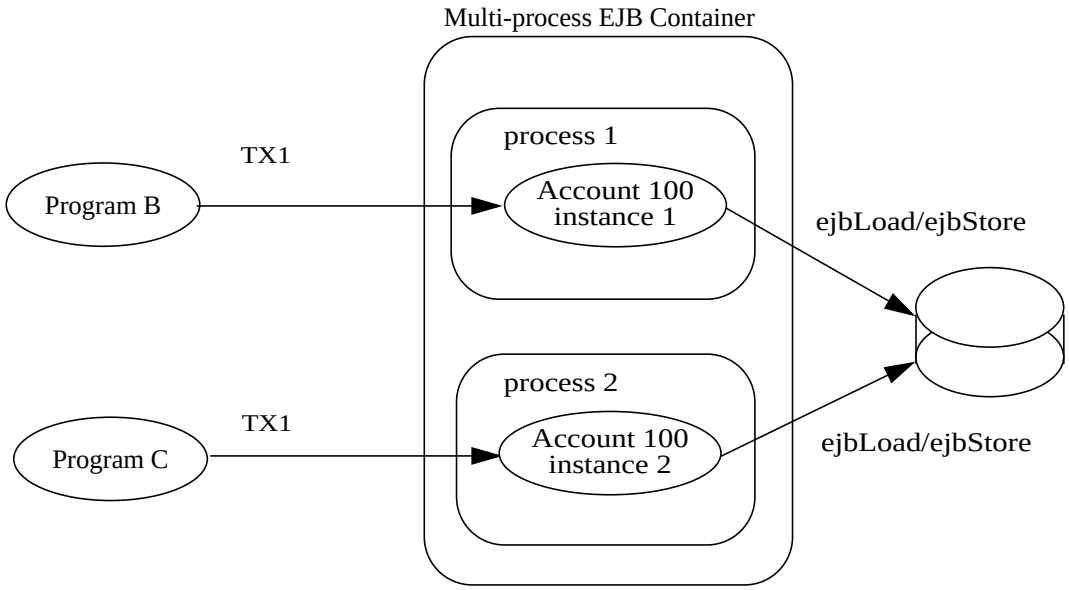
If the Container Provider chooses to support distributed diamonds, it should provide a consistent view of the entity state within a transaction. The Container Provider can implement the support in several ways. (The options that follow are illustrative, not prescriptive.)

- Always instantiate the entity bean instance for a given entity object in the same process, and route all clients' requests to this process. Within the process, the Container routes all the requests within the same transaction to the same enterprise bean instance.

- Instantiate the entity bean instance for a given entity object in multiple processes, and use the `ejbStore` and `ejbLoad` methods to synchronize the state of the instances within the same transaction. For example, the Container can issue `ejbStore` after each business method, and issue `ejbLoad` before the start of the next business method. This technique ensures that the instance used by a one client sees the updates done by other clients within the same transaction.

An illustration of the second approach follows. The illustration is illustrative, not prescriptive for the Container implementors.

Figure 72 Handling of diamonds by a multi-process container



Program B makes a call to an entity object representing Account 100. The request is routed to an instance in process 1. The Container invokes `ejbLoad` on the instance. The instance loads the state from the database in the `ejbLoad` method. The instance updates the state in the business method. When the method completes, the Container invokes `ejbStore`. The instance writes the updated state to the database in the `ejbStore` method.

Now program C makes a call to the same entity object in the same transaction. The request is routed to a different process (2). The Container invokes `ejbLoad` on the instance. The instance loads the state from the database in the `ejbLoad` method. The loaded state was written by the instance in process 1. The instance updates the state in the business method. When the method completes, the Container invokes `ejbStore`. The instance writes the updated state to the database in the `ejbStore` method.

In the above scenario, the Container presents the business methods operating on the entity object Account 100 with a consistent view of the entity object's state within the transaction.

Another implementation of the EJB Container might avoid calling `ejbLoad` and `ejbStore` on each business method by using a distributed lock manager.

16.7.3 Bean Provider's responsibilities

This Subsection specifies the Bean Provider's responsibilities with respect to the diamond case involving an entity object.

The diamond case is transparent to the Bean Provider—the Bean Provider does not have to code the enterprise bean differently for the bean to participate in a diamond. Any solution to the diamond problem implemented by the Container is transparent to the bean and does not change the semantics of the bean.

16.7.4 Application Assembler and Deployer's responsibilities

This Subsection specifies the Application Assembler and Deployer's responsibilities with respect to the diamond case involving an entity object.

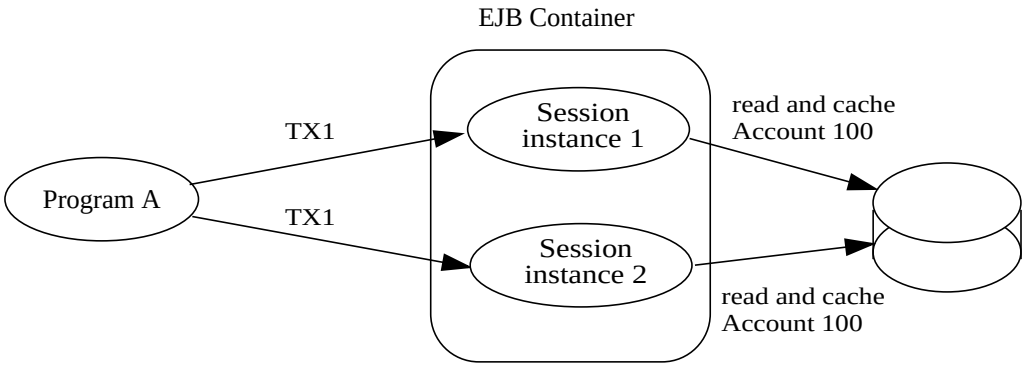
The Application Assembler and Deployer should be aware that distributed diamonds might occur. In general, the Application Assembler should try to avoid creating unnecessary distributed diamonds.

If a distributed diamond is necessary, the Deployer should advise the Container (using a Container-specific API) that an entity objects of the entity bean may be involved in distributed diamond scenarios.

16.7.5 Transaction diamonds involving session objects

While it is illegal for two clients to access the same session object, it is possible for applications that use session beans to encounter the diamond case. For example, program A starts a transaction and then invokes two different session objects.

Figure 73 Transaction diamond scenario with a session bean



If the session bean instances cache the same data item (e.g. the current balance of Account 100) across method invocations in the same transaction, most likely the program is going to produce incorrect results.

The problem may exist regardless of whether the two session objects are the same or different session beans. The problem may exist (and may be harder to discover) if there are intermediate objects between the transaction initiator and the session objects that cache the data.

There are no requirements for the Container Provider because it is impossible for the Container to detect this problem.

The Bean Provider and Application Assembler must avoid creating applications that would result in inconsistent caching of data in the same transaction by multiple session objects.

Chapter 17 Exception handling

17.1 Overview and Concepts

17.1.1 Application exceptions

An *application exception* is an exception defined in the throws clause of a method of the enterprise Bean's home and remote interfaces, other than the `java.rmi.RemoteException`.

Enterprise bean business methods use application exceptions to inform the client of abnormal application-level conditions, such as unacceptable values of the input arguments to a business method. A client can typically recover from an application exception. Application exceptions are not intended for reporting system-level problems.

For example, the Account *enterprise bean may throw an application exception to report that a* debit *operation cannot be performed because of an insufficient balance. The* Account *bean should not use an application exception to report, for example, the failure to obtain a database connection.*

The `javax.ejb.CreateException`, `javax.ejb.RemoveException`, `javax.ejb.FinderException`, and subclasses thereof are considered to be application exceptions. These exceptions are used as standard application exceptions to report errors to the client from the `create`, `remove`, and `finder` methods (see Subsections 9.6.8 and 11.1.9). These exceptions are covered by the rules on application exceptions that are defined in this chapter.

17.1.2 Goals for exception handling

The EJB specification for exception handling is designed to meet these high-level goals:

- An application exception thrown by an enterprise bean instance should be reported to the client *precisely* (i.e., the client gets the same exception).
- An application exception thrown by an enterprise bean instance should not automatically rollback a client's transaction. The client should typically be given a chance to recover a transaction from an application exception.
- An unexpected exception that may have left the instance's state variables and/or underlying persistent data in an inconsistent state can be handled safely.

17.2 Bean Provider's responsibilities

This section describes the view and responsibilities of the Bean Provider with respect to exception handling.

17.2.1 Application exceptions

The Bean Provider defines the application exceptions in the throws clauses of the methods of the remote and home interfaces. Because application exceptions are intended to be handled by the client, and not by the system administrator, they should be used only for reporting business logic exceptions, not for reporting system level problems.

The Bean Provider is responsible for throwing the appropriate application exception from the business method to report a business logic exception to the client. Because the application exception does not automatically result in marking the transaction for rollback, the Bean Provider must do one of the following to ensure data integrity before throwing an application exception from an enterprise bean instance:

- Ensure that the instance is in a state such that a client's attempt to continue and/or commit the transaction does not result in loss of data integrity. For example, the instance throws an application exception indicating that the value of an input parameter was invalid before the instance performed any database updates.
- Mark the transaction for rollback using the `EJBContext.setRollbackOnly()` method before throwing an application exception. Marking the transaction for rollback will ensure that the transaction can never commit.

An application exception class must be a subclass (direct or indirect) of `java.lang.Exception`. An application exception class must not be defined as a subclass of the `java.lang.RuntimeException`, or of the `java.rmi.RemoteException`. These are reserved for system exceptions (See next subsection).

The Bean Provider is also responsible for using the standard EJB application exceptions (`javax.ejb.CreateException`, `javax.ejb.RemoveException`, `javax.ejb.FinderException`, and subclasses thereof) as described in Subsections 9.6.8 and 11.1.9.

Bean Providers may define subclasses of the standard EJB application exceptions and throw instances of the subclasses in the entity bean methods. A subclass will typically provide more information to the client that catches the exception.

17.2.2 System exceptions

This subsection describes how the Bean Provider should handle various system-level exceptions and errors that an enterprise bean instance may encounter during the execution of a session or entity bean business method, a message-driven bean `onMessage` method, or a container callback method (e.g. `ejbLoad`).

The enterprise bean business method, `onMessage` method, or container callback method may encounter various exceptions or errors that prevent the method from successfully completing. Typically, this happens because the exception or error is unexpected, or the exception is expected but the EJB Provider does not know how to recover from it. Examples of such exceptions and errors are: failure to obtain a database connection, JNDI exceptions, unexpected `RemoteException` from invocation of other enterprise beans[27], unexpected `RuntimeException`, JVM errors, and so on.

If the enterprise bean method encounters a system-level exception or error that does not allow the method to successfully complete, the method should throw a suitable non-application exception that is compatible with the method's throws clause. While the EJB specification does not prescribe the exact usage of the exception, it encourages the Bean Provider to follow these guidelines:

- If the bean method encounters a `RuntimeException` or error, it should simply propagate the error from the bean method to the Container (i.e., the bean method does not have to catch the exception).
- If the bean method performs an operation that results in a checked exception[28] that the bean method cannot recover, the bean method should throw the `javax.ejb.EJBException` that wraps the original exception.
- Any other unexpected error conditions should be reported using the `javax.ejb.EJBException`.

[27] Note that the enterprise bean business method may attempt to recover from a RemoteException. The text in this subsection applies only to the case when the business method does not wish to recover from the RemoteException.

[28] A checked exception is one that is not a subclass of `java.lang.RuntimeException`.

Note that the `javax.ejb.EJBException` is a subclass of the `java.lang.RuntimeException`, and therefore it does not have to be listed in the throws clauses of the business methods.

The Container catches a non-application exception, logs it (which can result in alerting the System Administrator), and, unless the bean is a message-driven bean, throws the `java.rmi.RemoteException` (or subclass thereof) to the client. The Bean Provider can rely on the Container to perform the following tasks when catching a non-application exception:

- The transaction in which the bean method participated will be rolled back.
- No other method will be invoked on an instance that threw a non-application exception.

This means that the Bean Provider does not have to perform any cleanup actions before throwing a non-application exception. It is the Container that is responsible for the cleanup.

17.2.2.1 `javax.ejb.NoSuchEntityException`

The `NoSuchEntityException` is a subclass of `EJBException`. It should be thrown by the entity bean class methods to indicate that the underlying entity has been removed from the database.

An entity bean class typically throws this exception from the `ejbLoad` and `ejbStore` methods, and from the methods that implement the business methods defined in the remote interface.

17.3 Container Provider responsibilities

This section describes the responsibilities of the Container Provider for handling exceptions. The EJB architecture specifies the Container's behavior for the following exceptions:

- Exceptions from the business methods of session and entity beans.
- Exceptions from message-driven bean methods
- Exceptions from container-invoked callbacks on the enterprise bean.
- Exceptions from management of container-managed transaction demarcation.

17.3.1 Exceptions from a session or entity bean's business methods

Business methods are considered to be the methods defined in the enterprise bean's remote and home interface (including all their superinterfaces); and the following session bean or entity bean methods: `ejbCreate<METHOD>(...)`, `ejbPostCreate<METHOD>(...)`, `ejbRemove()`, and the `ejbFind<METHOD>` methods.

Table 17 specifies how the Container must handle the exceptions thrown by the business methods for beans with container-managed transaction demarcation. The table specifies the Container's action as a function of the condition under which the business method executes and the exception thrown by the business method. The table also illustrates the exception that the client will receive and how the client can recover from the exception. (Section 17.4 describes the client's view of exceptions in detail.)

Table 17 Handling of exceptions thrown by a business method of a bean with container-managed transaction demarcation

Method condition	Method exception	Container's action	Client's view
Bean method runs in the context of the caller's transaction [Note A]. This case may happen with `Required`, `Mandatory`, and `Supports` attributes.	AppException	Re-throw AppException	Receives AppException. Can attempt to continue computation in the transaction, and eventually commit the transaction (the commit would fail if the instance called `setRollbackOnly()`).
	all other exceptions and errors	Log the exception or error [Note B]. Mark the transaction for rollback. Discard instance [Note C]. Throw `TransactionRolledBackException` to the client.	Receives `TransactionRolledBackException`. Continuing transaction is fruitless.
Bean method runs in the context of a transaction that the Container started immediately before dispatching the business method. This case may happen with `Required` and `RequiresNew` attributes.	AppException	If the instance called setRollbackOnly(), then rollback the transaction, and re-throw AppException. Otherwise, attempt to commit the transaction, and then re-throw AppException.	Receives AppException. If the client executes in a transaction, the client's transaction is not marked for rollback, and client can continue its work.
	all other exceptions	Log the exception or error. Rollback the container-started transaction. Discard instance. Throw `RemoteException`.	Receives `RemoteException`. If the client executes in a transaction, the client's transaction is not marked for rollback, and client can continue its work.

Table 17 Handling of exceptions thrown by a business method of a bean with container-managed transaction demarcation

Method condition	Method exception	Container's action	Client's view
Bean method runs with an unspecified transaction context. This case may happen with the `NotSupported`, `Never`, and `Supports` attributes.	AppException	Re-throw AppException.	Receives AppException. If the client executes in a transaction, the client's transaction is not marked for rollback, and client can continue its work.
	all other exceptions	Log the exception or error. Discard instance. Throw `RemoteException`.	Receives `RemoteException`. If the client executes in a transaction, the client's transaction is not marked for rollback, and client can continue its work.

Notes:

[A] The caller can be another enterprise bean or an arbitrary client program.

[B] *Log the exception or error* means that the Container logs the exception or error so that the System Administrator is alerted of the problem.

[C] *Discard instance* means that the Container must not invoke any business methods or container callbacks on the instance.

Table 18 specifies how the Container must handle the exceptions thrown by the business methods for beans with bean-managed transaction demarcation. The table specifies the Container's action as a function of the condition under which the business method executes and the exception thrown by the business method. The table also illustrates the exception that the client will receive and how the client can recover from the exception. (Section 17.4 describes the client's view of exceptions in detail.)

Table 18 Handling of exceptions thrown by a business method of a session with bean-managed transaction demarcation

Bean method condition	Bean method exception	Container action	Client receives
Bean is stateful or stateless Session.	AppException	Re-throw AppException	Receives AppException.
	all other exceptions	Log the exception or error. Mark for rollback a transaction that has been started, but not yet completed, by the instance. Discard instance. Throw `RemoteException`.	Receives `RemoteException`.

17.3.2 Exceptions from message-driven bean methods

This section specifies the Container's handling of exceptions thrown from a message-driven bean's `onMessage`, `ejbCreate()`, and `ejbRemove()` methods.

Table 19 specifies how the Container must handle the exceptions thrown by the `onMessage`, `ejbCreate`, and `ejbRemove` methods for message-driven beans with container-managed transaction demarcation. The table specifies the Container's action as a function of the condition under which the method executes and the exception thrown by the method. Message-driven bean methods, unlike the business methods of session or entity beans, do not throw application exceptions and cannot throw exceptions to the client.

Table 19 Handling of exceptions thrown by a method of a message-driven bean with container-managed transaction demarcation.

Method condition	Method exception	Container's action
Bean method runs in the context of a transaction that the Container started immediately before dispatching the method. This case happens with `Required` attribute.	system exceptions	Log the exception or error[Note A]. Rollback the container-started transaction. Discard instance[Note B].
Bean method runs with an unspecified transaction context. This case happens with the `NotSupported` attribute.	system exceptions	Log the exception or error. Discard instance.

Notes:

[A] *Log the exception or error* means that the Container logs the exception or error so that the System Administrator is alerted of the problem.

[B] *Discard instance* means that the Container must not invoke any methods on the instance.

Table 20 specifies how the Container must handle the exceptions thrown by the `onMessage`, `ejbCreate`, and `ejbRemove` methods for message-driven beans with bean-managed transaction demarcation. The table specifies the Container's action as a function of the condition under which the method executes and the exception thrown by method.

Table 20 Handling of exceptions thrown by a method of a message-driven bean with bean-managed transaction demarcation.

Bean method condition	Bean method exception	Container action
Bean is message-driven bean	system exceptions	Log the exception or error. Mark for rollback a transaction that has been started, but not yet completed, by the instance. Discard instance.

17.3.3 Exceptions from container-invoked callbacks

This subsection specifies the Container's handling of exceptions thrown from the container-invoked callbacks on the enterprise bean. This subsection applies to the following callback methods:

- The `ejbActivate()`, `ejbLoad()`, `ejbPassivate()`, `ejbStore()`, `setEntityContext(EntityContext)`, and `unsetEntityContext()` methods of the `EntityBean` interface.
- The `ejbActivate()`, `ejbPassivate()`, and `setSessionContext(SessionContext)` methods of the `SessionBean` interface.
- The `setMessageDrivenContext(MessageDrivenContext)` method of the `MessageDrivenBean` interface.
- The `afterBegin(), beforeCompletion()` and `afterCompletion(boolean)` methods of the `SessionSynchronization` interface.

The Container must handle all exceptions or errors from these methods as follows:

- Log the exception or error to bring the problem to the attention of the System Administrator.
- If the instance is in a transaction, mark the transaction for rollback.
- Discard the instance (i.e., the Container must not invoke any business methods or container callbacks on the instance).

- If the exception or error happened during the processing of a client invoked method, throw the `java.rmi.RemoteException` to the client. If the instance executed in the client's transaction, the Container should throw the `javax.transaction.TransactionRolledBackException` because it provides more information to the client. (The client knows that it is fruitless to continue the transaction.)

17.3.4 `javax.ejb.NoSuchEntityException`

The `NoSuchEntityException` is a subclass of `EJBException`. If it is thrown by a method of an entity bean class, the Container must handle the exception using the rules for `EJBException` described in Sections 17.3.1, 17.3.2, and 17.3.3.

To give the client a better indication of the cause of the error, the Container should throw the `java.rmi.NoSuchObjectException` to the client (which is a subclass of `java.rmi.RemoteException`).

17.3.5 Non-existing session object

If a client makes a call to a session object that has been removed, the Container should throw the `java.rmi.NoSuchObjectException` to the client (which is a subclass of `java.rmi.RemoteException`).

17.3.6 Exceptions from the management of container-managed transactions

The container is responsible for starting and committing the container-managed transactions, as described in Subsection 16.6.2. This subsection specifies how the Container must deal with the exceptions that may be thrown by the transaction start and commit operations.

If the Container fails to start or commit a container-managed transaction, the Container must throw the `java.rmi.RemoteException` to the client. In the case where the Container fails to start or commit a container-managed transaction on behalf of a message-driven bean, the Container must throw and log the `javax.ejb.EJBException`.

However, the Container should not throw the `java.rmi.RemoteException` if the Container performs a transaction rollback because the instance has invoked the `setRollbackOnly()` method on its `EJBContext` object. In this case, the Container must rollback the transaction and pass the business method result or the application exception thrown by the business method to the client.

Note that some implementations of the Container may retry a failed transaction transparently to the client and enterprise bean code. Such a Container would throw the `java.rmi.RemoteException` *after a number of unsuccessful tries.*

17.3.7 Release of resources

When the Container discards an instance because of a system exception, the Container should release all the resources held by the instance that were acquired through the resource factories declared in the enterprise bean environment (See Subsection 19.4).

Note: While the Container should release the connections to the resource managers that the instance acquired through the resource factories declared in the enterprise bean environment, the Container cannot, in general, release "unmanaged" resources that the instance may have acquired through the JDK APIs. For example, if the instance has opened a TCP/IP connection, most Container implementations will not be able to release the connection. The connection will be eventually released by the JVM garbage collector mechanism.

17.3.8 Support for deprecated use of `java.rmi.RemoteException`

The EJB 1.0 specification allowed the business methods, `ejbCreate`, `ejbPostCreate`, `ejbFind<METHOD>`, `ejbRemove`, and the container-invoked callbacks (i.e., the methods defined in the `EntityBean`, `SessionBean`, and `SessionSynchronization` interfaces) implemented in the enterprise bean class to use the `java.rmi.RemoteException` to report non-application exceptions to the Container.

This use of the `java.rmi.RemoteException` was deprecated in EJB 1.1—enterprise beans written for the EJB 2.0 or EJB 1.1 specification should use the `javax.ejb.EJBException` instead.

The EJB 2.0 and EJB 1.1 specification require that a Container support the deprecated use of the `java.rmi.RemoteException`. The Container should treat the `java.rmi.RemoteException` thrown by an enterprise bean method in the same way as it is specified for the `javax.ejb.EJBException`.

Note: The use of the `java.rmi.RemoteException` *is deprecated only in the above-mentioned methods. The methods of the remote and home interface still must use the* `java.rmi.RemoteException` *as required by the EJB specification.*

17.4 Client's view of exceptions

This section describes the client's view of exceptions received from an enterprise bean invocation.

A client accesses an enterprise Bean through the enterprise Bean's remote and home interfaces. Both of these interfaces are Java RMI interfaces, and therefore the throws clauses of all their methods (including those inherited from superinterfaces) include the mandatory `java.rmi.RemoteException`. The throws clauses may include an arbitrary number of application exceptions.

17.4.1 Application exception

If a client program receives an application exception from an enterprise bean invocation, the client can continue calling the enterprise bean. An application exception does not result in the removal of the EJB object.

If a client program receives an application exception from an enterprise bean invocation while the client is associated with a transaction, the client can typically continue the transaction because an application exception does not automatically causes the Container to mark the transaction for rollback.

For example, if a client receives the ExceedLimitException *application exception from the* debit *method of an* Account *bean, the client may invoke the* debit *method again, possibly with a lower debit amount parameter. If the client executed in a transaction context, throwing the* ExceedLimitException *exception would not automatically result in rolling back, or marking for rollback, the client's transaction.*

Although the Container does not automatically mark for rollback a transaction because of a thrown application exception, the transaction might have been marked for rollback by the enterprise bean instance before it threw the application exception. There are two ways to learn if a particular application exception results in transaction rollback or not:

- Statically. Programmers can check the documentation of the enterprise bean's remote or home interface. The Bean Provider may have specified (although he is not required to) the application exceptions for which the enterprise bean marks the transaction for rollback before throwing the exception.

- Dynamically. Clients that are enterprise beans with container-managed transaction demarcation can use the `getRollbackOnly()` method of the `javax.ejb.EJBContext` object to learn if the current transaction has been marked for rollback; other clients may use the `getStatus()` method of the `javax.transaction.UserTransaction` interface to obtain the transaction status.

17.4.2 java.rmi.RemoteException

The client receives the `java.rmi.RemoteException` as an indication of a failure to invoke an enterprise bean method or to properly complete its invocation. The exception can be thrown by the Container or by the communication subsystem between the client and the Container.

If the client receives the `java.rmi.RemoteException` exception from a method invocation, the client, in general, does not know if the enterprise Bean's method has been completed or not.

If the client executes in the context of a transaction, the client's transaction may, or may not, have been marked for rollback by the communication subsystem or target bean's Container.

For example, the transaction would be marked for rollback if the underlying transaction service or the target Bean's Container doubted the integrity of the data because the business method may have been partially completed. Partial completion could happen, for example, when the target bean's method returned with a RuntimeException exception, or if the remote server crashed in the middle of executing the business method.

The transaction may not necessarily be marked for rollback. This might occur, for example, when the communication subsystem on the client-side has not been able to send the request to the server.

When a client executing in a transaction context receives a `RemoteException` from an enterprise bean invocation, the client may use either of the following strategies to deal with the exception:

- Discontinue the transaction. If the client is the transaction originator, it may simply rollback its transaction. If the client is not the transaction originator, it can mark the transaction for rollback or perform an action that will cause a rollback. For example, if the client is an enterprise bean, the enterprise bean may throw a `RuntimeException` which will cause the Container to rollback the transaction.

- Continue the transaction. The client may perform additional operations on the same or other enterprise beans, and eventually attempt to commit the transaction. If the transaction was marked for rollback at the time the `RemoteException` was thrown to the client, the commit will fail.

If the client chooses to continue the transaction, the client can first inquire about the transaction status to avoid fruitless computation on a transaction that has been marked for rollback. A client that is an enterprise bean with container-managed transaction demarcation can use the `EJBContext.getRollbackOnly()` method to test if the transaction has been marked for rollback; a client that is an enterprise bean with bean-managed transaction demarcation, and other client types, can use the `UserTransaction.getStatus()` method to obtain the status of the transaction.

Some implementations of EJB Servers and Containers may provide more detailed exception reporting by throwing an appropriate subclass of the `java.rmi.RemoteException` to the client. The following subsections describe the several subclasses of the `java.rmi.RemoteException` that may be thrown by the Container to give the client more information.

17.4.2.1 javax.transaction.TransactionRolledbackException

The `javax.transaction.TransactionRolledbackException` is a subclass of the `java.rmi.RemoteException.` It is defined in the JTA standard extension.

If a client receives the `javax.transaction.TransactionRolledbackException`, the client knows for certain that the transaction has been marked for rollback. It would be fruitless for the client to continue the transaction because the transaction can never commit.

17.4.2.2 javax.transaction.TransactionRequiredException

The `javax.transaction.TransactionRequiredException` is a subclass of the `java.rmi.RemoteException.` It is defined in the JTA standard extension.

The `javax.transaction.TransactionRequiredException` informs the client that the target enterprise bean must be invoked in a client's transaction, and that the client invoked the enterprise bean without a transaction context.

This error usually indicates that the application was not properly formed.

17.4.2.3 java.rmi.NoSuchObjectException

The `java.rmi.NoSuchObjectException` is a subclass of the `java.rmi.RemoteException`. It is thrown to the client if a remote business method cannot complete because the EJB object no longer exists.

17.5 System Administrator's responsibilities

The System Administrator is responsible for monitoring the log of the non-application exceptions and errors logged by the Container, and for taking actions to correct the problems that caused these exceptions and errors.

17.6 Differences from EJB 1.0

The EJB 2.0 and EJB 1.1 specification of exception handling preserve the rules defined in the EJB 1.0 specification, with the following exceptions:

- EJB 1.0 specified that the enterprise bean business methods and container-invoked callbacks use the `java.rmi.RemoteException` to report non-application exceptions. This practice was deprecated in EJB 1.1—the enterprise bean methods should use the `javax.ejb.EJBException`, or other suitable `RuntimeException` to report non-application exceptions.

- In EJB 2.0 and 1.1, all non-application exceptions thrown by the instance result in the rollback of the transaction in which the instance executed, and in discarding the instance. In EJB 1.0, the Container would not rollback a transaction and discard the instance if the instance threw the `java.rmi.RemoteException`.

- In EJB 2.0 and 1.1, an application exception does not cause the Container to automatically rollback a transaction. In EJB 1.0, the Container was required to rollback a transaction when an application exception was passed through a transaction boundary started by the Container. In EJB 1.1, the Container performs the rollback only if the instance have invoked the `setRollbackOnly()` method on its `EJBContext` object.

Chapter 18

Support for Distribution and Interoperability

This chapter describes the support for accessing enterprise beans from clients distributed over a network, and the interoperability requirements for invocations on enterprise beans from clients that are Java 2 Platform, Enterprise Edition (J2EE) components.

18.1 Support for distribution

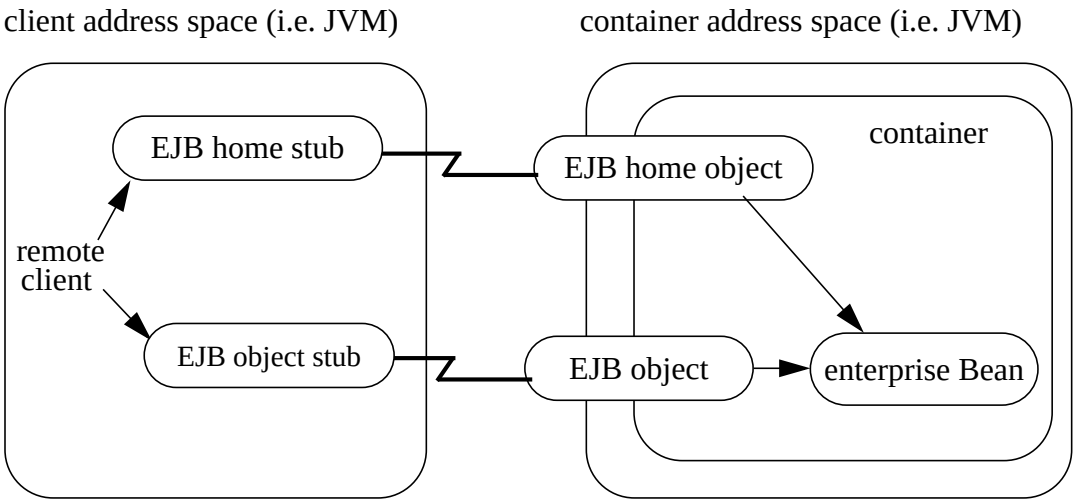
The home and remote interfaces of the enterprise bean's client view are defined as Java™ RMI [3] interfaces. This allows the Container to implement the home and remote interfaces as *distributed objects*. A client using the home and remote interfaces can reside on a different machine than the enterprise bean (location transparency), and the object references of the home and remote interfaces can be passed over the network to other applications.

The EJB specification further constrains the Java RMI types that can be used by enterprise beans to be legal RMI-IIOP types [7]. This makes it possible for EJB Container implementors to use RMI-IIOP as the object distribution protocol.

18.1.1 Client-side objects in distributed environment

When the RMI-IIOP protocol or similar distribution protocols are used, the client communicates with the enterprise bean using *stubs* for the server-side objects. The stubs implement the home and remote interfaces.

Figure 74 Location of EJB Client Stubs.



The communication stubs used on the client side are artifacts generated at the enterprise bean's deployment time by the EJB Container provider's tools. The stubs used on the client are specific to the wire protocol used for the remote invocation.

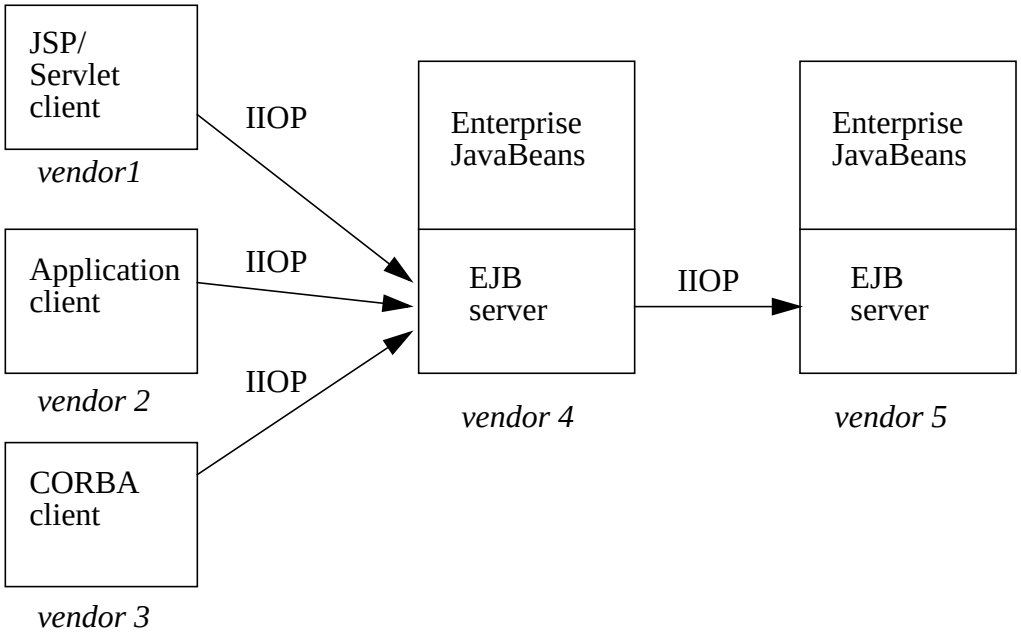
18.2 Interoperability overview

Session beans and entity beans that are deployed in one vendor's server product often need to be accessed from J2EE client components that are deployed in another vendor's product. EJB 2.0 defines a standard interoperability protocol based on CORBA/IIOP to address this need.

The interoperability protocols described here must be supported by compatible EJB products. Additional vendor-specific protocols may also be supported.

Figure 75 below shows a heterogeneous environment that includes systems from several vendors to illustrate the interoperability enabled by EJB 2.0.

Figure 75 Heterogeneous EJB Environment



The following sections in this chapter

- describe the goals for EJB invocation interoperability
- provide illustrative scenarios
- describe the interoperability requirements for remote invocations, transactions, naming, and security.

18.2.1 Interoperability goals

The goals of the interoperability requirements specified in this chapter are as follows:

- To allow clients in one application deployed in J2EE containers from one server provider to access services from session and entity beans in another application that is deployed in an EJB container from a different server provider. For example, web components (JavaServer Pages and Servlets) that are deployed on a J2EE-compliant web server provided by one server provider must be able to invoke the business methods of enterprise beans that are deployed on a J2EE-compliant EJB server from another server provider.
- To achieve interoperability without any new requirements on the J2EE application developer.

- To ensure out-of-the-box interoperability between compliant J2EE products. It must be possible for an enterprise customer to install multiple J2EE server products from different server providers (on potentially different operating systems), deploy applications in the J2EE servers, and have the multiple applications interoperate.

- To leverage the interoperability work done by standards bodies (including the IETF, W3C, and OMG) where possible, so that customers can work with industry standards and use standard protocols to access enterprise beans.

This specification does not address interoperability issues between enterprise beans and non-J2EE components. The J2EE platform specification [9] describes requirements for interoperability with Internet clients (using HTTP and XML) and interoperability with enterprise information systems (using the J2EE Connector architecture).

Since the interoperability protocol is based on CORBA/IIOP, CORBA clients written in Java, C++, or other languages can also invoke methods on enterprise beans.

This chapter subsumes the previous EJB1.1-to-CORBA mapping document [13].

18.3 Interoperability Scenarios

This section presents a number of interoperability scenarios that motivate the interoperability mechanisms described in later sections of this chapter. These scenarios are illustrative rather than prescriptive. There is no requirement that a J2EE product should support these scenarios in exactly the manner described here.

J2EE applications are multi-tier, web-enabled applications. Each application consists of one or more components, which are deployed in containers. The four types of containers are:

- EJB containers, which host enterprise beans.

- Web containers, which host JavaServer Pages (JSPs) and Servlet components as well as static documents including HTML pages.

- Application client containers, which host standalone applications.

- Applet containers, which host applets which may be downloaded from a web site. At this time, there is no requirement for an applet to be able to directly invoke the remote methods of enterprise beans.

The scenarios below describe interactions between components hosted in these various container types.

18.3.1 Interactions between web containers and EJB containers for e-commerce applications

This scenario occurs for business-to-business and business-to-consumer interactions over the Internet.

Scenario 1: *A customer wants to buy a book from an Internet bookstore. The bookstore's web site consists of a J2EE application containing JSPs that form the presentation layer, and another J2EE application containing enterprise beans that have the business logic and database access code. The JSPs and enterprise beans are deployed in containers from different vendors.*

At deployment time: *The enterprise beans are deployed, and their EJBHome objects are published in the EJB server's name service. The deployer links the EJB reference in the JSP's deployment descriptor to the URL of the enterprise bean's EJBHome object, which can be looked up from the name service. The transaction attribute specified in the enterprise bean's deployment descriptor is* `RequiresNew` *for all business methods. Because the "checkout" JSP requires secure access to set up payments for purchases, the bookstore's administrator configures the "checkout" JSP to require access over HTTPS with only server authentication. Customer authentication is done using form-based login. The "book search" JSP is accessed over normal HTTP. Both JSPs talk with enterprise beans which access the book database. The web and EJB containers use the same customer realm and have a trust relationship with each other. The network between the web and EJB servers is not guaranteed to be secure from attacks.*

At runtime: *The customer accesses the book search JSP using a browser. The JSP looks up the enterprise bean's EJBHome object in a name service, and calls* `findBooks(...)` *with the search criteria as parameters. The web container establishes a secure session with the EJB container with mutual authentication between the containers, and invokes the enterprise bean. The customer then decides to buy a book, and accesses the "checkout" JSP. The customer enters the necessary information in the login form, which is used by the web server to authenticate the customer. The JSP invokes the enterprise bean to update the book and customer databases. The customer's principal is propagated to the EJB container and used for authorization checks. The enterprise bean completes the updates and commits the transaction. The JSP sends back a confirmation page to the customer.*

18.3.2 Interactions between application client containers and EJB containers within an enterprise's intranet

Scenario 2.1: *An enterprise has an expense accounting application used by employees from their desktops. The server-side consists of a J2EE application containing enterprise beans that are deployed on one vendor's J2EE product, which is hosted in a datacenter. The client side consists of another J2EE application containing an application client deployed using another vendor's J2EE infrastructure. The network between the application client and the EJB container is insecure and needs to be protected against spoofing and other attacks.*

At deployment time: *The enterprise beans are deployed and their EJBHome objects are published in the enterprise's name service. The application clients are configured with the names of the EJBHome objects. The deployer maps employees to roles that are allowed access to the enterprise beans. The administrator configures the security settings of the application client and EJB container to require client and server authentication and message protection. The administrator also does the necessary client-side configuration to allow client authentication.*

At runtime: *The employee logs on using username and password. The application client container may interact with the enterprise's authentication service infrastructure to set up the employee's credentials. The client application does a remote invocation to the name server to look up the enterprise bean's EJBHome object, and creates the enterprise beans. The application client container uses a secure protocol to interact with the name server and EJB server, which does mutual authentication and also guarantees the confidentiality and integrity of messages. The employee then enters the expense information and submits it. This causes remote business methods of the enterprise beans to be invoked. The EJB container performs authorization checks and, if they succeed, executes the business methods.*

Scenario 2.2: *This is the same as Scenario 2.1, except that there is no client-side authentication infrastructure set up by the administrator. At runtime the client container needs to send the user's password to the server during the method invocation to authenticate the employee.*

18.3.3 Interactions between two EJB containers in an enterprise's intranet

Scenario 3: *An enterprise has an expense accounting application which needs to communicate with a payroll application. The applications use enterprise beans and are deployed on J2EE servers from different vendors. The J2EE servers and naming/authentication services may be in the enterprise's datacenter with a physically secure private network between them, or they may need to communicate across the intranet, which may be less secure. The applications need to update accounts and payroll databases. The employee (client) accesses the expense accounting application as described in Scenario 2.*

At deployment time: *The deployer configures both applications with the appropriate database resources. The accounts application is configured with the name of the EJBHome object of the payroll application. The payroll bean's deployment descriptor specifies the RequiresNew transaction attribute for all methods. The applications use the same principal-to-role mappings (e.g. the roles may be Employee, PayrollDept, AccountsDept). The deployer of these two applications has administratively set up a trust relationship between the two EJB containers, so that the containers do not need to authenticate principals propagated on calls to enterprise beans from the other container. The administrator also sets up the message protection parameters of the two containers if the network is not physically secure.*

At runtime: *An employee makes a request to the accounts application which requires it to access the payroll application. The accounts application does a lookup of the payroll application's EJBHome object in the naming/directory service and creates enterprise beans. It updates the accounts database and invokes a remote method of the payroll bean. The accounts bean's container propagates the employee's principal on the method call. The payroll bean's container maps the propagated employee principal to a role, does authorization checks, and sets up the payroll bean's transaction context. The container starts a new transaction, then the payroll bean updates the payroll database, and the container commits the transaction. The accounts bean receives a status reply from the payroll bean. If an error occurs in the payroll bean, the accounts bean executes code to recover from the error and restore the databases to a consistent state.*

18.3.4 Intranet application interactions between web containers and EJB containers

Scenario 4: *This is the same as scenario 2.1, except that instead of using a "fat-client" desktop application to access the enterprise's expense accounting application, employees use a web browser and connect to a web server in the intranet that hosts JSPs. The JSPs gather input from the user (e.g., through an HTML form), invoke enterprise beans that contain the actual business logic, and format the results returned by the enterprise beans (using HTML).*

At deployment time: *The enterprise deployer configures its expense accounting JSPs to require access over HTTPS with mutual authentication. The web and EJB containers use the same customer realm and have a trust relationship with each other.*

At run-time: *The employee logs in to the client desktop, starts the browser, and accesses the expense accounting JSP. The browser establishes an HTTPS session with the web server. Client authentication is performed (for example) using the employee's credentials which have been established by the operating system at login time (the browser interacts with the operating system to obtain the employee's credentials). The JSP looks up the enterprise bean's EJBHome object in a name service. The web container establishes a secure session with the EJB container with mutual authentication and integrity/confidentiality protection between the containers, and invokes methods on the enterprise beans.*

18.3.5 Overview of interoperability requirements

The following interoperable mechanisms are used to support the scenarios described above:

1. Remote method invocation on an enterprise bean's EJBObject and EJBHome object references (scenarios 1,2,3,4), described in section 18.4.

2. Name service lookup of the enterprise bean's EJBHome object (scenarios 1,2,3,4), described in section 18.6.

3. Integrity and confidentiality protection of messages (scenarios 1,2,3,4), described in section 18.7.

4. Authentication between an application client and EJB container (described in section 18.7):

- **4.1** Mutual authentication when there is client-side authentication infrastructure such as certificates (scenario 2.1).
- **4.2** Propagation of the user's authentication data from application client to EJB container to allow the EJB container to authenticate the client when there is no client-side authentication infrastructure (scenario 2.2).

5. Mutual authentication between two EJB containers or between a web and EJB container to establish trust before principals are propagated (scenarios 1,3,4), described in section 18.7.

6. Propagation of the Internet or intranet user's principal name for invocations on enterprise beans from web or EJB containers when the client and server containers have a trust relationship (scenarios 1,3,4), described in section 18.7.

18.4 Remote Invocation Interoperability

This section describes the interoperability mechanisms that enable remote invocations on EJBObject and EJBHome object references when client containers and EJB containers are provided by different vendors. This is needed to satisfy interoperability requirement one in section 18.3.5.

All EJB, web, and application client containers must support the IIOP 1.2 protocol for remote invocations on EJBObject and EJBHome references. EJB containers must be capable of servicing IIOP 1.2 based invocations on EJBObject and EJBHome objects. IIOP 1.2 is part of the CORBA 2.3.1 specification [14] from the OMG[29]. Containers may additionally support vendor-specific protocols.

CORBA Interoperable Object References (IORs) for EJBObject and EJBHome object references must include the GIOP version number 1.2. The IIOP infrastructure in all J2EE containers must be able to accept fragmented GIOP messages, although sending fragmented messages is optional. Bidirectional GIOP messages may optionally be supported by J2EE clients and servers: if a J2EE server receives an IIOP message from a client which contains the *BiDirIIOPServiceContext* structure, it may or may not use the same connection for sending requests back to the client.

Since Java applications use Unicode characters by default, J2EE containers are required to support the Unicode UTF16 code set for transmission of character and string data (in the IDL *wchar* and *wstring* datatypes). J2EE containers may optionally support additional code sets. EJBObject and EJBHome IORs must have the TAG_CODE_SETS tagged component which declares the codesets supported by the EJB container. IIOP messages which include *wchar* and *wstring* datatypes must have the code sets service context field. The CORBA 2.3.1 requirements for code set support must be followed by J2EE containers.

EJB containers are required to translate Java types to their on-the-wire representation in IIOP messages using the Java Language to IDL mapping specification [7] with the wire formats for IDL types as described in the GIOP specification in CORBA 2.3. The following subsections describe the mapping details for Java types.

18.4.1 Mapping Java Remote Interfaces to IDL

For each session bean or entity bean that is deployed in a container, there are two Java RMI remote interfaces—the bean's home interface and the bean's remote interface. The Java Language to IDL Mapping specification [7] describes precisely how these remote interfaces are mapped to IDL. This mapping to IDL is typically implicit when Java RMI over IIOP is used to invoke enterprise beans. J2EE clients use only the Java RMI APIs to invoke enterprise beans. The client container may use the CORBA portable Stub APIs for the client-side stubs. EJB containers may create CORBA Tie objects for each EJBObject or EJBHome object.

[29] CORBA APIs and earlier versions of the IIOP protocol are already included in the J2SE1.2, J2SE1.3 and J2EE1.2 platforms through JavaIDL and RMI-IIOP.

18.4.2 Mapping value objects to IDL

The Java interfaces that are passed by value during remote invocations on enterprise beans are `javax.ejb.Handle`, `javax.ejb.HomeHandle`, and `javax.ejb.EJBMetaData`. The `Enumeration` or `Collection` objects returned by entity bean finder methods are value types. There may also be application-specific value types that are passed as parameters or return values on enterprise bean invocations. In addition, several Java exception classes that are thrown by remote methods also result in concrete IDL value types. All these value types are mapped to IDL abstract value types or abstract interfaces using the rules in the Java Language to IDL Mapping.

18.4.3 Mapping of system exceptions

Java system exceptions, including the `java.rmi.RemoteException` and its subclasses, may be thrown by the EJB container. If the client's invocation was made over IIOP, the EJB server is required to map these exceptions to CORBA system exceptions and send them in the IIOP reply message to the client, as specified in the following table

System exception thrown by EJB container	CORBA system exception received by client ORB
javax.transaction. TransactionRolledbackException	TRANSACTION_ROLLEDBACK
javax.transaction. TransactionRequiredException	TRANSACTION_REQUIRED
javax.transaction. InvalidTransactionException	INVALID_TRANSACTION
java.rmi.NoSuchObjectException	OBJECT_NOT_EXIST
java.rmi.AccessException	NO_PERMISSION
java.rmi.MarshalException	MARSHAL
java.rmi.RemoteException	UNKNOWN

For EJB clients, the ORB's unmarshalling machinery maps CORBA system exceptions received in the IIOP reply message to the appropriate Java exception as specified in the Java Language to IDL mapping. This results in the original Java exception being received by the client J2EE component.

18.4.4 Obtaining stub and client view classes

When a J2EE component (application client, JSP, servlet or enterprise bean) receives a reference to an EJBObject or EJBHome object through JNDI lookup or as a parameter or return value of an invocation on an enterprise bean, an instance of an RMI-IIOP stub class (proxy) for the enterprise bean's home or remote RMI interface needs to be created. When a component receives a value object as a parameter or return value of an enterprise bean invocation, an instance of the value class needs to be created. The stub class, value class, and other client view classes must be available to the referencing container (the container hosting the component that receives the reference or value type).

The client view classes, including value classes, must be packaged with the referencing component's application, as described in Section 22.3. System value classes, including implementations of the javax.ejb.Handle, javax.ejb.HomeHandle, javax.ejb.EJBMetaData, java.util.Collection, and java.util.Iterator interfaces, must be provided in the form of a JAR file by the container hosting the referenced bean. For interoperability scenarios, if a referencing component would use such system value classes at runtime, the deployer must ensure that these system value clases provided by the container hosting the referenced bean are available to the referencing component. This may be done, for example, by including these system value classes in the classpath of the referencing container, or by deploying the system value classes with the referencing component's application by providing them to the deployment tool.

Implementations of these system value classes must be portable (they must use only J2SE and J2EE APIs) so that they can be instantiated in another vendor's container. If the system value class implementation needs to load application-specific classes (such as home or remote interfaces) at runtime, it must use the thread context class loader. The referencing container must make application-specific classes available to the system value class instance at runtime through the thread context class loader.

Stubs for invoking on EJBHome and EJBObject references must be provided by the referencing container, for example, by generating stub classes at deployment time for the EJBHome and EJBObject interfaces of the referenced beans that are packaged with the referencing component's application.

Containers may optionally support run-time downloading of stub and value classes needed by the referencing container. The CORBA 2.3.1 specification and the Java Language to IDL Mapping specify how stub and value type implementations are to be downloaded: using codebase URLs that are either embedded in the EJBObject or EJBHome's IOR, or sent in the IIOP message service context, or marshalled with the value type. The URLs for downloading may optionally include an HTTPS URL for secure downloading.

18.5 Transaction interoperability

Transaction interoperability between containers provided by different vendors is an optional feature in this version of the EJB specification. Vendors may choose to not implement transaction interoperability. However, vendors who choose to implement transaction interoperability must follow the requirements in sections 18.5.1 and 18.5.2, and vendors who choose not to implement transaction interoperability must follow the requirements in section 18.5.2.

18.5.1 Transaction interoperability requirements

A distributed transaction started by a web or EJB container must be able to propagate in a remote invocation to an enterprise bean in an EJB container provided by a different vendor, and the containers must participate in the distributed two-phase commit protocol.

18.5.1.1 Transaction context wire format

Transaction context propagation from client to EJB container uses the implicit propagation mechanism described in the CORBA Object Transaction Service (OTS) v1.2 specification [8].

The transaction context format in IIOP messages is specified in the *CosTransactions::PropagationContext* structure described in the OTS specification. EJB containers that support transaction interoperability are required to be capable of producing and consuming transaction contexts in IIOP messages in the format described in the OTS specification. Web containers that support transaction interoperability are required to include client-side libraries which can produce the OTS transaction context for sending over IIOP.

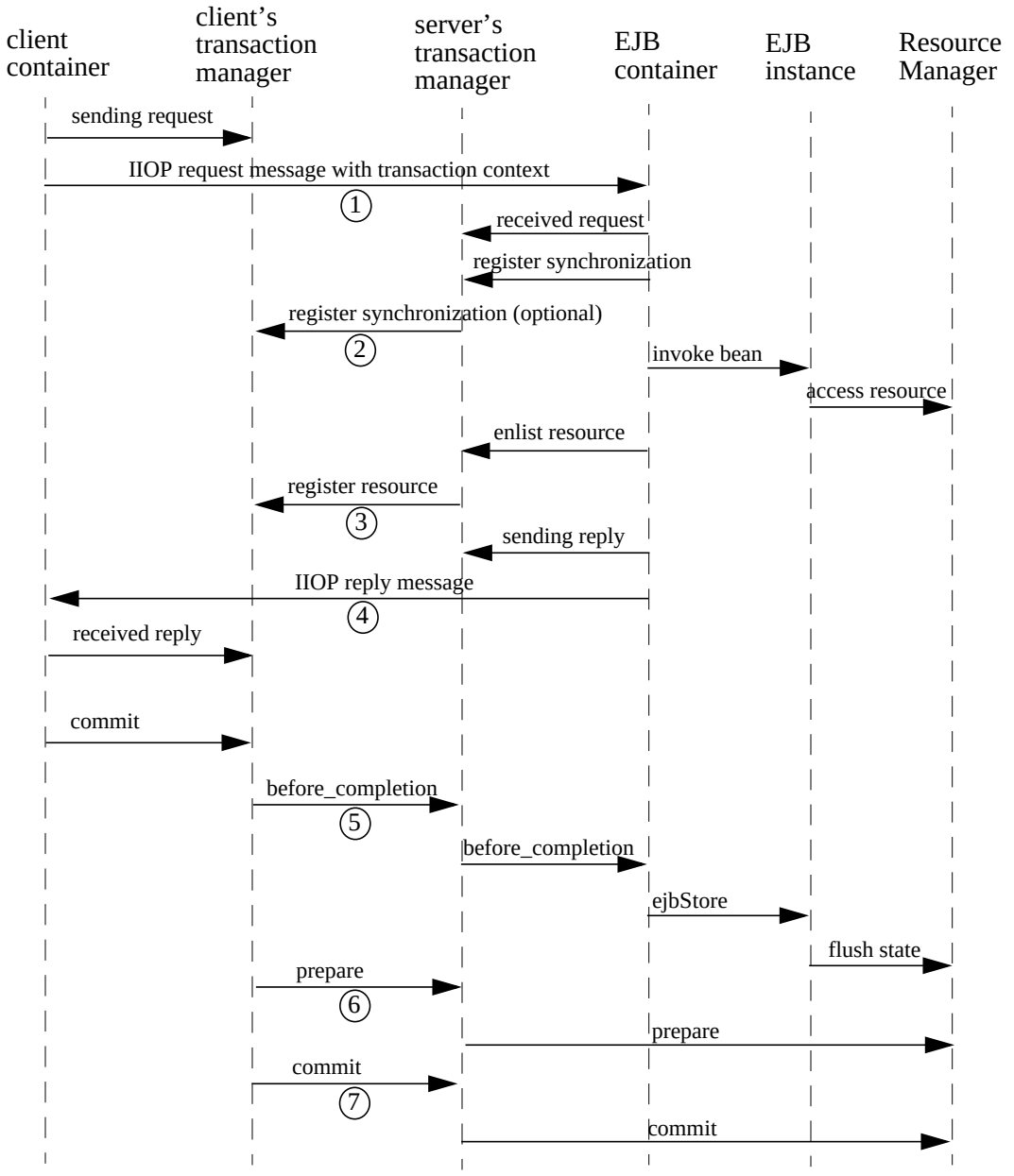
Note that it is not necessary for containers to include the Java mappings of the OTS APIs. A container may implement the requirements in the OTS specification in any manner, for example using a non-Java OTS implementation, or an on-the-wire bridge between an existing transaction manager and the OTS protocol, or an OTS wrapper around an existing transaction manager.

The *CosTransactions::PropagationContext* structure must be included in IIOP messages sent by web or EJB containers when required by the rules described in the OTS 1.2 specification. The target EJB container must process IIOP invocations based on the transaction policies of EJBObject or EJBHome references using the rules described in the OTS 1.2 specification [8].

18.5.1.2 Two-phase commit protocol

The object interaction diagram below (Figure 76) illustrates the interactions between the client and server transaction managers for transaction context propagation, resource and synchronization object registration, and two-phase commit. This diagram is an example of the interactions between the various entities; it is not intended to be prescriptive.

Figure 76 Transaction context propagation



Containers that perform transactional work within the scope of a transaction must register an OTS Resource object with the transaction coordinator whose object reference is included in the propagated transaction context (step 3), and may also register an OTS Synchronization object (step 2). If the server container does not register an OTS Synchronization object, it must still ensure that the `beforeCompletion` method of session beans and `ejbStore` method of entity beans are called with the proper transaction context. Containers must participate in the two-phase commit and recovery procedures performed by the transaction coordinator / terminator (steps 6,7), as described by the OTS specification.

Compliant J2EE containers must not use nested transactions in interoperability scenarios.

18.5.1.3 Transactional policies of enterprise bean references

The OTS1.2 specification describes the *CosTransactions::OTSPolicy* and *CosTransactions::InvocationPolicy* structures that are encoded in IORs as tagged components. EJBObject and EJBHome references must contain these tagged components[30] with policy values as described below.

The transaction attributes of enterprise beans can be specified per method, while in OTS the entire CORBA object has the same OTS transaction policy. The rules below ensure that the transaction context will be propagated if any method of an enterprise bean needs to execute in the client's transaction context. However, in some cases there may be extra performance overhead of propagating the client's transaction context even if it will not be used by the enterprise bean method.

EJBObject and EJBHome references may have the InvocationPolicy value as either *CosTransactions::SHARED* or *CosTransactions::EITHER*[31].

The following rules list the OTSPolicy values which must be assigned to EJBHome and EJBObject references:

- For session beans that use bean-managed transactions, the OTSPolicy value must be *CosTransactions::ADAPTS.*
- If all methods of an EJB interface (Home or Remote) have the transaction attribute `Mandatory`, the CORBA object for that interface must have the OTSPolicy value *CosTransactions::REQUIRES.*
- If all methods of an EJB interface (Home or Remote) have the transaction attribute `Never`, the CORBA object for that interface must have the OTSPolicy value *CosTransactions::FORBIDS.*
- CORBA objects for all other EJB interfaces must have the OTSPolicy value *CosTransactions::ADAPTS.*

The OTSPolicy and InvocationPolicy values must be a valid combination as described in the OTS1.2 specification.

The *CosTransactions::Synchronization* object registered by the EJB container with the transaction coordinator should have the OTSPolicy value *CosTransactions::ADAPTS* and InvocationPolicy value *CosTransactions::SHARED* to allow enterprise beans to do transactional work during the `beforeCompletion` notification from the transaction coordinator.

Client and EJB containers must set the NonTxTargetPolicy policy to *CosTransactions::PREVENT.*

---

[30] One way to include the tagged components in IORs is to create the object references using a Portable Object Adapter (POA) which is initialized with the appropriate transaction policies. Note that POA APIs are not required to be supported by server containers.

[31] If the InvocationPolicy is not present in the IOR, it is interpreted by the client as if the policy value was *CosTransactions::EITHER.*

18.5.1.4 Exception handling behavior

The exception handling behavior described in the OTS1.2 specification must be followed. In particular, if an application exception (an exception which is not a CORBA system exception and does not extend `java.rmi.RemoteException`) is returned by the server, the request is defined as being successful; hence the client-side OTS library must not roll back the transaction. This allows application exceptions to be propagated back to the client without rolling back the transaction, as required by the exception handling rules in Chapter 17.

18.5.2 Interoperating with containers that do not implement transaction interoperability

The requirements in this subsection are designed to ensure that when a J2EE container does not support transaction interoperability, the failure modes are well defined so that the integrity of an application's data is not compromised: at worst the transaction is rolled back. When a J2EE client component expects the client's transaction to propagate to the enterprise bean but the client or EJB container cannot satisfy this expectation, a `java.rmi.RemoteException` or subclass is thrown, which ensures that the client's transaction will roll back.

In addition, the requirements below allow a container that does not support transaction propagation to interoperate with a container that does support transaction propagation in the cases where the enterprise bean method's transaction attribute indicates that the method would not be executed in the client's transaction.

18.5.2.1 Client container requirements

If the client in another container invokes an enterprise bean's method when there is no active global transaction associated with the client's thread, the client container does not include a transaction context in the IIOP request message to the EJB server, i.e., there is no *CosTransactions::PropagationContext* structure in the IIOP request header.

The client application component expects a global transaction to be propagated to the server only if the client's thread has an active global transaction. In this scenario, if the client container does not support transaction interoperability, it has two options:

1. If the client container does not support transaction propagation or uses a non-OTS protocol, it must include the OTS *CosTransactions::PropagationContext* structure in the IIOP request to the server (step 1 in the object interaction diagram above), with the *CosTransactions::Coordinator* and *CosTransactions::Terminator* object references as null. The remaining fields in this "null transaction context," such as the transaction identifier, are not interpreted and may have any value. The "null transaction context" indicates that there is a global client transaction active but the client container is not capable of propagating it to the server. The presence of this "null transaction context" allows the EJB container to determine whether the J2EE client component expects the client's global transaction to propagate to the server.

2. Client containers that use the OTS transaction context format but still do not support transaction interoperability with other vendor's containers must reject the *Coordinator::register_resource* call (step 3 in the object interaction diagram above) if the server's Resource object reference indicates that it belongs to another vendor's container.

18.5.2.2 EJB container requirements

All EJB containers (including those that do not support transaction propagation) must include the *CosTransactions::OTSPolicy* and optionally the *CosTransactions::InvocationPolicy* tagged component in the IOR for EJBObject and EJBHome references as described in section 18.5.1.3.

18.5.2.2.1 Requirements for EJB containers supporting transaction interoperability

When an EJB container that supports transaction propagation receives the IIOP request message, it must behave as follows:

- If there is no OTS transaction context in the IIOP message, the container must follow the behavior described in Section 16.6.
- If there is a valid, complete OTS transaction context in the IIOP message, the container must follow the behavior described in Section 16.6.
- If there is a null transaction context (as defined in section 18.5.2.1 above) in the IIOP message, the container's required behavior is described in the table below. The entry "throw RemoteException" indicates that the EJB container must throw the `java.rmi.RemoteException` to the client after the "received request" interaction with the server's transaction manager (after step 1 in the object interaction diagram above).

EJB method's Transaction Attribute	EJB container behavior on receiving null OTS transaction context
Mandatory	throw RemoteException
Required	throw RemoteException
RequiresNew	follow Section 16.6
Supports	throw RemoteException
NotSupported	follow Section 16.6
Never	follow Section 16.6
Bean Managed	follow Section 16.6

18.5.2.2.2 Requirements for EJB containers not supporting transaction interoperability

When an EJB container that does not support transaction interoperability receives the IIOP request message, it must behave as follows:

- If there is no OTS transaction context in the IIOP message, the container must follow the behavior described in Section 16.6.

- If there is a valid, complete OTS transaction context in the IIOP message, the container's required behavior is described in the table below.

- If there is a null transaction context (as defined in section 18.5.2.1) in the IIOP message, the container's required behavior is described in the table below. Note that the container may not know whether the received transaction context in the IIOP message is valid or null.

EJB method's Transaction Attribute	EJB container behavior on receiving null or valid OTS transaction context
Mandatory	throw RemoteException
Required	throw RemoteException
RequiresNew	follow Section 16.6
Supports	throw RemoteException
NotSupported	follow Section 16.6
Never	follow Section 16.6
Bean Managed	follow Section 16.6

EJB containers that accept the OTS transaction context format but still do not support interoperability with other vendors' client containers must follow the column in the table above for "null or valid OTS transaction context" if the transaction identity or the Coordinator object reference in the propagated client transaction context indicate that the client belongs to a different vendor's container.

18.6 Naming Interoperability

This section describes the requirements for supporting interoperable access to naming services for looking up EJBHome object references (interoperability requirement two in section 18.3.5).

EJB containers are required to be able to publish EJBHome object references in a CORBA CosNaming service [15]. The CosNaming service must implement the IDL interfaces in the CosNaming module defined in [15] and allow clients to invoke the *resolve* and *list* operations over IIOP.

The CosNaming service must follow the requirements in the CORBA Interoperable Name Service specification [16] for providing the host, port, and object key for its root NamingContext object. The CosNaming service must be able to service IIOP invocations on the root NamingContext at the advertised host, port, and object key.

Client containers (i.e., EJB, web, or application client containers) are required to include a JNDI CosNaming service provider that uses the mechanisms defined in the Interoperable Name Service specification to contact the server's CosNaming service, and to resolve the EJBHome object using standard CosNaming APIs. The JNDI CosNaming service provider may or may not use the JNDI SPI architecture. The JNDI CosNaming service provider must access the root NamingContext of the server's CosNaming service by creating an object reference from the URL *corbaloc:iiop:1.2@<host>:<port>/<objectkey>* (where *<host>*, *<port>*, and *<objectkey>* are the values corresponding to the root NamingContext advertised by the server's CosNaming service), or by using an equivalent mechanism.

At deployment time, the deployer of the client container should obtain the host/port of the server's CosNaming service and the CosNaming name of the server EJBHome object (e.g. by browsing the server's namespace) for each `ejb-ref` element in the client component's deployment descriptor. The `ejb-ref-name` (which is used by the client code in the JNDI lookup call) should then be linked to the EJBHome object's CosNaming name. At run-time, the client component's JNDI lookup call uses the CosNaming service provider, which contacts the server's CosNaming service, resolves the CosNaming name, and returns the EJBHome object reference to the client component.

Since the EJBHome object's name is scoped within the namespace of the CosNaming service that is accessible at the provided host and port, it is not necessary to federate the namespaces of the client and server containers.

The advantage of using CosNaming is better integration with the IIOP infrastructure that is already required for interoperability, as well as interoperability with non-J2EE CORBA clients and servers. Since CosNaming stores only CORBA objects it is likely that vendors will use other enterprise directory services for storing other resources.

Security of CosNaming service access is achieved using the security interoperability protocol described in Section 18.7. The CosNaming service must support this protocol. Clients which construct the root NamingContext object reference from a URL should send an IIOP LocateRequest message to the CosNaming service to obtain the complete IOR (with SSL information) of the root NamingContext, and then initiate an SSL session with the CosNaming service.

18.7 Security Interoperability

This section describes the interoperable mechanisms that support secure invocations on enterprise beans in intranets. These mechanisms are based on the CORBA/IIOP protocol.

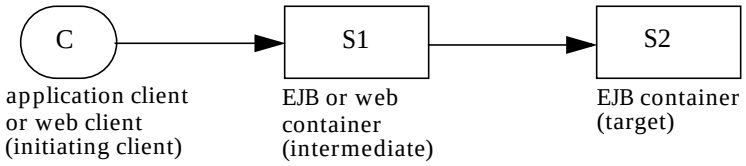
18.7.1 Introduction

The goal of the secure invocation mechanisms is to support the interoperability requirements described earlier in this chapter, as well as be capable of supporting security technologies that are expected to be widely deployed in enterprises, including Kerberos-based secret key mechanisms and X.509 certificate-based public key mechanisms.

The authentication identity (i.e. principal) associated with a J2EE component is usually that of the user on whose behalf the component is executing[32]. The principal under which an enterprise bean invocation is performed is either that of the bean's caller or the run-as principal which was configured by the deployer. When there is a chain of invocations across a web component and enterprise beans, an intermediate component may use the principal of the caller (the initiating client) or the intermediate component may use its run-as principal to perform an invocation on the callee, depending on the security identity specified for the intermediate component in its deployment descriptor.

The security principal associated with a container depends on the type of container. Application client containers usually do not have a separate principal associated with them (they operate under the user's principal). Web and EJB containers are typically associated with a security principal of their own (e.g., the operating system user for the container's process) which may be configured by the administrator at deployment time. When the client is a web or EJB container, the difference between the client component's principal and the client container's principal is significant for interoperability considerations.

18.7.1.1 Trust relationships between containers, principal propagation



When there is a chain of multiple invocations across web components and enterprise beans, intermediate components may not have access to the authentication data of the initiating client to provide proof of the client's identity to the target. In such cases, the target's authentication requirements can be satisfied if the target container trusts the intermediate container to vouch for the authenticity of the propagated principal. The call is made using the intermediate container's principal and authentication data, while also carrying the propagated principal of the initiating client. The invocation on the target enterprise bean is authorized and performed using the propagated principal. This procedure also avoids the overhead associated with authentication of clients on every remote invocation in a chain.

EJB containers are required to provide deployers or administrators with the tools to configure trust relationships for interactions with intermediate web or EJB containers[33]. If a trust relationship is set up, the containers are usually configured to perform mutual authentication, unless the security of the network can be ensured by some physical means. After a trust relationship is set up, the target EJB container does not need to independently authenticate the initiating client principal sent by the intermediate container on invocations. Thus only the principal name of the initiating client (which may include a realm) needs to be propagated.

---

[32] When there are concurrent invocations on a component from multiple clients, a different principal may be associated with the thread of execution for each invocation.

[33] One way to achieve this is to configure a "trusted container list" (TCL) for each EJB container which contains the list of intermediate client containers that are trusted. If the TCL is empty, then the target EJB container does not have a trust relationship with any intermediate container.

For the current interoperability needs of J2EE, it is assumed that trust relationships are transitive, such that if a target container trusts an intermediate container, it implicitly trusts all containers trusted by the intermediate container.

If no trust relationship has been set up between a target EJB container and intermediate web or EJB container, the target container needs to have access to and independently verify the initiating client principal's authentication data. Support for this scenario (where containers do not have a trust relationship) is not currently required.

Web and EJB containers are required to support at least caller propagation (where the initiating client's principal is propagated down the chain of calls on enterprise beans). This is needed for scenarios 1, 3 and 4 where the internet or intranet user's principal needs to be propagated to the target EJB container.

18.7.1.2 Application Client Authentication

Application client containers that have authentication infrastructure (such as certificates, Kerberos) can

- authenticate the user by interacting with an authentication service (e.g. the Kerberos KDC) in the enterprise
- inherit an authentication context which was established at system login time from the operating system process, or
- obtain the user's certificate from a client-side store.

These may be achieved by plugging in a Java Authentication and Authorization Service (JAAS) login module for the particular authentication service. After authentication is completed, a credential is associated with the client's thread of execution, which is used for all invocations on enterprise beans made from that thread.

If there is no authentication infrastructure installed in the client's environment, the client may send its private credentials (e.g. password) over a secure connection to the EJB server, which authenticates the user by interacting with an authentication service (e.g. a secure user/password database). This is similar to the basic authentication feature of HTTP.

18.7.2 Securing EJB invocations

This subsection describes the interoperable protocol requirements for providing authentication, protection of integrity and confidentiality, and principal propagation for invocations on enterprise beans. The invocation takes place over an enterprise's intranet as described in the scenarios in section 18.3. Since EJB invocations use the IIOP protocol, we need to secure IIOP messages between client and server containers. The client may be any of the J2EE containers and the server is an EJB container.

The secure interoperability requirements for EJB2.0 and other J2EE1.3 containers correspond to Conformance Level 0 of the Common Secure Interoperability version 2 (CSIv2) specification [20] which was developed by the OMG.

18.7.2.1 Secure transport protocol

The Secure Sockets Layer (SSL 3.0) protocol [19] and the related IETF standard Transport Layer Security (TLS 1.0) protocol [17] provide authentication and message protection (that is, integrity and/or confidentiality) at the transport layer. The original SSL and TLS specifications supported only X.509 certificates for authenticating principals. Recently, Kerberos-based authentication mechanisms and cipher suites have been defined for TLS (RFC 2712 [18]). Thus the TLS specification is capable of supporting the two main security technologies that are expected to be widely deployed in enterprises.

EJB, web and application client containers are required to support both SSL3.0 and TLS1.0 as security protocols for IIOP. This satisfies interoperability requirement 3 in section 18.3.5. The following public key SSL/TLS ciphersuites are required to be supported by compliant containers:

- TLS_RSA_WITH_RC4_128_MD5
- SSL_RSA_WITH_RC4_128_MD5
- TLS_DHE_DSS_WITH_3DES_EDE_CBC_SHA[34]
- SSL_DHE_DSS_WITH_3DES_EDE_CBC_SHA
- TLS_RSA_EXPORT_WITH_RC4_40_MD5
- SSL_RSA_EXPORT_WITH_RC4_40_MD5
- TLS_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA
- SSL_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA

Support for Kerberos ciphersuites is not specified.

When using IIOP over SSL, a secure channel between client and server containers is established at the SSL layer. The SSL handshake layer handles authentication (either mutual or server-only) between containers, negotiation of cipher suite for bulk data encryption, and optionally provides a compression method. The SSL record layer performs confidentiality and integrity protection on application data. Since compliant J2EE products are already required to support SSL (HTTPS for Internet communication), the use of SSL/TLS provides a relatively easy route to interoperable security at the transport layer.

18.7.2.2 Security information in IORs

Before initiating a secure connection to the EJB container, the client needs to know the hostname and port number at which the server is listening for SSL connections, and the security protocols supported or required by the server object. This information is obtained from the EJBObject or EJBHome reference's IOR.

[34] This ciphersuite is mandatory for compliant TLS implementations as specified in [17].

The CSIv2 specification [20] describes the TAG_CSI_SEC_MECH_LIST tagged component which must be included in EJBHome and EJBObject IORs.[35] This component contains a sequence of *CSIIOP::CompoundSecMech* structures (in decreasing order of the server's preference) that contain the target object's security information for transport layer and service context layer mechanisms. This information includes the server's SSL/TLS port, its security principal and supported/required security mechanisms.

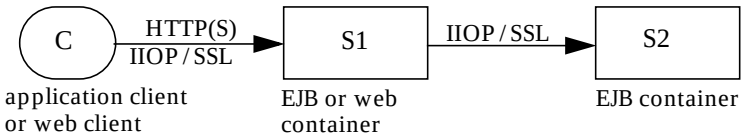
EJB containers must be capable of inserting the information in these structures into the IORs for EJBObject and EJBHome references, based on the deployer or administrator's security policy settings. Compliant EJB containers must follow the Conformance Level 0 rules described in the CSIv2 specification for constructing these IORs.

EJB containers must also be capable of creating IORs that allow access to enterprise beans over unprotected IIOP, based on the security policies set by the deployer or administrator.

18.7.2.3 Propagating principals and authentication data in IIOP messages

In scenarios where client authentication does not occur at the transport layer or where an intermediate client container does not have authentication data for the initiating client, it is necessary to support propagation of client principals and transfer of authentication data between two containers in the IIOP message service context.

It is assumed that all information exchanged between client and server at the transport layer is also available to the containers: e.g. the certificates used for authentication at the SSL layer may be used by the server container for authorization.



The following cases are required to be supported:

1. Application client invocations on enterprise beans with mutual authentication between the application client and EJB container (C and S1) at the SSL layer (scenario 2.1 in section 18.3.2, interoperability requirement 4.1 in section 18.3.5). E.g. this is possible when the enterprise has a Kerberos-based authentication infrastructure or when client-side certificates have been installed. In this case no additional information is required to be included in the security context of the IIOP message sent from C to S1.

[35] The standard tagged component and service context identifier values assigned by OMG are available at http://cgi.omg.org/cgi-bin/doc?standard-tags.

2. Application client invocations on enterprise beans with server-only authentication between the application client and EJB container (C and S1) at the SSL layer (scenario 2.2 in section 18.3.2, interoperability requirement 4.2 in section 18.3.5). This usually happens when there is no client-side authentication infrastructure. In this case, the client container must be capable of inserting into the IIOP message a CSIv2 security context with a client authentication token that contains the client C's authentication data. Once the EJB container S1 has authenticated the client, it may or may not maintain state about the client, so subsequent invocations from the client on the same network connection may need to be authenticated again. The client and server containers must follow the Conformance Level 0 rules in the CSIv2 specification for client authentication. In particular, support for the GSSUP username-password authentication mechanism is required. Support for other GSSAPI mechanisms (such as Kerberos) to perform client authentication at the IIOP layer is optional.

3. Invocations from Web/EJB clients to enterprise beans with a trust relationship between the client container S1 and server container S2 (scenario 3 in section 18.3.3, interoperability requirements five and six in section 18.3.5). S2 does not need to independently authenticate the initiating client C. In this case the client container S1 must be capable of inserting into the IIOP message a CSIv2 security context with an identity token. The identity token contains a principal name and realm (authentication domain). The principal may be propagated as an X.509 certificate chain or as a X.501 distinguished name or as a typed principal name encoded using the formats described in the CSIv2 specification. The identity propagated is determined as follows:
    - If the client Web/EJB component is configured to use caller identity, and the caller C authenticated itself to S1, then the identity token contains the initiating client C's identity.
    - If the client component is configured to use caller identity, and the caller C did not authenticate itself to S1, then the identity token contains the anonymous type.
    - If the client component is configured to use a run-as identity then the identity token contains the run-as identity.

J2EE containers are required to support the stateless mode of propagating principal and authentication information defined in CSIv2 (where the server does not store any state for a particular client principal across invocations), and may optionally support the stateful mode.

The caller principal String provided by `EJBContext.getCallerPrincipal().getName()` is defined as follows:

- For case one, the principal is derived from the distinguished name obtained from the client's X.509 certificate that was provided to the server during SSL mutual authentication.

- For case two, the principal is derived from the username obtained from the client authentication token in the CSIv2 security context of the IIOP message. For the GSSUP username-password mechanism, the principal is derived from the *Security::ScopedName* structure.

- For case three, the principal depends on the identity token type in the CSIv2 security context:
    - If the type is X.509 certificate chain, then the principal is derived from the distinguished name from the first certificate in the chain.
    - If the type is distinguished name, then the principal is derived from the distinguished name.

- If the type is principal name propagated as a GSS exported name, then the mechanism-specific principal name is returned. For the GSSUP username-password mechanism, the principal is derived from the *Security::ScopedName* structure.
- If the anonymous or absent principal type was propagated, then `EJBContext.getCallerPrincipal().getName()` returns a product-specific unauthenticated principal name.

18.7.2.4 Security configuration for containers

Since the interoperability scenarios involve IIOP/SSL usage in intranets, it is assumed that client and server container administrators cooperatively configure a consistent set of security policies for the enterprise.

At product installation or application deployment time, client and server container administrators may optionally configure the container and SSL infrastructure as described below. These preferences may be specified at any level of granularity (e.g. per host or per container process or per enterprise bean).

- Configure the list of supported SSL cipher suites in preference order.
- For server containers, configure a list of trusted client container principals with whom the server has a trust relationship.
- Configure authentication preferences and requirements (e.g. if the server prefers authenticated clients to anonymous clients). In particular, if a trust relationship has been configured between two servers, then mutual authentication should be required unless there is physical network security.
- If the client and server are using certificates for authentication, configure a trusted common certificate authority for both client and server. If using Kerberos, configure the client and server with the same KDC or cooperating KDCs.
- Configure a restricted list of trusted server principals that a client container is allowed to interact with, to prevent the client's private credentials such as password from being sent to untrusted servers.

18.7.2.5 Runtime behavior

Client containers should determine whether to use SSL for an enterprise bean invocation by using the security policies configured by the client administrator for interactions with the target host or enterprise bean, and the "*target_requires*" information in the CSIv2 tagged component in the target enterprise bean's IOR. If either the client configuration requires secure interactions with the enterprise bean, or the enterprise bean requires a secure transport, the client should initiate an SSL connection to the server. The client must follow the rules described in the CSIv2 specification Conformance Level 0 for including security context information in IIOP messages.

When an EJB container receives an IIOP message, its behavior depends on deployment time configuration, run-time information exchanged with the client at the SSL layer, and principal/authentication data contained in the IIOP message service context. EJB containers are required to follow the protocol rules prescribed by the CSIv2 specification Conformance Level 0.

When the administrator changes the security policies associated with an enterprise bean, the IORs for EJB references should be updated. When the bean has existing clients holding IORs, it is recommended that the security policy change should be handled by the client and server containers transparently to the client application if the old security policy is compatible with the new one. This may be done by using interoperable GIOP 1.2 forwarding mechanisms.

Chapter 19

Enterprise bean environment

This chapter specifies the interfaces for accessing the enterprise bean environment.

19.1 Overview

The Application Assembler and Deployer should be able to customize an enterprise bean's business logic without accessing the enterprise bean's source code.

In addition, ISVs typically develop enterprise beans that are, to a large degree, independent from the operational environment in which the application will be deployed. Most enterprise beans must access resource managers and external information. The key issue is how enterprise beans can locate external information without prior knowledge of how the external information is named and organized in the target operational environment.

The enterprise bean environment mechanism attempts to address both of the above issues.

This chapter is organized as follows.

- Section 19.2 defines the interfaces that specify and access the enterprise bean's environment. The section illustrates the use of the enterprise bean's environment for generic customization of the enterprise bean's business logic.

- Section 19.3 defines the interfaces for obtaining the home interface of another enterprise bean using an *EJB reference*. An EJB reference is a special entry in the enterprise bean's environment.

- Section 19.4 defines the interfaces for obtaining a resource manager connection factory using a *resource manager connection factory reference*. A resource manager connection factory reference is a special entry in the enterprise bean's environment.

- Section 19.5 defines the interfaces for obtaining an administered object that is associated with a resource (e.g., a JMS destination) using a *resource environment reference*. A resource environment reference is a special entry in the enterprise bean's environment.

19.2 Enterprise bean's environment as a JNDI naming context

The enterprise bean's environment is a mechanism that allows customization of the enterprise bean's business logic during deployment or assembly. The enterprise bean's environment allows the enterprise bean to be customized without the need to access or change the enterprise bean's source code.

The Container implements the enterprise bean's environment, and provides it to the enterprise bean instance through the JNDI interfaces. The enterprise bean's environment is used as follows:

1. The enterprise bean's business methods access the environment using the JNDI interfaces. The Bean Provider declares in the deployment descriptor all the environment entries that the enterprise bean expects to be provided in its environment at runtime.

2. The Container provides an implementation of the JNDI naming context that stores the enterprise bean environment. The Container also provides the tools that allow the Deployer to create and manage the environment of each enterprise bean.

3. The Deployer uses the tools provided by the Container to create the environment entries that are declared in the enterprise bean's deployment descriptor. The Deployer can set and modify the values of the environment entries.

4. The Container makes the environment naming context available to the enterprise bean instances at runtime. The enterprise bean's instances use the JNDI interfaces to obtain the values of the environment entries.

Each enterprise bean defines its own set of environment entries. All instances of an enterprise bean within the same home share the same environment entries; the environment entries are not shared with other enterprise beans. Enterprise bean instances are not allowed to modify the bean's environment at runtime.

If an enterprise bean is deployed multiple times in the same Container, each deployment results in the creation of a distinct home. The Deployer may set different values for the enterprise bean environment entries for each home.

Terminology warning: The enterprise bean's "environment" should not be confused with the "environment properties" defined in the JNDI documentation.

The following subsections describe the responsibilities of each EJB Role.

19.2.1 Bean Provider's responsibilities

This section describes the Bean Provider's view of the enterprise bean's environment, and defines his or her responsibilities.

19.2.1.1 Access to enterprise bean's environment

An enterprise bean instance locates the environment naming context using the JNDI interfaces. An instance creates a `javax.naming.InitialContext` object by using the constructor with no arguments, and looks up the environment naming via the `InitialContext` under the name `java:comp/env`. The enterprise bean's environment entries are stored directly in the environment naming context, or in any of its direct or indirect subcontexts.

The value of an environment entry is of the Java type declared by the Bean Provider in the deployment descriptor.

The following code example illustrates how an enterprise bean accesses its environment entries.

```
public class EmployeeServiceBean implements SessionBean {

    ...
    public void setTaxInfo(int numberOfExemptions, ...)
            throws InvalidNumberOfExemptionsException {
        ...

        // Obtain the enterprise bean's environment naming context.
        Context initCtx = new InitialContext();
        Context myEnv = (Context)initCtx.lookup("java:comp/env");

        // Obtain the maximum number of tax exemptions
        // configured by the Deployer.
        Integer max = (Integer)myEnv.lookup("maxExemptions");

        // Obtain the minimum number of tax exemptions
        // configured by the Deployer.
        Integer min = (Integer)myEnv.lookup("minExemptions");

        // Use the environment entries to customize business logic.
        if (numberOfExeptions > Integer.intValue(max) ||
            numberOfExemptions < Integer.intValue(min))
                throw new InvalidNumberOfExemptionsException();

        // Get some more environment entries. These environment
        // entries are stored in subcontexts.
        String val1 = (String)myEnv.lookup("foo/name1");
        Boolean val2 = (Boolean)myEnv.lookup("foo/bar/name2");

        // The enterprise bean can also lookup using full pathnames.
        Integer val3 = (Integer)
            initCtx.lookup("java:comp/env/name3");
        Integer val4 = (Integer)
            initCtx.lookup("java:comp/env/foo/name4");
        ...
    }
}
```

19.2.1.2 Declaration of environment entries

The Bean Provider must declare all the environment entries accessed from the enterprise bean's code. The environment entries are declared using the `env-entry` elements in the deployment descriptor.

Each `env-entry` element describes a single environment entry. The `env-entry` element consists of an optional description of the environment entry, the environment entry name relative to the `java:comp/env` context, the expected Java type of the environment entry value (i.e., the type of the object returned from the JNDI `lookup` method), and an optional environment entry value.

An environment entry is scoped to the enterprise bean whose declaration contains the `env-entry` element. This means that the environment entry is inaccessible from other enterprise beans at runtime, and that other enterprise beans may define `env-entry` elements with the same `env-entry-name` without causing a name conflict.

The environment entry values may be one of the following Java types: `String`, `Integer`, `Boolean`, `Double`, `Byte`, `Short`, `Long`, and `Float`.

If the Bean Provider provides a value for an environment entry using the `env-entry-value` element, the value can be changed later by the Application Assembler or Deployer. The value must be a string that is valid for the constructor of the specified type that takes a single `String` parameter.

The following example is the declaration of environment entries used by the `EmployeeService-Bean` whose code was illustrated in the previous subsection.

```
<enterprise-beans>
     <session>
          ...
          <ejb-name>EmployeeService</ejb-name>
          <ejb-class>
               com.wombat.empl.EmployeeServiceBean
          </ejb-class>
          ...
          <env-entry>
               <description>
                    The maximum number of tax exemptions
                    allowed to be set.
               </description>
               <env-entry-name>maxExemptions</env-entry-name>
               <env-entry-type>java.lang.Integer</env-entry-type>
               <env-entry-value>15</env-entry-value>
          </env-entry>
          <env-entry>
               <description>
                    The minimum number of tax exemptions
                    allowed to be set.
               </description>
               <env-entry-name>minExemptions</env-entry-name>
               <env-entry-type>java.lang.Integer</env-entry-type>
               <env-entry-value>1</env-entry-value>
          </env-entry>
          <env-entry>
               <env-entry-name>foo/name1</env-entry-name>
               <env-entry-type>java.lang.String</env-entry-type>
               <env-entry-value>value1</env-entry-value>
          </env-entry>
          <env-entry>
               <env-entry-name>foo/bar/name2</env-entry-name>
               <env-entry-type>java.lang.Boolean</env-entry-type>
               <env-entry-value>true</env-entry-value>
          </env-entry>
          <env-entry>
               <description>Some description.</description>
               <env-entry-name>name3</env-entry-name>
               <env-entry-type>java.lang.Integer</env-entry-type>
          </env-entry>
          <env-entry>
               <env-entry-name>foo/name4</env-entry-name>
               <env-entry-type>java.lang.Integer</env-entry-type>
               <env-entry-value>10</env-entry-value>
          </env-entry>
          ...
     </session>
</enterprise-beans>
...
```

19.2.2 Application Assembler's responsibility

The Application Assembler is allowed to modify the values of the environment entries set by the Bean Provider, and is allowed to set the values of those environment entries for which the Bean Provider has not specified any initial values.

19.2.3 Deployer's responsibility

The Deployer must ensure that the values of all the environment entries declared by an enterprise bean are set to meaningful values.

The Deployer can modify the values of the environment entries that have been previously set by the Bean Provider and/or Application Assembler, and must set the values of those environment entries for which no value has been specified.

The `description` elements provided by the Bean Provider or Application Assembler help the Deployer with this task.

19.2.4 Container Provider responsibility

The container provider has the following responsibilities:

- Provide a deployment tool that allows the Deployer to set and modify the values of the enterprise bean's environment entries.
- Implement the `java:comp/env` environment naming context, and provide it to the enterprise bean instances at runtime. The naming context must include all the environment entries declared by the Bean Provider, with their values supplied in the deployment descriptor or set by the Deployer. The environment naming context must allow the Deployer to create subcontexts if they are needed by an enterprise bean.
- The Container must ensure that the enterprise bean instances have only read access to their environment variables. The Container must throw the `javax.naming.OperationNotSupportedException` from all the methods of the `javax.naming.Context` interface that modify the environment naming context and its subcontexts.

19.3 EJB references

This section describes the programming and deployment descriptor interfaces that allow the Bean Provider to refer to the homes of other enterprise beans using "logical" names called *EJB references*. The EJB references are special entries in the enterprise bean's environment. The Deployer binds the EJB references to the enterprise bean's homes in the target operational environment.

The deployment descriptor also allows the Application Assembler to *link* an EJB reference declared in one enterprise bean to another enterprise bean contained in the same ejb-jar file, or in another ejb-jar file in the same J2EE application unit. The link is an instruction to the tools used by the Deployer that the EJB reference must be bound to the home of the specified target enterprise bean.

19.3.1 Bean Provider's responsibilities

This subsection describes the Bean Provider's view and responsibilities with respect to EJB references.

19.3.1.1 EJB reference programming interfaces

The Bean Provider must use EJB references to locate the home interfaces of other enterprise beans as follows.

- Assign an entry in the enterprise bean's environment to the reference. (See subsection 19.3.1.2 for information on how EJB references are declared in the deployment descriptor.)
- *The EJB specification recommends, but does not require, that all references to other enterprise beans be organized in the* `ejb` *subcontext of the bean's environment (i.e., in the* `java:comp/env/ejb` *JNDI context).*
- Look up the home interface of the referenced enterprise bean in the enterprise bean's environment using JNDI.

The following example illustrates how an enterprise bean uses an EJB reference to locate the home interface of another enterprise bean.

```
public class EmployeeServiceBean implements SessionBean {

     public void changePhoneNumber(...) {
          ...

          // Obtain the default initial JNDI context.
          Context initCtx = new InitialContext();

          // Look up the home interface of the EmployeeRecord
          // enterprise bean in the environment.
          Object result = initCtx.lookup(
               "java:comp/env/ejb/EmplRecord");

          // Convert the result to the proper type.
          EmployeeRecordHome emplRecordHome = (EmployeeRecordHome)
               javax.rmi.PortableRemoteObject.narrow(result,
                         EmployeeRecordHome.class);
          ...
     }
}
```

In the example, the Bean Provider of the `EmployeeServiceBean` enterprise bean assigned the environment entry `ejb/EmplRecord` as the EJB reference name to refer to the home of another enterprise bean.

19.3.1.2 Declaration of EJB references in deployment descriptor

Although the EJB reference is an entry in the enterprise bean's environment, the Bean Provider must not use a `env-entry` element to declare it. Instead, the Bean Provider must declare all the EJB references using the `ejb-ref` elements of the deployment descriptor. This allows the ejb-jar consumer (i.e. Application Assembler or Deployer) to discover all the EJB references used by the enterprise bean.

Each `ejb-ref` element describes the interface requirements that the referencing enterprise bean has for the referenced enterprise bean. The `ejb-ref` element contains an optional `description` element; and the mandatory `ejb-ref-name`, `ejb-ref-type`, `home`, and `remote` elements.

The `ejb-ref-name` element specifies the EJB reference name; its value is the environment entry name used in the enterprise bean code. The `ejb-ref-type` element specifies the expected type of the enterprise bean; its value must be either `Entity` or `Session`. The `home` and `remote` elements specify the expected Java types of the referenced enterprise bean's home and remote interfaces.

An EJB reference is scoped to the enterprise bean whose declaration contains the `ejb-ref` element. This means that the EJB reference is not accessible to other enterprise beans at runtime, and that other enterprise beans may define `ejb-ref` elements with the same `ejb-ref-name` without causing a name conflict.

The following example illustrates the declaration of EJB references in the deployment descriptor.

```
...
<enterprise-beans>
     <session>
          ...
          <ejb-name>EmployeeService</ejb-name>
          <ejb-class>
               com.wombat.empl.EmployeeServiceBean
          </ejb-class>
          ...
          <ejb-ref>
               <description>
                    This is a reference to the entity bean that
                    encapsulates access to employee records.
               </description>
               <ejb-ref-name>ejb/EmplRecord</ejb-ref-name>
               <ejb-ref-type>Entity</ejb-ref-type>
               <home>com.wombat.empl.EmployeeRecordHome</home>
               <remote>com.wombat.empl.EmployeeRecord</remote>
          </ejb-ref>

          <ejb-ref>
               <ejb-ref-name>ejb/Payroll</ejb-ref-name>
               <ejb-ref-type>Entity</ejb-ref-type>
               <home>com.aardvark.payroll.PayrollHome</home>
               <remote>com.aardvark.payroll.Payroll</remote>
          </ejb-ref>

          <ejb-ref>
               <ejb-ref-name>ejb/PensionPlan</ejb-ref-name>
               <ejb-ref-type>Session</ejb-ref-type>
               <home>com.wombat.empl.PensionPlanHome</home>
               <remote>com.wombat.empl.PensionPlan</remote>
          </ejb-ref>
          ...
     </session>
     ...
</enterprise-beans>
...
```

19.3.2 Application Assembler's responsibilities

The Application Assembler can use the `ejb-link` element in the deployment descriptor to link an EJB reference to a target enterprise bean. The link will be observed by the deployment tools.

The Application Assembler specifies the link between two enterprise beans as follows:

- The Application Assembler uses the optional `ejb-link` element of the `ejb-ref` element of the referencing enterprise bean. The value of the `ejb-link` element is the name of the target enterprise bean. (It is the name defined in the `ejb-name` element of the target enterprise bean.) The target enterprise bean can be in any ejb-jar file in the same J2EE application as the referencing application component.

- Alternatively, to avoid the need to rename enterprise beans to have unique names within an entire J2EE application, the Application Assembler may use the following syntax in the `ejb-link` element of the referencing application component. The Application Assembler specifies the path name of the ejb-jar file containing the referenced enterprise bean and appends the `ejb-name` of the target bean separated from the path name by `#`. The path name is relative to the referencing application component jar file. In this manner, multiple beans with the same `ejb-name` may be uniquely identified when the Application Assembler cannot change ejb-names.

- The Application Assembler must ensure that the target enterprise bean is type-compatible with the declared EJB reference. This means that the target enterprise bean must be of the type indicated in the `ejb-ref-type` element, and that the home and remote interfaces of the target enterprise bean must be Java type-compatible with the interfaces declared in the EJB reference.

The following illustrates an `ejb-link` in the deployment descriptor.

```
...
<enterprise-beans>
     <session>
          ...
          <ejb-name>EmployeeService</ejb-name>
          <ejb-class>
               com.wombat.empl.EmployeeServiceBean
          </ejb-class>
          ...
          <ejb-ref>
               <ejb-ref-name>ejb/EmplRecord</ejb-ref-name>
               <ejb-ref-type>Entity</ejb-ref-type>
               <home>com.wombat.empl.EmployeeRecordHome</home>
               <remote>com.wombat.empl.EmployeeRecord</remote>
               <ejb-link>EmployeeRecord</ejb-link>
          </ejb-ref>
          ...
     </session>
     ...


     <entity>
          <ejb-name>EmployeeRecord</ejb-name>
          <home>com.wombat.empl.EmployeeRecordHome</home>
          <remote>com.wombat.empl.EmployeeRecord</remote>
          ...
     </entity>
     ...
</enterprise-beans>
...
```

The Application Assembler uses the `ejb-link` element to indicate that the EJB reference "EmplRecord" declared in the `EmployeeService` enterprise bean has been linked to the `EmployeeRecord` enterprise bean.

The following example illustrates using the `ejb-link` element to indicate an enterprise bean reference to the `ProductEJB` enterprise bean that is in the same J2EE application unit but in a different ejb-jar file.

```
<entity>
        ...
    <ejb-name>OrderEJB</ejb-name>
    <ejb-class>
        com.wombat.orders.OrderBean
    </ejb-class>
    ...
    <ejb-ref>
        <ejb-ref-name>ejb/Product</ejb-ref-name>
        <ejb-ref-type>Entity</ejb-ref-type>
        <home>com.acme.orders.ProductHome</home>
        <remote>com.acme.orders.Product</remote>
        <ejb-link>../products/product.jar#ProductEJB</ejb-link>
    </ejb-ref>
        ...
</entity>
```

19.3.3 Deployer's responsibility

The Deployer is responsible for the following:

- The Deployer must ensure that all the declared EJB references are bound to the homes of enterprise beans that exist in the operational environment. The Deployer may use, for example, the JNDI `LinkRef` mechanism to create a symbolic link to the actual JNDI name of the target enterprise bean's home.

- The Deployer must ensure that the target enterprise bean is type-compatible with the types declared for the EJB reference. This means that the target enterprise bean must of the type indicated in the `ejb-ref-type` element, and that the home and remote interfaces of the target enterprise bean must be Java type-compatible with the home and remote interfaces declared in the EJB reference.

- If an EJB reference declaration includes the `ejb-link` element, the Deployer must bind the enterprise bean reference to the home of the enterprise bean specified as the link's target.

19.3.4 Container Provider's responsibility

The Container Provider must provide the deployment tools that allow the Deployer to perform the tasks described in the previous subsection. The deployment tools provided by the EJB Container provider must be able to process the information supplied in the `ejb-ref` elements in the deployment descriptor.

At the minimum, the tools must be able to:

- Preserve the application assembly information in the `ejb-link` elements by binding an EJB reference to the home interface of the specified target session or entity bean.

- Inform the Deployer of any unresolved EJB references, and allow him or her to resolve an EJB reference by binding it to a specified compatible target session or entity bean.

19.4 Resource manager connection factory references

A resource manager connection factory is an object that is used to create connections to a resource manager. For example, an object that implements the `javax.sql.DataSource` interface is a resource manager connection factory for `java.sql.Connection` objects which implement connections to a database management system.

This section describes the enterprise bean programming and deployment descriptor interfaces that allow the enterprise bean code to refer to resource factories using logical names called *resource manager connection factory references*. The resource manager connection factory references are special entries in the enterprise bean's environment. The Deployer binds the resource manager connection factory references to the actual resource manager connection factories that are configured in the Container. Because these resource manager connection factories allow the Container to affect resource management, the connections acquired through the resource manager connection factory references are called *managed resources* (e.g., these resource manager connection factories allow the Container to implement connection pooling and automatic enlistment of the connection with a transaction).

19.4.1 Bean Provider's responsibilities

This subsection describes the Bean Provider's view of locating resource factories and defines his or her responsibilities.

19.4.1.1 Programming interfaces for resource manager connection factory references

The Bean Provider must use resource manager connection factory references to obtain connections to resources as follows.

- Assign an entry in the enterprise bean's environment to the resource manager connection factory reference. (See subsection 19.4.1.2 for information on how resource manager connection factory references are declared in the deployment descriptor.)

- *The EJB specification recommends, but does not require, that all resource manager connection factory references be organized in the subcontexts of the bean's environment, using a different subcontext for each resource manager type. For example, all JDBC™ DataSource references might be declared in the `java:comp/env/jdbc` subcontext, and all JMS connection factories in the `java:comp/env/jms` subcontext. Also, all JavaMail connection factories might be declared in the `java:comp/env/mail` subcontext and all URL connection factories in the `java:comp/env/url` subcontext.*

- Lookup the resource manager connection factory object in the enterprise bean's environment using the JNDI interface.

- Invoke the appropriate method on the resource manager connection factory to obtain a connection to the resource. The factory method is specific to the resource type. It is possible to obtain multiple connections by calling the factory object multiple times.

The Bean Provider can control the shareability of the connections acquired from the resource manager connection factory. By default, connections to a resource manager are shareable across other enterprise beans in the application that use the same resource in the same transaction context. The Bean Provider can specify that connections obtained from a resource manager connection factory reference are not shareable by specifying the value of the `res-sharing-scope` deployment descriptor element to be `Unshareable`. The sharing of connections to a resource manager allows the container to optimize the use of connections and enables the container's use of local transaction optimizations.

The Bean Provider has two choices with respect to dealing with associating a principal with the resource manager access:

- Allow the Deployer to set up principal mapping or resource manager sign-on information. In this case, the enterprise bean code invokes a resource manager connection factory method that has no security-related parameters.

- Sign on to the resource manager from the bean code. In this case, the enterprise bean invokes the appropriate resource manager connection factory method that takes the sign-on information as method parameters.

The Bean Provider uses the `res-auth` deployment descriptor element to indicate which of the two resource manager authentication approaches is used.

We expect that the first form (i.e., letting the Deployer set up the resource manager sign-on information) will be the approach used by most enterprise beans.

The following code sample illustrates obtaining a JDBC connection.

```
public class EmployeeServiceBean implements SessionBean {
     EJBContext ejbContext;

     public void changePhoneNumber(...) {
          ...

          // obtain the initial JNDI context
          Context initCtx = new InitialContext();

          // perform JNDI lookup to obtain resource manager
          // connection factory
          javax.sql.DataSource ds = (javax.sql.DataSource)
               initCtx.lookup("java:comp/env/jdbc/EmployeeAppDB");

          // Invoke factory to obtain a connection. The security
          // principal is not given, and therefore
          // it will be configured by the Deployer.
          java.sql.Connection con = ds.getConnection();
          ...
     }
}
```

19.4.1.2 Declaration of resource manager connection factory references in deployment descriptor

Although a resource manager connection factory reference is an entry in the enterprise bean's environment, the Bean Provider must not use an `env-entry` element to declare it.

Instead, the Bean Provider must declare all the resource manager connection factory references in the deployment descriptor using the `resource-ref` elements. This allows the ejb-jar consumer (i.e. Application Assembler or Deployer) to discover all the resource manager connection factory references used by an enterprise bean.

Each `resource-ref` element describes a single resource manager connection factory reference. The `resource-ref` element consists of the `description` element; the mandatory `res-ref-name`, `res-type`, and `res-auth` elements; and the optional `res-sharing-scope` element. The `res-ref-name` element contains the name of the environment entry used in the enterprise bean's code. The name of the environment entry is relative to the `java:comp/env` context (e.g., the name should be `jdbc/EmployeeAppDB` rather than `java:comp/env/jdbc/EmployeeAppDB`). The `res-type` element contains the Java type of the resource manager connection factory that the enterprise bean code expects. The `res-auth` element indicates whether the enterprise bean code performs resource manager sign-on programmatically, or whether the Container signs on to the resource manager using the principal mapping information supplied by the Deployer. The Bean Provider indicates the sign-on responsibility by setting the value of the `res-auth` element to `Application` or `Con-`

tainer. The res-sharing-scope element indicates whether connections to the resource manager obtained through the given resource manager connection factory reference can be shared or whether connections are unshareable. The value of the res-sharing-scope element is Shareable or Unshareable. If the res-sharing-scope element is not specified, connections are assumed to be shareable.

A resource manager connection factory reference is scoped to the enterprise bean whose declaration contains the resource-ref element. This means that the resource manager connection factory reference is not accessible from other enterprise beans at runtime, and that other enterprise beans may define resource-ref elements with the same res-ref-name without causing a name conflict.

The type declaration allows the Deployer to identify the type of the resource manager connection factory.

Note that the indicated type is the Java type of the resource factory, not the Java type of the resource.

The following example is the declaration of resource manager connection factory references used by the EmployeeService enterprise bean illustrated in the previous subsection.

```
...
<enterprise-beans>
     <session>
          ...
          <ejb-name>EmployeeService</ejb-name>
          <ejb-class>
               com.wombat.empl.EmployeeServiceBean
          </ejb-class>
          ...
          <resource-ref>
               <description>
                    A data source for the database in which
                    the EmployeeService enterprise bean will
                    record a log of all transactions.
               </description>
               <res-ref-name>jdbc/EmployeeAppDB</res-ref-name>
               <res-type>javax.sql.DataSource</res-type>
               <res-auth>Container</res-auth>
               <res-sharing-scope>Shareable</res-sharing-scope>
          </resource-ref>
          ...
     </session>
</enterprise-beans>
...
```

The following example illustrates the declaration of the JMS resource manager connection factory references used by the example on page 341.

```
...
<enterprise-beans>
     <session>
               ...
          ...
          <resource-ref>
               <description>
                    A queue connection factory used by the
                    MySession enterprise bean to send
                    notifications.
               </description>
               <res-ref-name>jms/qConnFactory</res-ref-name>
               <res-type>javax.jms.QueueConnectionFactory
                    </res-type>
               <res-auth>Container</res-auth>
               <res-sharing-scope>Unshareable</res-sharing-scope>
          </resource-ref>
          ...
     </session>
</enterprise-beans>
...
```

19.4.1.3 Standard resource manager connection factory types

The Bean Provider must use the `javax.sql.DataSource` resource manager connection factory type for obtaining JDBC connections, and the `javax.jms.QueueConnectionFactory` or the `javax.jms.TopicConnectionFactory` for obtaining JMS connections.

The Bean Provider must use the `javax.mail.Session` resource manager connection factory type for obtaining JavaMail connections, and the `java.net.URL` resource manager connection factory type for obtaining URL connections.

It is recommended that the Bean Provider names JDBC data sources in the `java:comp/env/jdbc` subcontext, and JMS connection factories in the `java:comp/env/jms` subcontext. It is also recommended that the Bean Provider names all JavaMail connection factories in the `java:comp/env/mail` subcontext, and all URL connection factories in the `java:comp/env/url` subcontext.

The Connector mechanism allows an enterprise bean to use the API described in this section to obtain resource objects that provide access to additional back-end systems. See [12].

19.4.2 Deployer’s responsibility

The Deployer uses deployment tools to bind the resource manager connection factory references to the actual resource factories configured in the target operational environment.

The Deployer must perform the following tasks for each resource manager connection factory reference declared in the deployment descriptor:

- Bind the resource manager connection factory reference to a resource manager connection factory that exists in the operational environment. The Deployer may use, for example, the JNDI `LinkRef` mechanism to create a symbolic link to the actual JNDI name of the resource manager connection factory. The resource manager connection factory type must be compatible with the type declared in the `res-type` element.

- Provide any additional configuration information that the resource manager needs for opening and managing the resource. The configuration mechanism is resource-manager specific, and is beyond the scope of this specification.

- If the value of the `res-auth` element is `Container`, the Deployer is responsible for configuring the sign-on information for the resource manager. This is performed in a manner specific to the EJB Container and resource manager; it is beyond the scope of this specification.

For example, if principals must be mapped from the security domain and principal realm used at the enterprise beans application level to the security domain and principal realm of the resource manager, the Deployer or System Administrator must define the mapping. The mapping is performed in a manner specific to the EJB Container and resource manager; it is beyond the scope of the current EJB specification.

19.4.3 Container provider responsibility

The EJB Container provider is responsible for the following:

- Provide the deployment tools that allow the Deployer to perform the tasks described in the previous subsection.

- Provide the implementation of the resource manager connection factory classes for the resource managers that are configured with the EJB Container.

- If the Bean Provider sets the `res-auth` of a resource manager connection factory reference to `Application`, the Container must allow the bean to perform explicit programmatic sign-on using the resource manager's API.

- If the Bean Provider sets the `res-sharing-scope` of a resource manager connection factory reference to `Unshareable`, the Container must not attempt to share the connections obtained from the resource manager connection factory *reference*[36].

- The Container must provide tools that allow the Deployer to set up resource manager sign-on information for the resource manager references whose `res-auth` element is set to `Container`. The minimum requirement is that the Deployer must be able to specify the user/password information for each resource manager connection factory reference declared by the enterprise bean, and the Container must be able to use the user/password combination for user

[36] Connections obtained from the same resource manager connection factory through a different resource manager connection factory reference may be shareable.

authentication when obtaining a connection to the resource by invoking the resource manager connection factory.

Although not required by the EJB specification, we expect that Containers will support some form of a single sign-on mechanism that spans the application server and the resource managers. The Container will allow the Deployer to set up the resource managers such that the EJB caller principal can be propagated (directly or through principal mapping) to a resource manager, if required by the application.

While not required by the EJB specification, most EJB Container providers also provide the following features:

- A tool to allow the System Administrator to add, remove, and configure a resource manager for the EJB Server.
- A mechanism to pool connections to the resources for the enterprise beans and otherwise manage the use of resources by the Container. The pooling must be transparent to the enterprise beans.

19.4.4 System Administrator's responsibility

The System Administrator is typically responsible for the following:

- Add, remove, and configure resource managers in the EJB Server environment.

In some scenarios, these tasks can be performed by the Deployer.

19.5 Resource environment references

This section describes the programming and deployment descriptor interfaces that allow the Bean Provider to refer to administered objects that are associated with resources (for example, JMS Destinations) by using "logical" names called *resource environment references*. Resource environment references are special entries in the enterprise bean's environment. The Deployer binds the resource environment references to administered objects in the target operational environment.

19.5.1 Bean Provider's responsibilities

This subsection describes the Bean Provider's view and responsibilities with respect to resource environment references.

19.5.1.1 Resource environment reference programming interfaces

The Bean Provider must use resource environment references to locate administered objects, such as JMS Destinations, which are associated with resources, as follows.

- Assign an entry in the enterprise bean's environment to the reference. (See subsection 19.5.1.2 for information on how resource environment references are declared in the deployment descriptor.)
- *The EJB specification recommends, but does not require, that all resource environment references be organized in the appropriate subcontext of the bean's environment for the resource type (e.g. in the* `java:comp/env/jms` *JNDI context for JMS Destinations).*
- Look up the administered object in the enterprise bean's environment using JNDI.

The following example illustrates how an enterprise bean uses a resource environment reference to locate a JMS Destination .

```
public class StockServiceBean implements SessionBean {

     public void processStockInfo(...) {
          ...

          // Obtain the default initial JNDI context.
          Context initCtx = new InitialContext();

          // Look up the JMS StockQueue in the environment.
          Object result = initCtx.lookup(
               "java:comp/env/jms/StockQueue");

          // Convert the result to the proper type.
          javax.jms.Queue queue = (javax.jms.Queue)result;
     }
}
```

In the example, the Bean Provider of the `StockServiceBean` enterprise bean has assigned the environment entry `jms/StockQueue` as the resource environment reference name to refer to a JMS queue.

19.5.1.2 Declaration of resource environment references in deployment descriptor

Although the resource environment reference is an entry in the enterprise bean's environment, the Bean Provider must not use a `env-entry` element to declare it. Instead, the Bean Provider must declare all references to administered objects associated with resources using the `resource-env-ref` elements of the deployment descriptor. This allows the ejb-jar consumer to discover all the resource environment references used by the enterprise bean.

Each `resource-env-ref` element describes the requirements that the referencing enterprise bean has for the referenced administered object. The `resource-env-ref` element contains an optional `description` element; and the mandatory `resource-env-ref-name` and `resource-env-ref-type` elements.

The `resource-env-ref-name` element specifies the resource environment reference name; its value is the environment entry name used in the enterprise bean code. The name of the environment entry is relative to the `java:comp/env` context (e.g., the name should be `jms/StockQueue` rather than `java:comp/env/jms/StockQueue`). The `resource-env-ref-type` element specifies the expected type of the referenced object. For example, in the case of a JMS Destination, its value must be either `javax.jms.Queue` or `javax.jms.Topic`.

A resource environment reference is scoped to the enterprise bean whose declaration contains the `resource-env-ref` element. This means that the resource environment reference is not accessible to other enterprise beans at runtime, and that other enterprise beans may define `resource-env-ref` elements with the same `resource-env-ref-name` without causing a name conflict.

The following example illustrates the declaration of resource environment references in the deployment descriptor.

```
...
<enterprise-beans>
     <session>
          ...
          <ejb-name>EmployeeService</ejb-name>
          <ejb-class>
               com.wombat.empl.EmployeeServiceBean
          </ejb-class>
          ...
          <resource-env-ref>
               <description>
                    This is a reference to a JMS queue used in the
                    processing of Stock info
               </description>
               <resource-env-ref-name>
                    jms/StockInfo
               </resource-env-ref-name>
               <resource-env-ref-type>
                    javax.jms.Queue
               </resource-env-ref-type>
          </resource-env-ref>
          ...
     </session>
     ...
</enterprise-beans>
...
```

19.5.2 Deployer's responsibility

The Deployer is responsible for the following:

- The Deployer must ensure that all the declared resource environment references are bound to administered objects that exist in the operational environment. The Deployer may use, for example, the JNDI `LinkRef` mechanism to create a symbolic link to the actual JNDI name of the target object.

- The Deployer must ensure that the target object is type-compatible with the type declared for the resource environment reference. This means that the target object must be of the type indicated in the `resource-env-ref-type` element.

19.5.3 Container Provider's responsibility

The Container Provider must provide the deployment tools that allow the Deployer to perform the tasks described in the previous subsection. The deployment tools provided by the EJB Container provider must be able to process the information supplied in the `resource-env-ref` elements in the deployment descriptor.

At the minimum, the tools must be able to inform the Deployer of any unresolved resource environment references, and allow him or her to resolve a resource environment reference by binding it to a specified compatible target object in the environment.

19.6 Deprecated `EJBContext.getEnvironment()` method

The *environment naming context* introduced in EJB 1.1 replaces the EJB 1.0 concept of *environment properties*.

An EJB 2.0 or EJB 1.1 compliant Container is not required to implement support for the EJB 1.0 style environment properties. If the Container does not implement the functionality, it should throw a RuntimeException (or subclass thereof) from the `EJBContext.getEnvironment()` method.

If an EJB 2.0 or EJB 1.1 compliant Container chooses to provide support for the EJB 1.0 style environment properties (so that it can support enterprise beans written to the EJB 1.0 specification), it should implement the support as described below.

When the tools convert the EJB 1.0 deployment descriptor to the EJB 1.1 XML format, they should place the definitions of the environment properties into the `ejb10-properties` subcontext of the environment naming context. The `env-entry` elements should be defined as follows: the `env-entry-name` element contains the name of the environment property, the `env-entry-type` must be `java.lang.String`, and the optional `env-entry-value` contains the environment property value.

For example, an EJB 1.0 enterprise bean with two environment properties `foo` and `bar`, should declare the following `env-entry` elements in its EJB 1.1 format deployment descriptor.

```
...
<env-entry>
     env-entry-name>ejb10-properties/foo</env-entry-name>
     <env-entry-type>java.lang.String</env-entry-type>
</env-entry>
<env-entry>
     <description>bar's description</description>
     <env-entry-name>ejb10-properties/bar</env-entry-name>
     <env-entry-type>java.lang.String</env-entry-type>
     <env-entry-value>bar value</env-entry-value>
</env-entry>
...
```

The Container should provide the entries declared in the `ejb10-properties` subcontext to the instances as a `java.util.Properties` object that the instances obtain by invoking the `EJBContext.getEnvironment()` method.

The enterprise bean uses the EJB 1.0 API to access the properties, as shown by the following example.

```
public class SomeBean implements SessionBean {
     SessionContext ctx;
     java.util.Properties env;

     public void setSessionContext(SessionContext sc) {
          ctx = sc;
          env = ctx.getEnvironment();
     }

     public someBusinessMethod(...) ... {
          String fooValue = env.getProperty("foo");
          String barValue = env.getProperty("bar");
     }
     ...

}
```

19.7 UserTransaction interface

Note: The requirement for the Container to publish the UserTransaction interface in the enterprise bean's JNDI context was added to make the requirements on UserTransaction uniform with the other Java 2, Enterprise Edition application component types.

The Container must make the `UserTransaction` interface available to the enterprise beans that are allowed to use this interface (only session and message-driven beans with bean-managed transaction demarcation are allowed to use this interface) in JNDI under the name `java:comp/UserTransaction`.

The Container must not make the `UserTransaction` interface available to the enterprise beans that are not allowed to use this interface. The Container should throw `javax.naming.NameNotFoundException` if an instance of an enterprise bean that is not allowed to use the `UserTransaction` interface attempts to look up the interface in JNDI.

The following code example

```
public MySessionBean implements SessionBean {
     ...
     public someMethod()
     {
          Context initCtx = new InitialContext();
          UserTransaction utx = (UserTransaction)initCtx.lookup(
               “java:comp/UserTransaction”);
          utx.begin();
          ...
          utx.commit();
     }
     ...
}
```

is functionally equivalent to

```
public MySessionBean implements SessionBean {
     SessionContext ctx;
     ...
     public someMethod()
     {
          UserTransaction utx = ctx.getUserTransaction();
          utx.begin();
          ...
          utx.commit();
     }
     ...
}
```

Chapter 20 Security management

This chapter defines the EJB support for security management.

20.1 Overview

We set the following goals for the security management in the EJB architecture:

- *Lessen the burden of the application developer (i.e. the Bean Provider) for securing the application by allowing greater coverage from more qualified EJB roles. The EJB Container provider provides the implementation of the security infrastructure; the Deployer and System Administrator define the security policies.*
- *Allow the security policies to be set by the Application Assembler or Deployer rather than being hard-coded by the Bean Provider at development time.*
- *Allow the enterprise bean applications to be portable across multiple EJB Servers that use different security mechanisms.*

The EJB architecture encourages the Bean Provider to implement the enterprise bean class without hard-coding the security policies and mechanisms into the business methods. In most cases, the enterprise bean's business methods should not contain any security-related logic. This allows the Deployer to configure the security policies for the application in a way that is most appropriate for the operational environment of the enterprise.

To make the Deployer's task easier, the Application Assembler (which could be the same party as the Bean Provider) may define *security roles* for an application composed of one or more enterprise beans. A security role is a semantic grouping of permissions that a given type of users of the application must have in order to successfully use the application. The Applications Assembler can define (declaratively in the deployment descriptor) *method permissions* for each security role. A method permission is a permission to invoke a specified group of methods of the enterprise beans' home and remote interfaces. The security roles defined by the Application Assembler present a simplified security view of the enterprise beans application to the Deployer—the Deployer's view of the application's security requirements is the small set of security roles rather than a large number of individual methods.

The security principal under which a method invocation is performed is typically that of the component's caller. By specifying a run-as identity, however, it is possible to specify that a different principal be substituted for the execution of the bean's methods and any methods of other enterprise beans that the bean may call. The Application Assembler specifies in the deployment descriptor whether the caller's security identity or a run-as security identity should be used for the execution of the bean's methods. See section 20.3.4.

The Application Assembler should specify the requirements for the caller's principal management of enterprise bean invocations by means of the `security-identity` deployment descriptor element and as part of the description. If `use-caller-identity` is specified as the value of the `security-identity` element, the caller principal is propagated from the caller to the callee. (That is, the called enterprise bean will see the same returned value of the `EJBContext.getCallerPrincipal()` as the calling enterprise bean.) If the `run-as-specified-identity` element is specified, a security principal that has been assigned to the specified security role will be used for the execution of the bean's methods.

The Deployer is responsible for assigning principals, or groups of principals, which are defined in the target operational environment, to the security roles defined by the Application Assembler for the enterprise beans in the deployment descriptor. The Deployer is also responsible for assigning principals for the run-as identities specified by the Application Assembler. The Deployer is further responsible for configuring other aspects of the security management of the enterprise beans, such as principal mapping for inter-enterprise bean calls, and principal mapping for resource manager access.

At runtime, a client will be allowed to invoke a business method only if the principal associated with the client call has been assigned by the Deployer to have at least one security role that is allowed to invoke the business method.

The Container Provider is responsible for enforcing the security policies at runtime, providing the tools for managing security at runtime, and providing the tools used by the Deployer to manage security during deployment.

Because not all security policies can be expressed declaratively, the EJB architecture provides a simple programmatic interface that the Bean Provider may use to access the security context from the business methods.

The following sections define the responsibilities of the individual EJB roles with respect to security management.

20.2 Bean Provider's responsibilities

This section defines the Bean Provider's perspective of the EJB architecture support for security, and defines his or her responsibilities.

20.2.1 Invocation of other enterprise beans

An enterprise bean business method can invoke another enterprise bean via the other bean's remote or home interface. The EJB architecture provides no programmatic interfaces for the invoking enterprise bean to control the principal passed to the invoked enterprise bean.

The management of caller principals passed on *inter-enterprise* bean invocations (i.e. principal delegation) is set up by the Deployer and System Administrator in a Container-specific way. The Bean Provider and Application Assembler should describe all the requirements for the caller's principal management of inter-enterprise bean invocations as part of the description.

20.2.2 Resource access

Section 19.4 defines the protocol for accessing resource managers, including the requirements for security management.

20.2.3 Access of underlying OS resources

The EJB architecture does not define the operating system principal under which enterprise bean methods execute. Therefore, the Bean Provider cannot rely on a specific principal for accessing the underlying OS resources, such as files. (See subsection 20.6.8 for the reasons behind this rule.)

We believe that most enterprise business applications store information in resource managers such as relational databases rather than in resources at the operating system levels. Therefore, this rule should not affect the portability of most enterprise beans.

20.2.4 Programming style recommendations

The Bean Provider should neither implement security mechanisms nor hard-code security policies in the enterprise beans' business methods. Rather, the Bean Provider should rely on the security mechanisms provided by the EJB Container, and should let the Application Assembler and Deployer define the appropriate security policies for the application.

The Bean Provider and Application Assembler may use the deployment descriptor to convey security-related information to the Deployer. The information helps the Deployer to set up the appropriate security policy for the enterprise bean application.

20.2.5 Programmatic access to caller's security context

Note: In general, security management should be enforced by the Container in a manner that is transparent to the enterprise beans' business methods. The security API described in this section should be used only in the less frequent situations in which the enterprise bean business methods need to access the security context information.

The `javax.ejb.EJBContext` interface provides two methods (plus two deprecated methods that were defined in EJB 1.0) that allow the Bean Provider to access security information about the enterprise bean's caller.

```
public interface javax.ejb.EJBContext {
     ...

     //
     // The following two methods allow the EJB class
     // to access security information.
     //
     java.security.Principal getCallerPrincipal();
     boolean isCallerInRole(String roleName);


     //
     // The following two EJB 1.0 methods are deprecated.
     //
     java.security.Identity getCallerIdentity();
     boolean isCallerInRole(java.security.Identity role);

     ...

}
```

The Bean Provider can invoke the `getCallerPrincipal` and `isCallerInRole` methods only in the enterprise bean's business methods for which the Container has a client security context, as specified in Table 2 on page 70, Table 3 on page 80, Table 4 on page 175, and Table 12 on page 259 . If they are invoked when no security context exists, they should throw the `java.lang.IllegalStateException` runtime exception.

The `getCallerIdentity()` and `isCallerInRole(Identity role)` methods were deprecated in EJB 1.1. The Bean Provider must use the `getCallerPrincipal()` and `isCallerInRole(String roleName)` methods for new enterprise beans.

An EJB 2.0 or 1.1 compliant container may choose to implement the two deprecated methods as follows.

- A Container that does not want to provide support for this deprecated method should throw a `RuntimeException` (or subclass of `RuntimeException`) from the `getCallerIdentity()` method.
- A Container that wants to provide support for the `getCallerIdentity()` method should return an instance of a subclass of the `java.security.Identity` abstract class from the method. The `getName()` method invoked on the returned object must return the same value that `getCallerPrincipal().getName()` would return.

- A Container that does not want to provide support for this deprecated method should throw a `RuntimeException` (or subclass of `RuntimeException`) from the `isCallerInRole(Identity identity)` method.

- A Container that wants to implement the `isCallerInRole(Identity identity)` method should implement it as follows:

```
public isCallerInRole(Identity identity) {
    return isCallerInRole(identity.getName());
}
```

20.2.5.1 Use of `getCallerPrincipal()`

The purpose of the getCallerPrincipal() method is to allow the enterprise bean methods to obtain the current caller principal's name. The methods might, for example, use the name as a key to information in a database.

An enterprise bean can invoke the `getCallerPrincipal()` method to obtain a `java.security.Principal` interface representing the current caller. The enterprise bean can then obtain the distinguished name of the caller principal using the `getName()` method of the `java.security.Principal` interface.

> *Note that `getCallerPrincipal()` returns the principal that represents the caller of the enterprise bean, not the principal that corresponds to the run-as security identity for the bean, if any.*

The meaning of the *current caller*, the Java class that implements the `java.security.Principal` interface, and the realm of the principals returned by the `getCallerPrincipal()` method depend on the operational environment and the configuration of the application.

An enterprise may have a complex security infrastructure that includes multiple security domains. The security infrastructure may perform one or more mapping of principals on the path from an EJB caller to the EJB object. For example, an employee accessing his or her company over the Internet may be identified by a userid and password (basic authentication), and the security infrastructure may authenticate the principal and then map the principal to a Kerberos principal that is used on the enterprise's intranet before delivering the method invocation to the EJB object. If the security infrastructure performs principal mapping, the getCallerPrincipal() method returns the principal that is the result of the mapping, not the original caller principal. (In the previous example, getCallerPrincipal() would return the Kerberos principal.) The management of the security infrastructure, such as principal mapping, is performed by the System Administrator role; it is beyond the scope EJB specification.

The following code sample illustrates the use of the `getCallerPrincipal()` method.

```
public class EmployeeServiceBean implements SessionBean {
    EJBContext ejbContext;

    public void changePhoneNumber(...) {
        ...

        // Obtain the default initial JNDI context.
        Context initCtx = new InitialContext();

        // Look up the home interface of the EmployeeRecord
        // enterprise bean in the environment.
        Object result = initCtx.lookup(
            "java:comp/env/ejb/EmplRecord");

        // Convert the result to the proper type.
        EmployeeRecordHome emplRecordHome = (EmployeeRecordHome)
            javax.rmi.PortableRemoteObject.narrow(result,
                    EmployeeRecordHome.class);

        // obtain the caller principal.
        callerPrincipal = ejbContext.getCallerPrincipal();

        // obtain the caller principal's name.
        callerKey = callerPrincipal.getName();

        // use callerKey as primary key to EmployeeRecord finder
        EmployeeRecord myEmployeeRecord =
            emplRecordHome.findByPrimaryKey(callerKey);

        // update phone number
        myEmployeeRecord.changePhoneNumber(...);

        ...
    }
}
```

In the previous example, the enterprise bean obtains the principal name of the current caller and uses it as the primary key to locate an `EmployeeRecord` Entity object. This example assumes that application has been deployed such that the current caller principal contains the primary key used for the identification of employees (e.g., employee number).

20.2.5.2 Use of `isCallerInRole(String roleName)`

The main purpose of the isCallerInRole(String roleName) method is to allow the Bean Provider to code the security checks that cannot be easily defined declaratively in the deployment descriptor using method permissions. Such a check might impose a role-based limit on a request, or it might depend on information stored in the database.

The enterprise bean code uses the `isCallerInRole(String roleName)` method to test whether the current caller has been assigned to a given security role. Security roles are defined by the Application Assembler in the deployment descriptor (see Subsection 20.3.1), and are assigned to principals or principal groups that exist in the operational environment by the Deployer.

Note that `isCallerInRole(String roleName)` tests the principal that represents the caller of the enterprise bean, not the principal that corresponds to the run-as security identity for the bean, if any.

The following code sample illustrates the use of the `isCallerInRole(String roleName)` method.

```
public class PayrollBean ... {
     EntityContext ejbContext;

     public void updateEmployeeInfo(EmplInfo info) {

          oldInfo = ... read from database;

          // The salary field can be changed only by callers
          // who have the security role "payroll"
          if (info.salary != oldInfo.salary &&
               !ejbContext.isCallerInRole("payroll")) {
                    throw new SecurityException(...);
          }
          ...
     }
     ...
}
```

20.2.5.3 Declaration of security roles referenced from the bean's code

The Bean Provider is responsible for declaring in the `security-role-ref` elements of the deployment descriptor all the security role names used in the enterprise bean code. Declaring the security roles references in the code allows the Application Assembler or Deployer to link the names of the security roles used in the code to the security roles defined for an assembled application through the `security-role` elements.

The Bean Provider must declare each security role referenced in the code using the `security-role-ref` element as follows:

- Declare the name of the security role using the `role-name` element. The name must be the security role name that is used as a parameter to the `isCallerInRole(String roleName)` method.
- Optional: Provide a description of the security role in the `description` element.

A security role reference, including the name defined by the `role-name` element, is scoped to the session or entity bean element whose declaration contains the `security-role-ref` element.

The following example illustrates how an enterprise bean's references to security roles are declared in the deployment descriptor.

```
...
<enterprise-beans>
     ...
     <entity>
          <ejb-name>AardvarkPayroll</ejb-name>
          <ejb-class>com.aardvark.payroll.PayrollBean</ejb-class>
          ...
          <security-role-ref>
               <description>
                    This security role should be assigned to the
                    employees of the payroll department who are
                    allowed to update employees' salaries.
               </description>
               <role-name>payroll</role-name>
          </security-role-ref>
          ...
     </entity>
     ...
</enterprise-beans>
...
```

The deployment descriptor above indicates that the enterprise bean `AardvarkPayroll` makes the security check using `isCallerInRole("payroll")` in its business method.

20.3 Application Assembler's responsibilities

The Application Assembler (which could be the same party as the Bean Provider) may define a *security view* of the enterprise beans contained in the ejb-jar file. Providing the security view in the deployment descriptor is optional for the Bean Provider and Application Assembler.

The main reason for the Application Assembler's providing the security view of the enterprise beans is to simplify the Deployer's job. In the absence of a security view of an application, the Deployer needs detailed knowledge of the application in order to deploy the application securely. For example, the Deployer would have to know what each business method does to determine which users can call it. The security view defined by the Application Assembler presents a more consolidated view to the Deployer, allowing the Deployer to be less familiar with the application.

The security view consists of a set of *security roles*. A security role is a semantic grouping of permissions that a given type of users of an application must have in order to successfully use the application.

The Application Assembler defines *method permissions* for each security role. A method permission is a permission to invoke a specified group of methods of the enterprise beans' home and remote interfaces.

It is important to keep in mind that the security roles are used to define the logical security view of an application. They should not be confused with the user groups, users, principals, and other concepts that exist in the target enterprise's operational environment.

In special cases, a qualified Deployer may change the definition of the security roles for an application, or completely ignore them and secure the application using a different mechanism that is specific to the operational environment.

If the Bean Provider has declared any security role references using the `security-role-ref` elements, the Application Assembler must link all the security role references listed in the `security-role-ref` elements to the security roles defined in the `security-role` elements. This is described in more detail in subsection 20.3.3.

20.3.1 Security roles

The Application Assembler can define one or more *security roles* in the deployment descriptor. The Application Assembler then assigns groups of methods of the enterprise beans' home and remote interfaces to the security roles to define the security view of the application.

Because the Application Assembler does not, in general, know the security environment of the operational environment, the security roles are meant to be *logical* roles (or actors), each representing a type of user that should have the same access rights to the application.

The Deployer then assigns user groups and/or user accounts defined in the operational environment to the security roles defined by the Application Assembler.

Defining the security roles in the deployment descriptor is optional[37]l for the Application Assembler. Their omission in the deployment descriptor means that the Application Assembler chose not to pass any security deployment related instructions to the Deployer in the deployment descriptor.

The Application Assembler is responsible for the following:

- Define each security role using a `security-role` element.
- Use the `role-name` element to define the name of the security role.
- Optionally, use the `description` element to provide a description of a security role.

The security roles defined by the `security-role` elements are scoped to the ejb-jar file level, and apply to all the enterprise beans in the ejb-jar file.

[37] If the Application Assembler does not define security roles in the deployment descriptor, the Deployer will have to define security roles at deployment time.

The following example illustrates a security role definition in a deployment descriptor.

```
    ...
<assembly-descriptor>
    <security-role>
        <description>
            This role includes the employees of the
            enterprise who are allowed to access the
            employee self-service application. This role
            is allowed only to access his/her own
            information.
        </description>
        <role-name>employee</role-name>
    </security-role>

    <security-role>
        <description>
            This role includes the employees of the human
            resources department. The role is allowed to
            view and update all employee records.
        </description>
        <role-name>hr-department</role-name>
    </security-role>

    <security-role>
        <description>
            This role includes the employees of the payroll
            department. The role is allowed to view and
            update the payroll entry for any employee.
        </description>
        <role-name>payroll-department</role-name>
    </security-role>

    <security-role>
        <description>
            This role should be assigned to the personnel
            authorized to perform administrative functions
            for the employee self-service application.
            This role does not have direct access to
            sensitive employee and payroll information.
        </description>
        <role-name>admin</role-name>
    </security-role>
    ...
</assembly-descriptor>
```

20.3.2 Method permissions

If the Application Assembler has defined security roles for the enterprise beans in the ejb-jar file, he or she can also specify the methods of the remote and home interface that each security role is allowed to invoke.

Method permissions are defined in the deployment descriptor as a binary relation from the set of security roles to the set of methods of the home and remote interfaces of session and entity beans, including all their superinterfaces (including the methods of the `EJBHome` and `EJBObject` interfaces). The method permissions relation includes the pair (*R*, *M*) if and only if the security role *R* is allowed to invoke the method *M*.

The Application Assembler defines the method permissions relation in the deployment descriptor using the `method-permission` elements as follows.

- Each `method-permission` element includes a list of one or more security roles and a list of one or more methods. All the listed security roles are allowed to invoke all the listed methods. Each security role in the list is identified by the `role-name` element, and each method (or a set of methods, as described below) is identified by the `method` element. An optional description can be associated with a `method-permission` element using the `description` element.

- The method permissions relation is defined as the union of all the method permissions defined in the individual `method-permission` elements.

- A security role or a method may appear in multiple `method-permission` elements.

It is possible that some methods are not assigned to any security roles. If the Application Assembler has assigned some methods (but not all) of an enterprise bean to security roles, the Deployer should assume that the Application Assembler's intent is that the methods not assigned to security roles are not to be callable. The Deployer should configure the enterprise bean's security such that no access is permitted to any method that is not associated with at least one security role.

The `method` element uses the `ejb-name`, `method-name`, and `method-params` elements to denote one or more methods of an enterprise bean's home and remote interfaces. There are three legal styles for composing the `method` element:

Style 1:

```
<method>
      <ejb-name>EJBNAME</ejb-name>
      <method-name>*</method-name>
</method>
```

This style is used for referring to all of the remote and home interface methods of a specified enterprise bean.

Style 2: :

```
<method>
      <ejb-name>EJBNAME</ejb-name>
      <method-name>METHOD</method-name>
</method>
```

This style is used for referring to a specified method of the remote or home interface of the specified enterprise bean. If there are multiple methods with the same overloaded name, this style refers to all of the overloaded methods.

Style 3:

```
<method>
     <ejb-name>EJBNAME</ejb-name>
     <method-name>METHOD</method-name>
     <method-params>
          <method-param>PARAMETER_1</method-param>
          ...
          <method-param>PARAMETER_N</method-param>
     </method-params>
</method>
```

This style is used to refer to a specified method within a set of methods with an overloaded name. The method must be defined in the specified enterprise bean's remote or home interface.

The optional `method-intf` element can be used to differentiate methods with the same name and signature that are defined in both the remote and home interfaces.

The following example illustrates how security roles are assigned method permissions in the deployment descriptor:

```
...
<method-permission>
     <role-name>employee</role-name>
     <method>
          <ejb-name>EmployeeService</ejb-name>
          <method-name>*</method-name>
     </method>
</method-permission>

<method-permission>
     <role-name>employee</role-name>
     <method>
          <ejb-name>AardvarkPayroll</ejb-name>
          <method-name>findByPrimaryKey</method-name>
     </method>
     <method>
          <ejb-name>AardvarkPayroll</ejb-name>
          <method-name>getEmployeeInfo</method-name>
     </method>
     <method>
          <ejb-name>AardvarkPayroll</ejb-name>
          <method-name>updateEmployeeInfo</method-name>
     </method>
</method-permission>

<method-permission>
     <role-name>payroll-department</role-name>
     <method>
          <ejb-name>AardvarkPayroll</ejb-name>
          <method-name>findByPrimaryKey</method-name>
     </method>
     <method>
          <ejb-name>AardvarkPayroll</ejb-name>
          <method-name>getEmployeeInfo</method-name>
     </method>
     <method>
          <ejb-name>AardvarkPayroll</ejb-name>
          <method-name>updateEmployeeInfo</method-name>
     </method>
     <method>
          <ejb-name>AardvarkPayroll</ejb-name>
          <method-name>updateSalary</method-name>
     </method>
</method-permission>

<method-permission>
     <role-name>admin</role-name>
     <method>
          <ejb-name>EmployeeServiceAdmin</ejb-name>
          <method-name>*</method-name>
     </method>
</method-permission>
...
```

20.3.3 Linking security role references to security roles

If the Application Assembler defines the `security-role` elements in the deployment descriptor, he or she is also responsible for linking all the security role references declared in the `security-role-ref` elements to the security roles defined in the `security-role` elements.

The Application Assembler links each security role reference to a security role using the `role-link` element. The value of the `role-link` element must be the name of one of the security roles defined in a `security-role` element.

A `role-link` element must be used even if the value of `role-name` is the same as the value of the `role-link` reference.

The following deployment descriptor example shows how to link the security role reference named `payroll` to the security role named `payroll-department`.

```
...
<enterprise-beans>
     ...
     <entity>
          <ejb-name>AardvarkPayroll</ejb-name>
          <ejb-class>com.aardvark.payroll.PayrollBean</ejb-class>
          ...
          <security-role-ref>
               <description>
                    This role should be assigned to the
                    employees of the payroll department.
                    Members of this role have access to
                    anyone's payroll record.
                    The role has been linked to the
                    payroll-department role.
               </description>
               <role-name>payroll</role-name>
               <role-link>payroll-department</role-link>
          </security-role-ref>
          ...
     </entity>
     ...
</enterprise-beans>
...
```

20.3.4 Specification of security identities in the deployment descriptor

The Application Assembler typically specifies whether the caller's security identity should be used for the execution of the methods of an enterprise bean or whether a specific run-as identity should be used.

The Application Assembler uses the `security-identity` deployment descriptor element for this purpose. The value of the `security-identity` element is either `use-caller-identity` or `run-as-specified-identity`. The `use-caller-identity` element cannot be specified for message-driven beans.

Defining the security identities in the deployment descriptor is optional for the Application Assembler. Their omission in the deployment descriptor means that the Application Assembler chose not to pass any instructions related to security identities to the Deployer in the deployment descriptor.

20.3.4.1 Run-as

The Application Assembler can use the `run-as-specified-identity` element to define a run-as identity for an enterprise bean in the deployment descriptor. The run-as identity applies to the enterprise bean as a whole, that is, to all methods of the enterprise bean's home and remote interfaces or to the `onMessage` method of a message-driven bean, and all internal methods of the bean that they might in turn call.

Because the Application Assembler does not, in general, know the security environment of the operational environment, the run-as identity is designated by a *logical* role-name, which corresponds to one of the security roles defined by the Application Assembler in the deployment descriptor.

The Deployer then assigns a security principal defined in the operational environment to be used as the principal for the run-as identity. The security principal assigned by the Deployer should be a principal that has been assigned to the security role specified by the `role-name` element.

The Application Assembler is responsible for the following in the specification of run-as identities:

- Use the `role-name` element to define the name of the security role.
- Optionally, use the description element to provide a description of the principal that is expected to be bound to the run-as identity in terms of its security role.

The following example illustrates the definition of a run-as identity in the deployment descriptor.

```
...
<enterprise-beans>
     ...
     <session>
          <ejb-name>EmployeeService</ejb-name>
          ...
          <security-identity>
               <run-as-specified-identity>
                    <role-name>admin</role-name>
               </run-as-specified-identity>
          </security-identity>
          ...
     </session>
     ...
</enterprise-beans>
...
```

20.4 Deployer's responsibilities

The Deployer is responsible for ensuring that an assembled application is secure after it has been deployed in the target operational environment. This section defines the Deployer's responsibility with respect to EJB security management.

The Deployer uses deployment tools provided by the EJB Container Provider to read the security view of the application supplied by the Application Assembler in the deployment descriptor. The Deployer's job is to map the security view that was specified by the Application Assembler to the mechanisms and policies used by the security domain in the target operational environment. The output of the Deployer's work includes an application security policy descriptor that is specific to the operational environment. The format of this descriptor and the information stored in the descriptor are specific to the EJB Container.

The following subsections describe the security related tasks performed by the Deployer.

20.4.1 Security domain and principal realm assignment

The Deployer is responsible for assigning the security domain and principal realm to an enterprise bean application.

Multiple principal realms within the same security domain may exist, for example, to separate the realms of employees, trading partners, and customers. Multiple security domains may exist, for example, in application hosting scenarios.

20.4.2 Assignment of security roles

The Deployer assigns principals and/or groups of principals (such as individual users or user groups) used for managing security in the operational environment to the security roles defined in the `security-role` elements of the deployment descriptor.

Typically, the Deployer does not need to change the method permissions assigned to each security role in the deployment descriptor.

The Application Assembler linked all the security role references used in the bean's code to the security roles defined in the `security-role` elements. The Deployer does not assign principals and/or principal groups to the security role references—the principals and/or principals groups assigned to a security role apply also to all the linked security role references. For example, the Deployer of the `AardvarkPayroll` enterprise bean in subsection 20.3.3 would assign principals and/or principal groups to the security-role `payroll-department`, and the assigned principals and/or principal groups would be implicitly assigned also to the linked security role `payroll`.

The EJB architecture does not specify how an enterprise should implement its security architecture. Therefore, the process of assigning the logical security roles defined in the application's deployment descriptor to the operational environment's security concepts is specific to that operational environment. Typically, the deployment process consists of assigning to each security role one or more user groups (or individual users) defined in the operational environment. This assignment is done on a per-application basis. (That is, if multiple independent ejb-jar files use the same security role name, each may be assigned differently.)

20.4.3 Principal delegation

The Deployer is responsible for configuring the principal delegation for inter-component calls. The Deployer must follow any instructions supplied by the Application Assembler (for example, provided in the `run-as-specified-identity` elements of the deployment descriptor, in the `description` elements of the deployment descriptor, or in a deployment manual).

If the `use-caller-identity` element is specified, the caller principal is propagated from one component to another (i.e., the caller principal of the first enterprise bean in a call-chain is passed to the enterprise beans down the chain). This ensures that the returned value of `getCallerPrincipal()` will be the same for all the enterprise beans involved in a call chain. Note that if the security infrastructure performs principal mapping in the course of the call chain, however, the `getCallerPrincipal()` method returns the principal that is the result of the mapping, not the original caller principal.

If the Application Assembler specifies that a run-as identity be used on behalf of a particular enterprise bean, the Deployer must configure the enterprise beans such that the run-as principal is used as the caller principal on any calls that the enterprise bean makes to other beans, and that the run-as principal is propagated along the call-chain of those other beans (in the absence of the specification of any further `run-as-specified-identity` elements).

20.4.4 Security management of resource access

The Deployer's responsibilities with respect to securing resource managers access are defined in subsection 19.4.2.

20.4.5 General notes on deployment descriptor processing

The Deployer can use the security view defined in the deployment descriptor by the Bean Provider and Application Assembler merely as "hints" and may change the information whenever necessary to adapt the security policy to the operational environment.

Since providing the security information in the deployment descriptor is optional for the Application Assembler, the Deployer is responsible for performing any tasks that have not been done by the Application Assembler. (For example, if the definition of security roles and method permissions is missing in the deployment descriptor, the Deployer must define the security roles and method permissions for the application.) It is not required that the Deployer store the output of this activity in the standard ejb-jar file format.

20.5 EJB Client Responsibilities

This section defines the rules that the EJB client program must follow to ensure that the security context passed on the client calls, and possibly imported by the enterprise bean, do not conflict with the EJB Server's capabilities for association between a security context and transactions.

These rules are:

- A transactional client cannot change its principal association within a transaction. This rule ensures that all calls from the client within a transaction are performed with the same security context.

- A Session Bean's client must not change its principal association for the duration of the communication with the session object. This rule ensures that the server can associate a security identity with the session instance at instance creation time, and never have to change the security association during the session instance lifetime.

- If transactional requests within a single transaction arrive from multiple clients (this could happen if there are intermediary objects or programs in the transaction call-chain), all requests within the same transaction must be associated with the same security context.

20.6 EJB Container Provider's responsibilities

This section describes the responsibilities of the EJB Container and Server Provider.

20.6.1 Deployment tools

The EJB Container Provider is responsible for providing the deployment tools that the Deployer can use to perform the tasks defined in Section 20.4.

The deployment tools read the information from the deployment descriptor and present the information to the Deployer. The tools guide the Deployer through the deployment process, and present him or her with choices for mapping the security information in the deployment descriptor to the security management mechanisms and policies used in the target operational environment.

The deployment tools' output is stored in an EJB Container specific manner, and is available at runtime to the EJB Container.

20.6.2 Security domain(s)

The EJB Container provides a security domain and one or more principal realms to the enterprise beans. The EJB architecture does not specify how an EJB Server should implement a security domain, and does not define the scope of a security domain.

A security domain can be implemented, managed, and administered by the EJB Server. For example, the EJB Server may store X509 certificates or it might use an external security provider such as Kerberos.

The EJB specification does not define the scope of the security domain. For example, the scope may be defined by the boundaries of the application, EJB Server, operating system, network, or enterprise.

The EJB Server can, but is not required to, provide support for multiple security domains, and/or multiple principal realms.

The case of multiple domains on the same EJB Server can happen when a large server is used for application hosting. Each hosted application can have its own security domain to ensure security and management isolation between applications owned by multiple organizations.

20.6.3 Security mechanisms

The EJB Container Provider must provide the security mechanisms necessary to enforce the security policies set by the Deployer. The EJB specification does not specify the exact mechanisms that must be implemented and supported by the EJB Server.

The typical security functions provided by the EJB Server include:

- *Authentication of principals.*
- *Access authorization for EJB calls and resource manager access.*
- *Secure communication with remote clients (privacy, integrity, etc.).*

20.6.4 Passing principals on EJB calls

The EJB Container Provider is responsible for providing the deployment tools that allow the Deployer to configure the principal delegation for calls from one enterprise bean to another. The EJB Container is responsible for performing the principal delegation as specified by the Deployer.

The EJB Container must be capable of allowing the Deployer to specify that, for all calls from a single application within a single J2EE product, the caller principal is propagated on calls from one enterprise bean to another (i.e., the multiple beans in the call chain will see the same return value from `getCallerPrincipal()`).

This requirement is necessary for applications that need a consistent return value of getCallerPrincipal() across a chain of calls between enterprise beans.

The EJB Container must be capable of allowing the Deployer to specify that a run-as principal be used for the execution of the home and remote methods of a session or entity bean or for the `onMessage` method of a message-driven bean.

20.6.5 Security methods in `javax.ejb.EJBContext`

The EJB Container must provide access to the caller's security context information from the enterprise beans' instances via the `getCallerPrincipal()` and `isCallerInRole(String roleName)` methods. The EJB Container must provide this context information during the execution of a business method invoked via the enterprise bean's remote or home interface, as defined in Table 2 on page 70, Table 3 on page 80, Table 4 on page 175, and Table 12 on page 259.

The Container must ensure that all enterprise bean method invocations received through the home and remote interface are associated with some principal. The Container must never return a null from the `getCallerPrincipal()` method.

20.6.6 Secure access to resource managers

The EJB Container Provider is responsible for providing secure access to resource managers as described in Subsection 19.4.3.

20.6.7 Principal mapping

If the application requires that its clients are deployed in a different security domain, or if multiple applications deployed across multiple security domains need to interoperate, the EJB Container Provider is responsible for the mechanism and tools that allow mapping of principals. The tools are used by the System Administrator to configure the security for the application's environment.

20.6.8 System principal

The EJB specification does not define the "system" principal under which the JVM running an enterprise bean's method executes.

Leaving the principal undefined makes it easier for the EJB Container vendors to provide the runtime support for EJB on top of their existing server infrastructures. For example, while one EJB Container implementation can execute all instances of all enterprise beans in a single JVM, another implementation can use a separate JVM per ejb-jar per client. Some EJB Containers may make the system principal the same as the application-level principal; others may use different principals, potentially from different principal realms and even security domains.

20.6.9 Runtime security enforcement

The EJB Container is responsible for enforcing the security policies defined by the Deployer. The implementation of the enforcement mechanism is EJB Container implementation specific. The EJB Container may, but does not have to, use the Java programming language security as the enforcement mechanism.

For example, to isolate multiple executing enterprise bean instances, the EJB Container can load the multiple instances into the same JVM and isolate them via using multiple class loaders, or it can load each instance into its own JVM and rely on the address space protection provided by the operating system.

The general security enforcement requirements for the EJB Container follow:

- The EJB Container must provide enforcement of the client access control per the policy defined by the Deployer. A caller is allowed to invoke a method if, and only if, the caller principal is assigned **at least one** of the security roles that includes the method in its method permissions definition. (That is, it is not meant that the caller must be assigned **all** the roles associated with the method.) If the Container denies a client access to a business method, the Container must throw the `java.rmi.RemoteExcetion` to the client.

- The EJB Container must isolate an enterprise bean instance from other instances and other application components running on the server. The EJB Container must ensure that other enterprise bean instances and other application components are allowed to access an enterprise bean only via the enterprise bean's remote and home interfaces.

- The EJB Container must isolate an enterprise bean instance at runtime such that the instance does not gain unauthorized access to privileged system information. Such information includes the internal implementation classes of the container, the various runtime state and context maintained by the container, object references of other enterprise bean instances, or resource managers used by other enterprise bean instances. The EJB Container must ensure that the interactions between the enterprise beans and the container are only through the EJB architected interfaces.

- The EJB Container must ensure the security of the persistent state of the enterprise beans.

- The EJB Container must manage the mapping of principals on calls to other enterprise beans or on access to resource managers according to the security policy defined by the Deployer.

- The Container must allow the same enterprise bean to be deployed independently multiple times, each time with a different security policy[38]. The Container must allow multiple-deployed enterprise beans to co-exist at runtime.

20.6.10 Audit trail

The EJB Container may provide a security audit trail mechanism. A security audit trail mechanism typically logs all *java.security.Exception*s. It also logs all denials of access to EJB Servers, EJB Container, EJB remote interfaces, and EJB home interfaces.

20.7 System Administrator's responsibilities

This section defines the security-related responsibilities of the System Administrator. Note that some responsibilities may be carried out by the Deployer instead, or may require cooperation of the Deployer and the System Administrator.

20.7.1 Security domain administration

The System Administrator is responsible for the administration of principals. Security domain administration is beyond the scope of the EJB specification.

Typically, the System Administrator is responsible for creating a new user account, adding a user to a user group, removing a user from a user group, and removing or freezing a user account.

20.7.2 Principal mapping

If the client is in a different security domain than the target enterprise bean, the system administrator is responsible for mapping the principals used by the client to the principals defined for the enterprise bean. The result of the mapping is available to the Deployer.

The specification of principal mapping techniques is beyond the scope of the EJB architecture.

[38] The enterprise bean is installed each time using a different JNDI name.

20.7.3 Audit trail review

If the EJB Container provides an audit trail facility, the System Administrator is responsible for its management.

Chapter 21 Deployment descriptor

This chapter defines the deployment descriptor that is part of the ejb-jar file. Section 21.1 provides an overview of the deployment descriptor. Sections 21.2 through 21.4 describe the information in the deployment descriptor from the perspective of the EJB roles responsible for providing the information. Section 21.5 defines the deployment descriptor's XML DTD.

21.1 Overview

The deployment descriptor is part of the contract between the ejb-jar file producer and consumer. This contract covers both the passing of enterprise beans from the Bean Provider to Application Assembler, and from the Application Assembler to the Deployer.

An ejb-jar file produced by the Bean Provider contains one or more enterprise beans and typically does not contain application assembly instructions. An ejb-jar file produced by an Application Assembler contains one or more enterprise beans, plus application assembly information describing how the enterprise beans are combined into a single application deployment unit.

The J2EE specification defines how enterprise beans and other application components contained in multiple ejb-jar files can be assembled into an application.

The role of the deployment descriptor is to capture the declarative information (i.e. information that is not included directly in the enterprise beans' code) that is intended for the consumer of the ejb-jar file.

There are two basic kinds of information in the deployment descriptor:

- *Enterprise beans' structural* information. Structural information describes the structure of an enterprise bean and declares an enterprise bean's external dependencies. Providing structural information in the deployment descriptor is mandatory for the ejb-jar file producer. The structural information cannot, in general, be changed because doing so could break the enterprise bean's function.

- *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. Providing assembly information in the deployment descriptor is optional for the ejb-jar file producer. Assembly level information can be changed without breaking the enterprise bean's function, although doing so may alter the behavior of an assembled application.

21.2 Bean Provider's responsibilities

The Bean Provider is responsible for providing the structural information for each enterprise bean in the deployment descriptor.

The Bean Provider must use the `enterprise-beans` element to list all the enterprise beans in the ejb-jar file.

The Bean Provider must provide the following information for each enterprise bean:

- **Enterprise bean's name**. The Bean Provider must assign a logical name to each enterprise bean in the ejb-jar file. There is no architected relationship between this name and the JNDI name that the Deployer will assign to the enterprise bean. The Bean Provider specifies the enterprise bean's name in the `ejb-name` element.

- **Enterprise bean's class**. The Bean Provider must specify the fully-qualified name of the Java class that implements the enterprise bean's business methods. The Bean Provider specifies the enterprise bean's class name in the `ejb-class` element.

- **Enterprise bean's home interfaces**. The Bean Provider must specify the fully-qualified name of the enterprise bean's home interface in the `home` element, unless the bean is a Message-driven bean.

- **Enterprise bean's remote interfaces**. The Bean Provider must specify the fully-qualified name of the enterprise bean's remote interface in the `remote` element, unless the bean is a Message-driven bean.

- **Enterprise bean's type**. The enterprise bean types are: session, entity, and message-driven. The Bean Provider must use the appropriate `session`, `entity`, or `message-driven` element to declare the enterprise bean's structural information.

- **Re-entrancy indication**. The Bean Provider must specify whether an entity bean is re-entrant or not. Session beans and Message-driven beans are never re-entrant.

- **Session bean's state management type**. If the enterprise bean is a Session bean, the Bean Provider must use the `session-type` element to declare whether the session bean is stateful or stateless.

- **Session or Message-driven bean's transaction demarcation type**. If the enterprise bean is a Session or a Message-driven bean, the Bean Provider must use the `transaction-type` element to declare whether transaction demarcation is performed by the enterprise bean or by the Container.

- **Entity bean's persistence management**. If the enterprise bean is an Entity bean, the Bean Provider must use the `persistence-type` element to declare whether persistence management is performed by the enterprise bean or by the Persistence Manager.

- **Entity bean's primary key class**. If the enterprise bean is an Entity bean, the Bean Provider specifies the fully-qualified name of the Entity bean's primary key class in the `prim-key-class` element. The Bean Provider *must* specify the primary key class for an Entity with bean-managed persistence.

- **Entity Bean's abstract schema name.** If the enterprise bean is an Entity Bean with container managed persistence and `cmp-version 2.x`, the Bean Provider must specify the abstract schema name of the entity bean using the `abstract-schema-name` element.

- **Container-managed fields**. If the enterprise bean is an Entity bean with container-managed persistence, the Bean Provider must specify the container-managed fields using the `cmp-fields` elements.

- **Dependent classes.** If the enterprise bean is an Entity bean with container-managed persistence and `cmp-version 2.x`, the Bean Provider must specify any dependent object classes involved in container-managed relationships using the `dependents` element.

- **Container-managed relationships.** If the enterprise bean is an Entity bean with container-managed persistence and `cmp-version 2.x`, the Bean Provider must specify the container-managed relationships using the `relationships` element.

- **Finder and select queries.** If the enterprise bean is an Entity bean with container-managed persistence and `cmp-version 2.x`, the Bean Provider must use the `query` element to specify any EJB QL finder or select query other than that for `findByPrimaryKey`.

- **Environment entries**. The Bean Provider must declare all the enterprise bean's environment entries as specified in Subsection 19.2.1.

- **Resource manager connection factory references**.The Bean Provider must declare all the enterprise bean's resource manager connection factory references as specified in Subsection 19.4.1.

- **Resource environment references**. The Bean Provider must declare all the enterprise bean's references to administered objects that are associated with resources as specified in Subsection 19.5.1.

- **EJB references**. The Bean Provider must declare all the enterprise bean's references to the homes of other enterprise beans as specified in Subsection 19.3.1.

- **Security role references**. The Bean Provider must declare all the enterprise bean's references to security roles as specified in Subsection 20.2.5.3.

- **Message-driven bean's destination.** The Bean Provider may provide advice to the Deployer as to the destination type to which a Message-driven bean should be assigned.

- **Message-driven bean's message selector**. The Bean Provider may declare the JMS message selector to be used in determining which messages the Message-driven bean is to receive.

- **Message-driven bean's acknowledgment mode**. The Bean Provider may declare the JMS acknowledgment mode option that should be used for a message-driven bean with bean managed transaction demarcation.

The deployment descriptor produced by the Bean Provider must be well formed in the XML sense, and valid with respect to the DTD in Section 21.5. The content of the deployment descriptor must conform to the semantics rules specified in the DTD comments and elsewhere in this specification. The deployment descriptor must refer to the DTD using the following statement:

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

21.3 Application Assembler's responsibility

The Application Assembler assembles enterprise beans into a single deployment unit. The Application Assembler's input is one or more ejb-jar files provided by one or more Bean Providers, and the output is also one or more ejb-jar files. The Application Assembler can combine multiple input ejb-jar files into a single output ejb-jar file, or split an input ejb-jar file into multiple output ejb-jar files. Each output ejb-jar file is either a deployment unit intended for the Deployer, or a partially assembled application that is intended for another Application Assembler.

The Bean Provider and Application Assembler may be the same person or organization. In such a case, the person or organization performs the responsibilities described both in this and the previous sections.

The Application Assembler may modify the following information that was specified by the Bean Provider:

- **Values of environment entries**. The Application Assembler may change existing and/or define new values of environment properties.

- **Description fields**. The Application Assembler may change existing or create new `description` elements.

- **Relationship names.** If multiple ejb-jar files use the same names for relationships and are merged into a single ejb-jar file, it is the responsibility of the Application Assembler to modify the relationship names defined in the `ejb-relation-name` elements.

In general, the Application Assembler should never modify any of the following.

- **Enterprise bean's name**. The Application Assembler should not change the enterprise bean's name defined in the `ejb-name` element since EJB QL queries may depend on the content of this element.

- **Dependent class name.** The Application Assembler should not change the value of the `dependent-name` element since EJB QL queries may depend on the content of this element**.**

- **Remote enterprise bean's name.** The Application Assembler should not change an enterprise bean's name designated by the `remote-ejb-name` element since EJB QL queries may depend on the content of this element.

- **Role source element**. The Application Assembler should not change the content of an `ejb-name`, `remote-ejb-name` or a `dependent-name` element in the `role-source` element since they are used as references.

If any of these elements must be modified by the Application Assembler in order to resolve name clashes during the merging two ejb-jar files into one, the Application Assembler must also modify all `ejb-ql` query strings that depend on the value of the modified element(s).

The Application Assembler must not, in general, modify any other information listed in Section 21.2 that was provided in the input ejb-jar file.

The Application Assembler may, but is not required to, specify any of the following *application assembly* information:

- **Binding of enterprise bean references**. The Application Assembler may link an enterprise bean reference to another enterprise bean in the ejb-jar file or in the same J2EE application unit. The Application Assembler creates the link by adding the `ejb-link` element to the referencing bean. The Application Assembler uses the ejb-name of the referenced bean for the link. If there are multiple enterprise beans with the same ejb-name, the Application Assembler uses the path name specifying the location of the ejb-jar file that contains the referenced component. The path name is relative to the referencing ejb-jar file. The Application Assembler appends the ejb-name of the referenced bean to the path name separated by "#". In this manner, multiple beans with the same name may be uniquely identified.

- **Security roles**. The Application Assembler may define one or more security roles. The security roles define the *recommended* security roles for the clients of the enterprise beans. The Application Assembler defines the security roles using the `security-role` elements.

- **Method permissions**. The Application Assembler may define method permissions. Method permission is a binary relation between the security roles and the methods of the remote and home interfaces of the enterprise beans. The Application Assembler defines method permissions using the `method-permission` elements.

- **Linking of security role references**. If the Application Assembler defines security roles in the deployment descriptor, the Application Assembler must link the security role references declared by the Bean Provider to the security roles. The Application Assembler defines these links using the `role-link` element.

- **Security identity**. The Application Assembler may specify whether the caller's security identity should be used for the execution of the methods of an enterprise bean or whether a specific run-as security identity should be used.

- **Transaction attributes**. The Application Assembler may define the value of the transaction attributes for the methods of the remote and home interfaces of the enterprise beans that require container-managed transaction demarcation. All Entity beans and the Session and Message-driven beans declared by the Bean Provider as transaction-type `Container` require container-managed transaction demarcation. The Application Assembler uses the `container-transaction` elements to declare the transaction attributes.

- **Message-driven bean message selector**. The Application Assembler may further restrict, but not replace, the value of the `message-selector` element of a Message-driven bean.

If an input ejb-jar file contains application assembly information, the Application Assembler is allowed to change the application assembly information supplied in the input ejb-jar file. (This could happen when the input ejb-jar file was produced by another Application Assembler.)

The deployment descriptor produced by the Bean Provider must be well formed in the XML sense, and valid with respect to the DTD in Section 21.5. The content of the deployment descriptor must conform to the semantic rules specified in the DTD comments and elsewhere in this specification. The deployment descriptor must refer to the DTD using the following statement:

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

21.4 Container Provider's responsibilities

The Container provider provides tools that read and import the information contained in the XML deployment descriptor.

All EJB 2.0 implementations must support EJB 1.1 as well as EJB 2.0 deployment descriptors.

The EJB 1.1 deployment descriptor is defined in Appendix B.

21.5 Deployment descriptor DTD

This section defines the XML DTD for the EJB 2.0 deployment descriptor. The comments in the DTD specify additional requirements for the syntax and semantics that cannot be easily expressed by the DTD mechanism.

The content of the XML elements is in general case sensitive. This means, for example, that

```
<reentrant>True</reentrant>
```

must be used, rather than:

```
<reentrant>true</reentrant>.
```

All valid ejb-jar deployment descriptors must contain the following DOCTYPE declaration:

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

We plan to provide an ejb-jar file verifier that can be used by the Bean Provider and Application Assembler Roles to ensure that an ejb-jar is valid. The verifier would check all the requirements for the ejb-jar file and the deployment descriptor stated by this specification.

```
<!--
This is the XML DTD for the EJB 2.0 deployment descriptor.
-->

<!--
The abstract-schema-name element specifies the name of the abstract
schema type of an entity bean with cmp-version 2.x. It is used in EJB
QL queries.

For example, the abstract-schema-name for an entity bean whose entity
bean class is com.acme.commerce.OrderBean might be OrderBean.

Used in: entity.
-->
<!ELEMENT abstract-schema-name (#PCDATA)>

<!--
The acknowledge-mode element specifies whether JMS AUTO_ACKNOWLEDGE
or DUPS_OK_ACKNOWLEDGE message acknowledgment semantics should be
used for the onMessage message of a message-driven bean that uses bean
managed transaction demarcation.

The acknowledge-mode element must be one of the two following:
     <acknowledge-mode>Auto-acknowledge</acknowledge-mode>
     <acknowledge-mode>Dups-ok-acknowledge</acknowledgemode>

Used in: message-driven
-->
<!ELEMENT acknowledge-mode (#PCDATA)>

<!--
The assembly-descriptor element contains application-assembly infor-
mation.

The application-assembly information consists of the following parts:
the definition of security roles, the definition of method permis-
sions, and the definition of transaction attributes for enterprise
beans with container-managed transaction demarcation.

All the parts are optional in the sense that they are omitted if the
lists represented by them are empty.

Providing an assembly-descriptor in the deployment descriptor is
optional for the ejb-jar file producer.

Used in: ejb-jar
-->
<!ELEMENT assembly-descriptor (security-role*, method-permission*,
          container-transaction*)>

<!--
The cascade-delete element specifies that, within a particular rela-
tionship, the lifetime of a dependent object (or a collection of
dependent objects) is dependent upon the lifetime of an entity bean or
```

```
dependent object. The cascade-delete element can only be specified in
an ejb-relationship-role element in which the role-source element
specifies a dependent object class. The cascade-delete element can
only be specified for an ejb-relationship-role element contained in
an ejb-relation element in which the other ejb-relationship-role ele-
ment specifies a multiplicity of One.

Used in: ejb-relationship-role
-->
<!ELEMENT cascade-delete EMPTY>

<!--
The cmp-field element describes a container-managed field. The field
element includes an optional description of the field, and the name of
the field.

Used in: entity
-->
<!ELEMENT cmp-field (description?, field-name)>

<!--
The cmp-version element specifies the version of an entity bean with
container-managed persistence.

The cmp-version element must be one of the two following:
     <cmp-version>1.x</cmp-version>
     <cmp-version>2.x</cmp-version>

The default value of the cmp-version element is 2.x.

Used in: entity
-->
<!ELEMENT cmp-version (#PCDATA)>

<!--
The cmr-field element describes the bean provider's view of a rela-
tionship. It consists of an optional description, and the name and the
class type of a field in the source of a role of a relationship. The
cmr-field-name element corresponds to the name used for the get and
set accessor methods for the relationship. The cmr-field-type element
is used only for collection-valued cmr-fields. It specifies the type
of the collection that is used.

Used in: ejb-relationship-role
-->
<!ELEMENT cmr-field (description?, cmr-field-name, cmr-field-type?)>

<!--
The cmr-field-name element specifies the name of a relationship field
in the entity bean or dependent object class. The name of the
cmr-field must begin with a lowercase letter. This field is accessed
by methods whose names consists of the name of the field specified by
cmr-field-name in which the first letter is uppercased, prefixed by
"get" or "set".

Used in: cmr-field
-->
<!ELEMENT cmr-field-name (#PCDATA)>
```

```
<!--
The cmr-field-type element specifies the class of a collection-valued
relationship field in the entity bean or dependent class. The value of
the cmr-field-type element must be either: java.util.Collection or
java.util.Set.

Used in: cmr-field
-->
<!ELEMENT cmr-field-type (#PCDATA)>

<!--
The container-transaction element specifies how the container must
manage transaction scopes for the enterprise bean's method invoca-
tions. The element consists of an optional description, a list of
method elements, and a transaction attribute. The transaction
attribute is to be applied to all the specified methods.

Used in: assembly-descriptor
-->
<!ELEMENT container-transaction (description?, method+,
         trans-attribute)>

<!--
The dependent element specifies a dependent object class of an entity
bean with container managed persistence. The element consists of an
optional description; the dependent object's class; the unique name
that is used for the dependent object class; a list of the con-
tainer-managed fields of the dependent object class; a list of
pk-field elements if the Bean Provider wishes to specify the primary
key fields of the dependent object; and an optional list of query ele-
ments. The values specified by the pk-field elements must be a subset
of the field-names of the cmp-field elements. The query elements, if
present, must specify ejbSelect methods.

Used in: dependents.

Example:
     <dependent>
          <dependent-class>com.acme.Address</dependent-class>
          <dependent-name>Address</dependent-name>
          <cmp-field><field-name>street</field-name></cmp-field>
          <cmp-field><field-name>city</field-name></cmp-field>
          <cmp-field><field-name>zip</field-name></cmp-field>
          <cmp-field><field-name>country</field-name></cmp-field>
     </dependent>

-->
<!ELEMENT dependent (description?, dependent-class, dependent-name,
cmp-field*, pk-field*, query*)>

<!--
The dependents element contains an optional description and the dec-
laration of one or more dependent object classes that are used
(directly or indirectly) in relationships with entity beans with con-
tainer managed persistence.

Used in: ejb-jar.
-->
<!ELEMENT dependents (description?, dependent+)>
```

```
<!--
The dependent-class element contains the fully qualified name of the
dependent object class.

Used in: dependent.
-->
<!ELEMENT dependent-class (#PCDATA)>

<!--
The dependent-name element specifies a name that uniquely designates
a dependent object class.

Used in: dependent, role-source.
-->
<!ELEMENT dependent-name (#PCDATA)>

<!--
The description element is used by the ejb-jar file producer to pro-
vide text describing the parent element.

The description element should include any information that the
ejb-jar file producer wants to provide to the consumer of the ejb-jar
file (i.e. to the Deployer). Typically, the tools used by the ejb-jar
file consumer will display the description when processing the parent
element.

Used in: cmp-field, cmr-field, container-transaction, dependent,
dependents, ejb-entity-ref, ejb-jar, ejb-ref, ejb-relation, ejb-rela-
tionship-role, entity, env-entry, message-driven, method, method-per-
mission, relationships, role-source, run-as-specified-identity,
resource-env-ref, resource-ref, security-identity, security-role,
security-role-ref, and session.
-->
<!ELEMENT description (#PCDATA)>

<!--
The destination-type element specifies the type of the JMS destina-
tion. The type is specified by the Java interface expected to be
implemented by the destination.

The destination-type element must be one of the two following:
     <destination-type>javax.jms.Queue</destination-type>
     <destination-type>javax.jms.Topic</destination-type>

Used in: message-driven-destination
-->
<!ELEMENT destination-type (#PCDATA)>

<!--
The display-name element contains a short name that is intended to be
displayed by tools.

Used in: ejb-jar, session, entity, and message-driven

Example:
     <display-name>Employee Self Service</display-name>
-->
<!ELEMENT display-name (#PCDATA)>
```

```
<!--
The ejb-class element contains the fully-qualified name of the enter-
prise bean's class.

Used in: entity, message-driven, and session

Example:
      <ejb-class>com.wombat.empl.EmployeeServiceBean</ejb-class>
-->
<!ELEMENT ejb-class (#PCDATA)>

<!--
The optional ejb-client-jar element specifies a JAR file that con-
tains the class files necessary for a client program to access the
enterprise beans in the ejb-jar file.

Used in: ejb-jar

Example:
     <ejb-client-jar>employee_service_client.jar</ejb-client-jar>
-->
<!ELEMENT ejb-client-jar (#PCDATA)>

<!--
The ejb-entity-ref element, like the ejb-ref element, is used for
declaring a reference to an enterprise bean's home and remote. How-
ever, the ejb-entity-ref element also contains the remote-ejb-name
element, which provides a name for the remote entity bean. This name
is used within relationships and EJB QL queries to refer to the entity
bean specified by the ejb-entity-ref element.

The declaration consists of an optional description; the
remote-ejb-name element; the expected home and remote interfaces of
the referenced enterprise bean; and an optional ejb-link element,
which is used to specify the referenced enterprise bean.

Used in: relationships

Example:
     <ejb-entity-ref>
          <description>
          This is a reference descriptor for an order bean
          </description>
          <remote-ejb-name>OrderEJB</remote-ejb-name>
          <home>com.commercewarehouse.catalog.OrderHome</home>
          <remote>com.commercewarehouse.catalog.Order</remote>
          <ejb-link>../orders/orders.jar#OrderEJB</ejb-link>
     </ejb-entity-ref>
-->
<!ELEMENT ejb-entity-ref (description?, remote-ejb-name,
     home, remote, ejb-link?)>

<!--
The ejb-jar element is the root element of the EJB deployment descrip-
tor. It contains an optional description of the ejb-jar file; optional
display name; optional small icon file name; optional large icon file
name; mandatory structural information about all included enterprise
beans; structural information about any dependent object classes used
```

```
in container-managed relationships; a descriptor for container man-
aged relationships, if any; an optional application-assembly descrip-
tor; and an optional name of an ejb-client-jar file for the ejb-jar.
-->
<!ELEMENT ejb-jar (description?, display-name?, small-icon?,
        large-icon?, enterprise-beans, dependents?,
        relationships?, assembly-descriptor?, ejb-client-jar?)>

<!--
The ejb-link element is used in the ejb-ref and ejb-entity-ref ele-
ments to specify that an EJB reference is linked to another enterprise
bean.

The value of the ejb-link element must be the ejb-name of an enter-
prise bean in the same ejb-jar file or in another ejb-jar file in the
same J2EE application unit.

Alternatively, the name in the ejb-link element may be composed of a
path name specifying the ejb-jar containing the referenced enterprise
bean with the ejb-name of the target bean appended and separated from
the path name by "#". The path name is relative to the jar file con-
taining the referencing component. This allows multiple enterprise
beans with the same ejb-name to be uniquely identified.

Used in: ejb-entity-ref, ejb-ref

Examples:
      <ejb-link>EmployeeRecord</ejb-link>
      <ejb-link>../products/product.jar#ProductEJB</ejb-link>

-->
<!ELEMENT ejb-link (#PCDATA)>

<!--
The ejb-name element specifies an enterprise bean's name. This name is
assigned by the ejb-jar file producer to name the enterprise bean in
the ejb-jar file's deployment descriptor. The name must be unique
among the names of the enterprise beans in the same ejb-jar file.

There is no architected relationship between the ejb-name in the
deployment descriptor and the JNDI name that the Deployer will assign
to the enterprise bean's home.

The name for an entity bean with cmp-version 2.x must conform to the
lexical rules for an NMTOKEN. The name for an entity bean with
cmp-version 2.x must not be a reserved literal in EJB QL.

Used in: entity, message-driven, session, method, and role-source.

Example:
      <ejb-name>EmployeeService</ejb-name>
-->
<!ELEMENT ejb-name (#PCDATA)>

<!--
The ejb-ql element contains the EJB QL query string that defines a
finder or select query. This element is defined within the scope of a
query element whose contents specify the finder or the select method
that uses the query. The content must be a valid EJB QL query string
```

```
for the entity bean or dependent object class for which the query is
specified.

The ejb-ql element must be specified for all queries that are express-
ible in EJB QL.

Used in: query

Example:
<query>
     <query-method>
          <method-name>ejbSelectPendingLineitems</method-name>
          <method-params/>
     </query-method>
     <ejb-ql>SELECT l FROM LineItems l WHERE l.shipped is FALSE
     </ejb-ql>
</query>

-->
<!ELEMENT ejb-ql (#PCDATA)>

<!--
The ejb-ref element is used for the declaration of a reference to
another enterprise bean's home. The declaration consists of an
optional description; the EJB reference name used in the code of the
referencing enterprise bean; the expected type of the referenced
enterprise bean; the expected home and remote interfaces of the ref-
erenced enterprise bean; and an optional ejb-link information.

The optional ejb-link element is used to specify the referenced enter-
prise bean.

Used in: entity, message-driven, and session
-->
<!ELEMENT ejb-ref (description?, ejb-ref-name, ejb-ref-type, home,
          remote, ejb-link?)>

<!--
The ejb-ref-name element contains the name of an EJB reference. The
EJB reference is an entry in the enterprise bean's environment.

It is recommended that name is prefixed with "ejb/".

Used in: ejb-ref

Example:
       <ejb-ref-name>ejb/Payroll</ejb-ref-name>
-->
<!ELEMENT ejb-ref-name (#PCDATA)>

<!--
The ejb-ref-type element contains the expected type of the referenced
enterprise bean.

The ejb-ref-type element must be one of the following:
          <ejb-ref-type>Entity</ejb-ref-type>
          <ejb-ref-type>Session</ejb-ref-type>

Used in: ejb-ref
```

```
-->
<!ELEMENT ejb-ref-type (#PCDATA)>

<!--
The ejb-relation element describes a relationship between two
entity-beans, between an entity bean and a dependent object class, or
between two dependent object classes.  An ejb-relation element con-
tains a description; an optional ejb-relation-name element; and
exactly two relationship role declarations, defined by the ejb-rela-
tionship-role elements. The name of the relationship, if specified,
is unique within the ejb-jar file.

Used in: relationships
-->
<!ELEMENT ejb-relation (description?, ejb-relation-name?,
    ejb-relationship-role, ejb-relationship-role)>

<!--
The ejb-relation-name element provides a unique name for a relation-
ship.

Used in: ejb-relation
-->
<!ELEMENT ejb-relation-name (#PCDATA)>

<!--
The ejb-relationship-role element describes a role within a relation-
ship. There are two roles in each relationship.

The ejb-relationship-role element contains an optional description;
an optional name for the relationship role; a specification of the
multiplicity of the role; an optional specification of cascade-delete
functionality for the role; the role source; and a declaration of the
cmr-field, if any, by means of which the other side of the relation-
ship is accessed from the perspective of the role source.

The multiplicity and role-source element are mandatory.

The role-source element designates an entity-bean or dependent object
class by means of an ejb-name, remote-ejb-name or dependent-name ele-
ment. For bidirectional relationships, both roles of a relationship
must declare a role-source element that specifies a cmr-field in terms
of which the relationship is accessed. The lack of a cmr-field element
in an ejb-relationship-role specifies that the relationship is unidi-
rectional in navigability and that entity bean or the dependent class
that participates in the relationship is "not aware" of the relation-
ship.

Used in: ejb-relation

Example:
    <ejb-relation>
        <ejb-relation-name>Product-LineItem</ejb-relation-name>
        <ejb-relationship-role>
            <ejb-relationship-role-name>
            product-has-lineitems
            </ejb-relationship-role-name>
            <multiplicity>One</multiplicity>
            <role-source>
```

```
                    <ejb-name>ProductEJB</ejb-name>
                </role-source>
            </ejb-relationship-role>

-->
<!ELEMENT ejb-relationship-role (description?,
     ejb-relationship-role-name?, multiplicity, cascade-delete?,
     role-source, cmr-field?)>

<!--
The ejb-relationship-role-name element defines a name for a role that
is unique within an ejb-relation. Different relationships can use the
same name for a role.

Used in: ejb-relationship-role
-->
<!ELEMENT ejb-relationship-role-name (#PCDATA)>

<!--
The enterprise-beans element contains the declarations of one or more
enterprise beans.
-->
<!ELEMENT enterprise-beans (session | entity | message-driven)+>

<!--
The entity element declares an entity bean. The declaration consists
of: an optional description; optional display name; optional small
icon file name; optional large icon file name; a unique name assigned
to the enterprise bean in the deployment descriptor; the names of the
entity bean's home and remote interfaces; the entity bean's implemen-
tation class; the entity bean's persistence management type; the
entity bean's primary key class name; an indication of the entity
bean's reentrancy; an optional specification of the entity bean's
cmp-version; an optional specification of the entity bean's abstract
schema name; an optional list of container-managed fields; an
optional specification of the primary key field; an optional declara-
tion of the bean's environment entries; an optional declaration of the
bean's EJB references; an optional declaration of the security role
references; an optional declaration of the security identity to be
used for the execution of the bean's methods; an optional declaration
of the bean's resource manager connection factory references; an
optional declaration of the bean's resource environment references;
an optional set of query declarations for finder and select methods
for an entity bean with cmp-version 2.x.

The optional abstract-schema-name element must be specified for an
entity bean with container managed persistence and cmp-version 2.x.

The optional primkey-field may be present in the descriptor if the
entity's persistence-type is Container.

The optional cmp-version element may be present in the descriptor if
the entity's persistence-type is Container. If the persistence-type
is Container and the cmp-version element is not specified, its value
defaults to 2.x.

The optional query elements must be present if the persistence-type is
Container and the cmp-version is 2.x and query methods other than
findByPrimaryKey have been defined for the entity bean.
```

```
The other elements that are optional are “optional” in the sense that
they are omitted if the lists represented by them are empty.

At least one cmp-field element must be present in the descriptor if
the entity’s persistence-type is Container and the cmp-version is
1.x, and none must not be present if the entity’s persistence-type is
Bean.

Used in: enterprise-beans
-->
<!ELEMENT entity (description?, display-name?, small-icon?,
         large-icon?, ejb-name, home, remote, ejb-class,
         persistence-type, prim-key-class, reentrant, cmp-version?,
         abstract-schema-name?, cmp-field*, primkey-field?,
         env-entry*, ejb-ref*, security-role-ref*,
         security-identity?, resource-ref*, resource-env-ref*,
         query*)>

<!--
The env-entry element contains the declaration of an enterprise
bean’s environment entry. The declaration consists of an optional
description, the name of the environment entry, and an optional value.

Used in: entity, message-driven, and session
-->
<!ELEMENT env-entry (description?, env-entry-name, env-entry-type,
         env-entry-value?)>

<!--
The env-entry-name element contains the name of an enterprise bean’s
environment entry.

Used in: env-entry

Example:
      <env-entry-name>minAmount</env-entry-name>
-->
<!ELEMENT env-entry-name (#PCDATA)>

<!--
The env-entry-type element contains the fully-qualified Java type of
the environment entry value that is expected by the enterprise bean’s
code.

The following are the legal values of env-entry-type: java.lang.Bool-
ean, java.lang.String, java.lang.Integer, java.lang.Double,
java.lang.Byte, java.lang.Short, java.lang.Long, and java.lang.Float.

Used in: env-entry

Example:
      <env-entry-type>java.lang.Boolean</env-entry-type>
-->
<!ELEMENT env-entry-type (#PCDATA)>

<!--
The env-entry-value element contains the value of an enterprise
bean’s environment entry. The value must be a String that is valid for
```

```
the constructor of the specified type that takes a single String
parameter.

Used in: env-entry

Example:
      <env-entry-value>100.00</env-entry-value>
-->
<!ELEMENT env-entry-value (#PCDATA)>

<!--
The field-name element specifies the name of a container managed
field. The name must be a public field of the enterprise bean class or
one of its superclasses.

The name of the cmp-field of an entity bean with cmp-version 2.x must
begin with a lowercase letter. This field is accessed by methods whose
names consists of the name of the field specified by field-name in
which the first letter is uppercased, prefixed by “get” or “set”.

Used in: cmp-field

Example:
      <field-name>firstName</field-Name>
-->
<!ELEMENT field-name (#PCDATA)>

<!--
The home element contains the fully-qualified name of the enterprise
bean’s home interface.

Used in: ejb-ref, ejb-entity-ref, entity, and session

Example:
      <home>com.aardvark.payroll.PayrollHome</home>
-->
<!ELEMENT home (#PCDATA)>

<!--
The large-icon element contains the name of a file containing a large
(32 x 32) icon image. The file name is relative path within the
ejb-jar file.

The image must be either in the JPEG or GIF format, and the file name
must end with the suffix ".jpg" or ".gif" respectively.
The icon can be used by tools.

Example:
     <large-icon>employee-service-icon32x32.jpg</large-icon>
-->
<!ELEMENT large-icon (#PCDATA)>

<!--
The message-driven element declares a message-driven bean. The decla-
ration consists of: an optional description; optional display name;
optional small icon file name; optional large icon file name; a name
assigned to the enterprise bean in the deployment descriptor; the mes-
sage-driven bean’s implementation class; the message-driven bean’s
transaction management type; an optional declaration of the mes-
```

```
sage-driven bean's message selector; an optional declaration of the
acknowledgment mode for the message-driven bean if bean-managed
transaction demarcation is used; an optional declaration of the
intended destination type of the message-driven bean; an optional
declaration of the bean's environment entries; an optional declara-
tion of the bean's EJB references; an optional declaration of the
security identity to be used for the execution of the bean's methods;
an optional declaration of the bean's resource manager connection
factory references; and an optional declaration of the bean's
resource environment references.

Used in: enterprise-beans
-->
<!ELEMENT message-driven (description?, display-name?, small-icon?,
        large-icon?, ejb-name?, ejb-class, transaction-type,
        message-selector?, acknowledge-mode?,
        message-driven-destination?, env-entry*, ejb-ref*,
        security-identity?, resource-ref*, resource-env-ref*)>

<!--
The message-driven-destination element provides advice to the
Deployer as to whether a message-driven bean is intended for a Queue
or a Topic. The declaration consists of: the type of the mes-
sage-driven bean's intended destination and an optional declaration
of whether a durable or non-durable subscription should be used if the
destination-type is javax.jms.Topic.

Used in: message-driven
-->
<!ELEMENT message-driven-destination (destination-type,
        subscription-durability?)>

<!--
The message-selector element is used to specify the JMS message selec-
tor to be used in determining which messages a message-driven bean is
to receive.

Example:
<message-selector>JMSType = 'car' AND color = 'blue' AND weight &gt;
2500</message-selector>

Used in: message-driven
-->
<!ELEMENT message-selector (#PCDATA)>

<!--
The method element is used to denote a method of an enterprise bean's
home or remote interface, or, in the case of message-driven beans, the
bean's onMessage method, or a set of methods. The ejb-name element
must be the name of one of the enterprise beans declared in the
deployment descriptor; the optional method-intf element allows to
distinguish between a method with the same signature that is defined
in both the home and remote interface; the method-name element speci-
fies the method name; and the optional method-params elements iden-
tify a single method among multiple methods with an overloaded method
name.

There are three possible styles of the method element syntax:
```

1.
```
<method>
     <ejb-name>EJBNAME</ejb-name>
     <method-name>*</method-name>
</method>
```

This style is used to refer to all the methods of the specified enterprise bean's home and remote interfaces.

2.
```
<method>
     <ejb-name>EJBNAME</ejb-name>
     <method-name>METHOD</method-name>
</method>>
```

This style is used to refer to the specified method of the specified enterprise bean. If there are multiple methods with the same overloaded name, the element of this style refers to all the methods with the overloaded name.

3.
```
<method>
     <ejb-name>EJBNAME</ejb-name>
     <method-name>METHOD</method-name>
     <method-params>
          <method-param>PARAM-1</method-param>
          <method-param>PARAM-2</method-param>
          ...
          <method-param>PARAM-n</method-param>
     </method-params>
<method>
```

This style is used to refer to a single method within a set of methods with an overloaded name. PARAM-1 through PARAM-n are the fully-qualified Java types of the method's input parameters (if the method has no input arguments, the method-params element contains no method-param elements). Arrays are specified by the array element's type, followed by one or more pair of square brackets (e.g. int[][]).

Used in: method-permission, container-transaction and entity.

Examples:

Style 1: The following method element refers to all the methods of the EmployeeService bean's home and remote interfaces:

```
<method>
   <ejb-name>EmployeeService</ejb-name>
   <method-name>*</method-name>
</method>
```

Style 2: The following method element refers to all the *create* methods of the EmployeeService bean's home interface:

```
<method>
   <ejb-name>EmployeeService</ejb-name>
   <method-name>create</method-name>
</method>
```

Style 3: The following method element refers to the *create(String firstName, String LastName)* method of the EmployeeService bean's home interface.

```
<method>
   <ejb-name>EmployeeService</ejb-name>
   <method-name>create</method-name>
   <method-params>
      <method-param>java.lang.String</method-param>
      <method-param>java.lang.String</method-param>
   </method-params>
</method>
```

The following example illustrates a Style 3 element with more complex parameter types. The method

foobar(char s, int i, int[] iar, mypackage.MyClass mycl, mypackage.MyClass[][] myclaar)

would be specified as:

```
<method>
   <ejb-name>EmployeeService</ejb-name>
   <method-name>foobar</method-name>
   <method-params>
      <method-param>char</method-param>
      <method-param>int</method-param>
      <method-param>int[]</method-param>
      <method-param>mypackage.MyClass</method-param>
      <method-param>mypackage.MyClass[][]</method-param>
   </method-params>
</method>
```

The optional method-intf element can be used when it becomes necessary to differentiate between a method defined in the home interface and a method with the same name and signature that is defined in the remote interface.

For example, the method element

```
<method>
   <ejb-name>EmployeeService</ejb-name>
   <method-intf>Remote</method-intf>
   <method-name>create</method-name>
   <method-params>
      <method-param>java.lang.String</method-param>
      <method-param>java.lang.String</method-param>
   </method-params>
</method>
```

can be used to differentiate the create(String, String) method defined in the remote interface from the create(String, String) method defined in the home interface, which would be defined as

```
<method>
   <ejb-name>EmployeeService</ejb-name>
   <method-intf>Home</method-intf>
   <method-name>create</method-name>
   <method-params>
      <method-param>java.lang.String</method-param>
```

```
            <method-param>java.lang.String</method-param>
         </method-params>
      </method>

   The method-intf element can be used with all three Styles of the
method element usage. For example, the following method element exam-
ple could be used to refer to all the methods of the EmployeeService
bean's home interface.

      <method>
         <ejb-name>EmployeeService</ejb-name>
         <method-intf>Home</method-intf>
         <method-name>*</method-name>
      </method>

-->
<!ELEMENT method (description?, ejb-name, method-intf?, method-name,
             method-params?)>

<!--
The method-intf element allows a method element to differentiate
between the methods with the same name and signature that are defined
in both the remote and home interfaces.

The method-intf element must be one of the following:
         <method-intf>Home</method-intf>
         <method-intf>Remote</method-intf>

Used in: method
-->
<!ELEMENT method-intf (#PCDATA)>

<!--
The method-name element contains a name of an enterprise bean method,
or the asterisk (*) character. The asterisk is used when the element
denotes all the methods of an enterprise bean's remote and home inter-
faces.

Used in: method, query-method
-->
<!ELEMENT method-name (#PCDATA)>

<!--
The method-param element contains the fully-qualified Java type name
of a method parameter.

Used in: method-params
-->
<!ELEMENT method-param (#PCDATA)>

<!--
The method-params element contains a list of the fully-qualified Java
type names of the method parameters.

Used in: method, query-method
-->
<!ELEMENT method-params (method-param*)>

<!--
```

```
The method-permission element specifies that one or more security
roles are allowed to invoke one or more enterprise bean methods. The
method-permission element consists of an optional description, a list
of security role names, and a list of method elements.

The security roles used in the method-permission element must be
defined in the security-role element of the deployment descriptor,
and the methods must be methods defined in the enterprise bean's
remote and/or home interfaces.

Used in: assembly-descriptor
-->
<!ELEMENT method-permission (description?, role-name+, method+)>

<!--
The multiplicity element describes the multiplicity of the role that
participates in a relation.

The multiplicity element must be one of the two following:
         <multiplicity>One</multiplicity>
         <multiplicity>Many</multiplicity>

Used in: ejb-relationship-role
-->
<!ELEMENT multiplicity (#PCDATA)>

<!--
The persistence-type element specifies an entity bean's persistence
management type.

The persistence-type element must be one of the two following:
         <persistence-type>Bean</persistence-type>
         <persistence-type>Container</persistence-type>

Used in: entity
-->
<!ELEMENT persistence-type (#PCDATA)>

<!--
The pk-field element is used to specify the name of a primary key
field for a dependent object class.

The value of the pk-field must be the name of one of the fields
declared in the cmp-field elements for the dependent object class.

Used in: dependent

Example:
   <pk-field>creditCardNumber</pk-field>
-->
<!ELEMENT pk-field (#PCDATA)>

<!--
The prim-key-class element contains the fully-qualified name of an
entity bean's primary key class.

If the definition of the primary key class is deferred to deployment
time, the prim-key-class element should specify java.lang.Object.
```

```
Used in: entity

Examples:
   <prim-key-class>java.lang.String</prim-key-class>
   <prim-key-class>com.wombat.empl.EmployeeID</prim-key-class>
   <prim-key-class>java.lang.Object</prim-key-class>

-->
<!ELEMENT prim-key-class (#PCDATA)>

<!--
The primkey-field element is used to specify the name of the primary
key field for an entity with container-managed persistence.

The primkey-field must be one of the fields declared in the cmp-field
element, and the type of the field must be the same as the primary key
type.

The primkey-field element is not used if the primary key maps to mul-
tiple container-managed fields (i.e. the key is a compound key). In
this case, the fields of the primary key class must be public, and
their names must correspond to the field names of the entity bean
class that comprise the key.

Used in: entity

Example:
   <primkey-field>EmployeeId</primkey-field>
-->
<!ELEMENT primkey-field (#PCDATA)>

<!--
The query element is used to specify a finder or select query. It con-
tains an optional description of the query, the specification of the
finder or select method it is used by, and the EJB QL query string
that defines the query. Queries that are expressible in EJB QL must
use the ejb-ql element to specify the query. If a query is not
expressible in EJB QL, the description element should be used to
describe the semantics of the query and the ejb-ql element should be
empty.

Used in: entity, dependent
-->
<!ELEMENT query (description?, query-method, ejb-ql)>

<!--
The query-method element is used to specify the method for a finder or
select query.

The method-name element specifies the name of a finder or select
method in the entity bean's implementation class or a select method in
the dependent object class.

Each method-param must be defined for a query-method using the
method-params element.

Used in: query

Example:
```

```
   <query>
      <description>Method finds large orders</description>
      <query-method>
         <method-name>findLargeOrders</method-name>
         <method-params></method-params>
      </query-method>
      <ejb-ql>FROM OrderBean o WHERE o.amount &gt; 1000</ejb-ql>
   </query>

-->
<!ELEMENT query-method (method-name, method-params)>

<!--
The reentrant element specifies whether an entity bean is reentrant or
not.

The reentrant element must be one of the two following:
         <reentrant>True</reentrant>
         <reentrant>False</reentrant>

Used in: entity
-->
<!ELEMENT reentrant (#PCDATA)>

<!--
The relationships element describes the relationships in which con-
tainer managed persistence entity beans and dependent objects partic-
ipate. The relationships element contains an optional description; a
list of ejb-entity-ref elements (references to entity beans that par-
ticipate in container managed relationships but whose abstract per-
sistence schemas are not included in the ejb-jar file); and a list of
ejb-relation elements, which specify the container managed relation-
ships.

Used in: ejb-jar
-->
<!ELEMENT relationships (description?, ejb-entity-ref*,
    ejb-relation+)>

<!--
The remote element contains the fully-qualified name of the enter-
prise bean's remote interface.

Used in: ejb-ref, entity, and session

Example:
      <remote>com.wombat.empl.EmployeeService</remote>
-->
<!ELEMENT remote (#PCDATA)>

<!--
The remote-ejb-name element specifies the name of an entity bean that
participates in relationships, but whose abstract persistence schema
is not provided in the ejb-jar file. Remote entity beans may include
entity beans with bean managed persistence, EJB 1.1 entity beans with
container managed persistence, and entity beans that are defined in
another ejb-jar file. The name is declared in the ejb-entity-ref ele-
ment and used to designate a role in a relationship.
```

```
Used in: ejb-entity-ref, role-source
-->
<!ELEMENT remote-ejb-name (#PCDATA)>

<!--
The res-auth element specifies whether the enterprise bean code signs
on programmatically to the resource manager, or whether the Container
will sign on to the resource manager on behalf of the bean. In the
latter case, the Container uses information that is supplied by the
Deployer.

The value of this element must be one of the two following:
         <res-auth>Application</res-auth>
         <res-auth>Container</res-auth>
-->
<!ELEMENT res-auth (#PCDATA)>

<!--
The res-ref-name element specifies the name of a resource manager con-
nection factory reference.

Used in: resource-ref
-->
<!ELEMENT res-ref-name (#PCDATA)>

<!--
The res-sharing-scope element specifies whether connections obtained
through the given resource manager connection factory reference can
be shared. The value of this element, if specified, must be one of the
two following:
         <res-sharing-scope>Shareable</res-sharing-scope>
         <res-sharing-scope>Unshareable</res-sharing-scope>

The default value is Shareable.

Used in: resource-ref
-->
<!ELEMENT res-sharing-scope (#PCDATA)>

<!--
The res-type element specifies the type of the data source. The type
is specified by the Java interface (or class) expected to be imple-
mented by the data source.

Used in: resource-ref
-->
<!ELEMENT res-type (#PCDATA)>

<!--
The resource-env-ref element contains a declaration of an enterprise
bean's reference to an administered object associated with a resource
in the enterprise bean's environment. It consists of an optional
description, the resource environment reference name, and an indica-
tion of the resource environment reference type expected by the enter-
prise bean code.

Used in: entity, message-driven and session

Examples:
```

```
      <resource-env-ref>
         <resource-env-ref-name>jms/StockQueue
            </resource-env-ref-name>
         <resource-env-ref-type>javax.jms.Queue
            </resource-env-ref-type>
      </resource-env-ref>
-->
<!ELEMENT resource-env-ref (description?, resource-env-ref-name,
resource-env-ref-type)>

<!--
The resource-env-ref-name element specifies the name of a resource
environment reference; its value is the environment entry name used in
the enterprise bean code.

Used in: resource-env-ref
-->
<!ELEMENT resource-env-ref-name (#PCDATA)>

<!--
The resource-env-ref-type element specifies the type of a resource
environment reference.

Used in: resource-env-ref
-->
<!ELEMENT resource-env-ref-type (#PCDATA)>

<!--
The resource-ref element contains a declaration of enterprise bean's
reference to an external resource. It consists of an optional descrip-
tion, the resource manager connection factory reference name, the
indication of the resource manager connection factory type expected
by the enterprise bean code, the type of authentication (Application
or Container), and an optional specification of the shareability of
connections obtained from the resource (Shareable or Unshareable).

Used in: entity, message-driven, and session

Example:
      <resource-ref>
         <res-ref-name>jdbc/EmployeeAppDB</res-ref-name>
         <res-type>javax.sql.DataSource</res-type>
         <res-auth>Container</res-auth>
         <res-sharing-scope>Shareable</res-sharing-scope>
      </resource-ref>
-->
<!ELEMENT resource-ref (description?, res-ref-name, res-type,
            res-auth, res-sharing-scope?)>

<!--
The role-link element is used to link a security role reference to a
defined security role. The role-link element must contain the name of
one of the security roles defined in the security-role elements.

Used in: security-role-ref
-->
<!ELEMENT role-link (#PCDATA)>

<!--
```

```
The role-name element contains the name of a security role.

The name must conform to the lexical rules for an NMTOKEN.

Used in: method-permission, security-role, and security-role-ref
-->
<!ELEMENT role-name (#PCDATA)>

<!--
The role-source element designates the source of a role that partici-
pates in a relationship. A role-source element contains a reference
which uniquely identifies an entity bean or dependent object class.
The Bean Provider must ensure that the content of each role-source
element refers to an existing entity bean, entity bean reference, or
dependent object class.

Used in: ejb-relationship-role
-->
<!ELEMENT role-source (description?, (ejb-name|remote-ejb-name|depen-
dent-name))>

<!--
The run-as-specified-identity element specifies the run-as identity
to be used for the execution of the methods of an enterprise bean. It
contains an optional description, and the name of a security role.

Used in: security-identity
-->
<!ELEMENT run-as-specified-identity (description?, role-name)>

<!--
The security-identity element specifies whether the caller's security
identity is to be used for the execution of the methods of the enter-
prise bean or whether a specific run-as identity is to be used. It
contains an optional description and a specification of the security
identity to be used.

Used in: session, entity, message-driven
-->
<!ELEMENT security-identity (description?, (use-caller-iden-
tity|run-as-specified-identity))>

<!--
The security-role element contains the definition of a security role.
The definition consists of an optional description of the security
role, and the security role name.

Used in: assembly-descriptor

Example:
      <security-role>
         <description>
            This role includes all employees who are authorized
            to access the employee service application.
         </description>
         <role-name>employee</role-name>
      </security-role>
-->
<!ELEMENT security-role (description?, role-name)>
```

```
<!--
The security-role-ref element contains the declaration of a security
role reference in the enterprise bean's code. The declaration con-
sists of an optional description, the security role name used in the
code, and an optional link to a defined security role.

The value of the role-name element must be the String used as the
parameter to the EJBContext.isCallerInRole(String roleName) method.

The value of the role-link element must be the name of one of the
security roles defined in the security-role elements.

Used in: entity and session

-->
<!ELEMENT security-role-ref (description?, role-name, role-link?)>

<!--
The session-type element describes whether the session bean is a
stateful session, or stateless session.

The session-type element must be one of the two following:
        <session-type>Stateful</session-type>
        <session-type>Stateless</session-type>
-->
<!ELEMENT session-type (#PCDATA)>

<!--
The session element declares an session bean. The declaration con-
sists of: an optional description; optional display name; optional
small icon file name; optional large icon file name; a name assigned
to the enterprise bean in the deployment description; the names of the
session bean's home and remote interfaces; the session bean's imple-
mentation class; the session bean's state management type; the ses-
sion bean's transaction management type; an optional declaration of
the bean's environment entries; an optional declaration of the bean's
EJB references; an optional declaration of the security role refer-
ences; an optional declaration of the security identity to be used for
the execution of the bean's methods; an optional declaration of the
bean's resource manager connection factory references; and an
optional declaration of the bean's resource environment references.

The elements that are optional are "optional" in the sense that they
are omitted when if lists represented by them are empty.

Used in: enterprise-beans
-->
<!ELEMENT session (description?, display-name?, small-icon?,
        large-icon?, ejb-name, home, remote, ejb-class,
        session-type, transaction-type, env-entry*, ejb-ref*,
        security-role-ref*, security-identity?, resource-ref*,
        resource-env-ref*)>

<!--
The small-icon element contains the name of a file containing a small
(16 x 16) icon image. The file name is relative path within the
ejb-jar file.
```

```
The image must be either in the JPEG or GIF format, and the file name
must end with the suffix ".jpg" or ".gif" respectively.

The icon can be used by tools.

Example:
     <small-icon>employee-service-icon16x16.jpg</small-icon>
-->
<!ELEMENT small-icon (#PCDATA)>

<!--
The subscription-durability element specifies whether a JMS topic
subscription is intended to be durable or nondurable.

The subscription-durability element must be one of the two following:
     <subscription-durability>Durable</subscription-durability>
     <subscription-durability>NonDurable</subscription-durability>

Used in: message-driven-destination
-->
<!ELEMENT subscription-durability (#PCDATA)>

<!--
The transaction-type element specifies an enterprise bean's transac-
tion management type.

The transaction-type element must be one of the two following:
          <transaction-type>Bean</transaction-type>
          <transaction-type>Container</transaction-type>

Used in: session and message-driven
-->
<!ELEMENT transaction-type (#PCDATA)>

<!--
The trans-attribute element specifies how the container must manage
the transaction boundaries when delegating a method invocation to an
enterprise bean's business method.

The value of trans-attribute must be one of the following:
          <trans-attribute>NotSupported</trans-attribute>
          <trans-attribute>Supports</trans-attribute>
          <trans-attribute>Required</trans-attribute>
          <trans-attribute>RequiresNew</trans-attribute>
          <trans-attribute>Mandatory</trans-attribute>
          <trans-attribute>Never</trans-attribute>

Used in: container-transaction
-->
<!ELEMENT trans-attribute (#PCDATA)>

<!--
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.

Used in: security-identity
-->
<!ELEMENT use-caller-identity EMPTY>
```

```
<!--
The ID mechanism is to allow tools that produce additional deployment
information (i.e., information beyond the standard EJB deployment
descriptor information) to store the non-standard information in a
separate file, and easily refer from these tools-specific files to the
information in the standard deployment descriptor.

The EJB architecture does not allow the tools to add the non-standard
information into the EJB deployment descriptor.
-->
<!ATTLIST abstract-schema-name id ID #IMPLIED>
<!ATTLIST acknowledge-mode id ID #IMPLIED>
<!ATTLIST assembly-descriptor id ID #IMPLIED>
<!ATTLIST cascade-delete id ID #IMPLIED>
<!ATTLIST cmp-field id ID #IMPLIED>
<!ATTLIST cmp-version id ID #IMPLIED>
<!ATTLIST cmr-field id ID #IMPLIED>
<!ATTLIST cmr-field-name id ID #IMPLIED>
<!ATTLIST cmr-field-type id ID #IMPLIED>
<!ATTLIST container-transaction id ID #IMPLIED>
<!ATTLIST dependent id ID #IMPLIED>
<!ATTLIST dependents id ID #IMPLIED>
<!ATTLIST dependent-class id ID #IMPLIED>
<!ATTLIST dependent-name id ID #IMPLIED>
<!ATTLIST description id ID #IMPLIED>
<!ATTLIST destination-type id ID #IMPLIED>
<!ATTLIST display-name id ID #IMPLIED>
<!ATTLIST ejb-class id ID #IMPLIED>
<!ATTLIST ejb-client-jar id ID #IMPLIED>
<!ATTLIST ejb-entity-ref id ID #IMPLIED>
<!ATTLIST ejb-jar id ID #IMPLIED>
<!ATTLIST ejb-link id ID #IMPLIED>
<!ATTLIST ejb-name id ID #IMPLIED>
<!ATTLIST ejb-ql id ID #IMPLIED>
<!ATTLIST ejb-ref id ID #IMPLIED>
<!ATTLIST ejb-ref-name id ID #IMPLIED>
<!ATTLIST ejb-ref-type id ID #IMPLIED>
<!ATTLIST ejb-relation id ID #IMPLIED>
<!ATTLIST ejb-relation-name id ID #IMPLIED>
<!ATTLIST ejb-relationship-role id ID #IMPLIED>
<!ATTLIST ejb-relationship-role-name id ID #IMPLIED>
<!ATTLIST enterprise-beans id ID #IMPLIED>
<!ATTLIST entity id ID #IMPLIED>
<!ATTLIST env-entry id ID #IMPLIED>
<!ATTLIST env-entry-name id ID #IMPLIED>
<!ATTLIST env-entry-type id ID #IMPLIED>
<!ATTLIST env-entry-value id ID #IMPLIED>
<!ATTLIST field-name id ID #IMPLIED>
<!ATTLIST home id ID #IMPLIED>
<!ATTLIST large-icon id ID #IMPLIED>
<!ATTLIST message-driven id ID #IMPLIED>
<!ATTLIST message-driven-destination id ID #IMPLIED>
<!ATTLIST message-selector id ID #IMPLIED>
<!ATTLIST method id ID #IMPLIED>
<!ATTLIST method-intf id ID #IMPLIED>
<!ATTLIST method-name id ID #IMPLIED>
<!ATTLIST method-param id ID #IMPLIED>
<!ATTLIST method-params id ID #IMPLIED>
```

```
<!ATTLIST method-permission id ID #IMPLIED>
<!ATTLIST multiplicity id ID #IMPLIED>
<!ATTLIST persistence-type id ID #IMPLIED>
<!ATTLIST pk-field id ID #IMPLIED>
<!ATTLIST prim-key-class id ID #IMPLIED>
<!ATTLIST primkey-field id ID #IMPLIED>
<!ATTLIST query id ID #IMPLIED>
<!ATTLIST query-method id ID #IMPLIED>
<!ATTLIST reentrant id ID #IMPLIED>
<!ATTLIST relationships id ID #IMPLIED>
<!ATTLIST remote id ID #IMPLIED>
<!ATTLIST remote-ejb-name id ID #IMPLIED>
<!ATTLIST res-auth id ID #IMPLIED>
<!ATTLIST res-ref-name id ID #IMPLIED>
<!ATTLIST res-sharing-scope id ID #IMPLIED>
<!ATTLIST res-type id ID #IMPLIED>
<!ATTLIST resource-env-ref id ID #IMPLIED>
<!ATTLIST resource-env-ref-name id ID #IMPLIED>
<!ATTLIST resource-env-ref-type id ID #IMPLIED>
<!ATTLIST resource-ref id ID #IMPLIED>
<!ATTLIST role-link id ID #IMPLIED>
<!ATTLIST role-name id ID #IMPLIED>
<!ATTLIST role-source id ID #IMPLIED>
<!ATTLIST run-as-specified-identity id ID #IMPLIED>
<!ATTLIST security-identity id ID #IMPLIED>
<!ATTLIST security-role id ID #IMPLIED>
<!ATTLIST security-role-ref id ID #IMPLIED>
<!ATTLIST session-type id ID #IMPLIED>
<!ATTLIST session id ID #IMPLIED>
<!ATTLIST small-icon id ID #IMPLIED>
<!ATTLIST subscription-durability id ID #IMPLIED>
<!ATTLIST transaction-type id ID #IMPLIED>
<!ATTLIST trans-attribute id ID #IMPLIED>
<!ATTLIST use-caller-identity id ID #IMPLIED>
```

Chapter 22 Ejb-jar file

The ejb-jar file is the standard format for the packaging of enterprise Beans. The ejb-jar file format is used to package un-assembled enterprise beans (the Bean Provider's output), and to package assembled applications (the Application Assembler's output).

22.1 Overview

The ejb-jar file format is the contract between the Bean Provider and the Application Assembler, and between the Application Assembler and the Deployer.

An ejb-jar file produced by the Bean Provider contains one or more enterprise beans that typically do not contain application assembly instructions. An ejb-jar file produced by an Application Assembler (which can be the same person or organization as the Bean Provider) contains one or more enterprise beans, plus application assembly information describing how the enterprise beans are combined into a single application deployment unit.

22.2 Deployment descriptor

The ejb-jar file must contain the deployment descriptor in the format defined in Chapter 21. The deployment descriptor must be stored with the name `META-INF/ejb-jar.xml` in the ejb-jar file.

22.3 Class files

The ejb-jar file must contain, either by inclusion or by reference, the class files of each enterprise bean as follows:

- The enterprise bean class.
- The enterprise bean home and remote interfaces, unless the bean is a message-driven bean.
- The primary key class if the bean is an entity bean.

The ejb-jar file must also contain, either by inclusion or by reference, the class files for all the classes and interfaces that each enterprise bean class and the remote and home interfaces depend on, except J2EE and J2SE classes. This includes their superclasses and superinterfaces, dependent classes, and the classes and interfaces used as method parameters, results, and exceptions.

The ejb-jar file also contains the client view of the enterprise beans that are referenced by the enterprise beans whose implementations (enterprise bean classes) are provided in the ejb-jar file. The client view of an enterprise bean is defined in Section 4.2.1, and is comprised of the home and remote interfaces of the referenced enterprise bean and other classes that these interfaces depend on, such as their superclasses and superinterfaces, the classes and interfaces used as method parameters, results, and exceptions. The serializable value classes, including the classes which may be used as members of a collection in a remote method call to an enterprise bean, are part of the client view.

It is the responsibility of the Application Assembler to package all the application value classes that are needed for the client view in the ejb-jar file, either by inclusion or by reference. An example of an application value class might be an Address class used as a parameter in a method call.

It is the responsibility of the Container Provider's deployment tools to generate the additional value classes and make them available at deployment time as part of the client view. The additional value classes are the subclasses of the application value classes that may be needed as return values of method calls generated by the Container, as well as the system value classes. An example of a system value class might be an implementation class generated for the java.util.Collection interface by the Container which is returned as a result of a finder method. System value classes include the classes that implement the handle and metadata interfaces of an enterprise bean.

The Application Assembler must not package the stubs of the EJBHome and EJBObject interfaces in the ejb-jar file. This includes the stubs for the enterprise beans whose implementations are provided in the ejb-jar file as well as the referenced enterprise beans. Generating the stubs is the responsibility of the Container. The stubs are typically generated by the Container Provider's deployment tools for each class that extends the EJBHome or EJBObject interfaces, or they may be generated by the Container at runtime.

An ejb-jar file does not have to physically include the class files if the classes are defined in another jar file that is named in the Class-Path attribute in the Manifest file of the referencing ejb-jar file or in the transitive closure of such Class-Path references.

22.4 ejb-client JAR file

The ejb-jar file producer can create an ejb-client JAR file for the ejb-jar file. The ejb-client JAR file contains all the class files that a client program needs to use the client view of the enterprise beans that are contained in the ejb-jar file. The classes that comprise the client view are described in Section 22.3. If this option is used, it is the responsibility of the Application Assembler to include all the classes necessary to comprise the client view of an enterprise bean in the ejb-client JAR file. It is the responsibility of the container to provide the necessary stubs and system value classes as described in Section 22.3.

The ejb-client JAR file is specified in the deployment descriptor of the ejb-jar file using the optional `ejb-client-jar` element. The value of the `ejb-client-jar` element is the path name specifying the location of the ejb-client JAR file in the containing J2EE Enterprise Application Archive (.ear) file. The path name is relative to the location of the referencing ejb-jar file. When a client is contained in the same (.ear) file as the referenced enterprise beans (i.e. when a client is in the same application as the referenced enterprise beans), the Deployer should ensure that the specified ejb-client JAR file is accessible to the client program's class loader. If the `ejb-client-jar` element is not specified, the deployer of the component should make the entire ejb-jar file accessible to the client's class loader.

When clients refer to enterprise beans that are not part of the same (.ear) file, the jar file which contains the client, e.g. an ejb-jar file, must contain, either by inclusion or by reference, all the client view classes of the referenced beans, including the system and additional value classes that are generated at deployment time by the Container Provider's tools. The client view classes may have been packaged in an ejb-client JAR.

The EJB specification does not specify whether the ejb-jar file should include by copy or by reference the classes that are in the ejb-client JAR file. If the by-copy approach is used, the producer simply includes all the class files in the ejb-client JAR file also in the ejb-jar file. If the by-reference approach is used, the ejb-jar file producer does not duplicate the content of the ejb-client JAR file in the ejb-jar file, but instead uses a Manifest Class-Path entry in the ejb-jar file to specify that the ejb-jar file depends on the ejb-client JAR at runtime. The use of the Class-Path entries in the JAR files is explained in the Java 2 Platform, Enterprise Edition specification [9].

22.5 Deprecated in EJB 1.1

This section describes the deployment information that was defined in EJB 1.0, and was deprecated in EJB 1.1.

22.5.1 ejb-jar Manifest

The JAR Manifest file is no longer used by the EJB architecture to identify the enterprise beans contained in an ejb-jar file.

EJB 1.0 used the Manifest file to identify the individual enterprise beans that were included in the ejb-jar file. As of EJB 1.1, the enterprise beans are identified in the deployment descriptor, so the information in the Manifest is no longer needed.

22.5.2 Serialized deployment descriptor JavaBeans™ components

The mechanism of using serialized JavaBeans components as deployment descriptors has been replaced by the XML-based deployment descriptor.

Chapter 23 Runtime environment

This chapter defines the application programming interfaces (APIs) that a compliant EJB 2.0 Container must make available to the enterprise bean instances at runtime. These APIs can be used by portable enterprise beans because the APIs are guaranteed to be available in all EJB 2.0 Containers.

The chapter also defines the restrictions that the EJB 2.0 Container Provider can impose on the functionality that it provides to the enterprise beans. These restrictions are necessary to enforce security and to allow the Container to properly manage the runtime environment.

23.1 Bean Provider's responsibilities

This section describes the view and responsibilities of the Bean Provider.

23.1.1 APIs provided by Container

The EJB Provider can rely on the EJB 2.0 Container Provider to provide the following APIs:

- Java 2 Platform, Standard Edition, v1.3 (J2SE) APIs
- EJB 2.0 Standard Extension

- JDBC 2.0 Standard Extension (support for row sets only)
- JNDI 1.2 Standard Extension
- JTA 1.0.1 Standard Extension (the `UserTransaction` interface only)
- JMS 1.0.2 Standard Extension
- JavaMail 1.1 Standard Extension (for sending mail only)
- JAXP 1.0

23.1.2 Programming restrictions

This section describes the programming restrictions that a Bean Provider must follow to ensure that the enterprise bean is *portable* and can be deployed in any compliant EJB 2.0 Container. The restrictions apply to the implementation of the business methods. These restrictions also extend to the dependent classes that are used by an entity bean with container managed persistence. Section 23.2, which describes the Container's view of these restrictions, defines the programming environment that all EJB Containers must provide.

- An enterprise Bean must not use read/write static fields. Using read-only static fields is allowed. Therefore, it is recommended that all static fields in the enterprise bean class be declared as `final`.

This rule is required to ensure consistent runtime semantics because while some EJB Containers may use a single JVM to execute all enterprise bean's instances, others may distribute the instances across multiple JVMs.

- An enterprise Bean must not use thread synchronization primitives to synchronize execution of multiple instances.

Same reason as above. Synchronization would not work if the EJB Container distributed enterprise bean's instances across multiple JVMs.

- An enterprise Bean must not use the AWT functionality to attempt to output information to a display, or to input information from a keyboard.

Most servers do not allow direct interaction between an application program and a keyboard/display attached to the server system.

- An enterprise bean must not use the `java.io` package to attempt to access files and directories in the file system.

The file system APIs are not well-suited for business components to access data. Business components should use a resource manager API, such as JDBC, to store data.

- An enterprise bean must not attempt to listen on a socket, accept connections on a socket, or use a socket for multicast.

The EJB architecture allows an enterprise bean instance to be a network socket client, but it does not allow it to be a network server. Allowing the instance to become a network server would conflict with the basic function of the enterprise bean-- to serve the EJB clients.

- The enterprise bean must not attempt to query a class to obtain information about the declared members that are not otherwise accessible to the enterprise bean because of the security rules of the Java language. The enterprise bean must not attempt to use the Reflection API to access information that the security rules of the Java programming language make unavailable.

Allowing the enterprise bean to access information about other classes and to access the classes in a manner that is normally disallowed by the Java programming language could compromise security.

- The enterprise bean must not attempt to create a class loader; obtain the current class loader; set the context class loader; set security manager; create a new security manager; stop the JVM; or change the input, output, and error streams.

These functions are reserved for the EJB Container. Allowing the enterprise bean to use these functions could compromise security and decrease the Container's ability to properly manage the runtime environment.

- The enterprise bean must not attempt to set the socket factory used by ServerSocket, Socket, or the stream handler factory used by URL.

These networking functions are reserved for the EJB Container. Allowing the enterprise bean to use these functions could compromise security and decrease the Container's ability to properly manage the runtime environment.

- The enterprise bean must not attempt to manage threads. The enterprise bean must not attempt to start, stop, suspend, or resume a thread; or to change a thread's priority or name. The enterprise bean must not attempt to manage thread groups.

These functions are reserved for the EJB Container. Allowing the enterprise bean to manage threads would decrease the Container's ability to properly manage the runtime environment.

- The enterprise bean must not attempt to directly read or write a file descriptor.

Allowing the enterprise bean to read and write file descriptors directly could compromise security.

- The enterprise bean must not attempt to obtain the security policy information for a particular code source.

Allowing the enterprise bean to access the security policy information would create a security hole.

- The enterprise bean must not attempt to load a native library.

This function is reserved for the EJB Container. Allowing the enterprise bean to load native code would create a security hole.

- The enterprise bean must not attempt to gain access to packages and classes that the usual rules of the Java programming language make unavailable to the enterprise bean.

This function is reserved for the EJB Container. Allowing the enterprise bean to perform this function would create a security hole.

- The enterprise bean must not attempt to define a class in a package.

This function is reserved for the EJB Container. Allowing the enterprise bean to perform this function would create a security hole.

- The enterprise bean must not attempt to access or modify the security configuration objects (Policy, Security, Provider, Signer, and Identity).

These functions are reserved for the EJB Container. Allowing the enterprise bean to use these functions could compromise security.

- The enterprise bean must not attempt to use the subclass and object substitution features of the Java Serialization Protocol.

Allowing the enterprise bean to use these functions could compromise security.

- The enterprise bean must not attempt to pass `this` as an argument or method result. The enterprise bean must pass the result of `SessionContext.getEJBObject()` or `EntityContext.getEJBObject()` instead.

To guarantee portability of the enterprise bean's implementation across all compliant EJB 2.0 Containers, the Bean Provider should test the enterprise bean using a Container with the security settings defined in Table 21. The tables define the minimal functionality that a compliant EJB Container must provide to the enterprise bean instances at runtime.

23.2 Container Provider's responsibility

This section defines the Container's responsibilities for providing the runtime environment to the enterprise bean instances. The requirements described here are considered to be the minimal requirements; a Container may choose to provide additional functionality that is not required by the EJB specification.

An EJB 2.0 Container must make the following APIs available to the enterprise bean instances at runtime:

- Java 2 Platform, Standard Edition v1.3 (J2SE) APIs
- EJB 2.0 APIs
- JNDI 1.2
- JTA 1.0.1, the `UserTransaction` interface only
- JDBC™ 2.0 extension

- JMS 1.0.2
- JavaMail 1.1, sending mail only
- JAXP 1.0

The following subsections describes the requirements in more detail.

23.2.1 Java 2 APIs requirements

The Container must provide the full set of Java 2 Platform, Standard Edition, v1.3 (J2SE) APIs. The Container is not allowed to subset the Java 2 platform APIs.

The EJB Container is allowed to make certain Java 2 platform functionality unavailable to the enterprise bean instances by using the Java 2 platform security policy mechanism. The primary reason for the Container to make certain functions unavailable to enterprise bean instances is to protect the security and integrity of the EJB Container environment, and to prevent the enterprise bean instances from interfering with the Container's functions.

The following table defines the Java 2 platform security permissions that the EJB Container must be able to grant to the enterprise bean instances at runtime. The term "grant" means that the Container must be able to grant the permission, the term "deny" means that the Container should deny the permission.

Table 21 Java 2 Platform Security policy for a standard EJB Container

Permission name	EJB Container policy
java.security.AllPermission	deny
java.awt.AWTPermission	deny
java.io.FilePermission	deny
java.net.NetPermission	deny
java.util.PropertyPermission	grant "read", "*" deny all other
java.lang.reflect.ReflectPermission	deny
java.lang.RuntimePermission	grant "queuePrintJob", deny all other
java.lang.SecurityPermission	deny
java.io.SerializablePermission	deny
java.net.SocketPermission	grant "connect", "*" [Note A], deny all other

Notes:

[A] This permission is necessary, for example, to allow enterprise beans to use the client functionality of the Java IDL and RMI-IIOP packages that are part of the Java 2 platform.

Some Containers may allow the Deployer to grant more, or fewer, permissions to the enterprise bean instances than specified in Table 21. Support for this is not required by the EJB specification. Enterprise beans that rely on more or fewer permissions will not be portable across all EJB Containers.

23.2.2 EJB 2.0 requirements

The container must implement the EJB 2.0 interfaces as defined in this documentation.

23.2.3 JNDI 1.2 requirements

At the minimum, the EJB Container must provide a JNDI API name space to the enterprise bean instances. The EJB Container must make the name space available to an instance when the instance invokes the `javax.naming.InitialContext` default (no-arg) constructor.

The EJB Container must make available at least the following objects in the name space:

- The home interfaces of other enterprise beans.
- The resource factories used by the enterprise beans.

The EJB specification does not require that all the enterprise beans deployed in a Container be presented with the same JNDI API name space. However, all the instances of the same enterprise bean must be presented with the same JNDI API name space.

23.2.4 JTA 1.0.1 requirements

The EJB Container must include the JTA 1.0.1 extension, and it must provide the `javax.transaction.UserTransaction` interface to enterprise beans with bean-managed transaction demarcation through the `javax.ejb.EJBContext` interface, and also in JNDI under the name `java:comp/UserTransaction`, in the cases required by the EJB specification.

The other JTA interfaces are low-level transaction manager and resource manager integration interfaces, and are not intended for direct use by enterprise beans.

23.2.5 JDBC™ 2.0 extension requirements

The EJB Container must include the JDBC 2.0 extension and provide its functionality to the enterprise bean instances, with the exception of the low-level XA and connection pooling interfaces. These low-level interfaces are intended for integration of a JDBC driver with an application server, not for direct use by enterprise beans.

23.2.6 JMS 1.0.2 requirements

The EJB Container must include the JMS 1.0.2 extension and provide its functionality to the enterprise bean instances, with the exception of the low-level interfaces that are intended for integration of a JMS provider with an application server, not for direct use by enterprise beans. These interfaces include: `javax.jms.ServerSession`, `javax.jms.ServerSessionPool`, `javax.jms.ConnectionConsumer`, and all the `javax.jms` XA interfaces.

In addition, the following methods are for use by the Container only. Enterprise beans must not call these methods: `javax.jms.Session.setMessageListener`, `javax.jms.Session.getMessageListener`, `javax.jms.Session.run`, `javax.jms.QueueConnection.createConnectionConsumer`, `javax.jms.TopicConnection.createConnectionConsumer`, `javax.jms.TopicConnection.createDurableConnectionConsumer`.

The following methods must not be called by enterprise beans because they may interfere with the connection management by the Container: `javax.jms.Connection.setExceptionListener`, `javax.jms.Connection.stop`, `javax.jms.Connection.setClientID`.

Enterprise beans must not call the `javax.jms.MessageConsumer.setMessageListener` or `javax.jms.MessageConsumer.getMessageListener` method.

This specification recommends, but does not require, that the Container throw the `javax.jms.JMSException` if enterprise beans call any of the methods listed in this section.

23.2.7 Argument passing semantics

The enterprise bean's home and remote interfaces are *remote interfaces* for Java RMI. The Container must ensure the semantics for passing arguments conform to Java RMI. Non-remote objects must be passed by value.

Specifically, the EJB Container is not allowed to pass non-remote objects by reference on inter-EJB invocations when the calling and called enterprise beans are collocated in the same JVM. Doing so could result in the multiple beans sharing the state of a Java object, which would break the enterprise bean's semantics.

Chapter 24 Responsibilities of EJB Roles

This chapter provides the summary of the responsibilities of each EJB Role.

24.1 Bean Provider's responsibilities

This section highlights the requirements for the Bean Provider. Meeting these requirements is necessary to ensure that the enterprise beans developed by the Bean Provider can be deployed in all compliant EJB Containers.

24.1.1 API requirements

The enterprise beans must meet all the API requirements defined in the individual chapters of this document.

24.1.2 Packaging requirements

The Bean Provider is responsible for packaging the enterprise beans in an ejb-jar file in the format described in Chapter 22.

The deployment descriptor must include the *structural* information described in Section 21.2.

The deployment descriptor may optionally include any of the *application assembly* information as described in Section 21.3.

24.2 Application Assembler's responsibilities

The requirements for the Application Assembler are in defined in Section 21.3.

24.3 EJB Container Provider's responsibilities

The EJB Container Provider is responsible for providing the deployment tools used by the Deployer to deploy enterprise beans packaged in the ejb-jar file. The requirements for the deployment tools are defined in the individual chapters of this document.

The EJB Container Provider is responsible for implementing its part of the EJB contracts, and for providing all the runtime services described in the individual chapters of this document.

24.4 Deployer's responsibilities

The Deployer uses the deployment tools provided by the EJB Container provider to deploy ejb-jar files produced by the Bean Providers and Application Assemblers.

The individual chapters of this document describe the responsibilities of the Deployer in more detail.

24.5 System Administrator's responsibilities

The System Administrator is responsible for configuring the EJB Container and server, setting up security management, integrating resource managers with the EJB Container, and runtime monitoring of deployed enterprise beans applications.

The individual chapters of this document describe the responsibilities of the System Administrator in more detail.

24.6 Client Programmer's responsibilities

The EJB client programmer writes applications that access enterprise beans via their home and remote interfaces or via JMS messages.

Chapter 25 Enterprise JavaBeans™ API Reference

The following interfaces and classes comprise the Enterprise JavaBeans API:

package *javax.ejb*

Interfaces:

```
public interface EJBContext
public interface EJBHome
public interface EJBMetaData
public interface EJBObject
public interface EnterpriseBean
public interface EntityBean
public interface EntityContext
public interface Handle
public interface HomeHandle
public interface MessageDrivenBean
public interface MessageDrivenContext
public interface SessionBean
public interface SessionContext
public interface SessionSynchronization
```

Classes:

```
public class CreateException
public class DuplicateKeyException
public class EJBException
public class FinderException
public class ObjectNotFoundException
public class RemoveException
```

package *javax.ejb.deployment*

The `javax.ejb.deployment` package that was defined in the EJB 1.0 specification was deprecated in EJB 1.1. The EJB 1.0 deployment descriptor format should not be used by ejb-jar file producer, and the support for it is not required by EJB 1.1 and later compliant Containers.

The Javadoc specification of the EJB interface is included in a ZIP file distributed with this document.

Chapter 26 Related documents

[1] JavaBeans. *http://java.sun.com/beans*.

[2] Java Naming and Directory Interface (JNDI). *http://java.sun.com/products/jndi*.

[3] Java Remote Method Invocation (RMI). *http://java.sun.com/products/rmi.*

[4] Java Security. *http://java.sun.com/security.*

[5] Java Transaction API (JTA). *http://java.sun.com/products/jta*.

[6] Java Transaction Service (JTS). *http://java.sun.com/products/jts.*

[7] Java Language to IDL Mapping Specification. *http://www.omg.org/cgi-bin/doc?ptc/00-01-06*.

[8] CORBA Object Transaction Service v1.2. *http://www.omg.org/cgi-bin/doc?ptc/00-09-04*.

[9] Java 2 Platform, Enterprise Edition, v1.3 (J2EE).

[10] Java Message Service (JMS). *http://java.sun.com/products/jms*.

[11] JDBC 2.0 Standard Extension API. *http://java.sun.com/products/jdbc*.

[12] Java 2 Enterprise Edition Connector Architecture.

[13] Enterprise JavaBeans to CORBA Mapping v1.1. *http://java.sun.com/products/ejb/docs.html*.

[14] CORBA 2.3.1 Specification. *http://www.omg.org/cgi-bin/doc?formal/99-10-07*.

[15] CORBA COSNaming Service. *http://www.omg.org/cgi-bin/doc?formal/00-06-19*.

[16] Interoperable Name Service FTF document. *http://www.omg.org/cgi-bin/doc?ptc/00-08-07.*

[17] RFC 2246: The TLS Protocol. *ftp://ftp.isi.edu/in-notes/rfc2246.txt.*

[18] RFC 2712: Addition of Kerberos Cipher Suites to Transport Layer Security. *ftp://ftp.isi.edu/in-notes/rfc2712.txt.*

[19] The SSL Protocol Version 3.0. *http://home.netscape.com/eng/ssl3/draft302.txt.*

[20] Common Secure Interoperability Version 2 Final Submission. *http://www.omg.org/cgi-bin/doc?orbos/00-08-04.*

[21] Database Language SQL. ANSI X3.135-1992 or ISO/IEC 9075:1992.

Appendix A Features deferred to future releases

We plan to provide the following in future releases of the Enterprise JavaBeans specification:

- support for method interceptors
- support for component-level inheritance
- read-only Entity Beans with container managed persistence
- aggregate operations for EJB QL finder methods
- support for other types of messaging in addition to JMS
- specification for the pluggability of Persistence Managers

We plan to provide an SPI-level interface for attaching a JMS provider to the EJB Container as part of a future release of the Connector API.

Appendix B EJB 1.1 Deployment descriptor

This appendix defines the EJB 1.1 deployment descriptor. All EJB 2.0 compliant implementations must support EJB 1.1 as well as EJB 2.0 deployment descriptors. Section B.1 provides an overview of the deployment descriptor. Sections B.2 through B.4 describe the information in the deployment descriptor from the perspective of the EJB roles responsible for providing the information. Section B.5 defines the deployment descriptor's XML DTD. Section B.6 provides a complete example of a deployment descriptor of an assembled application.

B.1 Overview

The deployment descriptor is part of the contract between the ejb-jar file producer and consumer. This contract covers both the passing of enterprise beans from the Bean Provider to Application Assembler, and from the Application Assembler to the Deployer.

An ejb-jar file produced by the Bean Provider contains one or more enterprise beans and typically does not contain application assembly instructions. An ejb-jar file produced by an Application Assembler contains one or more enterprise beans, plus application assembly information describing how the enterprise beans are combined into a single application deployment unit.

The J2EE specification defines how enterprise beans and other application components contained in multiple ejb-jar files can be assembled into an application.

The role of the deployment descriptor is to capture the declarative information (i.e information that is not included directly in the enterprise beans' code) that is intended for the consumer of the ejb-jar file.

There are two basic kinds of information in the deployment descriptor:

- *Enterprise beans' structural* information. Structural information describes the structure of an enterprise bean and declares an enterprise bean's external dependencies. Providing structural information in the deployment descriptor is mandatory for the ejb-jar file producer. The structural information cannot, in general, be changed because doing so could break the enterprise bean's function.

- *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. Providing assembly information in the deployment descriptor is optional for the ejb-jar file producer. Assembly level information can be changed without breaking the enterprise bean's function, although doing so may alter the behavior of an assembled application.

B.2 Bean Provider's responsibilities

The Bean Provider is responsible for providing the structural information for each enterprise bean in the deployment descriptor.

The Bean Provider must use the `enterprise-beans` element to list all the enterprise beans in the ejb-jar file.

The Bean Provider must provide the following information for each enterprise bean:

- **Enterprise bean's name**. The Bean Provider must assign a logical name to each enterprise bean in the ejb-jar file. There is no architected relationship between this name, and the JNDI API name that the Deployer will assign to the enterprise bean. The Bean Provider specifies the enterprise bean's name in the `ejb-name` element.

- **Enterprise bean's class**. The Bean Provider must specify the fully-qualified name of the Java class that implements the enterprise bean's business methods. The Bean Provider specifies the enterprise bean's class name in the `ejb-class` element.

- **Enterprise bean's home interfaces**. The Bean Provider must specify the fully-qualified name of the enterprise bean's home interface in the `home` element.

- **Enterprise bean's remote interfaces**. The Bean Provider must specify the fully-qualified name of the enterprise bean's remote interface in the `remote` element.

- **Enterprise bean's type**. The enterprise beans types are session and entity. The Bean Provider must use the appropriate `session` or `entity` element to declare the enterprise bean's structural information.

- **Re-entrancy indication**. The Bean Provider must specify whether an entity bean is re-entrant or not. Session beans are never re-entrant.

- **Session bean's state management type**. If the enterprise bean is a Session bean, the Bean Provider must use the `session-type` element to declare whether the session bean is stateful or stateless.

- **Session bean's transaction demarcation type**. If the enterprise bean is a Session bean, the Bean Provider must use the `transaction-type` element to declare whether transaction demarcation is performed by the enterprise bean or by the Container.

- **Entity bean's persistence management**. If the enterprise bean is an Entity bean, the Bean Provider must use the `persistence-type` element to declare whether persistence management is performed by the enterprise bean or by the Container.

- **Entity bean's primary key class**. If the enterprise bean is an Entity bean, the Bean Provider specifies the fully-qualified name of the Entity bean's primary key class in the `prim-key-class` element. The Bean Provider *must* specify the primary key class for an Entity with bean-managed persistence, and *may* (but is not required to) specify the primary key class for an Entity with container-managed persistence.

- **Container-managed fields**. If the enterprise bean is an Entity bean with container-managed persistence, the Bean Provider must specify the container-managed fields using the `cmp-field` elements.

- **Environment entries**. The Bean Provider must declare all the enterprise bean's environment entries as specified in Subsection 19.2.1.

- **Resource manager connection factory references**.The Bean Provider must declare all the enterprise bean's resource manager connection factory references as specified in Subsection 19.4.1.

- **EJB references**. The Bean Provider must declare all the enterprise bean's references to the homes of other enterprise beans as specified in Subsection 19.3.1.

- **Security role references**. The Bean Provider must declare all the enterprise bean's references to security roles as specified in Subsection 20.2.5.3.

The deployment descriptor produced by the Bean Provider must be well formed in the XML sense, and valid with respect to the DTD in Section B.5. The content of the deployment descriptor must conform to the semantics rules specified in the DTD comments and elsewhere in this specification. The deployment descriptor must refer to the DTD using the following statement:

```
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise
JavaBeans 1.1//EN" "http://java.sun.com/j2ee/dtds/ejb-jar_1_1.dtd">
```

B.3 Application Assembler's responsibility

The Application Assembler assembles enterprise beans into a single deployment unit. The Application Assembler's input is one or more ejb-jar files provided by one or more Bean Providers, and the output is also one or more ejb-jar files. The Application Assembler can combine multiple input ejb-jar files into a single output ejb-jar file, or split an input ejb-jar file into multiple output ejb-jar files. Each output ejb-jar file is either a deployment unit intended for the Deployer, or a partially assembled application that is intended for another Application Assembler.

The Bean Provider and Application Assembler may be the same person or organization. In such a case, the person or organization performs the responsibilities described both in this and the previous sections.

The Application Assembler may modify the following information that was specified by the Bean Provider:

- **Enterprise bean's name**. The Application Assembler may change the enterprise bean's name defined in the `ejb-name` element.
- **Values of environment entries**. The Application Assembler may change existing and/or define new values of environment properties.
- **Description fields**. The Application Assembler may change existing or create new `description` elements.

The Application Assembler must not, in general, modify any other information listed in Section B.2 that was provided in the input ejb-jar file.

In addition, the Application Assembler may, but is not required to, specify any of the following *application assembly* information:

- **Binding of enterprise bean references**. The Application Assembler may link an enterprise bean reference to another enterprise bean in the ejb-jar file. The Application Assembler creates the link by adding the `ejb-link` element to the referencing bean.
- **Security roles**. The Application Assembler may define one or more security roles. The security roles define the *recommended* security roles for the clients of the enterprise beans. The Application Assembler defines the security roles using the `security-role` elements.
- **Method permissions**. The Application Assembler may define method permissions. Method permission is a binary relation between the security roles and the methods of the remote and home interfaces of the enterprise beans. The Application Assembler defines method permissions using the `method-permission` elements.
- **Linking of security role references**. If the Application Assembler defines security roles in the deployment descriptor, the Application Assembler must link the security role references declared by the Bean Provider to the security roles. The Application Assembler defines these links using the `role-link` element.

- **Transaction attributes**. The Application Assembler may define the value of the transaction attributes for the methods of the remote and home interfaces of the enterprise beans that require container-managed transaction demarcation. All Entity beans and the Session beans declared by the Bean Provider as transaction-type `Container` require container-managed transaction demarcation. The Application Assembler uses the `container-transaction` elements to declare the transaction attributes.

If an input ejb-jar file contains application assembly information, the Application Assembler is allowed to change the application assembly information supplied in the input ejb-jar file. (This could happen when the input ejb-jar file was produced by another Application Assembler.)

The deployment descriptor produced by the Bean Provider must be well formed in the XML sense, and valid with respect to the DTD in Section B.5. The content of the deployment descriptor must conform to the semantic rules specified in the DTD comments and elsewhere in this specification. The deployment descriptor must refer to the DTD using the following statement:

```
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise
JavaBeans 1.1//EN" "http://java.sun.com/j2ee/dtds/ejb-jar_1_1.dtd">
```

B.4 Container Provider's responsibilities

The Container provider provides tools that read and import the information contained in the XML deployment descriptor.

B.5 Deployment descriptor DTD

This section defines the XML DTD for the EJB 1.1 deployment descriptor. The comments in the DTD specify additional requirements for the syntax and semantics that cannot be easily expressed by the DTD mechanism.

The content of the XML elements is in general case sensitive. This means, for example, that

```
<reentrant>True</reentrant>
```

must be used, rather than:

```
<reentrant>true</reentrant>.
```

All valid ejb-jar deployment descriptors must contain the following DOCTYPE declaration:

```
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise
JavaBeans 1.1//EN" "http://java.sun.com/j2ee/dtds/ejb-jar_1_1.dtd">
```

We provide an EJB 1.1 ejb-jar file verifier that can be used by the Bean Provider and Application Assembler Roles to ensure that an ejb-jar is valid. The verifier checks all the requirements for the ejb-jar file and the deployment descriptor stated in this chapter.

```
<!--
This is the XML DTD for the EJB 1.1 deployment descriptor.
-->


<!--
The assembly-descriptor element contains application-assembly infor-
mation.

The application-assembly information consists of the following parts:
the definition of security roles, the definition of method permis-
sions, and the definition of transaction attributes for enterprise
beans with container-managed transaction demarcation.

All the parts are optional in the sense that they are omitted if the
lists represented by them are empty.

Providing an assembly-descriptor in the deployment descriptor is
optional for the ejb-jar file producer.

Used in: ejb-jar
-->
<!ELEMENT assembly-descriptor (security-role*, method-permission*,
          container-transaction*)>

<!--
The cmp-field element describes a container-managed field. The field
element includes an optional description of the field, and the name of
the field.

Used in: entity
-->
<!ELEMENT cmp-field (description?, field-name)>

<!--
The container-transaction element specifies how the container must
manage transaction scopes for the enterprise bean's method invoca-
tions. The element consists of an optional description, a list of
method elements, and a transaction attribute.The transaction
attribute is to be applied to all the specified methods.

Used in: assembly-descriptor
-->
<!ELEMENT container-transaction (description?, method+,
          trans-attribute)>

<!--
The description element is used by the ejb-jar file producer to pro-
vide text describing the parent element.

The description element should include any information that the
ejb-jar file producer wants to provide to the consumer of the ejb-jar
file (i.e. to the Deployer). Typically, the tools used by the ejb-jar
file consumer will display the description when processing the parent
```

```
element.

Used in: cmp-field, container-transaction, ejb-jar, entity,
env-entry, ejb-ref, method, method-permission, resource-ref, secu-
rity-role, security-role-ref, and session.
-->
<!ELEMENT description (#PCDATA)>

<!--
The display-name element contains a short name that is intended to be
display by tools.

Used in: ejb-jar, session, and entity

Example:
     <display-name>Employee Self Service</display-name>
-->
<!ELEMENT display-name (#PCDATA)>

<!--
The ejb-class element contains the fully-qualified name of the enter-
prise bean's class.

Used in: entity and session

Example:
      <ejb-class>com.wombat.empl.EmployeeServiceBean</ejb-class>
-->
<!ELEMENT ejb-class (#PCDATA)>

<!--
The optional ejb-client-jar element specifies a JAR file that con-
tains the class files necessary for a client program to access the
enterprise beans in the ejb-jar file. The Deployer should make the
ejb-client JAR file accessible to the client's class-loader.

Used in: ejb-jar

Example:
     <ejb-client-jar>employee_service_client.jar</ejb-client-jar>
-->
<!ELEMENT ejb-client-jar (#PCDATA)>

<!--
The ejb-jar element is the root element of the EJB deployment descrip-
tor. It contains an optional description of the ejb-jar file, optional
display name, optional small icon file name, optional large icon file
name, mandatory structural information about all included enterprise
beans, optional application-assembly descriptor, and an optional name
of an ejb-client-jar file for the ejb-jar.
-->
<!ELEMENT ejb-jar (description?, display-name?, small-icon?,
        large-icon?, enterprise-beans, assembly-descriptor?,
        ejb-client-jar?)>

<!--
The ejb-link element is used in the ejb-ref element to specify that an
EJB reference is linked to another enterprise bean in the ejb-jar
file.
```

```
The value of the ejb-link element must be the ejb-name of an enter-
prise bean in the same ejb-jar file, or in another ejb-jar file in the
same J2EE application unit.

Used in: ejb-ref

Example:
      <ejb-link>EmployeeRecord</ejb-link>
-->
<!ELEMENT ejb-link (#PCDATA)>

<!--
The ejb-name element specifies an enterprise bean's name. This name is
assigned by the ejb-jar file producer to name the enterprise bean in
the ejb-jar file's deployment descriptor. The name must be unique
among the names of the enterprise beans in the same ejb-jar file.

The enterprise bean code does not depend on the name; therefore the
name can be changed during the application-assembly process without
breaking the enterprise bean's function.

There is no architected relationship between the ejb-name in the
deployment descriptor and the JNDI name that the Deployer will assign
to the enterprise bean's home.

The name must conform to the lexical rules for an NMTOKEN.

Used in: entity, method, and session

Example:
      <ejb-name>EmployeeService</ejb-name>
-->
<!ELEMENT ejb-name (#PCDATA)>

<!--
The ejb-ref element is used for the declaration of a reference to
another enterprise bean's home. The declaration consists of an
optional description; the EJB reference name used in the code of the
referencing enterprise bean; the expected type of the referenced
enterprise bean; the expected home and remote interfaces of the ref-
erenced enterprise bean; and an optional ejb-link information.

The optional ejb-link element is used to specify the referenced enter-
prise bean. It is used typically in ejb-jar files that contain an
assembled application.

Used in: entity and session
-->
<!ELEMENT ejb-ref (description?, ejb-ref-name, ejb-ref-type, home,
         remote, ejb-link?)>

<!--
The ejb-ref-name element contains the name of an EJB reference. The
EJB reference is an entry in the enterprise bean's environment.

It is recommended that name is prefixed with "ejb/".

Used in: ejb-ref
```

```
Example:
      <ejb-ref-name>ejb/Payroll</ejb-ref-name>
-->
<!ELEMENT ejb-ref-name (#PCDATA)>

<!--
The ejb-ref-type element contains the expected type of the referenced
enterprise bean.

The ejb-ref-type element must be one of the following:
         <ejb-ref-type>Entity</ejb-ref-type>
         <ejb-ref-type>Session</ejb-ref-type>

Used in: ejb-ref
-->
<!ELEMENT ejb-ref-type (#PCDATA)>

<!--
The enterprise-beans element contains the declarations of one or more
enterprise beans.
-->
<!ELEMENT enterprise-beans (session | entity)+>

<!--
The entity element declares an entity bean. The declaration consists
of: an optional description; optional display name; optional small
icon file name; optional large icon file name; a name assigned to the
enterprise bean in the deployment descriptor; the names of the entity
bean's home and remote interfaces; the entity bean's implementation
class; the entity bean's persistence management type; the entity
bean's primary key class name; an indication of the entity bean's
reentrancy; an optional list of container-managed fields; an optional
specification of the primary key field; an optional declaration of the
bean's environment entries; an optional declaration of the bean's EJB
references; an optional declaration of the security role references;
and an optional declaration of the bean's resource manager connection
factory references.

The optional primkey-field may be present in the descriptor if the
entity's persistency-type is Container.

The other elements that are optional are "optional" in the sense that
they are omitted if the lists represented by them are empty.

At least one cmp-field element must be present in the descriptor if
the entity's persistency-type is Container, and none must not be
present if the entity's persistence-type is Bean.

Used in: enterprise-beans
-->
<!ELEMENT entity (description?, display-name?, small-icon?,
         large-icon?, ejb-name, home, remote, ejb-class,
         persistence-type, prim-key-class, reentrant,
         cmp-field*, primkey-field?, env-entry*,
         ejb-ref*, security-role-ref*, resource-ref*)>

<!--
The env-entry element contains the declaration of an enterprise
```

```
bean's environment entries. The declaration consists of an optional
description, the name of the environment entry, and an optional value.

Used in: entity and session
-->
<!ELEMENT env-entry (description?, env-entry-name, env-entry-type,
         env-entry-value?)>

<!--
The env-entry-name element contains the name of an enterprise bean's
environment entry.

Used in: env-entry

Example:
      <env-entry-name>minAmount</env-entry-name>
-->
<!ELEMENT env-entry-name (#PCDATA)>

<!--
The env-entry-type element contains the fully-qualified Java type of
the environment entry value that is expected by the enterprise bean's
code.

The following are the legal values of env-entry-type: java.lang.Bool-
ean, java.lang.String, java.lang.Integer, java.lang.Double,
java.lang.Byte, java.lang.Short, java.lang.Long, and java.lang.Float.

Used in: env-entry

Example:
      <env-entry-type>java.lang.Boolean</env-entry-type>
-->
<!ELEMENT env-entry-type (#PCDATA)>

<!--
The env-entry-value element contains the value of an enterprise
bean's environment entry.

Used in: env-entry

Example:
      <env-entry-value>100.00</env-entry-value>
-->
<!ELEMENT env-entry-value (#PCDATA)>

<!--
The field-name element specifies the name of a container managed
field. The name must be a public field of the enterprise bean class or
one of its superclasses.

Used in: cmp-field

Example:
      <field-name>firstName</field-Name>
-->
<!ELEMENT field-name (#PCDATA)>

<!--
```

```
The home element contains the fully-qualified name of the enterprise
bean's home interface.

Used in: ejb-ref, entity, and session

Example:
      <home>com.aardvark.payroll.PayrollHome</home>
-->
<!ELEMENT home (#PCDATA)>

<!--
The large-icon element contains the name of a file containing a large
(32 x 32) icon image. The file name is relative path within the
ejb-jar file.

The image must be either in the JPEG or GIF format, and the file name
must end with the suffix ".jpg" or ".gif" respectively.
The icon can be used by tools.

Example:
     <large-icon>employee-service-icon32x32.jpg</large-icon>
-->
<!ELEMENT large-icon (#PCDATA)>

<!--
The method element is used to denote a method of an enterprise bean's
home or remote interface, or a set of methods. The ejb-name element
must be the name of one of the enterprise beans in declared in the
deployment descriptor; the optional method-intf element allows to
distinguish between a method with the same signature that is defined
in both the home and remote interface; the method-name element speci-
fies the method name; and the optional method-params elements iden-
tify a single method among multiple methods with an overloaded method
name.

There are three possible styles of the method element syntax:

1.   <method>
         <ejb-name>EJBNAME</ejb-name>
         <method-name>*</method-name>
     </method>

   This style is used to refer to all the methods of the specified
   enterprise bean's home and remote interfaces.

2.   <method>
         <ejb-name>EJBNAME</ejb-name>
         <method-name>METHOD</method-name>
     </method>>

   This style is used to refer to the specified method of the
   specified enterprise bean. If there are multiple methods with
   the same overloaded name, the element of this style refers to
   all the methods with the overloaded name.
```

3.
```
<method>
    <ejb-name>EJBNAME</ejb-name>
    <method-name>METHOD</method-name>
    <method-params>
        <method-param>PARAM-1</method-param>
        <method-param>PARAM-2</method-param>
        ...
        <method-param>PARAM-n</method-param>
    </method-params>
<method>
```

This style is used to refer to a single method within a set of methods with an overloaded name. PARAM-1 through PARAM-n are the fully-qualified Java types of the method's input parameters (if the method has no input arguments, the method-params element contains no method-param elements). Arrays are specified by the array element's type, followed by one or more pair of square brackets (e.g. int[][]).

Used in: method-permission and container-transaction

Examples:

Style 1: The following method element refers to all the methods of the EmployeeService bean's home and remote interfaces:

```
<method>
   <ejb-name>EmployeeService</ejb-name>
   <method-name>*</method-name>
</method>
```

Style 2: The following method element refers to all the *create* methods of the EmployeeService bean's home interface:

```
<method>
   <ejb-name>EmployeeService</ejb-name>
   <method-name>create</method-name>
</method>
```

Style 3: The following method element refers to the *create(String firstName, String LastName)* method of the EmployeeService bean's home interface.

```
<method>
   <ejb-name>EmployeeService</ejb-name>
   <method-name>create</method-name>
   <method-params>
      <method-param>java.lang.String</method-param>
      <method-param>java.lang.String</method-param>
   </method-params>
</method>
```

The following example illustrates a Style 3 element with more complex parameter types. The method

foobar(char s, int i, int[] iar, mypackage.MyClass mycl, mypackage.MyClass[][] myclaar)

would be specified as:

```
      <method>
         <ejb-name>EmployeeService</ejb-name>
         <method-name>foobar</method-name>
         <method-params>
            <method-param>char</method-param>
            <method-param>int</method-param>
            <method-param>int[]</method-param>
            <method-param>mypackage.MyClass</method-param>
            <method-param>mypackage.MyClass[][]</method-param>
         </method-params>
      </method>

   The optional method-intf element can be used when it becomes
   necessary to differentiate between a method defined in the home
   interface and a method with the same name and signature that is
   defined in the remote interface.

   For example, the method element

      <method>
         <ejb-name>EmployeeService</ejb-name>
         <method-intf>Remote</method-intf>
         <method-name>create</method-name>
         <method-params>
            <method-param>java.lang.String</method-param>
            <method-param>java.lang.String</method-param>
         </method-params>
      </method>

   can be used to differentiate the create(String, String) method
   defined in the remote interface from the create(String, String)
   method defined in the home interface, which would be defined as

      <method>
         <ejb-name>EmployeeService</ejb-name>
         <method-intf>Home</method-intf>
         <method-name>create</method-name>
         <method-params>
            <method-param>java.lang.String</method-param>
            <method-param>java.lang.String</method-param>
         </method-params>
      </method>

   The method-intf element can be used with all three Styles of the
method element usage. For example, the following method element exam-
ple could be used to refer to all the methods of the EmployeeService
bean's home interface.

      <method>
         <ejb-name>EmployeeService</ejb-name>
         <method-intf>Home</method-intf>
         <method-name>*</method-name>
      </method>


-->
<!ELEMENT method (description?, ejb-name, method-intf?, method-name,
             method-params?)>
```

```
<!--
The method-intf element allows a method element to differentiate
between the methods with the same name and signature that are defined
in both the remote and home interfaces.

The method-intf element must be one of the following:
        <method-intf>Home</method-intf>
        <method-intf>Remote</method-intf>

Used in: method
-->
<!ELEMENT method-intf (#PCDATA)>

<!--
The method-name element contains a name of an enterprise bean method,
or the asterisk (*) character. The asterisk is used when the element
denotes all the methods of an enterprise bean's remote and home inter-
faces.

Used in: method
-->
<!ELEMENT method-name (#PCDATA)>

<!--
The method-param element contains the fully-qualified Java type name
of a method parameter.

Used in: method-params
-->
<!ELEMENT method-param (#PCDATA)>

<!--
The method-params element contains a list of the fully-qualified Java
type names of the method parameters.

Used in: method
-->
<!ELEMENT method-params (method-param*)>

<!--
The method-permission element specifies that one or more security
roles are allowed to invoke one or more enterprise bean methods. The
method-permission element consists of an optional description, a list
of security role names, and a list of method elements.

The security roles used in the method-permission element must be
defined in the security-role element of the deployment descriptor,
and the methods must be methods defined in the enterprise bean's
remote and/or home interfaces.

Used in: assembly-descriptor
-->
<!ELEMENT method-permission (description?, role-name+, method+)>

<!--
The persistence-type element specifies an entity bean's persistence
management type.
```

```
The persistence-type element must be one of the two following:
         <persistence-type>Bean</persistence-type>
         <persistence-type>Container</persistence-type>

Used in: entity
-->
<!ELEMENT persistence-type (#PCDATA)>

<!--
The prim-key-class element contains the fully-qualified name of an
entity bean's primary key class.

If the definition of the primary key class is deferred to deployment
time, the prim-key-class element should specify java.lang.Object.

Used in: entity

Examples:
   <prim-key-class>java.lang.String</prim-key-class>
   <prim-key-class>com.wombat.empl.EmployeeID</prim-key-class>
   <prim-key-class>java.lang.Object</prim-key-class>

-->
<!ELEMENT prim-key-class (#PCDATA)>

<!--
The primkey-field element is used to specify the name of the primary
key field for an entity with container-managed persistence.

The primkey-field must be one of the fields declared in the cmp-field
element, and the type of the field must be the same as the primary key
type.

The primkey-field element is not used if the primary key maps to mul-
tiple container-managed fields (i.e., the key is a compound key). In
this case, the fields of the primary key class must be public, and
their names must correspond to the field names of the entity bean
class that comprise the key.

Used in: entity

Example:
   <primkey-field>EmployeeId</primkey-field>
-->
<!ELEMENT primkey-field (#PCDATA)>

<!--
The reentrant element specifies whether an entity bean is reentrant or
not.

The reentrant element must be one of the two following:
         <reentrant>True</reentrant>
         <reentrant>False</reentrant>

Used in: entity
-->
<!ELEMENT reentrant (#PCDATA)>

<!--
```

```
The remote element contains the fully-qualified name of the enter-
prise bean's remote interface.

Used in: ejb-ref, entity, and session

Example:
      <remote>com.wombat.empl.EmployeeService</remote>
-->
<!ELEMENT remote (#PCDATA)>

<!--
The res-auth element specifies whether the enterprise bean code signs
on programmatically to the resource manager, or whether the Container
will sign on to the resource manager on behalf of the bean. In the
latter case, the Container uses information that is supplied by the
Deployer.

The value of this element must be one of the two following:
         <res-auth>Application</res-auth>
         <res-auth>Container</res-auth>
-->
<!ELEMENT res-auth (#PCDATA)>

<!--
The res-ref-name element specifies the name of a resource manager con-
nection factory reference.

Used in: resource-ref
-->
<!ELEMENT res-ref-name (#PCDATA)>

<!--
The res-type element specifies the type of the data source. The type
is specified by the Java interface (or class) expected to be imple-
mented by the data source.

Used in: resource-ref
-->
<!ELEMENT res-type (#PCDATA)>

<!--
The resource-ref element contains a declaration of enterprise bean's
reference to an external resource. It consists of an optional descrip-
tion, the resource manager connection factory reference name, the
indication of the resource manager connection factory type expected
by the enterprise bean code, and the type of authentication (bean or
container).

Used in: entity and session

Example:
      <resource-ref>
         <res-ref-name>jdbc/EmployeeAppDB</res-ref-name>
         <res-type>javax.sql.DataSource</res-type>
         <res-auth>Container</res-auth>
      </resource-ref>
-->
<!ELEMENT resource-ref (description?, res-ref-name, res-type,
              res-auth)>
```

```
<!--
The role-link element is used to link a security role reference to a
defined security role. The role-link element must contain the name of
one of the security roles defined in the security-role elements.

Used in: security-role-ref
-->
<!ELEMENT role-link (#PCDATA)>

<!--
The role-name element contains the name of a security role.

The name must conform to the lexical rules for an NMTOKEN.

Used in: method-permission, security-role, and security-role-ref
-->
<!ELEMENT role-name (#PCDATA)>

<!--
The security-role element contains the definition of a security role.
The definition consists of an optional description of the security
role, and the security role name.

Used in: assembly-descriptor

Example:
      <security-role>
         <description>
            This role includes all employees who are authorized
            to access the employee service application.
         </description>
         <role-name>employee</role-name>
      </security-role>
-->
<!ELEMENT security-role (description?, role-name)>

<!--
The security-role-ref element contains the declaration of a security
role reference in the enterprise bean's code. The declaration con-
sists of an optional description, the security role name used in the
code, and an optional link to a defined security role.

The value of the role-name element must be the String used as the
parameter to the EJBContext.isCallerInRole(String roleName) method.

The value of the role-link element must be the name of one of the
security roles defined in the security-role elements.

Used in: entity and session

-->
<!ELEMENT security-role-ref (description?, role-name, role-link?)>

<!--
The session-type element describes whether the session bean is a
stateful session, or stateless session.

The session-type element must be one of the two following:
```

```
        <session-type>Stateful</session-type>
        <session-type>Stateless</session-type>
-->
<!ELEMENT session-type (#PCDATA)>

<!--
The session element declares an session bean. The declaration con-
sists of: an optional description; optional display name; optional
small icon file name; optional large icon file name; a name assigned
to the enterprise bean in the deployment description; the names of the
session bean's home and remote interfaces; the session bean's imple-
mentation class; the session bean's state management type; the ses-
sion bean's transaction management type; an optional declaration of
the bean's environment entries; an optional declaration of the bean's
EJB references; an optional declaration of the security role refer-
ences; and an optional declaration of the bean's resource manager con-
nection factory references.

The elements that are optional are "optional" in the sense that they
are omitted when if lists represented by them are empty.

Used in: enterprise-beans
-->
<!ELEMENT session (description?, display-name?, small-icon?,
        large-icon?, ejb-name, home, remote, ejb-class,
        session-type, transaction-type, env-entry*,
        ejb-ref*, security-role-ref*, resource-ref*)>

<!--
The small-icon element contains the name of a file containing a small
(16 x 16) icon image. The file name is relative path within the
ejb-jar file.

The image must be either in the JPEG or GIF format, and the file name
must end with the suffix ".jpg" or ".gif" respectively.

The icon can be used by tools.

Example:
    <small-icon>employee-service-icon16x16.jpg</small-icon>
-->
<!ELEMENT small-icon (#PCDATA)>

<!--
The transaction-type element specifies an enterprise bean's transac-
tion management type.

The transaction-type element must be one of the two following:
        <transaction-type>Bean</transaction-type>
        <transaction-type>Container</transaction-type>

Used in: session
-->
<!ELEMENT transaction-type (#PCDATA)>

<!--
The trans-attribute element specifies how the container must manage
the transaction boundaries when delegating a method invocation to an
enterprise bean's business method.
```

```
The value of trans-attribute must be one of the following:
         <trans-attribute>NotSupported</trans-attribute>
         <trans-attribute>Supports</trans-attribute>
         <trans-attribute>Required</trans-attribute>
         <trans-attribute>RequiresNew</trans-attribute>
         <trans-attribute>Mandatory</trans-attribute>
         <trans-attribute>Never</trans-attribute>

Used in: container-transaction
-->
<!ELEMENT trans-attribute (#PCDATA)>

<!--
The ID mechanism is to allow tools that produce additional deployment
information (i.e information beyond the standard EJB deployment
descriptor information) to store the non-standard information in a
separate file, and easily refer from these tools-specific files to the
information in the standard deployment descriptor.

The EJB architecture does not allow the tools to add the non-standard
information into the EJB deployment descriptor.
-->
<!ATTLIST assembly-descriptor id ID #IMPLIED>
<!ATTLIST cmp-field id ID #IMPLIED>
<!ATTLIST container-transaction id ID #IMPLIED>
<!ATTLIST description id ID #IMPLIED>
<!ATTLIST display-name id ID #IMPLIED>
<!ATTLIST ejb-class id ID #IMPLIED>
<!ATTLIST ejb-client-jar id ID #IMPLIED>
<!ATTLIST ejb-jar id ID #IMPLIED>
<!ATTLIST ejb-link id ID #IMPLIED>
<!ATTLIST ejb-name id ID #IMPLIED>
<!ATTLIST ejb-ref id ID #IMPLIED>
<!ATTLIST ejb-ref-name id ID #IMPLIED>
<!ATTLIST ejb-ref-type id ID #IMPLIED>
<!ATTLIST enterprise-beans id ID #IMPLIED>
<!ATTLIST entity id ID #IMPLIED>
<!ATTLIST env-entry id ID #IMPLIED>
<!ATTLIST env-entry-name id ID #IMPLIED>
<!ATTLIST env-entry-type id ID #IMPLIED>
<!ATTLIST env-entry-value id ID #IMPLIED>
<!ATTLIST field-name id ID #IMPLIED>
<!ATTLIST home id ID #IMPLIED>
<!ATTLIST large-icon id ID #IMPLIED>
<!ATTLIST method id ID #IMPLIED>
<!ATTLIST method-intf id ID #IMPLIED>
<!ATTLIST method-name id ID #IMPLIED>
<!ATTLIST method-param id ID #IMPLIED>
<!ATTLIST method-params id ID #IMPLIED>
<!ATTLIST method-permission id ID #IMPLIED>
<!ATTLIST persistence-type id ID #IMPLIED>
<!ATTLIST prim-key-class id ID #IMPLIED>
<!ATTLIST primkey-field id ID #IMPLIED>
<!ATTLIST reentrant id ID #IMPLIED>
<!ATTLIST remote id ID #IMPLIED>
<!ATTLIST res-auth id ID #IMPLIED>
<!ATTLIST res-ref-name id ID #IMPLIED>
<!ATTLIST res-type id ID #IMPLIED>
```

```
<!ATTLIST resource-ref id ID #IMPLIED>
<!ATTLIST role-link id ID #IMPLIED>
<!ATTLIST role-name id ID #IMPLIED>
<!ATTLIST security-role id ID #IMPLIED>
<!ATTLIST security-role-ref id ID #IMPLIED>
<!ATTLIST session-type id ID #IMPLIED>
<!ATTLIST session id ID #IMPLIED>
<!ATTLIST small-icon id ID #IMPLIED>
<!ATTLIST transaction-type id ID #IMPLIED>
<!ATTLIST trans-attribute id ID #IMPLIED>
```

B.6 Deployment descriptor example

The following example illustrates a sample deployment descriptor for the ejb-jar containing the Wombat's assembled application described in Section 3.2.

Note: The text in the <description> elements has been formatted by adding whitespace to appear properly indented in this document. In a real deployment descriptor document, the <description> elements would likely contain no extra whitespace characters.

```
<!DOCTYPE ejb-jar PUBLIC “-//Sun Microsystems, Inc.//DTD Enterprise
JavaBeans 1.1//EN” “http://java.sun.com/j2ee/dtds/ejb-jar_1_1.dtd”>
<ejb-jar>
   <description>
      This ejb-jar file contains assembled enterprise beans that are
      part of employee self-service application.
   </description>

   <enterprise-beans>
      <session>
         <description>
            The EmployeeService session bean implements a session
            between an employee and the employee self-service
            application.
         </description>
         <ejb-name>EmployeeService</ejb-name>
         <home>com.wombat.empl.EmployeeServiceHome</home>
         <remote>com.wombat.empl.EmployeeService</remote>
         <ejb-class>com.wombat.empl.EmployeeServiceBean</ejb-class>
         <session-type>Stateful</session-type>
         <transaction-type>Bean</transaction-type>

         <env-entry>
            <env-entry-name>envvar1</env-entry-name>
            <env-entry-type>String</env-entry-type>
            <env-entry-value>some value</env-entry-value>

         </env-entry>

         <ejb-ref>
            <ejb-ref-name>ejb/EmplRecords</ejb-ref-name>
            <ejb-ref-type>Entity</ejb-ref-type>
            <home>com.wombat.empl.EmployeeRecordHome</home>
            <remote>com.wombat.empl.EmployeeRecord</remote>
            <ejb-link>EmployeeRecord</ejb-link>
         </ejb-ref>

         <ejb-ref>
            <ejb-ref-name>ejb/Payroll</ejb-ref-name>
            <ejb-ref-type>Entity</ejb-ref-type>
            <home>com.aardvark.payroll.PayrollHome</home>
            <remote>com.aardvark.payroll.Payroll</remote>
            <ejb-link>AardvarkPayroll</ejb-link>
         </ejb-ref>

         <ejb-ref>
            <ejb-ref-name>ejb/PensionPlan</ejb-ref-name>
            <ejb-ref-type>Session</ejb-ref-type>
            <home>com.wombat.empl.PensionPlanHome</home>
            <remote>com.wombat.empl.PensionPlan</remote>
         </ejb-ref>

         <resource-ref>
            <description>
               This is a reference to a JDBC database.
```

```
                  EmployeeService keeps a log of all
                  transactions performed through the
                  EmployeeService bean for auditing
                  purposes.
               </description>
               <res-ref-name>jdbc/EmployeeAppDB</res-ref-name>
               <res-type>javax.sql.DataSource</res-type>
               <res-auth>Container</res-auth>
            </resource-ref>
         </session>

         <session>
            <description>
               The EmployeeServiceAdmin session bean implements
               the session used by the application's administrator.
            </description>

            <ejb-name>EmployeeServiceAdmin</ejb-name>
            <home>com.wombat.empl.EmployeeServiceAdminHome</home>
            <remote>com.wombat.empl.EmployeeServiceAdmin</remote>
            <ejb-class>com.wombat.empl.EmployeeServiceAdmin-
Bean</ejb-class>
            <session-type>Stateful</session-type>
            <transaction-type>Bean</transaction-type>

            <resource-ref>
               <description>
                  This is a reference to a JDBC database.
                  EmployeeService keeps a log of all
                  transactions performed through the
                  EmployeeService bean for auditing
                  purposes.
               </description>
               <res-ref-name>jdbc/EmployeeAppDB</res-ref-name>
               <res-type>javax.sql.DataSource</res-type>
               <res-auth>Container</res-auth>
            </resource-ref>
         </session>

         <entity>
            <description>
               The EmployeeRecord entity bean encapsulates access
               to the employee records.The deployer will use
               container-managed persistence to integrate the
               entity bean with the back-end system managing
               the employee records.
            </description>

            <ejb-name>EmployeeRecord</ejb-name>
            <home>com.wombat.empl.EmployeeRecordHome</home>
            <remote>com.wombat.empl.EmployeeRecord</remote>
            <ejb-class>com.wombat.empl.EmployeeRecordBean</ejb-class>
            <persistence-type>Container</persistence-type>
            <prim-key-class>com.wombat.empl.EmployeeID</prim-key-class>
            <reentrant>True</reentrant>

            <cmp-field><field-name>employeeID</field-name></cmp-field>
            <cmp-field><field-name>firstName</field-name></cmp-field>
            <cmp-field><field-name>lastName</field-name></cmp-field>
```

```
            <cmp-field><field-name>address1</field-name></cmp-field>
            <cmp-field><field-name>address2</field-name></cmp-field>
            <cmp-field><field-name>city</field-name></cmp-field>
            <cmp-field><field-name>state</field-name></cmp-field>
            <cmp-field><field-name>zip</field-name></cmp-field>
            <cmp-field><field-name>homePhone</field-name></cmp-field>
            <cmp-field><field-name>jobTitle</field-name></cmp-field>
            <cmp-field><field-name>managerID</field-name></cmp-field>
            <cmp-field><field-name>jobTitleHis-
tory</field-name></cmp-field>
         </entity>

         <entity>
            <description>
               The Payroll entity bean encapsulates access
               to the payroll system.The deployer will use
               container-managed persistence to integrate the
               entity bean with the back-end system managing
               payroll information.
            </description>

            <ejb-name>AardvarkPayroll</ejb-name>
            <home>com.aardvark.payroll.PayrollHome</home>
            <remote>com.aardvark.payroll.Payroll</remote>
            <ejb-class>com.aardvark.payroll.PayrollBean</ejb-class>
            <persistence-type>Bean</persistence-type>
            <prim-key-class>com.aardvark.payroll.Accoun-
tID</prim-key-class>
            <reentrant>False</reentrant>

            <security-role-ref>
               <role-name>payroll-org</role-name>
               <role-link>payroll-department</role-link>
            </security-role-ref>
         </entity>
      </enterprise-beans>

      <assembly-descriptor>
         <security-role>
            <description>
               This role includes the employees of the
               enterprise who are allowed to access the
               employee self-service application. This role
               is allowed only to access his/her own
               information.
            </description>
            <role-name>employee</role-name>
         </security-role>

         <security-role>
            <description>
               This role includes the employees of the human
               resources department. The role is allowed to
               view and update all employee records.
            </description>
            <role-name>hr-department</role-name>
         </security-role>

         <security-role>
```

```
   <description>
      This role includes the employees of the payroll
      department. The role is allowed to view and
      update the payroll entry for any employee.
   </description>
   <role-name>payroll-department</role-name>
</security-role>

<security-role>
   <description>
      This role should be assigned to the personnel
      authorized to perform administrative functions
      for the employee self-service application.
      This role does not have direct access to
      sensitive employee and payroll information.
   </description>
   <role-name>admin</role-name>
</security-role>

<method-permission>
   <role-name>employee</role-name>
   <method>
      <ejb-name>EmployeeService</ejb-name>
      <method-name>*</method-name>
   </method>
</method-permission>

<method-permission>
   <role-name>employee</role-name>
   <method>
      <ejb-name>EmployeeRecord</ejb-name>
      <method-name>findByPrimaryKey</method-name>
   </method>
   <method>
      <ejb-name>EmployeeRecord</ejb-name>
      <method-name>getDetail</method-name>
   </method>
   <method>
      <ejb-name>EmployeeRecord</ejb-name>
      <method-name>updateDetail</method-name>
   </method>
</method-permission>

<method-permission>
   <role-name>employee</role-name>
   <method>
      <ejb-name>AardvarkPayroll</ejb-name>
      <method-name>findByPrimaryKey</method-name>
   </method>
   <method>
      <ejb-name>AardvarkPayroll</ejb-name>
      <method-name>getEmployeeInfo</method-name>
   </method>
   <method>
      <ejb-name>AardvarkPayroll</ejb-name>
      <method-name>updateEmployeeInfo</method-name>
   </method>
</method-permission>
```

```
<method-permission>
   <role-name>admin</role-name>
   <method>
      <ejb-name>EmployeeServiceAdmin</ejb-name>
      <method-name>*</method-name>
   </method>
</method-permission>

<method-permission>
   <role-name>hr-department</role-name>
   <method>
      <ejb-name>EmployeeRecord</ejb-name>
      <method-name>create</method-name>
   </method>
   <method>
      <ejb-name>EmployeeRecord</ejb-name>
      <method-name>remove</method-name>
   </method>
   <method>
      <ejb-name>EmployeeRecord</ejb-name>
      <method-name>changeManager</method-name>
   </method>
   <method>
      <ejb-name>EmployeeRecord</ejb-name>
      <method-name>changeJobTitle</method-name>
   </method>
   <method>
      <ejb-name>EmployeeRecord</ejb-name>
      <method-name>findByPrimaryKey</method-name>
   </method>
   <method>
      <ejb-name>EmployeeRecord</ejb-name>
      <method-name>getDetail</method-name>
   </method>
   <method>
      <ejb-name>EmployeeRecord</ejb-name>
      <method-name>updateDetail</method-name>
   </method>
</method-permission>

<method-permission>
   <role-name>payroll-department</role-name>
   <method>
      <ejb-name>AardvarkPayroll</ejb-name>
      <method-name>findByPrimaryKey</method-name>
   </method>
   <method>
      <ejb-name>AardvarkPayroll</ejb-name>
      <method-name>getEmployeeInfo</method-name>
   </method>
   <method>
      <ejb-name>AardvarkPayroll</ejb-name>
      <method-name>updateEmployeeInfo</method-name>
   </method>
   <method>
      <ejb-name>AardvarkPayroll</ejb-name>
      <method-name>updateSalary</method-name>
   </method>
</method-permission>
```

```
        <container-transaction>
           <method>
              <ejb-name>EmployeeRecord</ejb-name>
              <method-name>*</method-name>
           </method>
           <trans-attribute>Required</trans-attribute>
        </container-transaction>

        <container-transaction>
           <method>
              <ejb-name>AardvarkPayroll</ejb-name>
              <method-name>*</method-name>
           </method>
           <trans-attribute>Required</trans-attribute>
        </container-transaction>
     </assembly-descriptor>
  </ejb-jar>
```

Appendix C EJB 1.1 Runtime environment

This appendix defines the application programming interfaces (APIs) that a compliant EJB 1.1 Container must make available to the EJB 1.1 enterprise bean instances at runtime. These APIs can be used by portable enterprise beans because the APIs are guaranteed to be available in all EJB 1.1 Containers.

This appendix also defines the restrictions that the EJB 1.1 Container Provider can impose on the functionality that it provides to the enterprise beans. These restrictions are necessary to enforce security and to allow the Container to properly manage the runtime environment.

C.1 EJB 1.1 Bean Provider's responsibilities

This section describes the view and responsibilities of the EJB 1.1 Bean Provider.

C.1.1 APIs provided by EJB 1.1 Container

The EJB 1.1 Provider can rely on the EJB 1.1 Container Provider to provide the following APIs:

- Java 2 Platform, Standard Edition, v1.2 (J2SE)
- EJB 1.1 Standard Extension
- JDBC 2.0 Standard Extension (support for row sets only)

- JNDI 1.2 Standard Extension
- JTA 1.0.1 Standard Extension (the `UserTransaction` interface only)
- JavaMail 1.1 Standard Extension (for sending mail only)

C.1.2 Programming restrictions

This section describes the programming restrictions that an EJB 1.1 Bean Provider must follow to ensure that the enterprise bean is *portable* and can be deployed in any compliant EJB 1.1 Container. The restrictions apply to the implementation of the business methods. Section C.2, which describes the Container's view of these restrictions, defines the programming environment that all EJB 1.1 Containers must provide.

- An enterprise Bean must not use read/write static fields. Using read-only static fields is allowed. Therefore, it is recommended that all static fields in the enterprise bean class be declared as `final`.

This rule is required to ensure consistent runtime semantics because while some EJB Containers may use a single JVM to execute all enterprise bean's instances, others may distribute the instances across multiple JVMs.

- An enterprise Bean must not use thread synchronization primitives to synchronize execution of multiple instances.

Same reason as above. Synchronization would not work if the EJB Container distributed enterprise bean's instances across multiple JVMs.

- An enterprise Bean must not use the AWT functionality to attempt to output information to a display, or to input information from a keyboard.

Most servers do not allow direct interaction between an application program and a keyboard/display attached to the server system.

- An enterprise bean must not use the `java.io` package to attempt to access files and directories in the file system.

The file system APIs are not well-suited for business components to access data. Business components should use a resource manager API, such as JDBC API, to store data.

- An enterprise bean must not attempt to listen on a socket, accept connections on a socket, or use a socket for multicast.

The EJB architecture allows an enterprise bean instance to be a network socket client, but it does not allow it to be a network server. Allowing the instance to become a network server would conflict with the basic function of the enterprise bean-- to serve the EJB clients.

- The enterprise bean must not attempt to query a class to obtain information about the declared members that are not otherwise accessible to the enterprise bean because of the security rules

of the Java language. The enterprise bean must not attempt to use the Reflection API to access information that the security rules of the Java programming language make unavailable.

Allowing the enterprise bean to access information about other classes and to access the classes in a manner that is normally disallowed by the Java programming language could compromise security.

- The enterprise bean must not attempt to create a class loader; obtain the current class loader; set the context class loader; set security manager; create a new security manager; stop the JVM; or change the input, output, and error streams.

These functions are reserved for the EJB Container. Allowing the enterprise bean to use these functions could compromise security and decrease the Container's ability to properly manage the runtime environment.

- The enterprise bean must not attempt to set the socket factory used by ServerSocket, Socket, or the stream handler factory used by URL.

These networking functions are reserved for the EJB Container. Allowing the enterprise bean to use these functions could compromise security and decrease the Container's ability to properly manage the runtime environment.

- The enterprise bean must not attempt to manage threads. The enterprise bean must not attempt to start, stop, suspend, or resume a thread; or to change a thread's priority or name. The enterprise bean must not attempt to manage thread groups.

These functions are reserved for the EJB Container. Allowing the enterprise bean to manage threads would decrease the Container's ability to properly manage the runtime environment.

- The enterprise bean must not attempt to directly read or write a file descriptor.

Allowing the enterprise bean to read and write file descriptors directly could compromise security.

- The enterprise bean must not attempt to obtain the security policy information for a particular code source.

Allowing the enterprise bean to access the security policy information would create a security hole.

- The enterprise bean must not attempt to load a native library.

This function is reserved for the EJB Container. Allowing the enterprise bean to load native code would create a security hole.

- The enterprise bean must not attempt to gain access to packages and classes that the usual rules of the Java programming language make unavailable to the enterprise bean.

This function is reserved for the EJB Container. Allowing the enterprise bean to perform this function would create a security hole.

- The enterprise bean must not attempt to define a class in a package.

This function is reserved for the EJB Container. Allowing the enterprise bean to perform this function would create a security hole.

- The enterprise bean must not attempt to access or modify the security configuration objects (Policy, Security, Provider, Signer, and Identity).

These functions are reserved for the EJB Container. Allowing the enterprise bean to use these functions could compromise security.

- The enterprise bean must not attempt to use the subclass and object substitution features of the Java Serialization Protocol.

Allowing the enterprise bean to use these functions could compromise security.

- The enterprise bean must not attempt to pass `this` as an argument or method result. The enterprise bean must pass the result of `SessionContext.getEJBObject()` or `EntityContext.getEJBObject()` instead.

To guarantee portability of the EJB 1.1 enterprise bean's implementation across all compliant EJB 1.1 Containers, the Bean Provider should test the enterprise bean using a Container with the security settings defined in Table 22. The table defines the minimal functionality that a compliant EJB 1.1 Container must provide to the enterprise bean instances at runtime.

C.2 EJB 1.1 Container Provider's responsibility

This section defines the EJB 1.1 Container's responsibilities for providing the runtime environment to the enterprise bean instances. The requirements described here are considered to be the minimal requirements; a Container may choose to provide additional functionality that is not required by the EJB specification.

An EJB 1.1 Container must make the following APIs available to the enterprise bean instances at runtime:

- Java 2 Platform, Standard Edition, v 1.2 (J2SE) APIs
- EJB 1.1 APIs
- JNDI 1.2
- JTA 1.0.1, the `UserTransaction` interface only
- JDBC™ 2.0 extension
- JavaMail 1.1, sending mail only

The following subsections describes the requirements in more detail.

C.2.1 Java 2 Platform, Standard Edition, v 1.2 (J2SE) APIs requirements

The Container must provide the full set of Java 2 Platform, Standard Edition, v 1.2 (J2SE) APIs. The Container is not allowed to subset the Java 2 platform APIs.

The EJB Container is allowed to make certain Java 2 platform functionality unavailable to the enterprise bean instances by using the Java 2 platform security policy mechanism. The primary reason for the Container to make certain functions unavailable to enterprise bean instances is to protect the security and integrity of the EJB Container environment, and to prevent the enterprise bean instances from interfering with the Container's functions.

The following table defines the Java 2 platform security permissions that the EJB Container must be able to grant to the enterprise bean instances at runtime. The term "grant" means that the Container must be able to grant the permission, the term "deny" means that the Container should deny the permission.

Table 22 Java 2 Platform Security policy for a standard EJB Container

Permission name	EJB Container policy
java.security.AllPermission	deny
java.awt.AWTPermission	deny
java.io.FilePermission	deny
java.net.NetPermission	deny
java.util.PropertyPermission	grant "read", "*" deny all other
java.lang.reflect.ReflectPermission	deny
java.lang.RuntimePermission	grant "queuePrintJob", deny all other
java.lang.SecurityPermission	deny
java.io.SerializablePermission	deny
java.net.SocketPermission	grant "`connect", "*"` [Note A], deny all other

Notes:

[A] This permission is necessary, for example, to allow enterprise beans to use the client functionality of the Java IDL API and RMI-IIOP packages that are part of Java 2 platform.

Some Containers may allow the Deployer to grant more, or fewer, permissions to the enterprise bean instances than specified in Table 22. Support for this is not required by the EJB specification. Enterprise beans that rely on more or fewer permissions will not be portable across all EJB Containers.

C.2.2 EJB 1.1 requirements

The container must implement the EJB 1.1 interfaces.

C.2.3 JNDI 1.2 requirements

At the minimum, the EJB Container must provide a JNDI API name space to the enterprise bean instances. The EJB Container must make the name space available to an instance when the instance invokes the `javax.naming.InitialContext` default (no-arg) constructor.

The EJB Container must make available at least the following objects in the name space:

- The home interfaces of other enterprise beans.
- The resource factories used by the enterprise beans.

The EJB specification does not require that all the enterprise beans deployed in a Container be presented with the same JNDI API name space. However, all the instances of the same enterprise bean must be presented with the same JNDI API name space.

C.2.4 JTA 1.0.1 requirements

The EJB Container must include the JTA 1.0.1 extension, and it must provide the `javax.transaction.UserTransaction` interface to enterprise beans with bean-managed transaction demarcation through the `javax.ejb.EJBContext` interface, and also in JNDI under the name `java:comp/UserTransaction`, in the cases required by the EJB specification.

The EJB Container is not required to implement the other interfaces defined in the JTA specification. The other JTA interfaces are low-level transaction manager and resource manager integration interfaces, and are not intended for direct use by enterprise beans.

C.2.5 JDBC™ 2.0 extension requirements

The EJB Container must include the JDBC 2.0 extension and provide its functionality to the enterprise bean instances, with the exception of the low-level XA and connection pooling interfaces. These low-level interfaces are intended for integration of a JDBC driver with an application server, not for direct use by enterprise beans.

C.2.6 Argument passing semantics

The enterprise bean's home and remote interfaces are *remote interfaces* for Java RMI. The Container must ensure the semantics for passing arguments conform to Java RMI. Non-remote objects must be passed by value.

Specifically, the EJB Container is not allowed to pass non-remote objects by reference on inter-EJB invocations when the calling and called enterprise beans are collocated in the same JVM. Doing so could result in the multiple beans sharing the state of a Java object, which would break the enterprise bean's semantics.

Appendix D Frequently asked questions

This Appendix provides the answers to a number of frequently asked questions.

D.1 Client-demarcated transactions

The Java 2, Enterprise Edition specification [9] defines how a client can obtain the `javax.transaction.UserTransaction` interface using JNDI.

The following is an example of how a Java application can obtain the `javax.transaction.UserTransaction` interface.

```
...
Context ctx = new InitialContext();
UserTransaction utx =
    (UserTransaction)ctx.lookup(“java:comp/UserTransaction”);

//
// Perform calls to enterprise beans in a transaction.
//
utx.begin();
... call one or more enterprise beans
utx.commit();
...
```

D.2 Container managed persistence

EJB 2.0 supports both field-based and method-based container managed persistence. The EJB 2.0 specification recommends that the new EJB 2.0 mechanism be used for new work because of the added functionality that it provides. Before making a decision, however, the Bean Provider should evaluate the advantages and limitations of both mechanisms and choose the one that best supports his or her needs.

The use of both EJB 2.0 method-based and EJB1.1 field-based container managed persistence entity beans can be combined in the same EJB 2.0 application. The beans that are written to the EJB1.1 container managed persistence API, however, must be indicated as such in the EJB 2.0 deployment descriptor.

EJB 2.0 containers must support the EJB 1.1 mechanism for container managed persistence. EJB 1.1 entity beans and deployment descriptors are supported by EJB 2.0. The EJB 1.1 container managed persistence mechanism has not been deprecated.

D.3 Inheritance

The current EJB specification does not specify the concept of *component inheritance*. There are complex issues that would have to be addressed in order to define component inheritance (for example, the issue of how the primary key of the derived class relates to the primary key of the parent class, and how component inheritance affects the parent component's persistence).

However, the Bean Provider can take advantage of the Java language support for inheritance as follows:

- *Interface inheritance*. It is possible to use the Java language interface inheritance mechanism for inheritance of the home and remote interfaces. A component may derive its home and remote interfaces from some "parent" home and remote interfaces; the component then can be used anywhere where a component with the parent interfaces is expected. This is a Java language feature, and its use is transparent to the EJB Container.

- *Implementation class inheritance*. It is possible to take advantage of the Java class implementation inheritance mechanism for the enterprise bean class. For example, the class `CheckingAccountBean` class can extend the `AccountBean` class to inherit the implementation of the business methods.

D.4 How to obtain database connections

Section 19.4 specifies how an enterprise bean should obtain connections to resources such as JDBC connections. The connection acquisition protocol uses resource manager connection factory references that are part of the enterprise bean's environment.

The following is an example of how an enterprise bean obtains a JDBC connection:

```
public class EmployeeServiceBean implements SessionBean {
     EJBContext ejbContext;

     public void changePhoneNumber(...) {
          ...

          // obtain the initial JNDI context
          Context initCtx = new InitialContext();

          // perform JNDI lookup to obtain resource manager
          // connection factory
          javax.sql.DataSource ds = (javax.sql.DataSource)
               initCtx.lookup("java:comp/env/jdbc/EmployeeAppDB");

          // Invoke factory to obtain a connection. The security
          // principal is not given, and therefore
          // it will be configured by the Deployer.
          java.sql.Connection con = ds.getConnection();
          ...
     }
}
```

D.5 Session beans and primary key

The EJB 1.1 specification specifies the Container's behavior for the cases when a client attempts to access the primary key of a session object. In summary, the Container must throw an exception on a client's attempt to access the primary key of a session object.

D.6 Copying of parameters required for EJB calls within the same JVM

The enterprise bean's home and remote interfaces are *remote interface* in the Java RMI sense. The Container must ensure the Java RMI argument passing semantics. Non-remote objects must be passed by value.

Specifically, the EJB Container is not allowed to pass local objects by reference on inter-enterprise bean invocations when the calling and called enterprise beans are collocated in the same JVM. Doing so could result in the multiple beans sharing the state of a Java object, which would break the enterprise bean's semantics.

Appendix E Revision History

This appendix lists the significant changes that have been made during the development of the EJB 2.0 specification.

E.1 Version 0.1

Created document from EJB 1.1 Public Draft 3.

Revised introductory chapters to reflect goals of EJB 2.0.

Added message-driven beans to the EJB architecture:

- New chapter: “Message-driven Bean Component Contract”.
- Additions to “Overview” chapter to reflect new message-driven bean component type.
- Additions to Transactions chapter: scenarios; examples; contracts for message-driven beans using container-managed transaction demarcation and bean-managed transaction demarcation.
- Additions to “Exception Handling” chapter for exceptions from message-driven beans.
- Introduction of JMS destination references in “Enterprise Bean Environment” chapter.
- Additions to deployment descriptor for message-driven bean component type, jms destination references, message-driven bean message-selector and concurrency-mode.

E.2 Version 0.2

Minor modifications to message-driven bean contracts:

- Removed serialized option for message-concurrency-mode. Message-driven beans must be prepared to handle out-of-order messages in any case, and the option inhibited ability of the container to provide concurrency.
- MessageDrivenBean modified to extend MessageListener.
- Minor clarifications to Transactions and Exceptions chapters for message-driven beans.

E.3 Version 0.3

Specified of new contracts for entity beans with container-managed persistence to address the limitations of the field-based approach to container-managed persistence in the EJB 1.1 specification. The new mechanisms are added:

- To support the requirement for container managed relationships among entity beans and between an entity bean and its dependent object classes.
- To provided the basis for a portable finder query syntax.
- To support more efficient vendor implementations leveraging lazy loading mechanisms, dirty detection, reduce memory footprints, avoid data aliasing problems, etc.
- To provide the foundation for pluggable persistence managers.

Changes in version 0.3:

- Added new chapter: “Entity Bean Component Contract for Container Managed Persistence”.
- Separated out component contract for bean-managed persistence into separate chapter, “Entity Bean Component Contract for Bean Managed Persistence.”
- Removed text related to EJB 1.1 component contracts for container-managed persistence.
- Added architected support for container-managed relationships for entity beans with container-managed persistence.
- Additions to component contract in Overview chapter.
- Added deployment descriptor elements for new container-managed persistence architecture to support relationships and dependent objects for entity beans with container managed persistence, and to support versioning of entity beans with regard to cmp-version.

E.4 Version 0.4

Changes to the EJB 2.0 container-managed persistence architecture:

- Shifted responsibility for management of persistent state and relationships entirely to persistence manager.
- Simplified Bean Provider’s view of entity beans with container managed persistence to JavaBeans-like API.
- Specified contract for persistent state management and lifecycle contract.
- Clarified distinction among client view, bean provider’s view, and persistence manager’s view.
- Clarified distinctions between dependent object classes and dependent value classes.

E.5 Version 0.5

Added home business methods for entity beans.

Generalized the naming of create methods for Session beans and Entity beans.

Introduced local transaction optimization in “Support for Transactions” chapter.

Added chapter to retain specification of EJB 1.1 contract for container-managed persistence.

Brought the EJB 2.0 specification into sync with the EJB 1.1 Final Release, by incorporating modifications that were made to the EJB 1.1 specification after the initial creation of the EJB 2.0 document.

E.6 Version 0.6

Added specification of EJB QL, a declarative syntax for finder methods for entity beans with container managed persistence.

Added connection and transaction management contracts between the container and the persistence manager for entity beans with container-managed persistence.

Added object interaction diagrams for entity beans with container-managed persistence.

Simplified the deployment descriptor elements for dependent objects and relationships.

Updated Appendixes: added list of items for future releases, added FAQ for container-managed persistence.

Incorporated changes specified in the EJB 1.1 Errata document.

E.7 Version 0.7

Introduced select methods and extended EJB QL to provide an internal query capability for entity bean classes.

Added clarifications to container-managed persistence runtime model for relationship management, assignment semantics, collection semantics, and differences between dependent object classes and dependent value classes.

Relaxed restrictions to allow sharing of dependent object classes among beans.

Added run-as security identity functionality.

Provided generalization of JMS destination references in terms of resource environment references and removed JMS destination references proper.

Revised chapters related to interoperability to reflect EJB Interoperability Architecture document.

Revision of the Runtime Environment chapter to reflect J2EE 1.3.

Added Appendix chapters for EJB 1.1 Runtime and EJB 1.1 Deployment descriptor.

E.8 Participant Draft

Minor clarification to description of Application Assembler's responsibilities in linking EJB references.

E.9 Public Draft

Removed restrictions on sharing of instances of dependent object classes.

Merged EJB Interoperability document into Chapter 18.

Renamed EJB-QL to EJB QL.

Corrected minor inconsistencies in treatment of finder methods.

E.10 Public Draft 2

Relaxed ownership restrictions on dependent object classes.

Added delete() method on dependent object classes.

Added cascade-delete deployment descriptor element for dependent object involved in one-to-one and one-to-many relationships.

Introduced primary keys for instances of dependent object classes.

Clarified semantics of detached instances of dependent object classes.

Removed the requirement for the deepCopy() method.

Clarified naming rules for accessor methods for entity beans with container managed persistence and cmp-fields and cmr-fields.

Revised creation protocol for dependent object classes; added ejbCreate and ejbPostCreate methods for dependent object classes.

Removed the requirement that dependent object classes not be serializable.

Added clarification that findByPrimaryKey(primaryKey) method for entity beans must not be overloaded.

Removed ejbSelectInEntity methods.

Added ejbSelect methods to dependent object classes.

Removed the query-spec element.

Clarified the semantics of mutation operations on relationships.

Removed requirement for persistence manager to raise DuplicateKeyException in ejbCreate methods.

Added restriction that dependent object instances not be created and cmr-fields not be modified in ejb-Create methods (but rather in ejbPostCreate methods).

Extended and aligned EJB QL to reflect above changes in container managed persistence.

Added full BNF for EJB QL

Added range variables for dependent object classes to EJB QL to provide for queries for searching detached dependent objects.

Clarified EJB QL equality semantics to utilize dependent object identity based on primary keys.

Further clarified EJB QL type system, naming, and path expression semantics.

Clarified allowable arguments for EJB QL finder expressions and introduced constructor expressions to convert primitive types to equivalent Java object types.

Extended EJB QL Select clause to allow casting to ejbObject types and to handle single valued references.

Removed requirement that MessageDrivenBean interface extend javax.jms.MessageListener; added requirement that message driven bean class must implement javax.jms.MessageListener interface.

Added requirement that container must support deployment of a message driven bean as a consumer of a JMS queue or a JMS durable subscription.

Removed the jms- prefix from deployment descriptor elements specific to message driven beans.

Modified discussion of use of local transaction optimization by the container to reflect changes being made to the J2EE platform specification [9].

Added requirement that the Bean Provider must use only the Required, RequiresNew, or Mandatory transaction attributes for methods defined in the home or remote interface of an entity bean with EJB 2.0 container managed persistence.

Modified discussion of use of connection sharing by the container to reflect changes being made to the J2EE platform specification [9].

Added res-sharing-scope deployment descriptor element to allow Bean Provider to be able to indicate whether connections were shareable or unshareable.

Renamed runAs-specified-identity deployment descriptor element to run-as-specified-identity.

E.11 Proposed Final Draft

Loosened the container's requirement for raising the java.rmi.RemoteException in the case of concurrent calls to a stateful session object.

Corrected inconsistency in specification of error behavior for javax.ejb.Home.remove(Object primaryKey) when called on session bean: javax.ejb.RemoveException should be thrown.

Clarified that the Bean Provider does not need to release a session bean's reference to a resource manager connection factory in ejbPassivate.

Renamed the delete() method for dependent object classes to remove().

Clarified that the Bean Provider should program a dependent object class to be able to handle loopback calls if it is possible that such loopback calls will occur.

Added requirement that Container should provide to the Persistence Manager the functionality to obtain an EJBObject from a primary key for the given transaction context.

Removed the restriction that the FROM clause of a finder method query specify a range variable only of the abstract schema type of the entity bean for which the finder method is defined: range variables of other abstract schema types are now allowed.

Clarified that the get and set accessor methods of entity beans with container managed persistence and dependent object classes must be defined as public (and not as public or protected).

Clarified the semantics for assignment for multi-valued cmr-fields and semantics of methods of java.util.Collection API applied to these fields.

Clarification of literal syntax for EJB QL.

Added clarification that the container must not attempt the local transaction optimization on transactions imported from a different container.

Clarified that enterprise beans must not call the javax.jms.MessageConsumer.setMessageListener or javax.jms.MessageConsumer.getMessageListener method.

Changed dependent deployment descriptor element to make it optional for a dependent object class to have cmp-fields.

Clarified that if the Application Assembler has assigned some methods (but not all) of an enterprise bean to security roles, the Deployer should configure the bean's security so that no access is permitted to the other methods.

Changed entity deployment descriptor element to make it optional for an entity bean with container managed persistence and cmp-version 2.x to have cmp-fields.

Corrected use-caller-identity deployment descriptor element to be EMPTY rather than PCDATA.

Changed EJB 2.0 DTD URL to http://java.sun.com/dtd/ejb-jar_2_0.dtd.

Removed ejb-ref-name from ejb-entity-ref element: Bean Provider does not specify this.

Changed capitalization of the values of the following deployment descriptor elements to maintain uniform deployment descriptor capitalization convention: multiplicity (One, Many), subscription-durability (Durable, NonDurable), Acknowledge-mode (Auto-acknowledge, Dups-ok-acknowledge).

Clarified that containers must implement the CORBA 2.3.1 requirements for code set support.

Added requirements for obtaining stub classes and client-view classes: containers are required to provide stubs for all beans that are referenced from a J2EE application; containers are required to provide portable system value classes that may be instantiated by clients in other vendor's containers; client-view classes are packaged in the referencing J2EE application by the assembler and deployer.

Updated the transaction interoperability requirements to follow the CORBA Object Transaction Service v1.2; updated the requirements on transaction policies in EJB references accordingly.

Updated the requirements for lookup of EJBHome objects using the CORBA Interoperable Naming Service: the root NamingContext is accessed at the host, port, and object key advertised by the server's COSNaming service.

Updated the list of required SSL/TLS ciphersuites for transport-layer security interoperability.

Updated the requirements for security information in EJB references based on the CORBA Common Secure Interoperability version 2 (CSIv2) specification.

Updated the requirements for carrying principals and authentication data in IIOP message security contexts, based on the CSIv2 specification.

Clarified that the String returned from EJBContext.getCallerPrincipal().getName() is derived from (i.e. may not be exactly the same as) the caller information obtained from the IIOP message and transport layers.

Aligned ejb-jar file packaging with interoperability requirements; clarified responsibilities of roles.

Index