Sun Microsystems

Enterprise JavaBeans™ Specification

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.

Please send technical comments on this Draft by June 10, 1999 to:

ejb-spec-comments@sun.com

Please send product and business questions to:

ejb-marketing@sun.com

Version 1.1, Public Draft

Vlada Matena & Mark Hapner

May 7, 1999

NOTICE

TRADEMARKS

DISCLAIMER OF WARRANTIES

PARTICULAR PURPOSE, OR NON-INFRINGEMENT; THAT THE CONTENTS OF THE SPECIFICATION ARE SUITABLE FOR ANY PURPOSE; NOR THAT ANY PRACTICE OR IMPLEMENTATION OF SUCH CONTENTS WILL NOT INFRINGE ANY THIRD PARTY PATENTS, COPYRIGHTS, TRADE SECRETS OR OTHER RIGHTS. This document does not represent any commitment to release or implement any portion of this Specification in any product(s).

THIS SPECIFICATION COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS. CHANGES ARE PERIODICALLY ADDED TO THE INFORMATION HEREIN; THESE CHANGES WILL BE INCORPORATED IN NEW VERSIONS OF THE SPECIFICATION, IF ANY. SUN MAY MAKE IMPROVEMENTS AND/OR CHANGES IN THE PRODUCT(S) AND/OR THE PROGRAM(S) DESCRIBED IN THIS SPECIFICATION AT ANY TIME. Any use of such changes in the Specification will be governed by the then current terms and conditions 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 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 hold Sun harmless from any claims based: (i) your use of the Specification, (ii) from the use or distribution of your pre-FCS Java application, applet and/or clean room implementation, and (iii) 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

Use, duplication, or disclosure by the U.S. Government is subject to the restrictions set forth in this license and as provided in DFARS 227.7202-1(a) and 227.7202-3(a) (1995), DFARS 252.227-7013(c)(1)(ii)(Oct 1988), FAR 12.212(a) (1995), FAR 52.227-19 (June 1987), or FAR 52.227-14(ALT III) (June 1987), as applicable.

REPORT

As an Evaluation Posting of this Specification, 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 that such Feedback is provided on a non-proprietary and non-confidential basis and (ii) grant to Sun a perpetual, non-exclusive, worldwide, fully paid-up, irrevocable license to incorporate, disclose, and use without limitation the Feedback for any purpose relating to the Specification and future versions, implementations, and test suites thereof.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

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 1.1

We have tightened the Entity bean specification, and made support for Entity beans mandatory for Container Providers.

The other changes in the EJB 1.1 specification were motivated by the goal to improve the support for the development, application assembly, and deployment of ISV-produced enterprise beans. The primary changes we have made to the specification are as follows:

- Enhanced support for the enterprise bean's environment. The Bean Provider must specify all the bean's environmental dependencies using entries in a JNDI naming context.
- Added support for Application Assembly in the deployment descriptor.
- Clearly separated the responsibilities of the Bean Provider and Application Assembler.
- Removed the EJB 1.0 deployment descriptor features that describe the Deployer's output. The role of the deployment descriptor is to describe the information that is the *input* to the Deployer, not the Deployer's *output*.

The changes affected mainly Chapters 11, 14, 15, and 16. We minimized the impact on the server vendors who implemented support for EJB 1.0 in their runtime. The only change to the runtime API of the EJB Container is the replacement of the `java.security.Identity` class with the `java.security.Principal` interface, necessitated by changes in JDK 1.2.

We have also added a number of clarifications and corrections to the specification based on the input that we have received from the reviewers.

1.3 Application compatibility and interoperability

EJB 1.1 attempts to provide a high degree of application compatibility for enterprise beans that were written for the EJB 1.0 specification. While the deployment descriptor of EJB 1.0 based enterprise beans must be converted to the EJB 1.1 XML format, the enterprise bean code does not have to be changed or re-compiled to run in an EJB 1.1 Container, except in the following situations:

- The bean uses the `javax.jts.UserTransaction` interface. The package name of the `javax.jts` interface has changed to `javax.transaction`, and there has been minor changes to the exceptions thrown by the methods of this interface. An enterprise bean that uses the `javax.jts.UserTransaction` interface needs to be modified to use the new name `javax.transaction.UserTransaction`.

- The bean uses the `getCallerIdentity()` or `isCallerInRole(Identity identity)` methods of the `javax.ejb.EJBContext` interface. These method were deprecated in EJB 1.1 because the class `java.security.Principal` is deprecated in Java 2. While a Container Provider may choose to provide a backward compatible implementation of these two methods, the Container Provider is not required to do so. An enterprise bean written to the EJB 1.0 specification needs to be modified to use the new methods to work in *all* EJB 1.1. Containers.

- The bean is an entity bean with container-managed persistence. The required return value of `ejbPostCreate(...)` is different in EJB 1.1 than in EJB 1.0. An enterprise bean with container-managed persistence written to the EJB 1.0 specification needs to be recompiled to work with all EJB 1.1 compliant Containers.

- The bean is an entity bean whose finders do not define the `FinderException` in the methods' throws clauses. EJB 1.1 requires that all finders define the `FinderException.`

- The bean is an entity bean that uses a `UserTransaction` interface. In EJB 1.1, an entity bean must not use the `UserTransaction` interface.

- The bean uses the `UserTransaction` interface and implements the `SessionSynchronization` interface at the same time. This is disallowed in EJB 1.1.

- The bean violates any of the additional semantic restrictions that are defined in EJB 1.1 but were not defined in EJB 1.0.

The client view of an enterprise bean is the same in EJB 1.0 and EJB 1.1. This means that enterprise beans written to EJB 1.1 can seamlessly interoperate with those written to EJB 1.0, and vice versa.

1.4 Acknowledgments

Rick Cattell, Linda Demichiel, Shel Finkelstein, Graham Hamilton, Li Gong, Rohit Garg, Susan Cheung, Hans Hrasna, Sanjeev Krishnan, Kevin Osborn, Bill Shannon, Anil Vijendran, and Larry Cable have provided invaluable input to the design of Enterprise JavaBeans architecture.

The Enterprise JavaBeans architecture is a broad effort that includes contributions from numerous groups at Sun and at partner companies. The ongoing specification review process has been extremely valuable, and the many comments that we have received helped us to define the specification.

We would also like to thank all the reviewers who sent us feedback during the public review period. Their input helped us to improve the specification.

1.5 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 10 define Entity Beans: Chapter 8 discusses the client view, Chapter 9 presents the Entity Bean component contract, and Chapter 10 outlines an example Entity Bean scenario.

Chapters 11 through 15 discuss transactions, exceptions, distribution, environment, and security.

Chapters 16 and 17 describe the format of the ejb-jar file and its deployment descriptor.

Chapter 18 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 19 summarizes the responsibilities of the individual EJB Roles.

Chapter 20 is the Enterprise JavaBeans API Reference.

Chapter 21 provides a list of related documents.

1.6 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.`

Chapter 2 Goals

2.1 Overall goals

We have set the following goals for the Enterprise JavaBeans (EJB) architecture:

- *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; resource 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 non-Java programming language applications.*
- *The Enterprise JavaBeans architecture will be compatible with the CORBA protocols.*

2.2 Goals for Release 1.0

In Release 1.0, we focused on the following aspects:

- *Define the distinct "EJB Roles" that are assumed by the component architecture.*
- *Define the client view of enterprise Beans.*
- *Define the enterprise Bean developer's view.*
- *Define 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.*
- *Define the format of the ejb-jar file, EJB's unit of deployment.*

2.3 Goals for Release 1.1

In the EJB 1.1 Release, we want to focus on the following aspects:

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

Chapter 3 EJB Roles and Scenarios

3.1 EJB Roles

The Enterprise JavaBeans architecture defines six 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 mainly 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 EJB Role of the Enterprise Bean Provider and the EJB Role of the Application Assembler.*

The following sections define the six 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 bean(s). 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 and 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 of the Application Assembler is one or more ejb-jar files produced by Bean Provider(s), and the output is one or more ejb-jar files that contain the enterprise beans with their application assembly instructions. The application assembly instruction have been inserted 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 Beans' remote and home interfaces, he or she does not have to have any knowledge of the enterprise Beans' implementation.

3.1.3 Deployer

The Deployer takes one or more ejb-jar file 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 factories used by the enterprise beans are present in the operational environment, and bind them to the resource factory references declared in the deployment descriptor), and must follow the application assembly instructions defined by the Application Assembler. To perform his 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 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) is responsible for providing the deployment tools necessary for the deployment of enterprise beans, and for providing the runtime support for the deployed enterprise beans' instances.

From the perspective of the enterprise beans, the Container is a part of the target operational environment. It is the only part of the operational environment with which the enterprise bean instances interact directly at runtime; all interactions of the instances with the operational environment are through the Container.

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).

For Entity Beans with container-managed persistence, the entity container is responsible for persistence of the Entity Beans installed in the container. The Container Provider's tools are used to generate code that moves data between the enterprise Bean's instance variables, and a database or an existing application.

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 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 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.

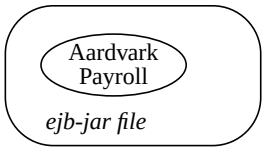
The Wombat application is packaged as a collection of JAR files. A single ejb-jar file contains all the enterprise beans developed by Wombat and also the AardvarkPayroll *enterprise bean developed by Aardvark; the other JAR files contain the non-EJB application components, such as the JSP components. 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 an* ***EJB Server Provider*** *roles.*

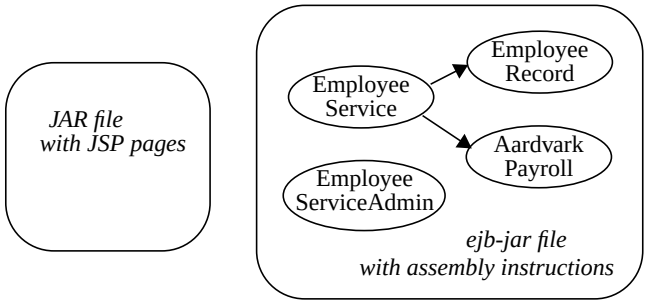
The ABC Enterprise wants to enable its employees to access and update employee records, payroll information, and pension plan information over the Web. The information is stored in ABC's ERP system. ABC buys the employee self-service application from Wombat. To host the application, ABC buys the EJB Container and Server from Acme. ABC's 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 performed 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, which includes tasks 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****.*

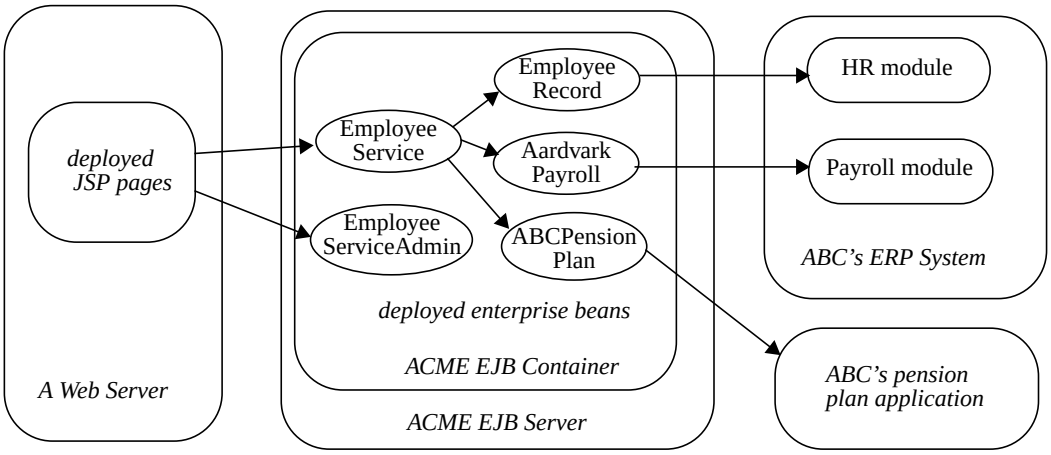
The following diagrams illustrates the products of the various EJB Roles.



(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 metadata, such as a transaction and security attributes, are separate from the enterprise Bean class. This allows the metadata to be managed by tools during application assembly a or deployment (or both).
- 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 the service.
- An enterprise Bean can be included in an assembled application without requiring source code changes or recompilation of the enterprise Bean.
- A client view of an enterprise Bean is defined by the Bean Provider. The client view can be manually defined by the Bean developer, or 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 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 components 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.

A client always uses the same API for object creation, lookup, method invocation, and destruction, regardless of how an enterprise Bean is implemented, and what function it provides to the client.

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 class). 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 typically locates 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; the Container uses the primary key to identify the Entity Object. 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.)

4.2.2 Component contract

This subsection describes the contract between an enterprise Bean and its Container. 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.

- The requirement for the Bean Provider to implement the `ejbCreate`, `ejbPostCreate`, 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 define the enterprise bean's home and remote interfaces. 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 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 Container 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, and the `javax.ejb.EntityContext` interface to entity 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.

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.

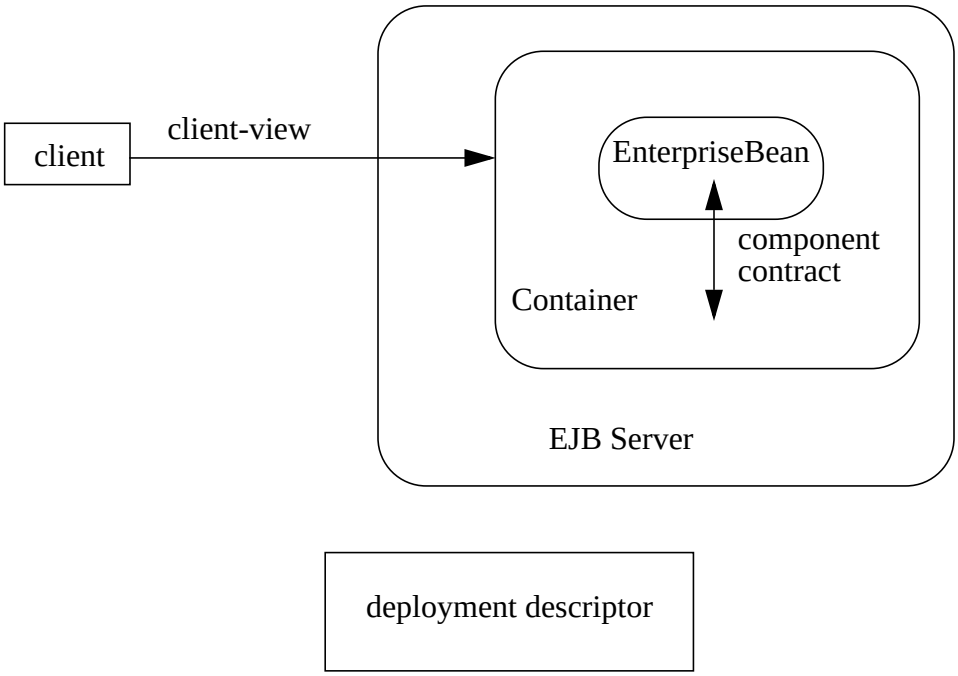
The ejb-jar file includes:

- 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 and entity objects

The Enterprise JavaBeans architecture defines two types of enterprise Beans:

- A session object type.
- An entity object type.

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*** *as a special case of a Session Bean. There are minor differences in the API between stateful (normal) Sessions, and stateless Sessions.*

4.3.2 Entity objects

A typical entity object has the following characteristics:

- *Provides an object view of transactional 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 server 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 on a server 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.

4.4 Standard mapping to CORBA protocols

To help interoperability for EJB environments that include systems from multiple vendors, we define a standard mapping of the Enterprise JavaBeans client-view contract to the CORBA protocols.

The use of the EJB to CORBA mapping by the EJB Server is not a requirement for EJB 1.1 compliance. A later release of the J2EE platform is likely to require the that the J2EE platform vendor implement the EJB to CORBA mapping.

The EJB-to-CORBA mapping covers:

1. Mapping of the EJB remote and home interfaces to RMI-IIOP. This mapping is an identity mapping because every remote and home interface is an RMI-IIOP interface.

2. Propagation of transaction context over IIOP.

3. Propagation of security context over IIOP.

4. Interoperable naming service.

The EJB-to-CORBA mapping not only enables on-the-wire interoperability among multiple vendors' implementations of the EJB Container, but also enables non-Java clients to access server-side applications written as enterprise Beans through standard CORBA APIs.

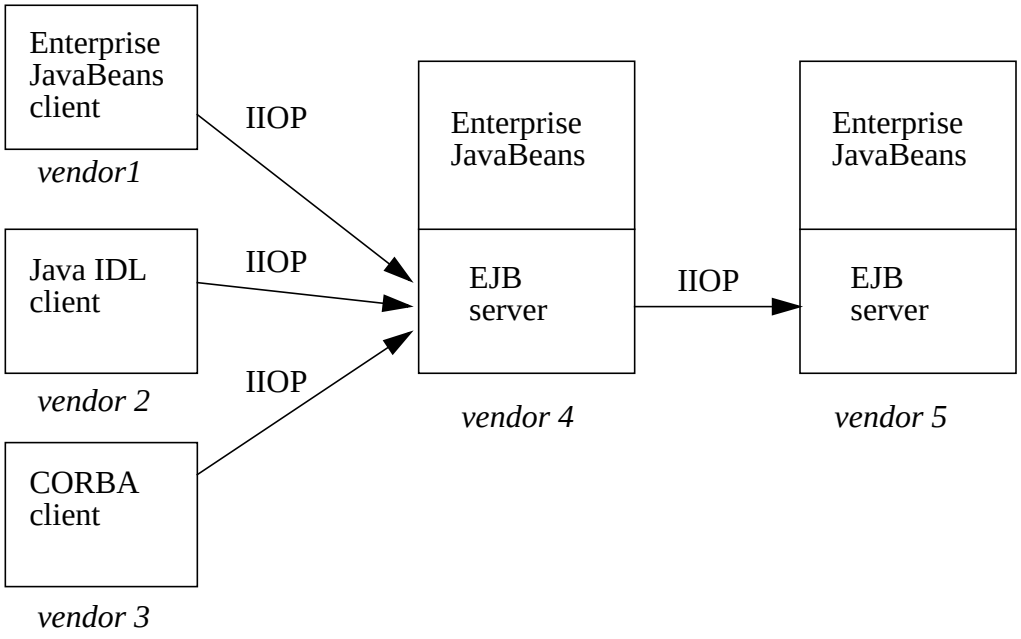
The EJB-to-CORBA mapping depends on the standard CORBA Object Services protocols for the propagation of the transaction and security context.

The CORBA mapping is defined in an accompanying document [8].

While the EJB-to-CORBA mapping defines the mapping of the EJB application interfaces and transaction interoperability, the mapping must be used in conjunction with other CORBA standards to ensure full "on-the-wire" interoperability. For example, multiple EJB servers must agree on the security protocol to achieve seamless interoperability.

The following figure illustrates a heterogeneous environment that includes systems from five different vendors.

Figure 2 Heterogeneous EJB Environment



Chapter 5 Client View of a Session Bean

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

Although the client view of the enterprise Bean is provided by classes implemented by the container, the container itself is transparent to the client.

5.1 Overview

For a client, a session enterprise Bean is a non-persistent object that implements some business logic running on the server. One way to think of a session object is that a session object is 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 enterprise Bean through the session Bean's remote interface. The object that implements this remote interface is called an **EJB object.** An EJB object is a remote Java programming language object accessible from a client through the standard Java™ APIs for remote object invocation [3].

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

Each session EJB 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 an EJB object is location-independent. A client running in the same JVM as the EJB object uses the same API as a client running in a different JVM on the same or different machine.

A client of an enterprise bean can be another enterprise bean deployed in the same or different Container; or 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.

Multiple EJB classes can be installed in a container. The container allows the clients to look up the home interfaces of the installed EJB classes via JNDI. Each home interface provides methods to create and remove the EJB objects of the corresponding EJB class.

The client view of an EJB object is the same, irrespective of the implementation of the enterprise Bean and its container.

5.2 EJB Container

An EJB Container (container for short) is a system that functions as the “container” for enterprise Beans. Enterprise Beans of multiple EJB classes can live in the same container. The client can look up the home interface for a specific EJB class using JNDI. The container is responsible for making the installed EJB classes available to the client through JNDI.

A container is where an enterprise Bean object lives, just as a record lives in a database, and a file or directory lives in a file system.

5.2.1 Locating an enterprise Bean’s home interface

A client locates an enterprise Bean’s home interface using JNDI. For example, a container for `Cart` EJB objects can be located using the following code segment:

```
Context initialContext = new InitialContext();
CartHome cartHome = (CartHome)javax.rmi.PortableRemoteObject.narrow(
     initialContext.lookup(“applications/mall/freds-carts”),
     CartHome.class);
```

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

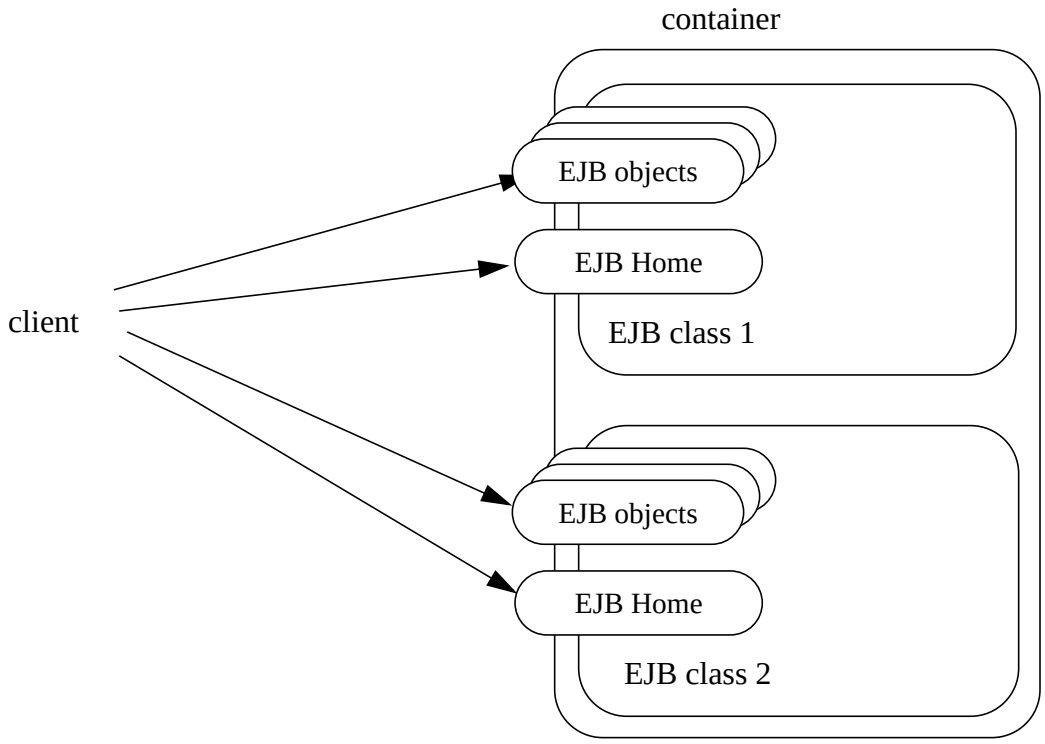
The lifecycle of the distributed object implementing the home interface is Container-specific. A client application should be able to retrieve a home interface object reference, and then use it multiple times during the client application's lifetime.

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 a home interface object reference obtained via JNDI.

5.2.2 What a container provides

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

Figure 3 Client View of a Session EJB Container.



5.3 Home interface

An EJB Container implements the home interface of each enterprise Bean installed in the container. The container makes the home interfaces available to the client through JNDI.

The home interface allows a client to do the following:

- Create a new EJB object.
- Remove an EJB object.
- Get the `javax.ejb.EJBMetaData` interface for the enterprise Bean. The `javax.ejb.EJBMetaData` interface is intended to allow application assembly tools to discover information about the enterprise Bean. The meta-data is defined to allow loose client/server binding and scripting.
- Obtain a handle for the home object. 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 object.

5.3.1 Creating an EJB object

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

The following example illustrates a home interface that defines a single `create(...)` method:

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

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

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

5.3.2 Removing an EJB object

A client may remove an EJB 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 Beans do not have primary keys that are accessible to clients, invoking the `javax.ejb.Home.remove(Object primaryKey)` method on a session results in the `javax.rmi.RemoteException`.

5.4 EJB object

A client never accesses instances of the enterprise Bean's class directly. A client always uses the enterprise Bean's remote interface to access an enterprise Bean's instance. The class that implements the enterprise Bean's remote interface is provided by the container. The distributed objects that this class implements are called `EJB objects`.

An EJB object supports:

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

The implementation of the methods defined in the `javax.ejb.EJBObject` interface is provided by the container.

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 EJB objects, from the client's perspective, appear anonymous. In contrast to entity EJB objects, which expose their identity as a primary key, session objects hide their identity. As a result, the `EJBObject.getPrimaryKey()` and `EJBHome.remove(Object primaryKey)` methods result in a `java.rmi.RemoteException` when called on a Session Bean. when the `EJBMeta-Data.getPrimaryKeyClass()`method is invoked on a meta-data object for a Session Bean, the `java.lang.RuntimeException` results.

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

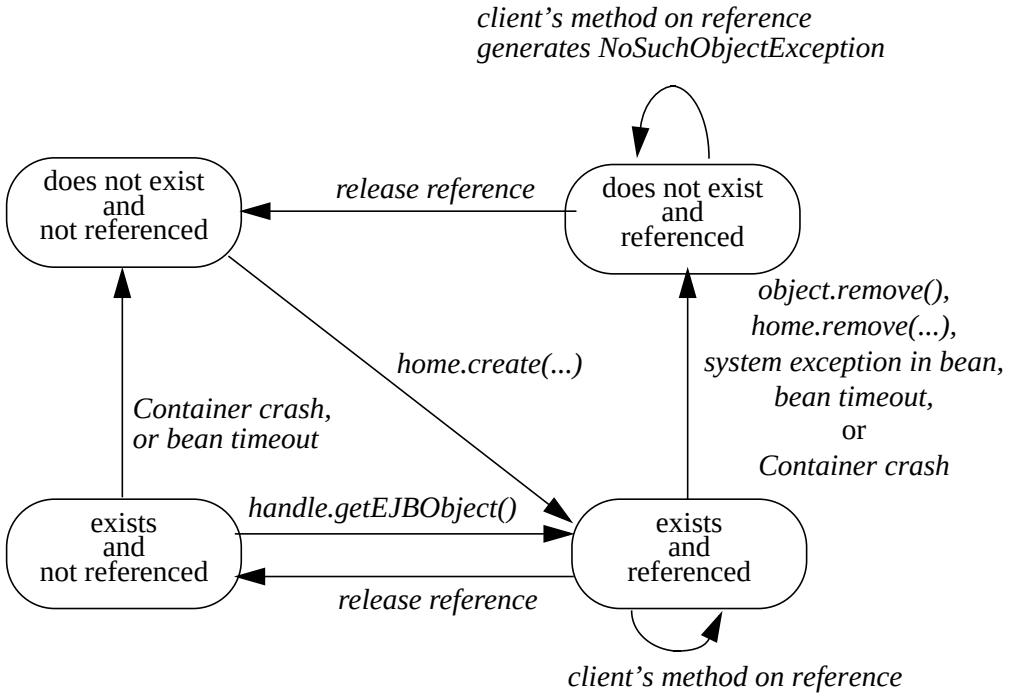
A session EJB object handle can be held beyond the life of a client process by serializing the handle to persistent store. When the handle is later deserialized, the session EJB object it returns will work as long as the object still exists on the server. (An earlier timeout or server crash may have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

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.

5.6 Client view of session Bean's life cycle

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

Figure 4 Lifecycle of a Session EJB.



An EJB object does not exist until it is created. When an object is created by a client, the client gets a reference to the newly created EJB object.

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

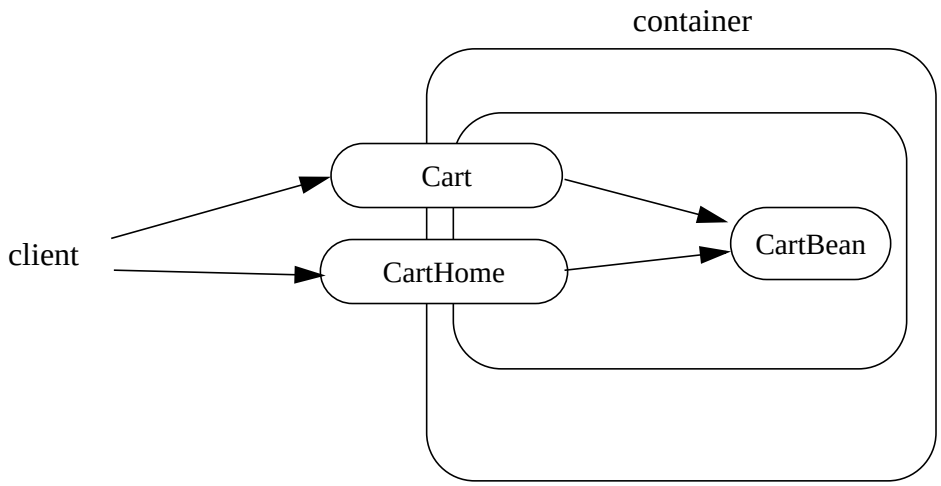
- Invoke application methods on the object through the session Bean's remote interface.
- Get a reference to the object's home interface.
- Get a handle for the object
- Pass the object as a parameter or return value within the scope of the client.
- Remove the object. A container may also remove the object automatically when the object's lifetime expires.

References to an EJB object that does not exist are invalid. Attempted invocations on an object that does not exist will throw `java.rmi.NoSuchObjectException`.

5.7 Creating and using a session Bean

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

Figure 5 Session Bean Example Objects



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

Suppose that the end-user wishes to start the shopping session on a work machine and later complete this session from a home machine. The client might implement this feature by getting the session's handle, sending the serialized handle to his home, and using it to reestablish access to the original `Cart`.

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

```
CartHome cartHome = (CartHome)javax.rmi.PortableRemoteObject.narrow(
    initialContext.lookup(...), CartHome.class);
Cart cart = cartHome.create(...);
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 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 and purchase the content 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 EJB 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 EJB object by invoking the `isIdentical(Object otherObject)` method on of the references.

The following example illustrates the use of the `isIdentical(Object otherObject)` method for a stateful Session Bean.

```
FooHome fooHome = ...; // obtain home of a stateful EJB
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 EJB objects of the same stateless Session Bean have the same object identity, which is assigned by the container.

The `isIdentical(Object otherObject)` method always returns true when used to compare object references of two EJB objects of the same stateless Session Bean.

The following example illustrates the use of the `isIdentical(Object otherObject)` method for a stateless Session Bean.

```
FooHome fooHome = ...; // obtain home of a stateless EJB
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 Bean is, in general, opaque to the client. The result of `getPrimaryKey()` on a Session Bean object 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 a session Bean instance.

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

6.1 Overview

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 client. This state describes the conversation represented by a specific client/instance 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 Bean.
- Its lifetime is typically that of its client.

A session Bean instance's life may also be terminated by a container-specified timeout or the failure of the server it is running on. For this reason, a client must always be prepared to recreate a new instance if it loses the one it is using.

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

On the other hand, cached database data must be explicitly managed by the Bean. A Bean must write any cached database updates prior to the Bean's 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, notifying it when Bean action may be necessary, and providing a full range of services to ensure that the 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

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

A container may only passivate a session Bean when that Bean is **not** in a transaction.

In order to help its container manage its state, a session Bean is specified at deployment as having one of the following state management modes:

- STATELESS - the Bean contains no conversational state between methods; any Bean instance can be used for any client.
- STATEFUL - the Bean contains conversational state which must be retained across methods and transactions.

6.4 Conversational state

A STATEFUL session Bean's conversational state is defined as its field values plus the transitive closure of the objects from the session Bean's fields by following Java object references.

In advanced cases, a session Bean's conversational state may contain open resources, such as open sockets and open database cursors. A container cannot retain such open resources while a session Bean is passivated. A developer of such a session Bean must close and open the resources using 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[1]
- A `null`.
- An EJB object 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 (i.e. the `java:comp/env` JNDI context) or any of its subcontexts.

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`.

The Bean Provider must assume that the content of transient fields may be lost between the `ejbPassivate` and `ejbActivate` notifications.

The enterprise Bean provider must not store the reference of the `SessionContext` object in a `transient` field.

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

The enterprise Bean provider must not store the reference of the environment JNDI naming context or and its subcontexts in a `transient` field.

The following are the requirements for the Container.

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

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 EJBs objects 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 EJB references.

If the EJB 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 EJB 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.

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

While a session 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[2]. Declaring the enterprise Bean's fields as "transient" is, in general, discouraged.

6.4.2 The effect of transaction rollback on conversational state

A session Bean's conversational state is not transactional. It is not automatically rolled back to its initial state if the Bean's transaction rolls back.

If a rollback could result in an inconsistency between a session Bean'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 The protocol between a session Bean and its container

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

[2] 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.

6.5.1 The required *SessionBean* interface

All session Beans must implement the `SessionBean` interface.

The `setSessionContext` method is called by the Bean's container to associate a session Bean instance with its context maintained by the **container**. Typically a session Bean 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.

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

6.5.2 The *SessionContext* interface

All Bean containers provide their Bean instances with a `SessionContext`. This gives the Bean instance access to the instance's context maintained by the container. The `SessionContext` interface has the following methods:

- The `getEJBObject` method returns the EJB object for the instance.
- The `getEJBHome` method returns the home interface for the instance's EJB class.
- The `getCallerPrincipal` method returns the `java.security.Principal` that identifies the invoker of the Bean instance's EJB object.
- The `isCallerInRole` predicate tests if the Bean 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 enterprise beans using container-managed transactions can use this method.
- The `getRollbackOnly` method allows the instance to test if the current transaction has been marked for rollback. Only enterprise beans using container-managed transactions can use this method.
- The `getUserTransaction` method returns the `javax.transaction.UserTransaction` interface that the Bean can use for explicit transaction demarcation, and for obtaining transaction status. Only enterprise beans using bean-managed transactions can use this interface.

6.5.3 The optional *SessionSynchronization* interface

A session Bean can optionally implement the `javax.ejb.SessionSynchronization` interface. This interface can provide the Bean with transaction synchronization notifications. Session Beans use these notifications to manage database data they may cache within transactions.

The `afterBegin` notification signals a session instance that a new transaction has begun. The container invokes this method before the first business method within a 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 instance's client has completed work on its current transaction but prior to committing the instance's resources. 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 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 spec.

The `SessionSynchronization` interface may be implemented only by a stateful Session Bean using container-managed transactions. The `SessionSynchronization` interface must not be implemented by a stateless Session Bean.

There is no need for a Session Bean with bean-managed transaction to rely on the synchronization call backs since 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 enterprise Bean's remote interface defines the business methods callable by a client. The enterprise Bean's remote interface is implemented by the EJB object class generated by the container tools. The EJB object class delegates an invocation of a business method to the matching business method that is implemented in the enterprise Bean class.

6.5.5 Session Bean's *ejbCreate(...)* methods

A client creates a session Bean instance using one of the `create` methods defined in the Bean's home interface. The 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 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 whose signature matches the signature of the `create` method invoked by the client. The input parameters sent from the client are passed to the `ejbCreate` method.

Each session Bean must have at least one `ejbCreate` method. The number and signatures of a session Bean's `create` methods are specific to each EJB 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 instance. Most containers will support many instances of a 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 (i.e. `ejbPassivate`, `beforeCompletion`, etc. methods), and it must serialize these callbacks with the client-invoked business method calls.

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, the container must throw the `javax.rmi.RemoteException` to the second client. Clients are not allowed to make concurrent calls to a session object.

One implication of this rule is that it is illegal to make a "loopback" call to a session Bean instance. An example of a loopback call is when a client calls instance A, instance A calls instance B, and B calls A. The loopback call attempt from B to A would result in the container throwing the `java.rmi.RemoteException` to B.

6.5.7 Transaction context of session Bean methods

A session Bean's `afterBegin` and `beforeCompletion` methods are always called with the proper global transaction context.

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.

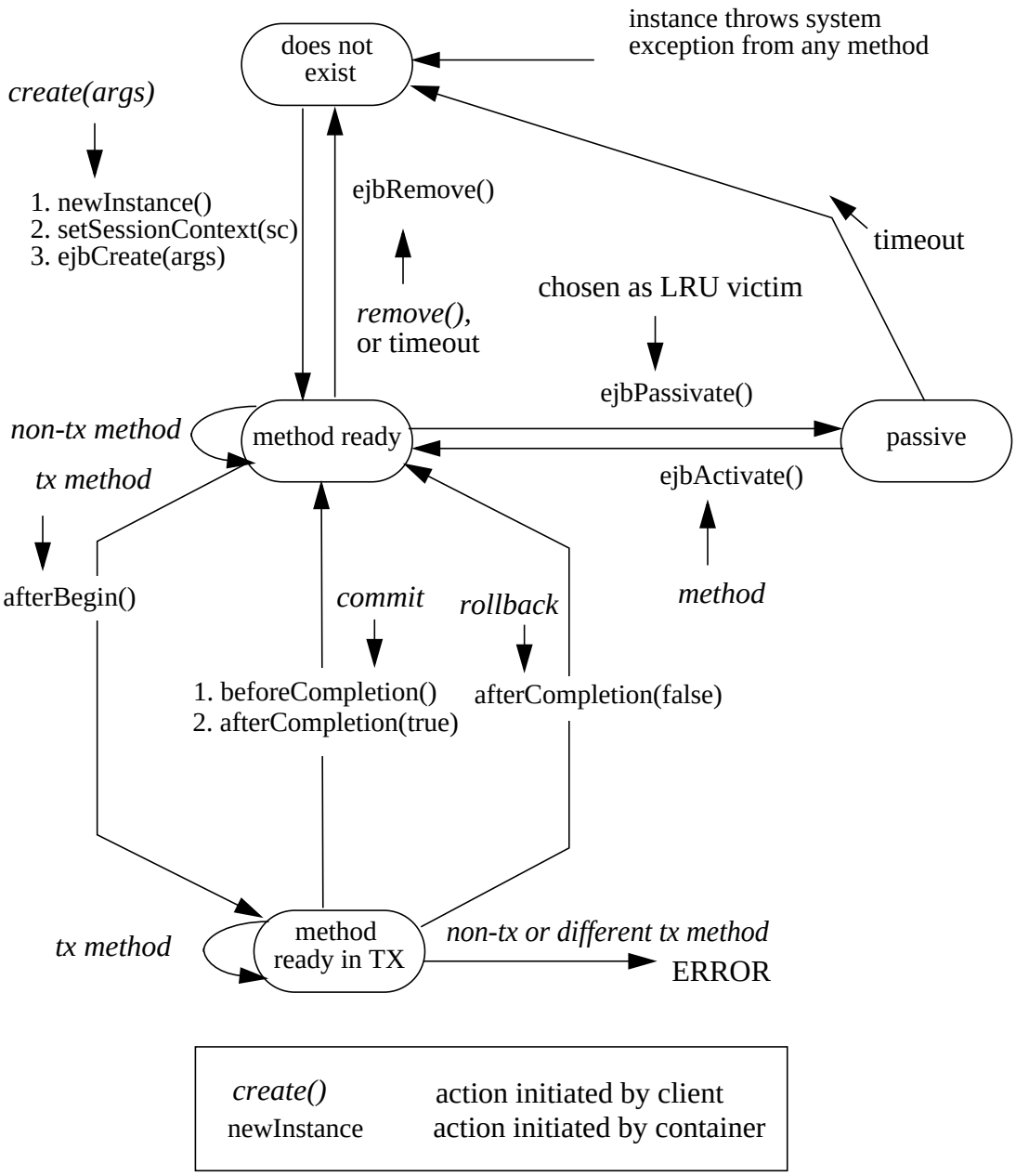
A session Bean's `newInstance`, `setSessionContext`, `ejbCreate`, `ejbRemove`, `ejbPassivate`, `ejbActivate`, and `afterCompletion` methods are called outside of the client's global transaction.

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 executed under the protection of a global 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 (see next section).*

6.6 STATEFUL Session Bean State Diagram

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

Figure 6 Lifecycle of a STATEFUL Session EJB



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

- A session Bean's life starts when a client invokes a `create(...)` method on the enterprise Bean's home interface. This causes the container to invoke `newInstance()` on the Bean

class to create a new memory object for the enterprise Bean. Next, the container calls `setSessionContext()` and `ejbCreate(...)` on the instance, and returns an EJB object reference to the client. The instance is now in the method ready state.

- The Bean instance is now ready for client's business methods. Based on the transaction attributes in the enterprise Bean's deployment descriptor and the transaction context associated with the client's invocation, a business method is or is not executed in a global transaction context (shown as tx method and non-tx method in the diagram). See Chapter 11 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 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 that 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.

- 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 Bean and the deployment descriptor for the method requires that the container invoke the method in a different transaction context than the one that the instance is currently associated with, or in no transaction context.

- If a transaction commit has been requested, the transaction service notifies the container, which issues a `beforeCompletion` on the instance. (The transaction service notifies the container before actually committing the transaction.) When `beforeCompletion` is invoked, the instance should write any cached updates to the database.

- The transaction service then attempts to commit the transaction, resulting in either a commit or rollback. If, in the previous step, a transaction rollback had been requested, the rollback status is reached without issuing `beforeCompletion`.

- When the transaction completes, the container issues `afterCompletion` on the instance, specifying the status of the completion (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 must save 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 instance and the associated EJB Object after a timeout specified by the deployer has expired. All object references and handles for the associated EJB Object become invalid. If a client attempts to invoke

the EJB Object, the Container will throw the `java.rmi.NoSuchObjectException` to the client.

- If a client invokes a passivated session Bean instance, the container will activate the instance. To activate the session instance, the container restores the instance's state from secondary storage and issues `ejbActivate` on it.
- The session Bean is again ready for client methods.
- When the client calls `remove()` on the EJB object, this causes the container to issue `ejbRemove()` on the Bean instance. This ends the life of the session Bean instance. Any subsequent attempt by its client to invoke the instance will cause the `java.rmi.NoSuchObjectException` to be thrown. (This exception is a subclass of `java.rmi.RemoteException`). Note that a container can implicitly invoke the `remove()` method on the instance after the lifetime of the EJB object has expired. The `remove()` method cannot be called when the instance is participating in a transaction. An attempt to remove a session instance while the instance is in a transaction will cause the container to throw the `javax.ejb.RemoveException` to the client.

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 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 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 the 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 with container-managed transactions

Bean method	**Bean method can perform the following operations**	
	Container-managed transactions	**Bean-managed transactions**
constructor	-	-
setSessionContext	SessionContext methods: *getEJBHome* JNDI access to java:comp/env	SessionContext methods: *getEJBHome, getUserTransaction* JNDI access to java:comp/env
ejbCreate ejbRemove ejbActivate ejbPassivate	SessionContext methods: *getEJBHome, getCallerPrincipal, isCallerInRole, getEJBObject* JNDI access to java:comp/env	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, getRollback-Only, isCallerInRole, setRollback-Only, getEJBObject* JNDI access to java:comp/env Resource manager access Enterprise bean access	SessionContext methods: *getEJBHome, getCallerPrincipal, getRollback-Only, isCallerInRole, setRollback-Only, getEJBObject, getUserTransaction* UserTransaction methods JNDI access to java:comp/env Resource manager access Enterprise bean access
afterBegin beforeCompletion	SessionContext methods: *getEJBHome, getCallerPrincipal, getRollback-Only, isCallerInRole, setRollback-Only, getEJBObject* JNDI access to java:comp/env Resource manager access Enterprise bean access	N/A (a bean with bean-managed transactions cannot implement the SessionSynchroni-zation interface)
afterCompletion	SessionContext methods: *getEJBHome, getCallerPrincipal, isCallerInRole, getEJBObject* JNDI access to java:comp/env	

Additional restrictions:

- The `getRollbackOnly` and `setRollbackOnly` methods of the `SessionContext` interface should be used only in the enterprise bean methods that execute in the context of a global transaction. The Container must throw the `java.lang.IllegalStateExcep-tion` if the methods are invoked while the instance is not associated with a global 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 EJB 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 to all session beans with bean-managed transactions.
- 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.
- Accessing resource managers and enterprise beans is disallowed in the `ejbCreate` and `ejbRemove` method of a stateful session bean with container-managed transactions even though the Container has a meaningful transaction and client security context. Allowing transactional resource manager and enterprise bean access in these operations would greatly complicate the instance life cycle because the life cycle diagram in Figure 6 would have to include transitions that deal with rollback of the transaction that includes the `ejbCreate` and `ejbRemove` methods.
- The `UserTransaction` interface is unavailable to the enterprise beans with container-managed transactions.

6.6.2 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 12.

From the client perspective, the corresponding EJB Object does not exist any more. Subsequent invocations through the object reference 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 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 and/or in the business methods, and releases normally 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 Bean in a different or no transaction context. It is also an error to invoke a method on the session Bean if the deployment descriptor would cause the container to execute the method in a different transaction context or no transaction context. The container will throw 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 Bean or its home interface. 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, 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

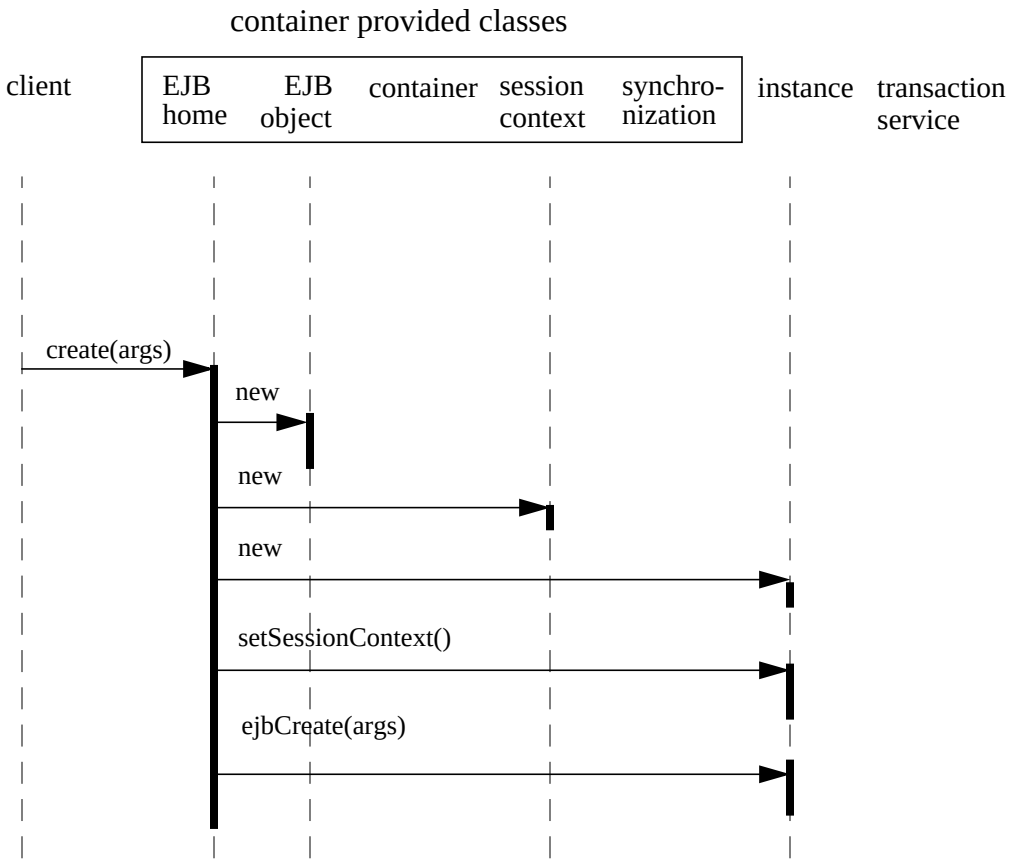
The object interaction diagrams illustrate a box labeled "container-provided classes." These are either classes that are part of the container, or classes that were generated by the container tools. These classes communicate with each other through protocols that are container-implementation specific. 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 transactional session enterprise Bean

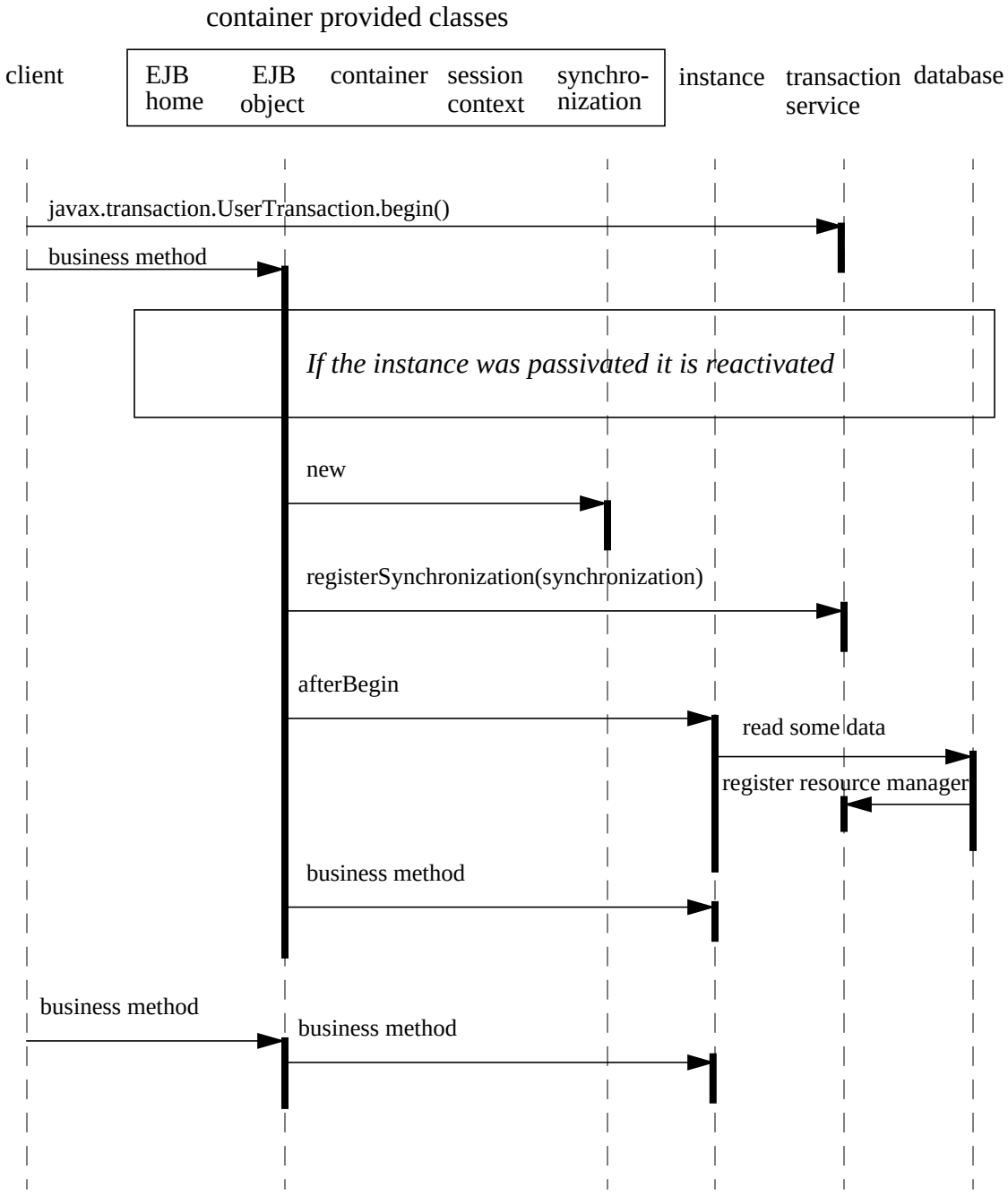
Figure 7 OID for Creation of a Transactional Session EJB.



6.7.3 Starting a transaction

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

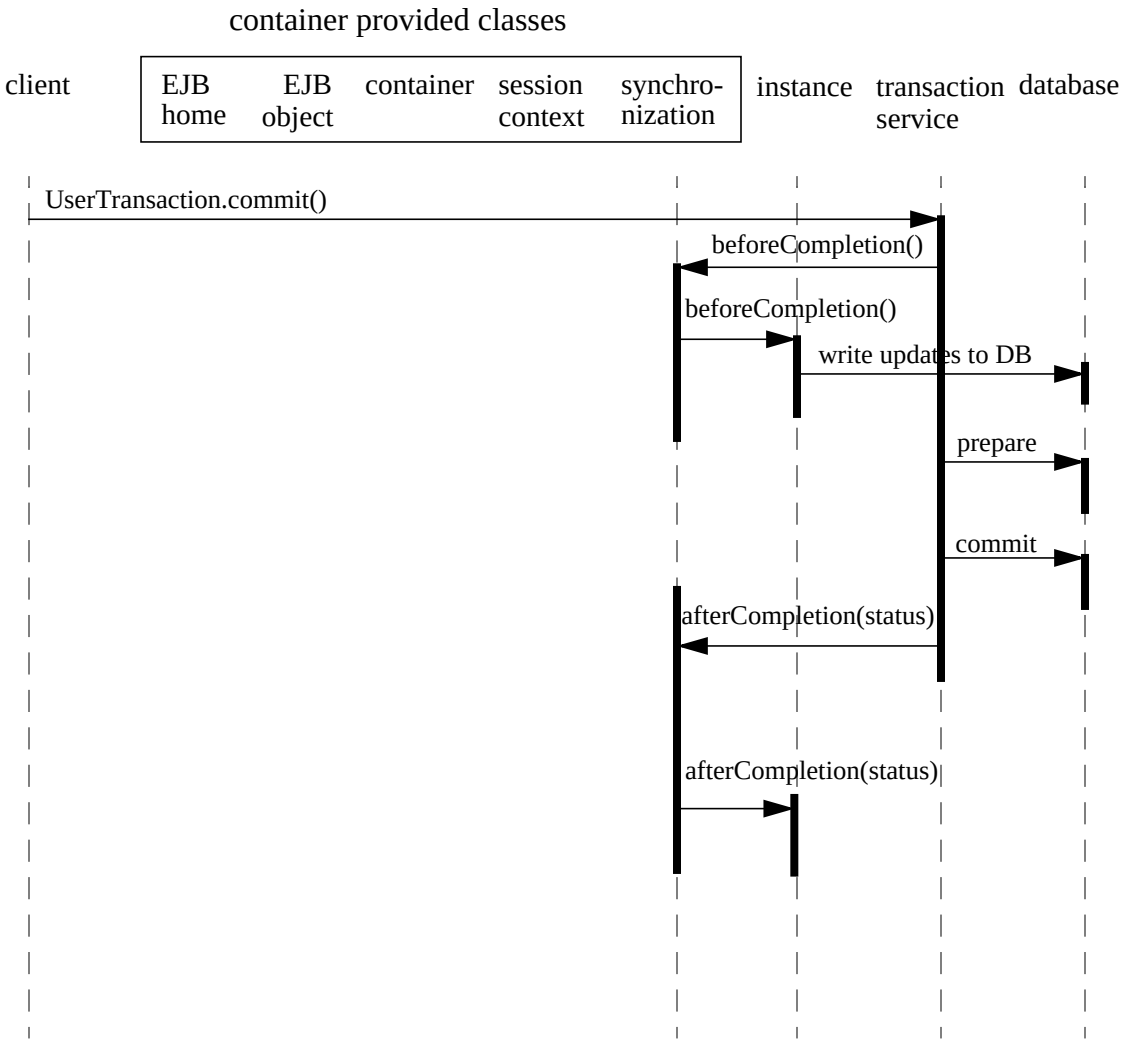
Figure 8 OID for protocol at start of Session EJB Transaction.



6.7.4 Committing a transaction

The following diagram illustrates the transaction synchronization protocol for a session enterprise Bean instance

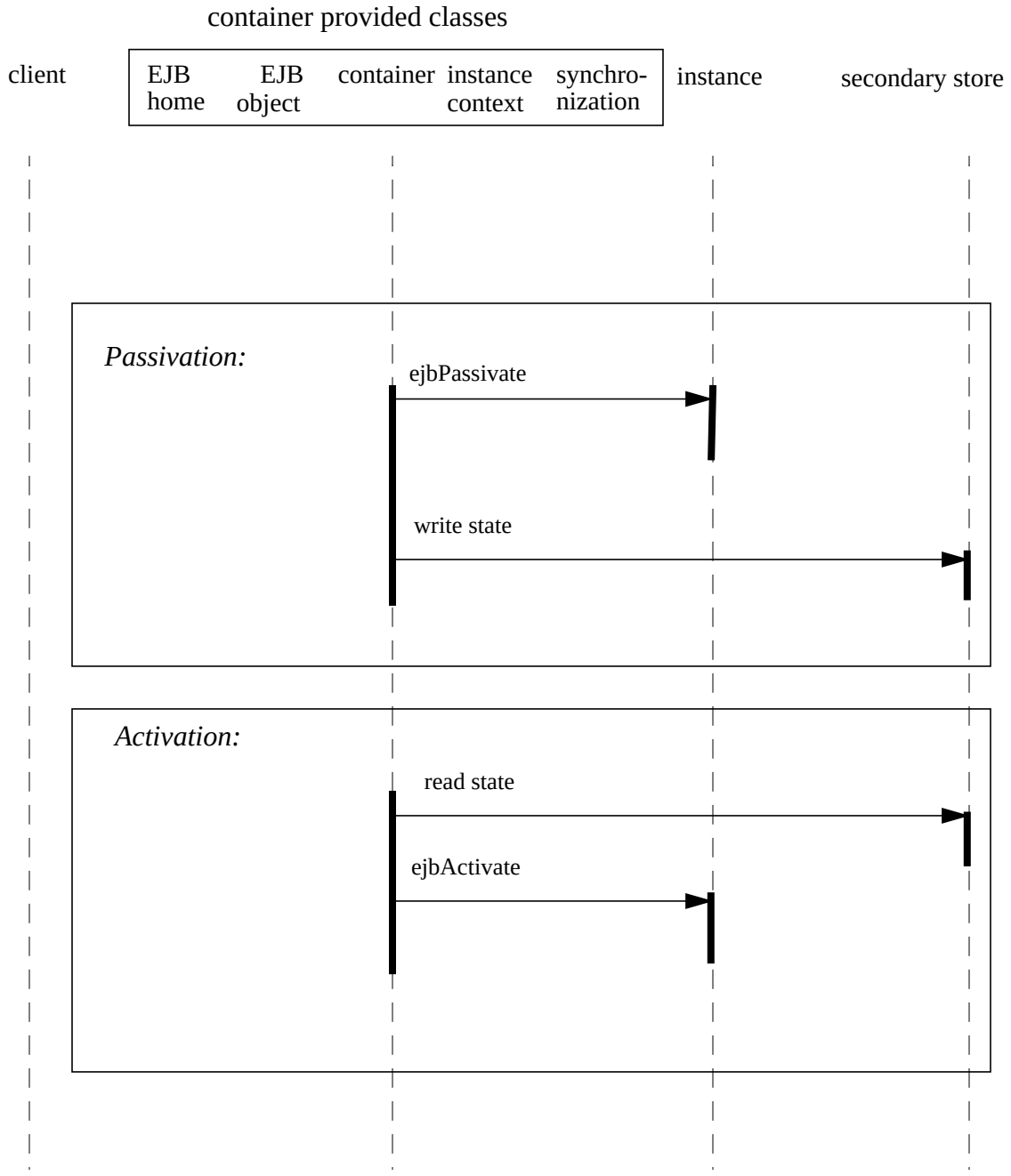
Figure 9 OID for Transaction Synchronization Protocol for a Session EJB.



6.7.5 Passivating and activating an instance between transactions

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

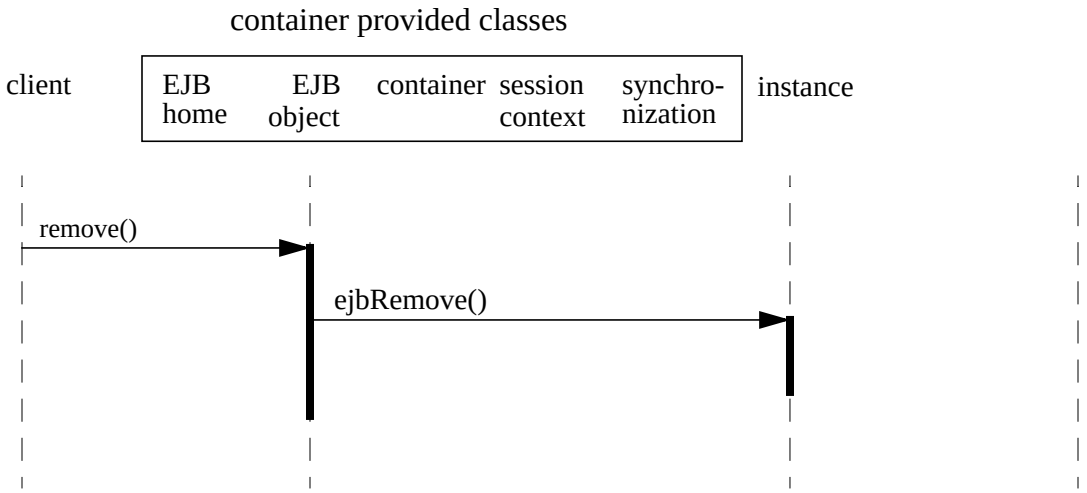
Figure 10 OID for Passivation and Activate of Session EJBs.



6.7.6 Removing a session object

The following diagram illustrates the destruction of a session Bean

Figure 11 OID for the Destruction of a Session EJB.



6.8 Stateless session Beans

Stateless session Beans are session Beans with no conversational state. This means that all Bean instances are equivalent when they are not involved in serving 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 a `create` method that takes no arguments, and returns the session Bean's remote interface. The home interface must not have any other `create` methods. The session enterprise Bean class must define a single `ejbCreate` method that takes no arguments.

Since all instances of a stateless session Bean are equivalent, the container can choose to delegate a client's work to any available 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. If another stateless session Bean instance is needed to handle an increase in client work load, the container creates one. If a stateless session Bean is not needed to handle the current client work load, the container can destroy it.

Since stateless session Beans minimize the resources needed to support a large population of clients, depending the implementation of the container, applications that use this approach may scale somewhat better than those using stateful session Beans. 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 that it appears as if the client is controlling the life cycle of an EJB instance. However, the container is handling 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.

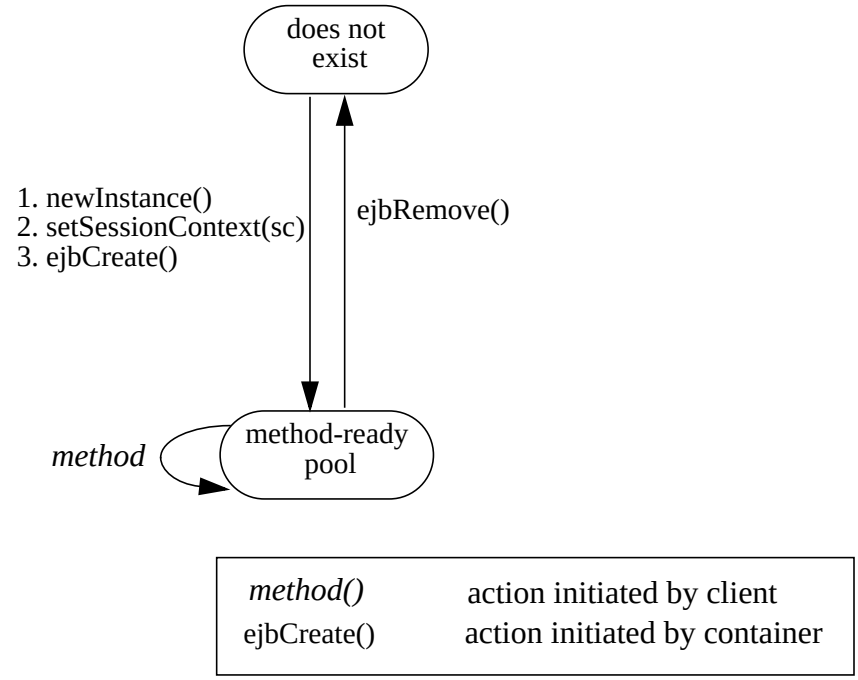
A stateless session must not implement the `javax.ejb.SessionSynchronization` interface.

6.8.1 Stateless session Bean state diagram

When a client calls a method on its stateless session Bean reference, 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 12 Lifecycle of a STATELESS Session Bean.



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

- A stateless session Bean's life starts when the container invokes `newInstance()` on the Bean class to create a new memory object for the enterprise Bean. Next, the container calls `setSessionContext()` followed by `ejbCreate()` on the instance. The container can perform the instance creation at anytime, with no relationship to a client's invoking the `create()` method.
- The Bean instance is now ready to be delegated a business method call from 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 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 with container-managed transactions

Bean method	**Bean method can perform the following operations**	
	Container-managed transactions	**Bean-managed transactions**
constructor	-	-
setSessionContext	SessionContext methods: *getEJBHome* JNDI access to java:comp/env	SessionContext methods: *getEJBHome, getUserTransaction* JNDI access to java:comp/env
ejbCreate ejbRemove	SessionContext methods: *getEJBHome, getRollbackOnly, setRollbackOnly, getEJBObject* JNDI access to java:comp/env	SessionContext methods: *getEJBHome, 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

Additional restrictions:

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

The reasons for disallowing operations in Table 3:

- Invoking the `getEJBObject` methods is disallowed in the session bean methods in which there is no EJB 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 to all session beans with bean-managed transactions.

- 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 transactions.

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 12.

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

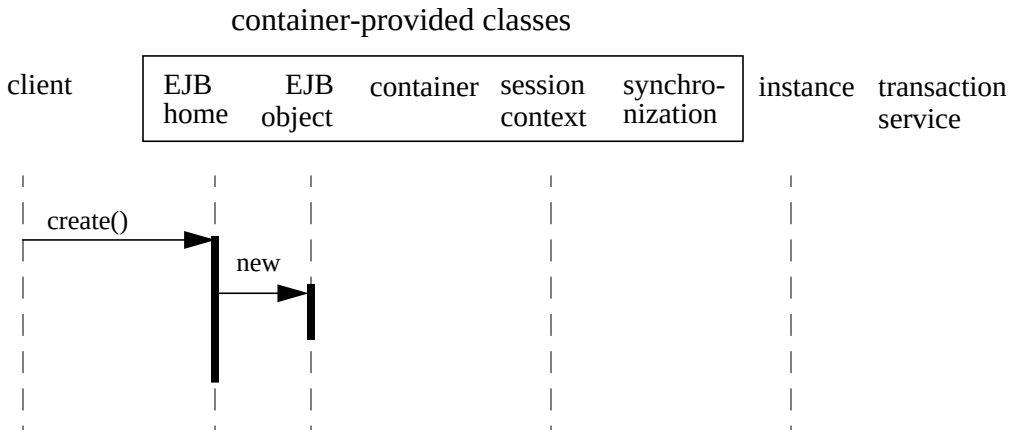
6.9 Object interaction diagrams for a STATELESS session Bean

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 an EJB object that is implemented by a stateless session Bean.

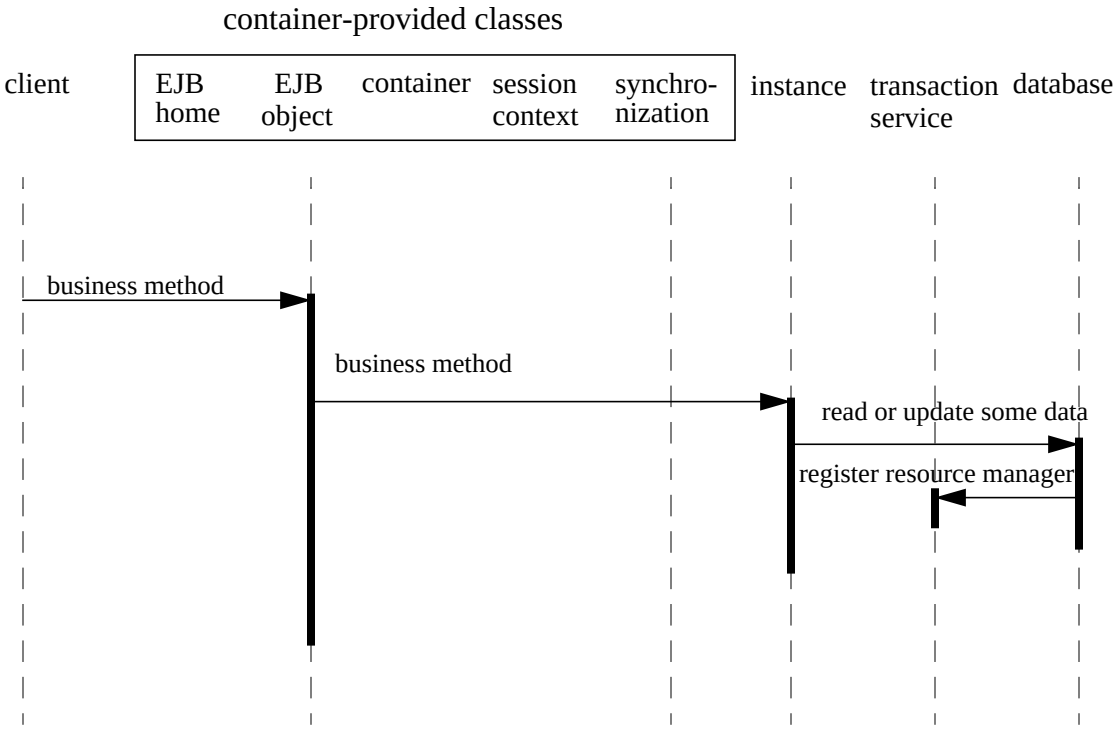
Figure 13 OID for creation of a STATELESS Session Bean



6.9.2 Business method invocation

The following diagram illustrates the invocation of a business method.

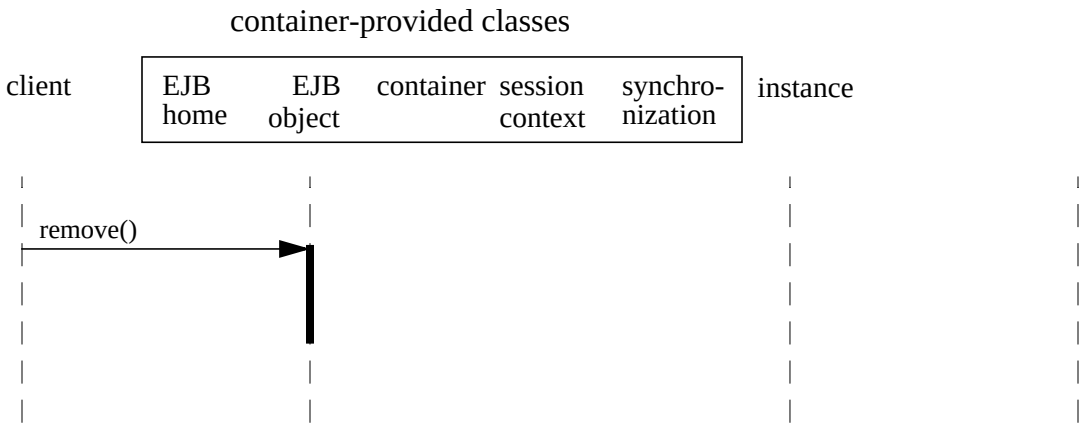
Figure 14 OID for invocation of business method on STATELESS Session Bean



6.9.3 Client-invoked *remove()*

The following diagram illustrates the destruction of an EJB object that is implemented by a stateless session Bean.

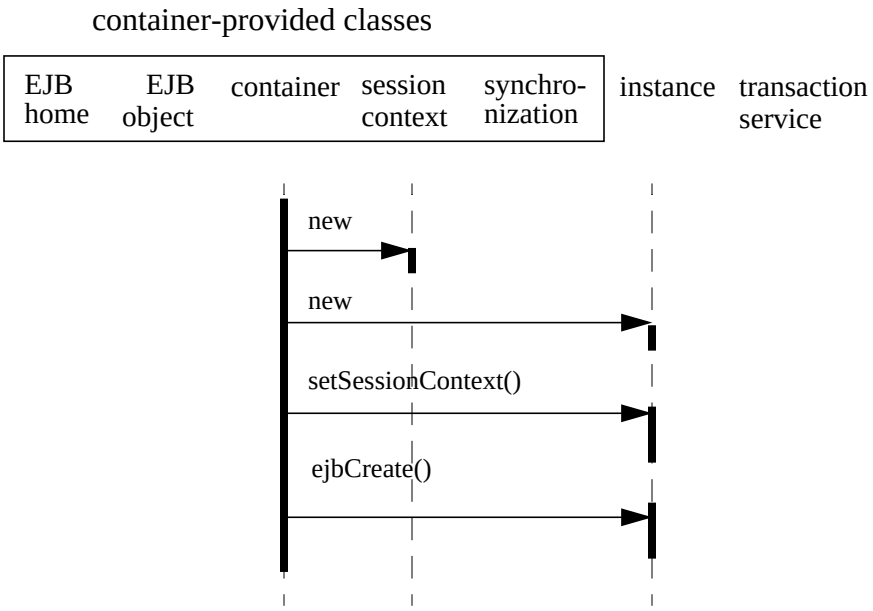
Figure 15 OID for removal of a STATELESS Session Bean



6.9.4 Adding instance to the pool

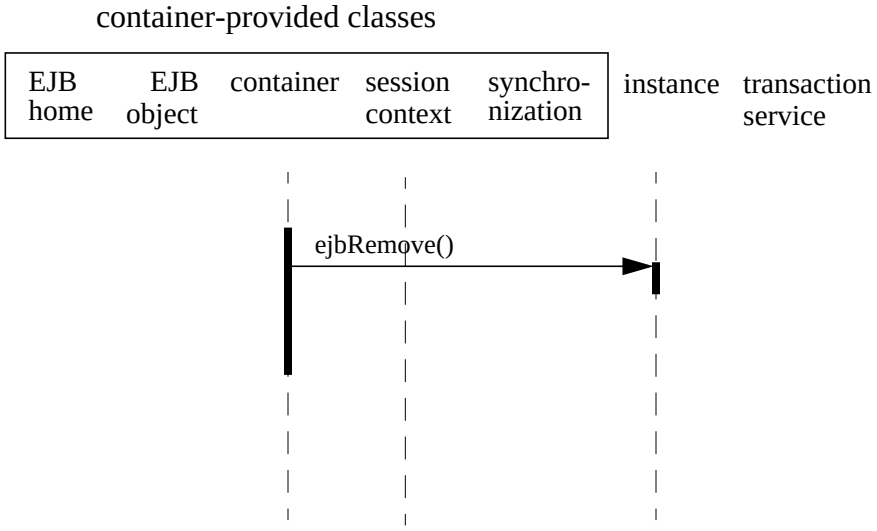
The following diagram illustrates the sequence for a container adding an instance to the method-ready pool.

Figure 16 OID for Container Adding Instance to a Method-Ready Pool of STATELESS Session Beans



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

Figure 17 OID for a Container Removing an Instance of STATELESS Session Bean from Ready Pool



6.10 The responsibilities of the enterprise Bean provider

This section describes the responsibilities of session enterprise Bean provider to ensure that an enterprise Bean can be deployed in any EJB Container.

6.10.1 Classes and interfaces

The enterprise Bean provider is responsible for providing the following class files:

- Enterprise Bean class.
- Enterprise Bean's remote interface.
- Enterprise Bean's home interface.

6.10.2 Enterprise Bean class

The following are the requirements for session enterprise 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 class must not define the `finalize()` method.

The class may, but is not required to, implement the enterprise Bean's remote interface[3].

The class must implement the business methods and the `ejbCreate` methods.

The class can optionally implement the `javax.ejb.SessionSynchronization` interface.

The enterprise bean class may have superclasses and/or superinterfaces. If the enterprise bean has superclasses, the business methods, the `ejbCreate` methods, the methods of the `SessionBean` interface, and the methods of the optional `SessionSynchronization` interface may be defined in the enterprise bean class, or any of its superclasses.

6.10.3 *ejbCreate* methods

The enterprise Bean class must define one or more `ejbCreate(...)` methods whose signatures must follow these rules:

[3] It is recommended that the enterprise bean class not implement the remote interface to prevent inadvertent passing of *this* as a method argument or result.

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 methods 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 is deprecated in EJB 1.1—an EJB 1.1 compliant enterprise bean should throw the javax.ejb.EJBException or another RuntimeException to indicate non-application exceptions to the Container (see Section 12.2.2).

6.10.4 Business methods

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

The function names can be arbitrary, but they must not conflict with the names of the methods defined by the EJB architecture (`ejbCreate`, `ejbActivate`, etc.).

The business method must be declared as `public`.

The method must not be declared as `final` or `static`.

The methods arguments and return value types 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 is deprecated in EJB 1.1—an EJB 1.1 compliant enterprise bean should throw the javax.ejb.EJBException or another RuntimeException to indicate non-application exceptions to the Container (see Section 12.2.2).

6.10.5 Enterprise Bean's remote interface

The following are the requirements for the enterprise 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 arguments and return values must be of valid types for RMI-IIOP, and their throws clause 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 enterprise 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 enterprise Bean class must be defined in the throws clause of the method of the remote interface.

6.10.6 Enterprise Bean's home interface

The following are the requirements for the enterprise Bean's home interface signature:

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 arguments and return values must be of valid types for RMI-IIOP, and that their throws clause 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 defines one or more `create(...)` methods.

Each `create` method must be named "**create**", and it must match one of the `ejbCreate` methods defined in the enterprise Bean class. The matching `ejbCreate` method must have the same number and types of arguments. (Note that the return type is different.)

The return type for a `create` method must be the enterprise Bean's remote interface type.

All the exceptions defined in the throws clause of an `ejbCreate` method of the enterprise Bean class must be defined in the throws clause of the matching `create` method of the remote 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 objects 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 enterprise 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 Bean's deployment descriptor.

The deployment tools must generate the following classes:

- A class that implements the enterprise Bean's home interface (EJB Home class).
- A class that implements the enterprise Bean's remote interface (EJB Object class).

The deployment tools may also generate a class that mixes some container-specific code with the enterprise 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.

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 EJB Home class

The EJB home class, which is generated by the deployment tools, implement the enterprise Bean's home interface. This class implements the methods of the `javax.ejb.EJBHome` interface, and the `create` methods specific to the enterprise Bean.

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

The implementation of the `remove(...)` methods defined in the `javax.ejb.EJBHome` interface must activate the instance (if the instance is in the passive state) and invoke the `ejbRemove` method on the instance.

6.11.3 EJB Object class

The EJB Object class, which is generated by the deployment tools, implements the enterprise Bean's remote interface. It implements the methods of the `javax.ejb.EJBObject` interface and the business methods specific to the enterprise Bean.

The implementation of the `remove()` method (defined in the `javax.ejb.EJBObject` interface) must activate the instance (if the instance is in the passive state) and invoke the `ejbRemove` method on the instance.

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 class

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

6.11.5 Meta-data 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.EJB-MetaData` 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 enterprise Bean is intended to support only a single client. Therefore, it would be an application error if two clients attempted to invoke the same session Bean.*

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 11, 15, and 12.

Chapter 7 Example Session Scenario

This chapter describes an example development and deployment scenario of a session enterprise Bean. We use the scenario to explain the responsibilities of the enterprise 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 enterprise Bean and its container in a different way, provided that it achieves an equivalent effect (from the perspectives of the enterprise Bean provider and the client-side programmer).

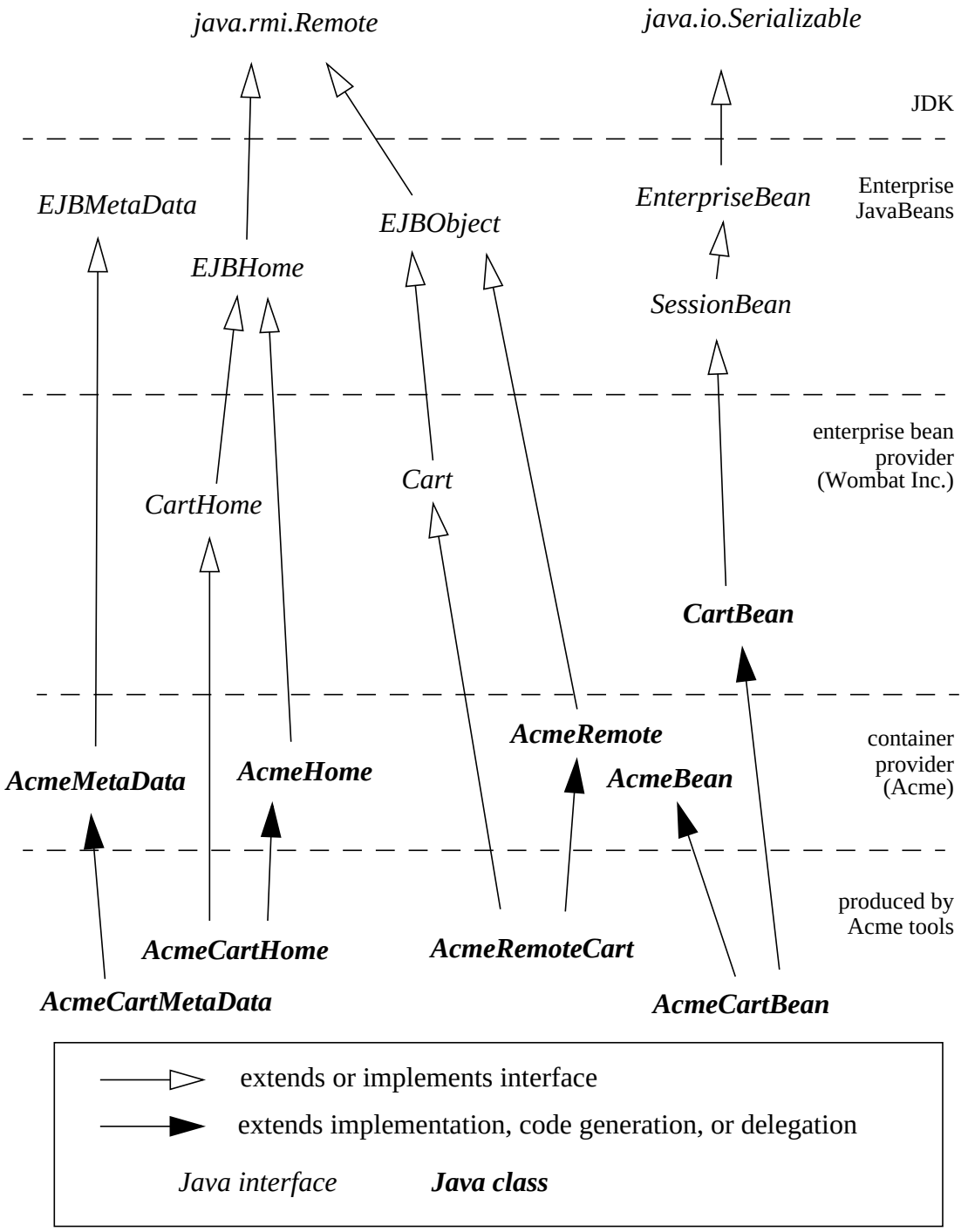
7.1 Overview

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 18 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(...) methods invoked at an 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 that specifies 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 remote Bean class (AcmeRemoteCart) for the session Bean. The remote Bean class is a "wrapper" class for the enterprise Bean and provides the client view of the enterprise Bean. The tools also generate the classes that implement the communication stub and skeleton for the remote Bean class.*

- *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 implementation class for the session Bean's home interface (AcmeCartHome). The tools also generate the classes that implement the communication stub and skeleton for the home class.*

- *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, which 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 EJB object. It is actually a contract fulfilled by an enterprise Bean's container in which the enterprise Bean is installed. Only the business methods are supplied by the enterprise Bean itself.

Although the client view of the enterprise Beans is provided by classes implemented by the container, the container is transparent to the client.

8.1 Overview

For a client, an entity enterprise Bean is an object that represents an object view of an entity stored in a persistent storage, such as a database, or an entity that is implemented by an existing enterprise application.

A client accesses an entity enterprise Bean through the Entity Bean's remote interface. The object that implements the remote interface is called an **EJB object.** An EJB object is a remote Java programming language object that is accessible from a client through the standard Java™ APIs for remote object invocation [3].

From its creation until its destruction, an EJB object lives in a container. Transparently to the client, the container provides security, concurrency, transactions, persistence, and other services for the EJB 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 installed properly synchronizes access to the entity 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.

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

A client of an enterprise bean can be another enterprise bean deployed in the same or different Container; or 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 not written in the Java programming language.

Multiple EJB classes can be installed in a container. For each EJB class installed in a container, the container implements the enterprise Bean's **home interface.** The home interface allows the client to create, look up, and remove entity EJB objects of a given enterprise Bean. A client can look up the enterprise Bean's home interface through JNDI; it is the responsibility of the container to make the enterprise Bean's home interface available in the JNDI name space.

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

8.2 EJB Container

An EJB Container (Container for short) is a system that functions as a "container" for enterprise Beans. A container is where an enterprise Bean object lives, just as a record lives in a database, and a file or directory lives in a file system.

Multiple EJB classes can be installed in a single container. For each EJB class installed in a container, the container provides a **home interface** that allows the client to create, look up, and remove EJB objects of the corresponding EJB class. The container makes the enterprise 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 enterprise Bean's home interface

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

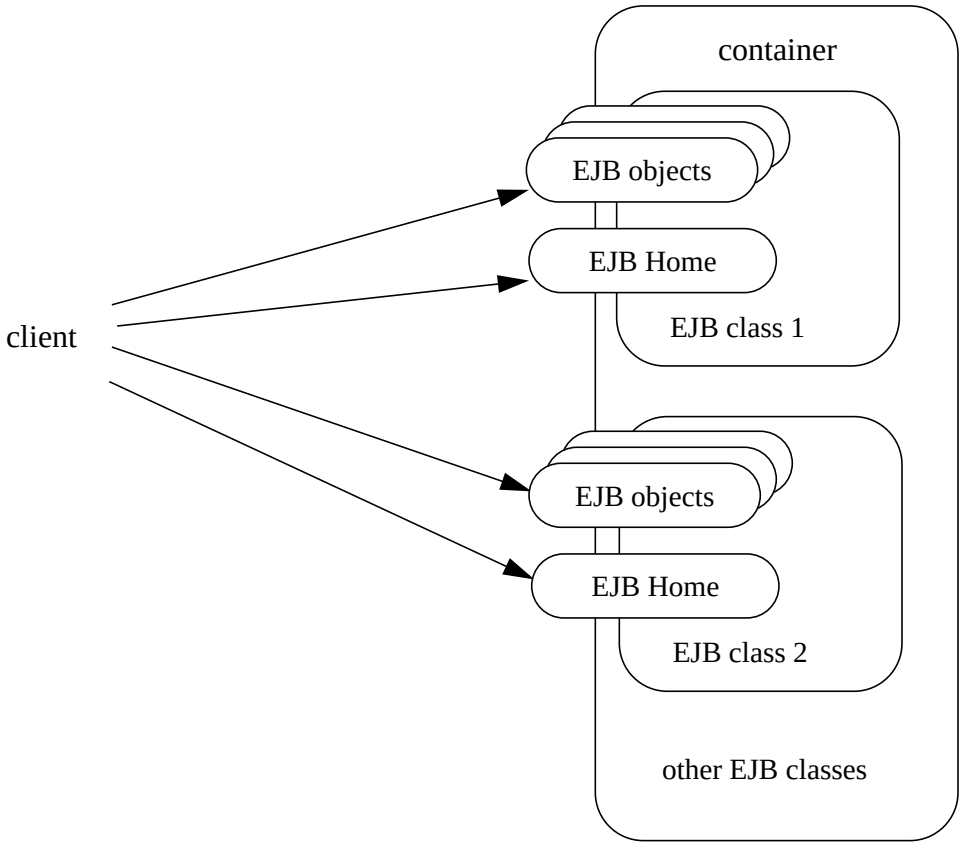
```
Context initialContext = new InitialContext();
AccountHome accountHome = (AccountHome)
         javax.rmi.PortableRemoteObject.narrow(
              initialContext.lookup("applications/bank/accounts"),
                                   AccountHome.class);
```

A client's JNDI name space may be configured to include the home interfaces of EJB classes installed 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 an entity container provides to its clients.

Figure 19 Client View of Entity Enterprise JavaBeans Architecture



8.3 Enterprise Bean’s home interface

The container provides the implementation of the home interface of each enterprise Bean installed in the container. The container makes the home interface of every enterprise Bean installed in the container accessible to the clients through JNDI. The implementation class of an enterprise Bean’s home interface is called **EJB home.**

The home interface of an Entity Bean allows a client to do the following:

- Create new EJB objects.
- Look up existing EJB objects.
- Remove an EJB object.
- Get the javax.ejb.EJBMetaData interface for the enterprise Bean. The javax.ejb.EJBMetaData interface is intended to allow application assembly tools to discover information about the enterprise Bean. The meta-data is defined to allow loose client/server binding and scripting.
- Obtain a handle for the home object. 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 object.

An enterprise 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(...)` methods, one for each way to create an EJB object. The arguments of the `create` methods are typically used to initialize the state of the created EJB object.

The return type of a `create` method is the enterprise Bean's remote interface.

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

The following home interface illustrates two 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;
      ...
}
```

The following example illustrates how a client creates a new EJB 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[4], one for each way to look up an EJB object, or collection of EJB objects of a particular type. The name of each finder method must start 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 enterprise Bean's remote interface, or a type representing a collection of EJB objects (see Subsection 9.1.8).

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

The home interface of every Entity Bean includes the `findByPrimaryKey(primaryKey)` method that allows a client to locate an Entity Bean using a primary key. The name of the method is always `findByPrimaryKey`; it has a single argument that is of the enterprise Bean's primary key type, and its return type is the enterprise Bean's remote interface. The implementation of the `findByPrimaryKey(primaryKey)` method must ensure that the entity exists in the underlying database.

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 EJB objects.

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

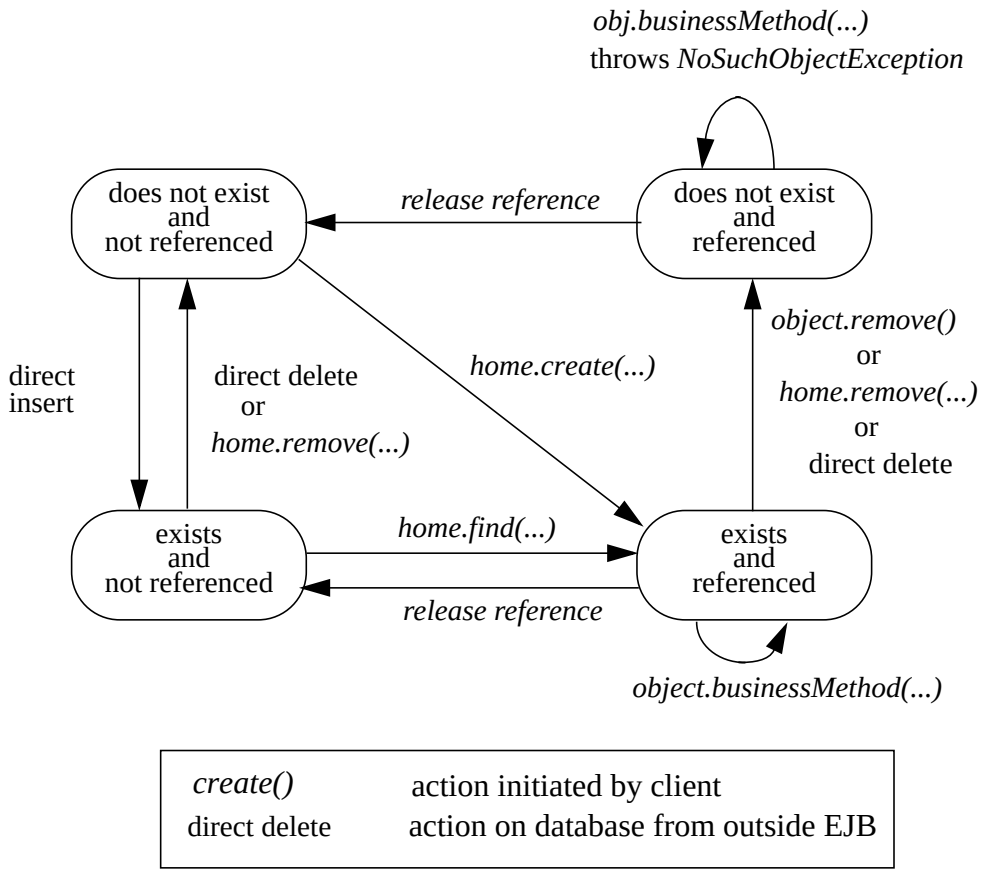
[4] The *findByPrimaryKey(primaryKey)* method is mandatory for all Entity Beans.

8.4 Entity EJB object life cycle

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

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

Figure 20 Client View of EJB Entity Object Life Cycle



An EJB 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 EJB object using the enterprise Bean's home interface that is implemented by the container. When an EJB object is created by a client, the client obtains a reference to the newly created EJB object.

In an environment with legacy data, EJB objects may “exist” before the container and EJB object are deployed. In addition, an entity EJB object may be “created” in the environment via a mechanism other than by invoking a `create(...)` 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 EJB 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 EJB object in any of the following ways:

- Receive a reference as a parameter in a method call (input parameter or result).
- Look up the EJB object using a finder method of the enterprise Bean’s home interface.
- Obtain the reference from a Bean’s handle. (see Section 8.7)

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

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

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

All entity EJB objects are considered **persistent objects**. The lifetime of an entity EJB object is not limited by the lifetime of the Java Virtual Machine process in which it executes. A crash of the Java Virtual Machine may result in a rollback of current transactions, but does not destroy previously created EJB entity objects, or invalidate their references held by clients.

Multiple clients can access the same EJB object concurrently. Transactions are used to isolate the clients’s work from each other.

8.5 Primary key and object identity

Every entity EJB object has a unique identity within its home. The object’s identity within its container is determined by the EJB object’s home and primary key. If two EJB 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. The primary key class is specific to an enterprise Bean class (i.e. each enterprise Bean class may have a different class for its primary key).

A client that holds a reference to an EJB object can determine the 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()` will always return the same value when called on the same entity reference.)

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

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

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

if (acct1.isIdentical(acc2)) {
     acc1 and acc2 are the same EJB objects
} else {
     acc2 and acc2 are different EJB object
}
```

A client that knows the primary key of an entity EJB object can obtain a reference to the object by invoking the `findByPrimaryKey(key)` method of the home interface implemented by the container.

Note that the Enterprise JavaBeans architecture does not specify "object equality" (i.e use of the `==` operator) for EJB 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 same object is not guaranteed to yield the same result. Therefore, a client should always use the `isIdentical` method to determine if two EJB object references refer to the same EJB object.

8.6 Entity Bean's remote interface

A client accesses an Entity Bean through the enterprise Bean's remote interface. An enterprise 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 methods that allow the client to perform the following operations on an EJB object's reference:

- Obtain the home interface for the EJB class.
- Remove the EJB object.
- Obtain the EJB object's handle.
- Obtain the EJB object's primary key.

The implementation of the methods defined in the `javax.ejb.EJBObject` interface is provided by the container. The business methods are delegated to the enterprise Bean class.

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

8.7 Entity Bean's handle

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

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

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

```
// A client obtains a handle of an account EJB 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 ressurect an object reference to the
// account EJB object.
//
ObjectInputStream stream = ...;
Handle handle = (Handle) stream.readObject(handle);
Account account = (Account)javax.rmi.PortableRemoteObject.narrow(
         handle.getEJBObject(), Account.class);
account.debit(100.00);
```

8.8 Entity Home handles

The EJB specification allows the client to obtain a handle for the home object. The client can use the home handle to store a reference to a 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 is defined by 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

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

Note: Container support for entity enterprise Beans is a mandatory feature starting in the EJB 1.1 release.

The Entity Bean component contract is the contract between an Entity Bean and its container. It defines the life cycle of an Entity Bean instance and the model for method delegation of the client-invoked business methods. The main goal of this contract is to ensure that a component is portable across all compliant EJB Containers.

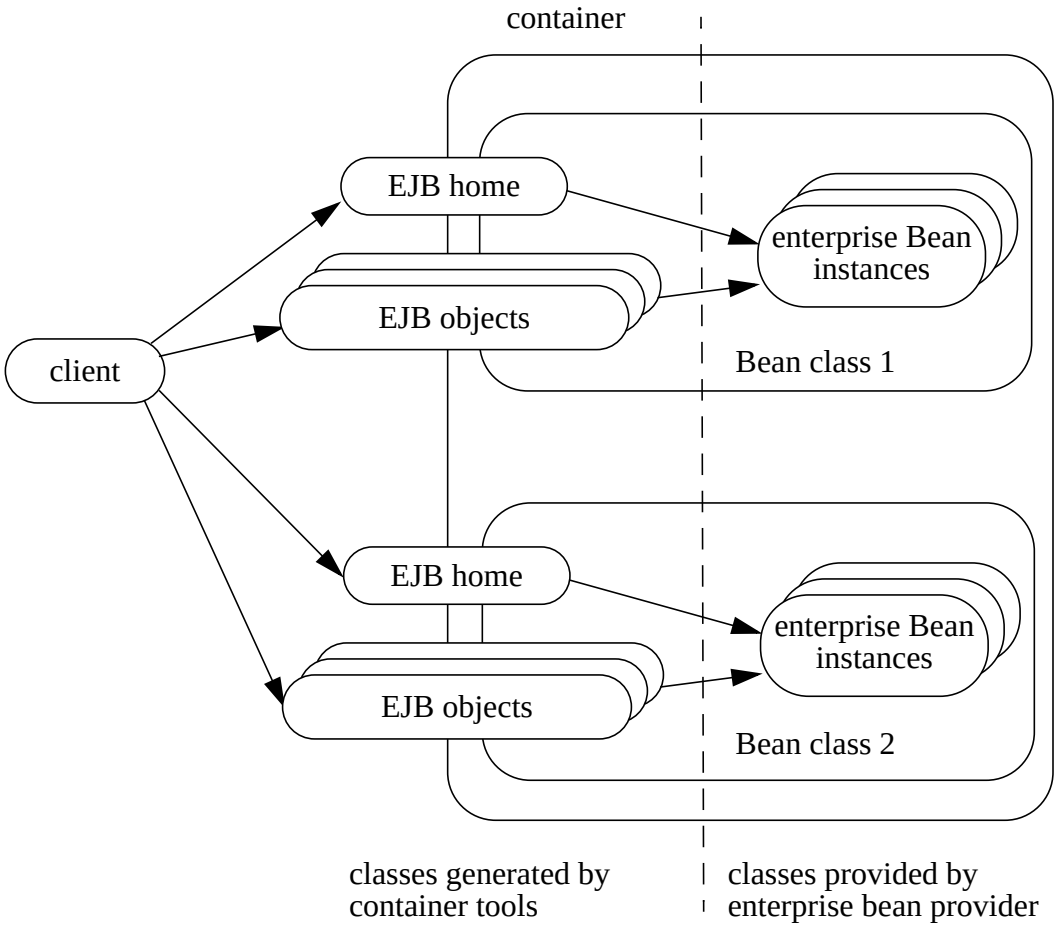
This chapter defines the enterprise Bean developer's view of this contract and the container's responsibility for managing the component's life cycle.

9.1 Concepts

9.1.1 The runtime execution model

This section describes the runtime model and the classes used in the description of the contract between an entity enterprise Bean and its container.

Figure 21 Overview of the Entity EJB Runtime Execution Model



An **enterprise Bean instance** is an object whose class was provided by the enterprise Bean developer.

An **EJB object** is an object whose class was generated at deployment time by the container provider's tools. The EJB object class implements the enterprise Bean's remote interface. A client never references an enterprise Bean instance directly—a client always references an EJB object whose implementation is provided by the container.

An **EJB home** object provides the life cycle operations (create, remove, find) for its EJB objects. The class for the EJB home object was generated by the container provider's tools at deployment time. The home object implements the enterprise Bean's home interface that was defined by the Bean Provider.

9.1.2 Granularity of entity objects

This section provides guidelines to the Bean Providers for modeling of business objects as EJB Entities.

In general, an EJB Entity 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 modeled as an EJB Entity. 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 that has dependent objects is often stored in multiple database records and spans multiple tables.

In addition, the Bean Provider must take into consideration the following factors 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 enterprise 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 most fine-grained object interactions.

- The EJB deployment descriptor does not provide a mechanism for describing object schemas (the relationships among the fine-grained objects, and how fine-grained objects are mapped to the underlying database).

9.1.3 Entity persistence (data access protocol)

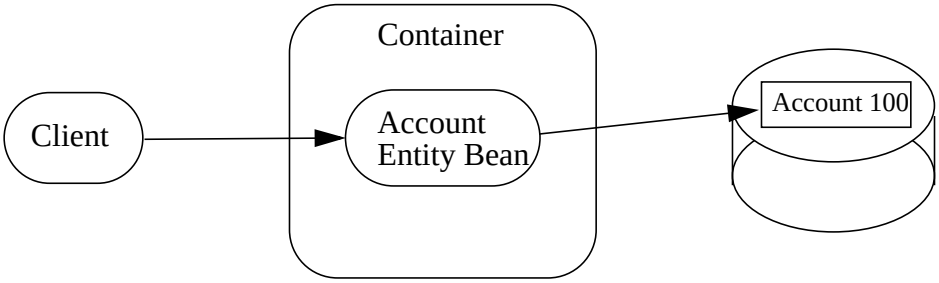
An entity enterprise Bean implements an object view of an entity stored in an underlying database, or an entity implemented by an existing enterprise application (for example, by a mainframe program or by a packaged application). The data access protocol for transferring the state of the entity between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

The entity component protocol allows the enterprise Bean provider either to implement the enterprise Bean's persistence directly in the enterprise Bean class or in one or more helper objects provided with the enterprise bean class (Bean-managed persistence), or to delegate the enterprise Bean's persistence to the container (container-managed persistence).

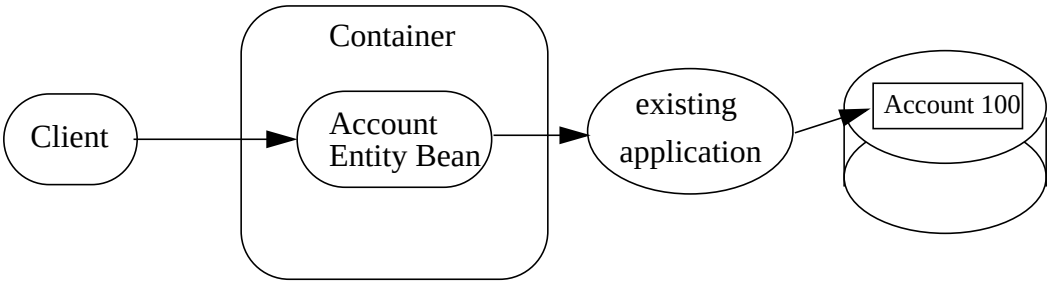
In many cases, the underlying data source may be an existing application rather than a database.

Figure 22 Client View of Underlying Data Sources Accessed Through Entity EJBs

(a) Entity bean is an object view of a record in the database



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



9.1.3.1 Bean-managed persistence

In the Bean-managed case, the enterprise 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(...)`, `ejbRemove()`, `ejbFind<METHOD>()`, `ejbLoad()`, and `ejbStore()` methods; and/or in the business methods.

The data access calls can be coded directly into the enterprise bean class, or can be encapsulated in a data access component that is part of the Entity bean.

We expect that most enterprise beans 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. This EJB specification does not define the architecture for data access objects or strategies.

If the data access calls are coded directly in the enterprise bean class, it may more difficult to adapt the Entity component to work with a database that has a different schema, or with a different type of database.

If the data access calls are encapsulated in data access components, the data access components may optionally provide deployment interfaces to allow adapting data access to different schemas, or even to a different database type. These data access component strategies are beyond the scope of the EJB specification.

9.1.3.2 Container-managed persistence

In the container-managed case, the Bean developer does not write the database access calls in the enterprise Bean. Instead, the container provider's tools generate the database access calls at the enterprise Bean's deployment time (i.e. when the enterprise Bean class is installed into a container). The enterprise Bean provider must specify in the deployment descriptor the list of instance fields for which the container provider tools must generate access calls.

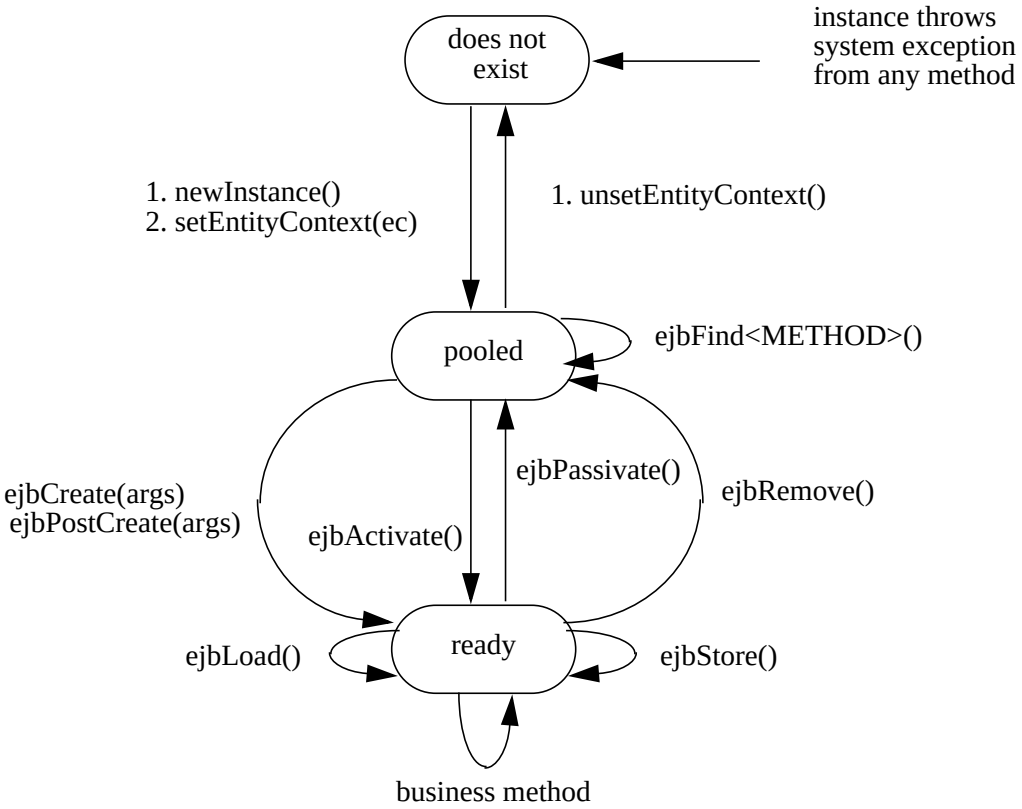
The advantage of using container-managed persistence is that the enterprise Bean class can be largely independent from the data source in which the entity is stored. The container tools can generate classes that use JDBC or SQLJ to access the entity state in a relational database, or classes that implement access to a non-relational data source, such as an IMS database, or classes that implement function calls to existing enterprise applications.

The disadvantage is that sophisticated tools must be used at deployment time to map the enterprise Bean's fields to a data source. These tools and containers are typically specific to each data source.

The essential difference between an entity with bean-managed and container-managed persistence is that in the bean-managed case, the data access components are provided as part of the Entity bean, whereas in the container-managed case, the data access components are generated at deployment time by the container tools.

9.1.4 Instance life cycle

Figure 23 Life cycle of an Enterprise Bean's instance.



An instance is in one of the three states:

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

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

- An enterprise 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 an entity context interface. The entity context object 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 of the enterprise Bean class. While the instance is in the available pool, the instance is not associated with an identity of a specific EJB object. All instances in the pool are equivalent, and therefore can be assigned by the container to any EJB object 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 enterprise Bean's finder methods (shown as ejbFind<METHOD>(...) in the diagram). The instance does **not** move to the ready state during the execution of a finder method.

- An instance transitions from the pooled state to the ready state when the container picks that instance to service a client call on an EJB object. There are two possible transitions from the pooled to the ready state: through the ejbCreate(...) and ejbPostCreate(...) methods, or through the ejbActivate() method. The container invokes the ejbCreate(...) and ejbPostCreate(...) methods when the instance is assigned to an EJB object during EJB object creation (i.e. when the client invokes a create method on the 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 EJB object because there is no suitable instance in the ready state.

- When an enterprise Bean instance is in the ready state, the instance is associated with a specific EJB object. 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 at anytime it determines that there is a need to synchronize the instance's state.

- Eventually, the container will transition the instance to the pooled state. There are two possible transitions from the ready to the pooled state: through the ejbPassivate() method, and through the ejbRemove() method. The container invokes the ejbPassivate() method when the container wants to disassociate the instance from the EJB object without removing the EJB object. The container invokes the ejbRemove() method when the container is removing the EJB object (i.e. when the client invoked the remove() method on the EJB object, or one of the remove() methods on the enterprise Bean's home interface).

- When the instance is put back into the pool, it is no longer associated with the identity of the EJB object. The container can assign the instance to any EJB object of the same enterprise Bean class.

- An instance in the pool can be removed by calling the unsetEntityContext() method on the instance.

Notes:

1. The entity context 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 `getCaller-`

`Principal()` and `isCallerInRole(...)` methods may be different in each business method.

2. 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. The Container must not invoke any method on the instance after a `RuntimeException` has been caught. From the client perspective, the corresponding Entity EJB Object continues to exist. The client can continue accessing the Entity EJB Object because the Container can use a different instance to delegate the client's requests. Exception handling is described further in Chapter 12.

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.

9.1.5 The Entity Bean component contract

This section specifies the contract between an Entity Bean and its container. The contract specified here assumes the use of Bean-managed persistence. The differences in the contract for container-managed persistence are defined in Section 9.4.

9.1.5.1 Enterprise Bean instance's view:

The following describes the enterprise Bean instance's side of the contract:

An enterprise Bean is responsible for implementing the following functionality in the enterprise Bean methods:

- 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 enterprise Bean instance. If the enterprise Bean instance needs to use the entity context during its lifetime, it must remember the entity context in an instance variable.

  It is unspecified in which transaction context this method is called. An identity of an EJB object is not available during this method.

  The instance can take advantage of the `setEntityContext(ic)` method to allocate any resources that are to be held by the instance for its lifetime. Such resources cannot be specific to an EJB object identity since the instance might be reused during its lifetime to serve multiple EJB objects.
- `public void unsetEntityContext();`

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

  It is unspecified in which transaction context this method is called. An identity of an EJB 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(...);`

  There are zero[5] or more `ejbCreate(...)` methods, whose signatures match the signatures of the `create(...)` methods of the enterprise Bean's home interface. The container invokes an `ejbCreate(...)` method on an enterprise Bean instance when a client invokes a matching `create(...)` function.

  The implementation of the `ejbCreate(...)` method typically validates the client-supplied arguments, and inserts a record representing the entity into the database. The method also initializes the instance's variables. The `ejbCreate(...)` method must return the primary key for the created entity.

  An `ejbCreate(...)` method executes in the transaction context determined by the transaction attribute of the matching `create(...)` method, as described in subsection 11.6.2. Depending on the value of the transaction attribute, the transaction context can be the client's transaction context, a new transaction context, or non-existent.

- `public void ejbPostCreate(...);`

  For each `ejbCreate(...)` method, there is a matching `ejbPostCreate(...)` method that has the same input parameters but the return value is void. The container invokes the matching `ejbPostCreate(...)` method after it invokes the `ejbCreate(...)` method, with the same arguments. The EJB object identity is available during the `ejbPostCreate(...)` method. The instance may, for example, pass its own EJB object reference to another EJB object as a method argument.

  An `ejbPostCreate(...)` method executes in the same transaction context as the previous `ejbCreate(...)` 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 EJB object identity. The `ejbActivate()` method gives the enterprise Bean instance the chance to acquire additional resources that it needs while it is in the ready state.

  This method executes in an unspecified transaction context. The instance can obtain the identity of the EJB object via the `getPrimaryKey()` or `getEJBObject()` method on the entity context. The instance can rely on the fact that the primary key and EJB 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 EJB object identity, and to put the instance back into the pool of available

[5] An entity enterprise Bean has no *ejbCreate(...)* and *ejbPostCreate(...)* 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.

instances. The ejbPassivate() method gives the enterprise Bean 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 in an unspecified transaction context. The instance can still obtain the identity of the EJB object via the `getPrimaryKey()` or `getEJBObject()` method on the entity context.

Note that instance should not use the `ejbPassivate()` method to write its state to the database; the 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 EJB object via the `getPrimaryKey()` or `getEJBObject()` method on the entity context.

  An enterprise Bean instance should use this method to remove its entity 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. Depending on the value of the transaction attribute, the transaction context can be the client's transaction context, a new transaction context, or non-existent. The transaction attributes are described in subsection 11.6.2.

- `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 invoked on the instance. All business methods between the previous `ejbLoad`

or `ejbCreate` method and this `ejbStore` methods are also invoked in this 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 EJB object identity) when the container selects the instance to execute the `ejbFind<METHOD>` method on it, and 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 11.6.2.

  The implementation of an `ejbFind<METHOD>` method typically uses the method's arguments to locate the requested object or a collection of objects in the database. The method must return a primary key or a collection of primary keys to the container (see Subsection 9.1.8).

9.1.5.2 Container's view:

This subsection describes the container's side 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 enterprise Bean's entity context to the enterprise Bean. The container must invoke this method after it creates the instance, and before it puts the instance into the pool of available instances.

  It does not matter whether the container calls this method inside or outside of a transaction context. At this point, the entity context is not associated with any EJB object.

- `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.

  It does not matter whether the container calls this method inside or outside of a transaction context.

- `public` *`PrimaryKeyClass`* `ejbCreate(...);`
  `public void ejbPostCreate(...);`

  The container invokes these two methods during the creation of an EJB entity object as a result of a client's invoking a `create(...)` method on the enterprise Bean's EJB home.

  The container first invokes the `ejbCreate(...)` method whose signature matches the `create(...)` method invoked by the client. The `ejbCreate(...)` method returns a primary key for the created entity. The container creates an EJB object reference for the primary key. The container then invokes a matching `ejbPostCreate(...)` method to allow the instance to fully initialize itself. Finally, the container returns the EJB object reference to the client.

  The container must invoke the `ejbCreate(...)` and `ejbPostCreate(...)` methods in the transaction context determined by the transaction attribute of the matching `create(...)` method, as described in subsection 11.6.2. Depending on the value of the transac-

tion attribute, the transaction context can be the client's transaction context, a new transaction context, or non-existent.

- `public void ejbActivate()`;

  The container invokes this method on an enterprise Bean instance at activation time (i.e., when the instance is taken from the pool and assigned to an EJB object). The container must ensure that the primary key of the associated EJB object is available to the instance if the instance invokes the `getPrimaryKey()` or `getEJBObject()` method on its entity context.

  A container may call this method inside or outside of a 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 enterprise Bean instance at passivation time (i.e., when the instance is being disassociated from an EJB object and moved into the pool). The container must ensure that the primary key of the associated EJB object is still available to the instance if the instance invokes the `getPrimaryKey()` or `getEJBObject()` method on its entity context.

  A container may call this method inside or outside of a 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 EJB object as a result of a client's invoking a remove operation.

  The container invokes this method in the transaction context determined by the transaction attribute of the invoked `remove` method. Depending on the value of the transaction attribute, the transaction context can be the client's transaction context, a new transaction context, or non-existent.
  The container must ensure that the primary key of the associated EJB 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).

- `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 from the database. After the first `ejbLoad` within a transaction, the container is not required to recognize that the state of the entity 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. Depending on the value

of the transaction attribute, the transaction context can be the client's transaction context, a new transaction context, or non-existent. Transaction attributes are described in subsection 11.6.2.

- `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 to another instance for the same entity in the same transaction (see Subsection 9.1.13).

  The container must invoke this method in the same transaction context as the previously invoked `ejbLoad` or `ejbCreate` 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 enterprise 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 EJB object) 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.

  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 11.6.2. Depending on the value of the transaction attribute, the transaction context can be the client's transaction context, a new transaction context, or non-existent.

  If the `ejbFind<METHOD>` method is declared to return a single primary key, the container creates an EJB object 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 EJB objects for the primary keys returned from `ejbFind<METHOD>`, and returns the collection to the client. (See Subsection 9.1.8 for information on collections.)

9.1.6 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*, *getEJBObject* 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	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*
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 global transaction. The Container must throw the `java.lang.IllegalStateException` if the methods are invoked while the instance is not associated with a global transaction.

Reasons for disallowing operations:

- Invoking the `getEJBObject` and `getPrimaryKey` methods is disallowed in the entity bean methods in which there is no EJB 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 is disallowed in the entity bean methods for which the Container does not have a meaningful transaction context or client security context.

9.1.7 Caching of entity state and the `ejbLoad` and `ejbStore` methods

An instance of an entity bean with bean-managed persistence can cache the entity state between business methods. An instance may choose to cache the entire entity state, parts 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 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 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 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 state:

- An instance loads the entire entity 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 state to the database.

- An instance loads the most frequently used part of the entity state in the `ejbLoad` method and caches it until the container invokes the `ejbStore` method. Additional parts of the entity state are loaded as needed by the business methods. The `ejbStore` method writes the updated parts of the entity state to the database.

- An instance does not cache any entity state between business methods. The business methods access and modify the entity state directly in the database. The `ejbLoad` and `ejbStore` methods have an empty implementation.

We expect that the most entity developers will not code the cache management and data access calls manually in the enterprise 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.

9.1.8 Finder method return type

9.1.8.1 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's home interface is the entity's remote interface. The result type of the corresponding `ejbFind<METHOD>(...)` implementation method defined in the entity's implementation class is the entity'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
    {
        ...
    }
    ...
}
```

9.1.8.2 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's home interface is a *collection* of objects implementing the entity's remote interface. The result type of the corresponding `ejbFind<METHOD>(...)` implementation method defined in the entity's implementation class is a collection of objects of the entity's primary key type.

The Bean Provider can choose two types to define a collection type for a finder:

- the JDK™ 1.1 `java.util.Enumeration` interface
- the Java™ 2 `java.util.Collection` interface

A Bean Provider that wants to ensure that the enterprise Bean is compatible with containers and clients based on JDK 1.1 s must use the `java.util.Enumeration` interface for the finder's result type[6].

A Bean Provider targeting only containers and clients based on Java 2 can use the `java.util.Collection` interface for the finder's result type.

The Bean Provider must ensure that the objects in the `java.util.Enumeration` or `java.util.Collection` returned from the `ejbFind<METHOD>(...)` method are compatible with the Entity's primary key class (i.e. their type is the primary key class or a subclass thereof).

The following is an example of a multi-object finder method definition that is 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
    {
        ...
    }
    ...
}
```

[6] The finder will be also compatible with Java 2-based Containers and Clients.

The following is an example of a multi-object finder method definition that is compatible with only 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
     {
          ...
     }
     ...
}
```

9.1.9 Standard application exceptions for Entities

The EJB specification defines the following standard EJB application exceptions:

- `javax.ejb.CreateException`
- `javax.ejb.DuplicateKeyException`
- `javax.ejb.FinderException`
- `javax.ejb.ObjectNotFoundException`
- `javax.ejb.RemoveException`

This section describes their use by entities with bean-managed persistence. The use of the exceptions by entity beans with container-manager persistence is the same, with one additional element: The responsibilities for throwing the exceptions apply to the data access methods generated by the Container Provider's tools.

9.1.9.1 CreateException

From the client's perspective, a `CreateException` (or a subclass of `CreateException`) indicates that an application level error occurred during the `create(...)` operation. If a client receives this exception, the client does not know, in general, whether the entity was created but not fully initialized, or not created at all. Also, the client also 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(...)` and `ejbPostCreate(...)` methods to indicate an application-level error from the entity 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(...) methods are invalid.

The Container treats the `CreateException` as any other application exception. See Section 12.3.

9.1.9.2 DuplicateKeyException

The `DuplicateKeyException` is a subclass of `CreateException`. It is thrown by the `ejbCreate(...)` methods to indicate to the client that the entity cannot be created because an entity 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.

9.1.9.3 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 12.3.

9.1.9.4 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 9.1.8) may throw this exception. Multi-object finders must not throw this exception.

9.1.9.5 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 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 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 RemoteException 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 12.3.

9.1.10 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 state.

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 5 Summary of commit-time options

	Write instance state to database	**Instance stays ready**	**Instance state remains valid**
Option A	Yes	Yes	Yes

Table 5 Summary of commit-time options

	Write instance state to database	Instance stays ready	Instance state remains valid
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 9.5.4 illustrate the three alternative commit options in detail.

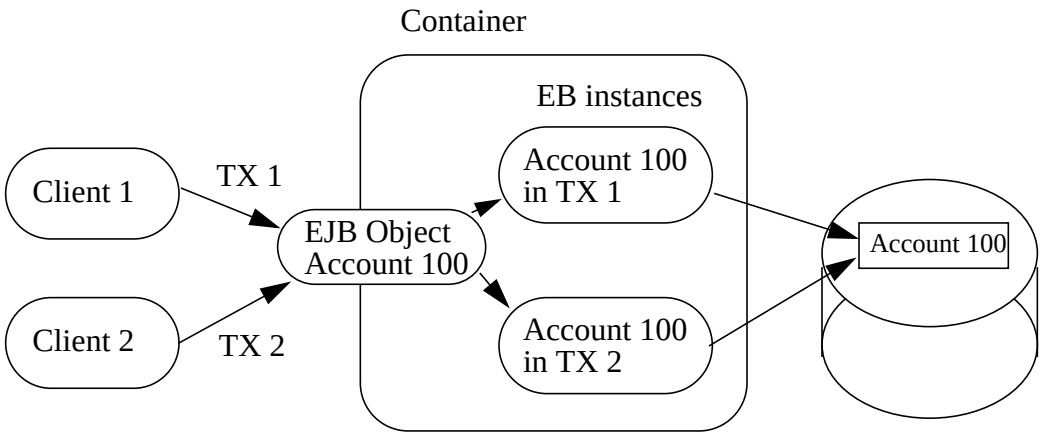
9.1.11 Concurrent access from multiple transactions

When writing the business methods, the enterprise Bean developer does not have to worry about concurrent access from multiple transactions. The enterprise Bean developer may assume that the container will ensure appropriate synchronization for Entity Beans that are accessed concurrently from multiple transactions.

The entity 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 enterprise Bean, one for each transaction in which the entity is being accessed. The transaction synchronization is performed automatically by the underlying database during the database access calls performed by the `ejbLoad`, `ejbCreate`, `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 9.5.4 apply to this type of container.

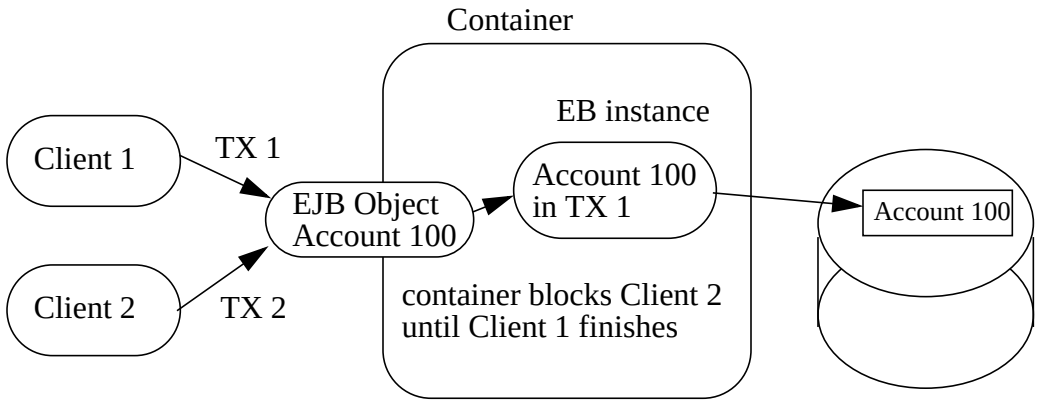
Figure 24 Multiple Clients Can Access the Same Entity EJB using multiple instances



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 instance’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 9.5.4 applies to this type of container.

Figure 25 Multiple Clients Can Access the Same Entity EJB using single instance



9.1.12 Non-reentrant and re-entrant instances

By default, an Entity Bean instance is not re-entrant. If an instance executes a client request in a given transaction context, and another request with the same transaction context arrives at the EJB object, the container will throw the `java.rmi.RemoteException` to the second request. This rules allows the Bean developer to program the 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 Bean A, A calls Bean B, and B calls back A in the same transaction context. The 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 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 try 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 Bean is re-entrant.

Re-entrant Beans must be programmed and used with great caution. First, the Bean programmer must code the 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 EJB 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 can be marked as non-reentrant in the deployment descriptor, allowing the container to detect and prevent illegal concurrent calls from clients.

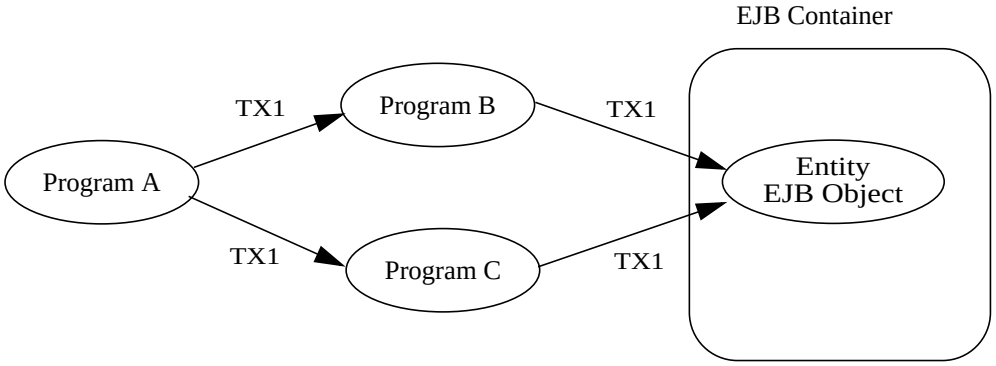
9.1.13 Access from multiple clients in the same transaction context

This section describes a more complex distributed transaction scenario, and specifies the required Container's behavior for this scenarios.

9.1.13.1 Transaction "diamond" topology scenario

An Entity EJB 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 EJB Object, the topology of the transaction creates a diamond.

Figure 26 Transaction diamond scenario



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 the Entity EJB Object through different network paths. This case is challenging because most high-end 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 processes of the EJB Container, and both B and C needs to access the same Entity EJB 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[7].

The above example illustrates a simple diamond. We use the term diamond to refer to any distributed transaction scenario in which an Entity EJB Object is accessed in the same transaction through multiple network paths.

Note that in the diamond scenario the clients B and C must access the Entity EJB Object serially. Concurrent access to an Entity EJB 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 in 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.

9.1.13.2 Container Provider's responsibilities

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 EJB 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 EJB 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.

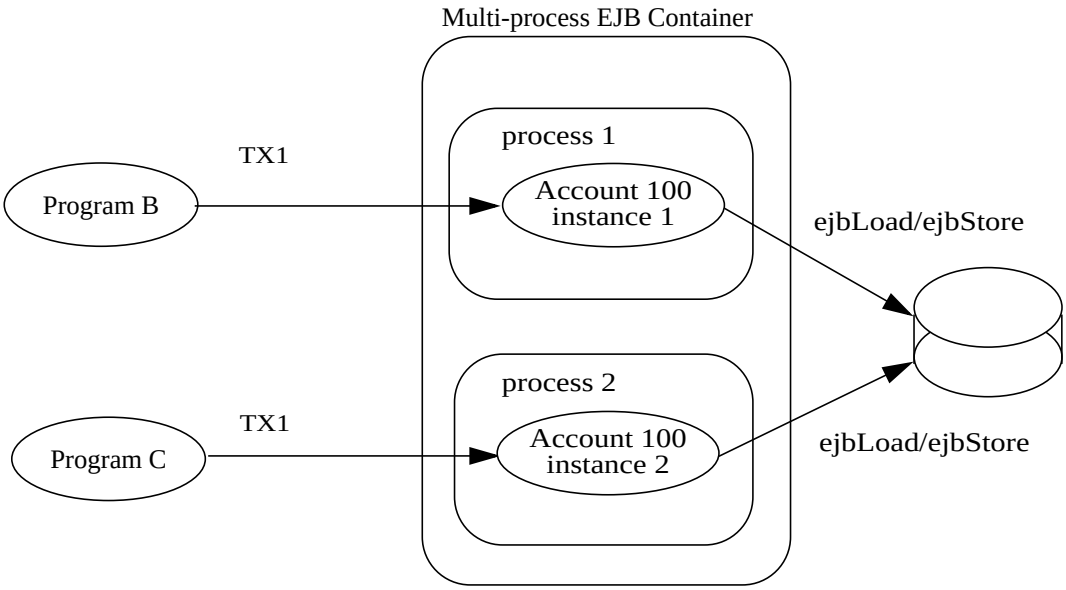
If the Container Provider chooses to support distributed diamonds, it should provide a consistent view of the entity state within a transaction. The Container Provider can implement the support in several ways. (The options that follow are illustrative, not prescriptive.)

- Always instantiate the EJB 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 EJB 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 this approach follows.

[7] This diamond problem applies only to the case when B and C are in the same transaction.

Figure 27 Handling of diamonds by a multi-process Container



Program B makes a call to an Entity bean 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 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 of the Entity Account 100 with a consistent view of the entity state within the transaction.

Note that a more sophisticated Container might avoid calling `ejbLoad` and `ejbStore` on each business method by using a distributed lock manager.

9.1.13.3 Bean Provider's responsibilities

The diamond case is transparent to the Bean Provider—the Bean Provider does not have 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.

9.1.13.4 Application Assembler and Deployer's responsibilities

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 bean may be involved in distributed diamond scenarios.

9.2 Responsibilities of the Enterprise Bean Provider

This section describes the responsibilities of an entity enterprise Bean provider to ensure that an enterprise Bean can be deployed in any EJB Container.

The requirements are stated for the provider of an entity bean with bean-managed persistence. The differences for entities with container-managed persistence are defined in Section 9.4.

9.2.1 Classes and interfaces

The enterprise Bean provider is responsible for providing the following class files:

- Enterprise Bean class
- Enterprise Bean's remote interface
- Enterprise Bean's home interface
- Primary key class

9.2.2 Enterprise Bean class

The following are the requirements for an entity enterprise 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 enterprise Bean's remote interface[8]. If the class implements the enterprise 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 class at runtime.

A no-op implementation of these methods is required to avoid defining the EJB class as abstract.

The class must implement the business methods, and the `ejbCreate, ejbPostCreate,` and `ejbFind<METHOD>` methods as described later in this section.

The enterprise bean class may have superclasses and/or superinterfaces. If the enterprise bean has superclasses, the business methods, the `ejbCreate and ejbPostCreate` methods, the finder methods, and the methods of the `EntityBean` interface may be defined in the enterprise bean class, or any of its superclasses.

9.2.3 *ejbCreate* methods

The enterprise Bean class may define zero or more `ejbCreate(...)` methods whose signatures 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 the primary key type.

The methods arguments 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 is deprecated in EJB 1.1—an EJB 1.1 compliant enterprise bean should throw the javax.ejb.EJBException or another java.lang.RuntimeException to indicate non-application exceptions to the Container (see Section 12.2.2).

[8] It is recommended that the enterprise bean class not implement the remote interface to prevent inadvertent passing of *this* as a method argument or result.

9.2.4 *ejbPostCreate* methods

For each `ejbCreate(...)` method, the enterprise Bean class must define a matching `ejbPostCreate(...)` method, using the following rules:

The method name must be `ejbPostCreate`.

The method must be declared as `public`.

The method must not be declared as `final` or `static`.

The return type must be `void`.

The methods arguments must be the same as the arguments of the matching `ejbCreate(...)` 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 is deprecated in EJB 1.1—an EJB 1.1 compliant enterprise bean should throw the javax.ejb.EJBException or another java.lang.RuntimeException to indicate non-application exceptions to the Container (see Section 12.2.2).

9.2.5 *ejbFind* methods

The enterprise 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 methods arguments types must be legal types for RMI-IIOP.

The return type of a finder method must be the enterprise Bean's primary key type, or an EJB primary key collection (see *Section* Subsection 9.1.8).

The throws clause may define arbitrary application specific exceptions, including the `javax.ejb.FinderException`.

Every entity enterprise Bean class 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).

Compatibility Note: EJB 1.0 allowed the finder methods to throw the `java.rmi.RemoteException` *to indicate a non-application exception. This practice is deprecated in EJB 1.1—an EJB 1.1 compliant enterprise bean should throw the javax.ejb.EJBException or another java.lang.RuntimeException to indicate non-application exceptions to the Container (see Section 12.2.2).*

9.2.6 Business methods

The 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 methods arguments 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 is deprecated in EJB 1.1—an EJB 1.1 compliant enterprise bean should throw the javax.ejb.EJBException or another java.lang.RuntimeException to indicate non-application exceptions to the Container (see Section 12.2.2).*

9.2.7 Enterprise Bean's remote interface

The following are the requirements for the enterprise 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 value types must be valid types for RMI-IIOP, and their throws clause 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 enterprise 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.

9.2.8 Enterprise Bean's home interface

The following are the requirements for the enterprise Bean's home interface signature:

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 clause 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.

Each `create` method must be named "**create**", and it must match one of the `ejbCreate` methods defined in the enterprise Bean class. The matching `ejbCreate` 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 must be the enterprise Bean's remote interface type.

All the exceptions defined in the throws clause of the matching `ejbCreate` and `ejbPostCreate` methods of the enterprise Bean class must be included in the throws clause of the matching `create` method of the remote 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` and `ejbPostCreate` methods)

The throws clause of a `create` 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 enterprise 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 enterprise 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 enterprise Bean class must be included in the throws clause of the matching `find` method of the remote interface.

The throws clause of a `finder` method must include the `javax.ejb.FinderException`.

9.2.9 Enterprise Bean's primary key class

The Bean provider must specify a primary key class in the deployment descriptor.

The primary key class 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.

9.3 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 Entity Bean objects 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.

9.3.1 Generation of implementation classes

The deployment tools provided by the container provider are responsible for the generation of additional classes when the enterprise 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 Bean's deployment descriptor.

The deployment tools must generate the following classes:

- A class that implements the enterprise Bean's home interface.
- A class that implements the enterprise Bean's remote interface.

The deployment tools may also generate a class that mixes some container-specific code with the enterprise Bean class. The 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.

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.

9.3.2 EJB Home class

The EJB home class, which is generated by deployment tools, implements the enterprise Bean's home interface. This class also implements the methods of the `javax.ejb.EJBHome` interface, and the type-specific `create` and `finder` methods specific to the enterprise Bean.

The implementation of each `create(...)` methods invokes a matching `ejbCreate(...)` method, followed by the matching `ejbPostCreate(...)` method, passing the `create(...)` parameters to these 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>(...)` methods invokes a matching `ejbFind<METHOD>(...)` method. The implementation of the `find<METHOD>(...)` method must create an EJB object for the primary key returned from the `ejbFind<METHOD>`, and return the EJB 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 EJB objects for the primary keys, and return the collection to the client.

9.3.3 EJB Object class

The EJB Object, which is generated by deployment tools, implements the enterprise Bean's remote interface. It also implements the methods of the `javax.ejb.EJBObject` interface and the business methods specific to the enterprise 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.

9.3.4 Handle class

The deployment tools responsible for implementing the handle class for the enterprise Bean. The handle class must be serializable by the Java programming language Serialization protocol.

As the handle class is not EJB type specific, the container may use a single class for all deployed enterprise beans.

9.3.5 Home Handle class

The deployment tools responsible for implementing the home handle class for the enterprise Bean. The handle class must be serializable by the Java programming language Serialization protocol.

Because the home handle class is not EJB type specific, the container may use a single class for the homes of all deployed enterprise beans.

9.3.6 Meta-data 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-IIOP Value Type, and must implement the `javax.ejb.EJBMetaData` interface.

Because the meta-data class is not EJB type specific, the container may use a single class for all deployed enterprise beans.

9.3.7 Instance's re-entrance

The container runtime must enforce the rules defined in Section 9.1.12.

9.3.8 Transaction scoping, security, exceptions

The container runtime must follow the rules on transaction scoping, security checking, and exception handling described in Chapters 11, 15, and 12.

9.4 Entity Beans with container-managed persistence

The previous sections described the component contract for Entity Beans with Bean-managed persistence. This section specifies the contract for the Entity Beans with container-managed persistence.

The deployment descriptor for an entity bean indicates whether the entity bean uses bean-managed persistence or container-managed persistence.

The contract for an entity bean with container-managed persistence is the same as that for an entity with bean-managed persistence (as described in the previous sections), except for the differences described in the following subsections.

9.4.1 Container-managed fields

An entity bean with container-managed persistence relies on the Container Provider's tools to generate methods that perform data access on behalf of the enterprise bean instances. The generated methods transfer data between the enterprise 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 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 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 container is responsible for transferring data between the 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 following requirements ensure that an Entity Bean can be deployed in any compliant container.

- The Bean Provider must ensure that the Java types assigned to the container-managed fields are one of the following: Java primitive types, Java serializable types, or references to enterprise beans' remote or home interfaces.

- The Container Provider may, but is not required to, use Java serialization to store the entity state 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 types for the container-managed fields, we expect that in practice the Bean Provider 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 resource 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 entities that expect mapping to an underlying database schema should use bean-managed persistence instead.

The provider of entities with container-managed persistence must take into account the following limitations of the 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 state into the container-managed fields before invoking the `ejbLoad` method. This approach may not be suitable for entities whose state is large, and whose business methods require access to only parts of the state.

An entity designer who runs into the limitations of the container-managed persistence should use bean-managed persistence instead.

9.4.2 ejbCreate, ejbPostCreate

With Bean-managed persistence, the enterprise Bean developer 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 enterprise Bean developer's responsibility is to initialize the container-managed fields in an `ejbCreate(...)` method from the input arguments such that when `ejbCreate(...)` returns, the container can extract the container-managed fields from the instance and insert them into the database.

The `ejbCreate(...)` method must be defined to return the primary key class type. Because the `ejbCreate(...)` method of an entity with container-managed persistence does not know the primary key (the primary key will be established by the Container later), the `ejbCreate(...)` method should return a `null`. The return value is ignored by the Container.

The container is responsible for extracting the primary key fields of the newly created entity representation in the database, and for creating an EJB object reference for the primary key. 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).

Then the container 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 in the transaction context determined by the transaction attribute of the matching `create(...)` method, as described in subsection 11.6.2. Depending on the value of the transaction attribute, the transaction context can be the client's transaction context, a new transaction context, or non-existent.

9.4.3 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 an EJB object reference or on the EJB home interface.

The enterprise Bean provider can use the `ejbRemove` method to implement any actions that must be done before the entity representation is removed from the database.

After `ejbRemove` returns, the container removes the entity 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 11.6.2. Depending on the value of the transaction attribute, the transaction context can be the client's transaction context, a new transaction context, or non-existent.

9.4.4 ejbLoad

When the container needs to synchronize the state of an instance with the entity state in the database, the container reads the entity state from the database into the container-managed fields and then it invokes the `ejbLoad()` method on the instance.

The enterprise Bean developer can rely on the container's having loaded the container-managed fields from the database just before the container invoked the `ejbLoad()` method. The enterprise 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).

9.4.5 ejbStore

When the container needs to synchronize the state of the entity state in the database with the state of the 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 enterprise Bean developer 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.

9.4.6 finder methods

The enterprise Bean provider does not write the finder (`ejbFind<METHOD>(...)`) methods.

The finder methods are generated at Bean deployment time using the container provider's tools. The tools can, for example, create a subclass of the enterprise 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 enterprise Bean's home interface.

Note that the `ejbFind<METHOD>` names and parameter signatures 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 bean Deployer uses container tools to generate the implementation of the finder methods based in the description supplied by the bean provider. The Enterprise JavaBeans architecture does not specify the format of the finder method description.

9.4.7 primary key type

The container must be able to manipulate the primary key type. Therefore, the primary key type for a Bean with container-managed persistence must follow the rules in this subsection, in addition to those specified in Subsection 9.2.9.

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 to include 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.

9.4.7.1 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 bean class that contains the primary key. The field's type must be the primary key type.

9.4.7.2 Primary key that maps to multiple fields in the entity bean class

The 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.)

As explained in Subsection 9.2.9, in special situations the Bean Provider may defer the specification of the primary key type to the entity bean Deployer.

9.4.7.3 Special case: Unknown primary key class

In special situations, the Bean Provider may choose not to specify the primary key class for an entity bean with container-managed persistence. This case happens if the Bean Provider wants to allow the Deployer to select the primary key fields at deployment time. The Deployer uses instructions supplied by the Bean Provider (these instructions are beyond the scope of the EJB spec.) to define a suitable primary key class.

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 when the Bean Provider develops enterprise beans 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 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 (for example, EJBHome.findByPrimaryKey(...)).

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.

9.5 Object interaction diagrams

This section uses object interaction diagrams to illustrate the interactions between an Entity Bean instance and its container.

9.5.1 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

9.5.2 Creating an entity object

Figure 28 OID of Creation of an enterprise Bean with Bean-managed persistence.

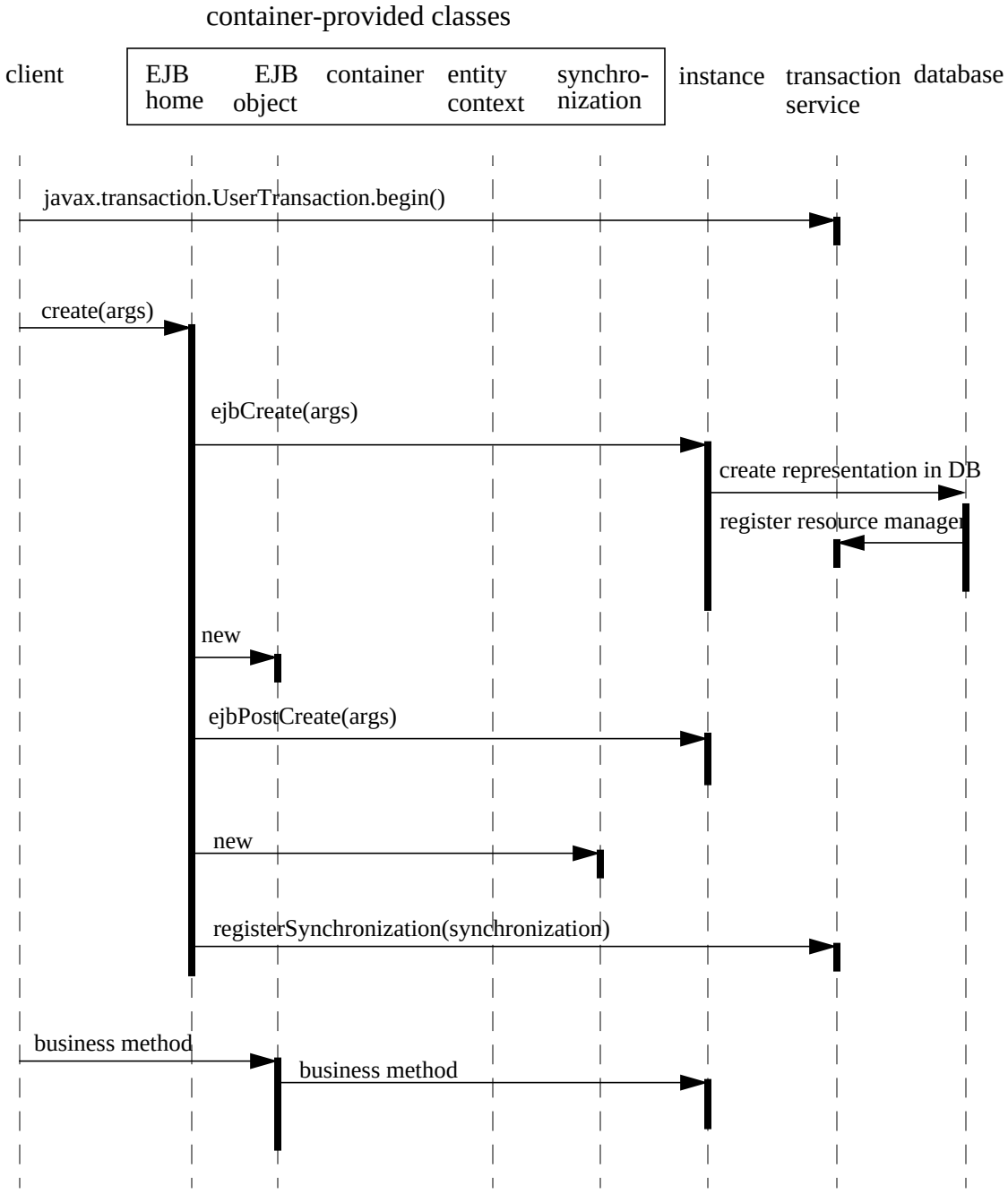
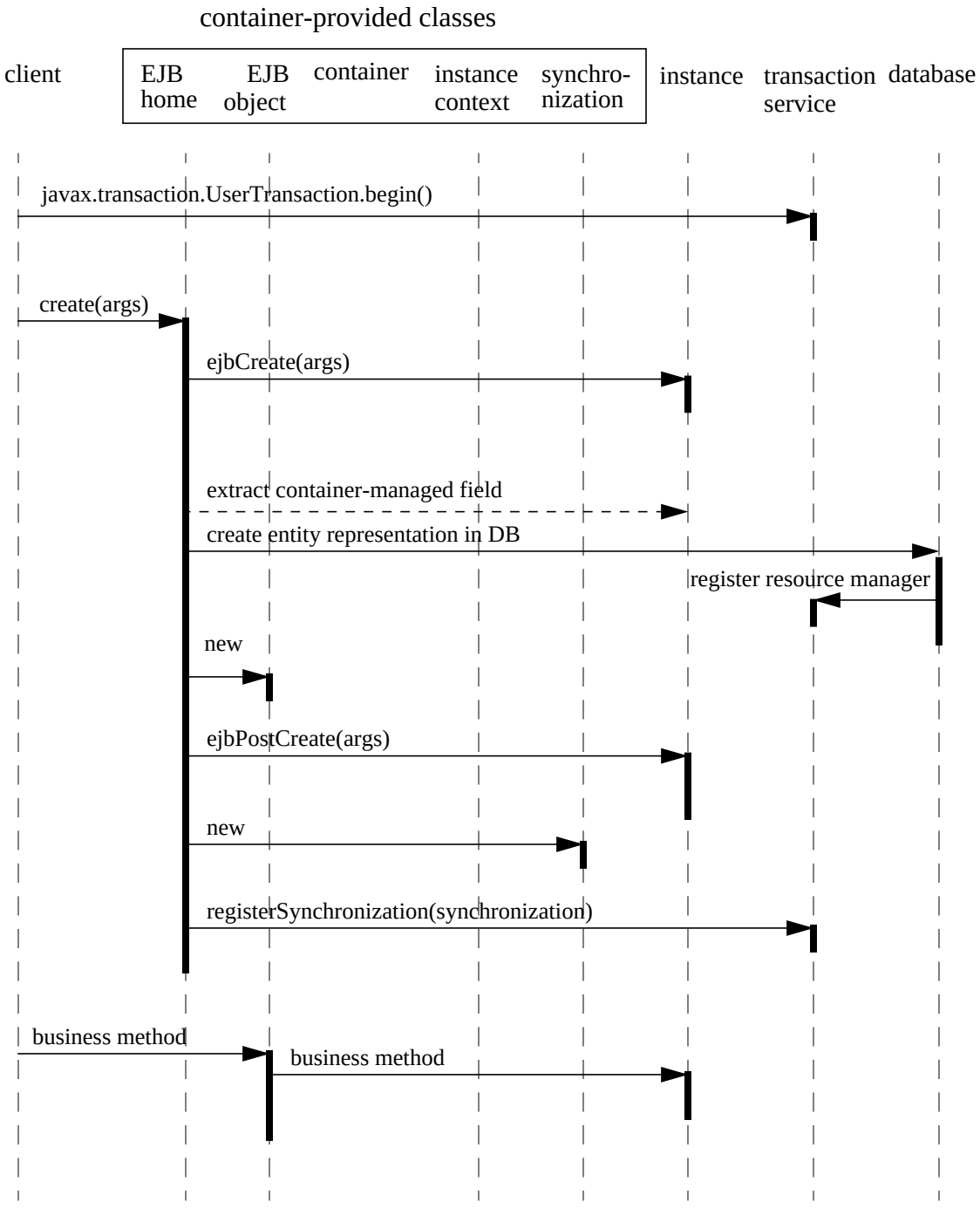


Figure 29 OID of Creation of an enterprise Bean with container-managed persistence:



9.5.3 Passivating and activating an instance in a transaction

Figure 30 OID of Passivation and Reactivation of an EJB instance with Bean-managed persistence.

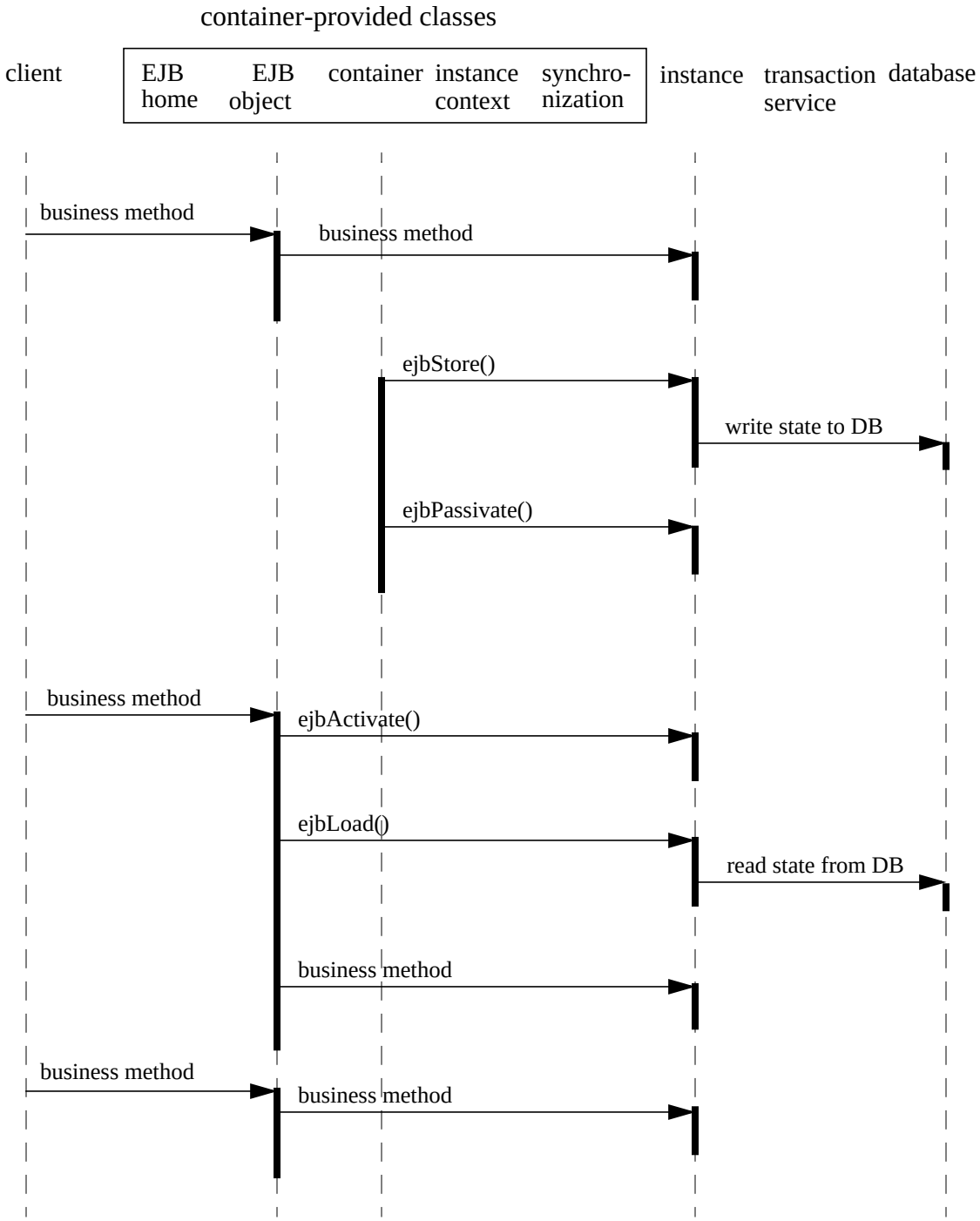
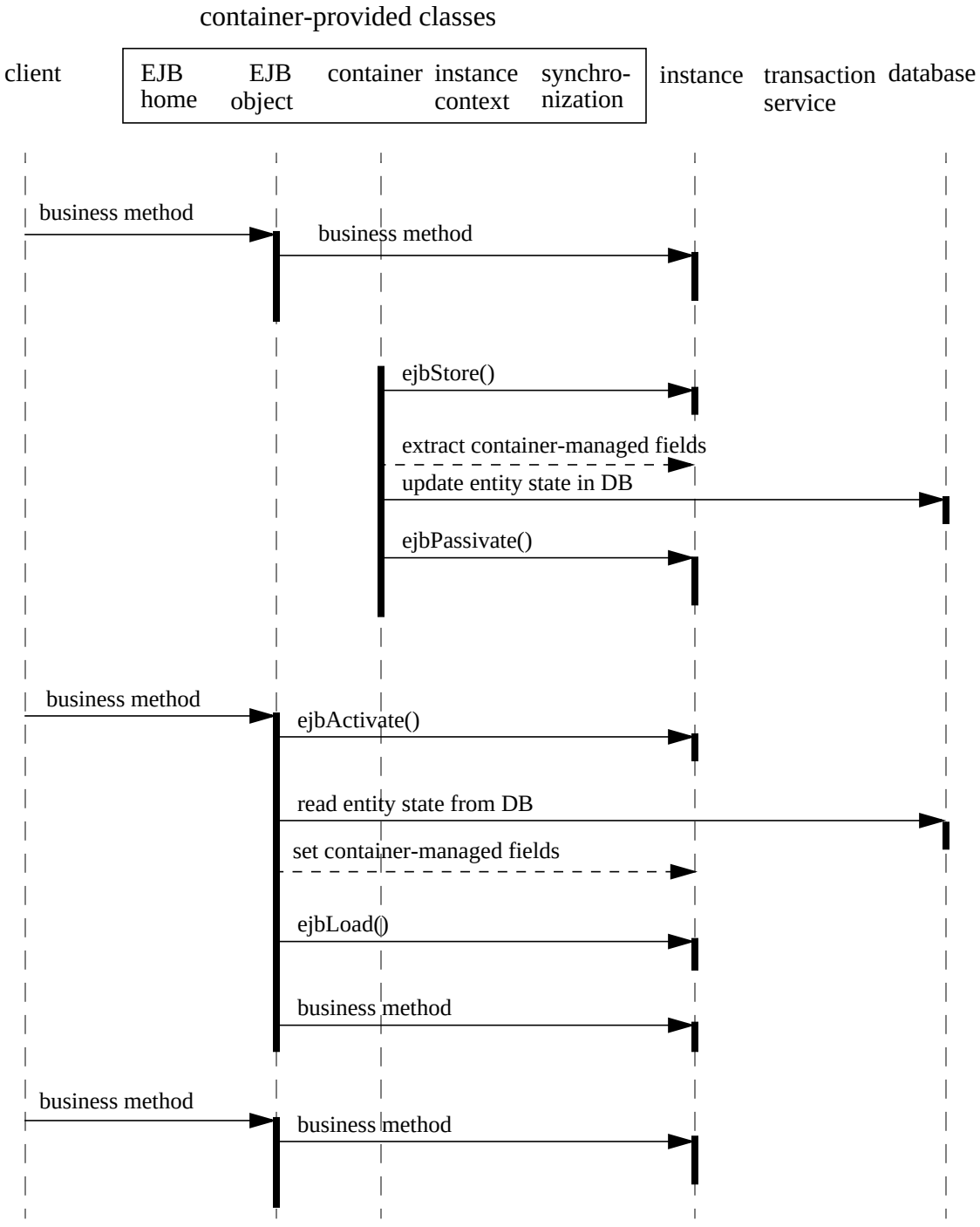


Figure 31 OID of Passivation and reactivation of an EJB instance with CMP.



9.5.4 Committing a transaction

Figure 32 OID of transaction commit protocol with an EJB instance with Bean-managed persistence.

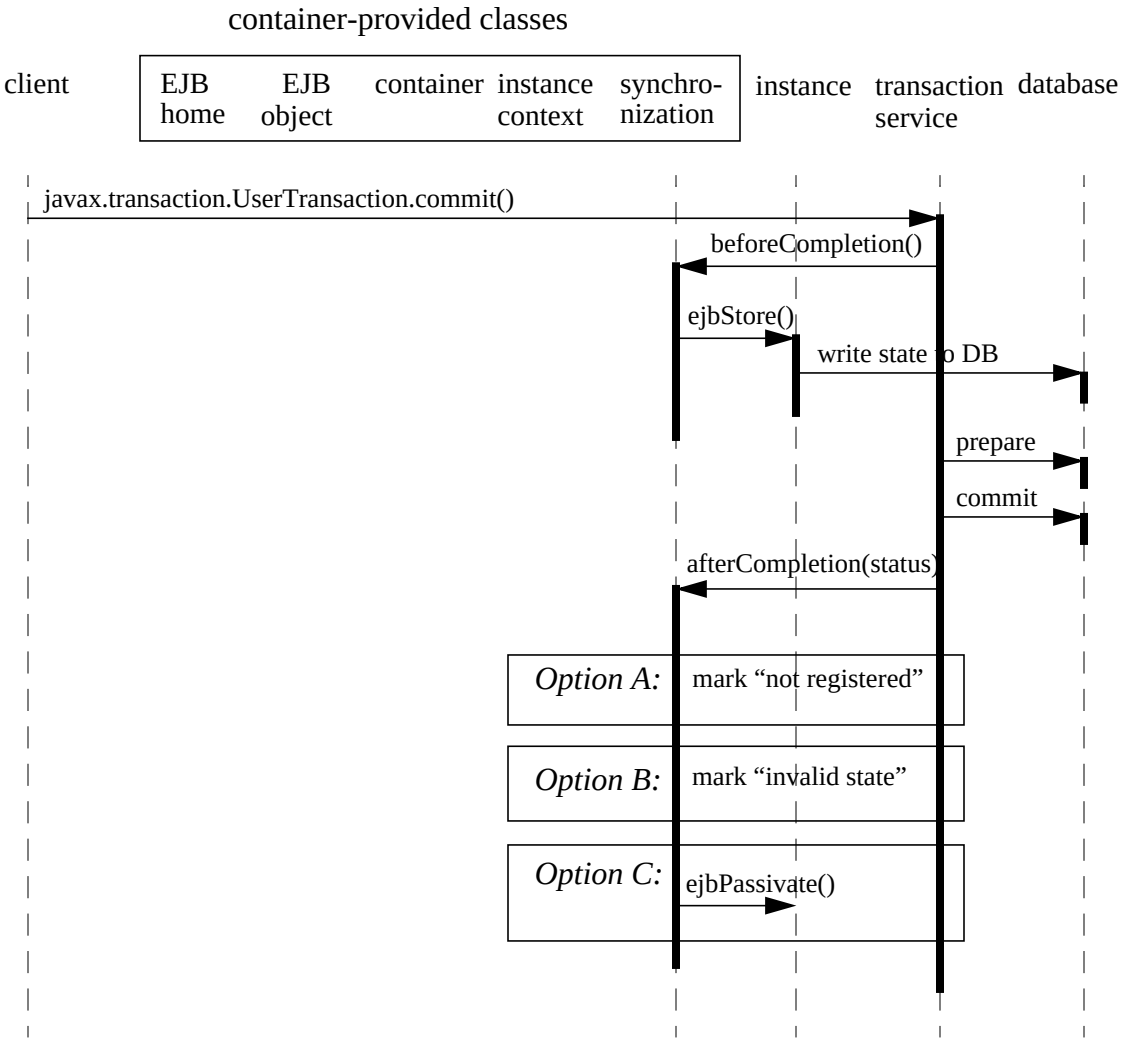
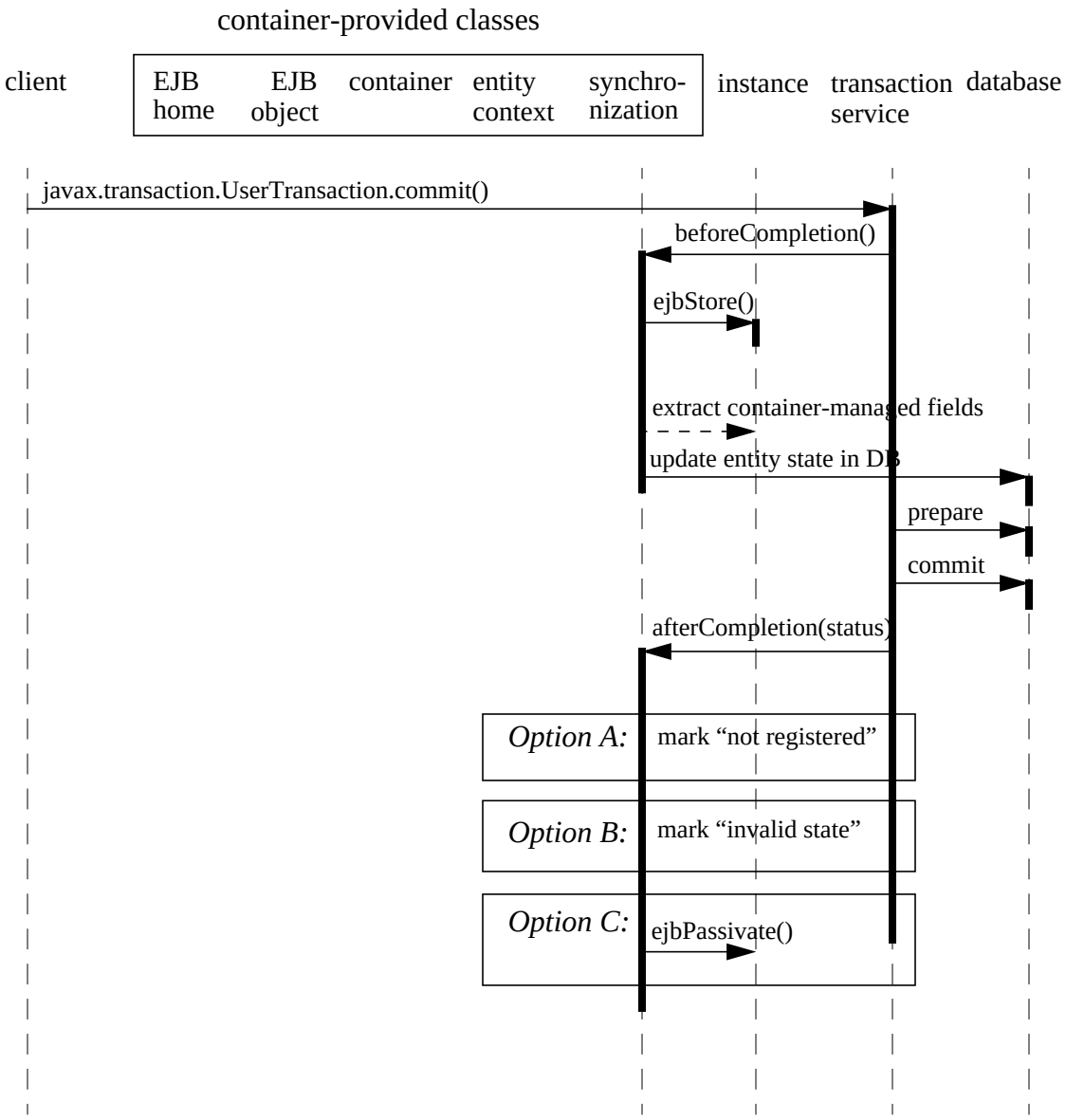


Figure 33 OID of transaction commit protocol for EJB instance with container-managed persistence.



9.5.5 Starting the next transaction

The following diagram illustrates the protocol performed for a Bean 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 34 OID of Start of Transaction for EJB instance using bean-managed persistence.

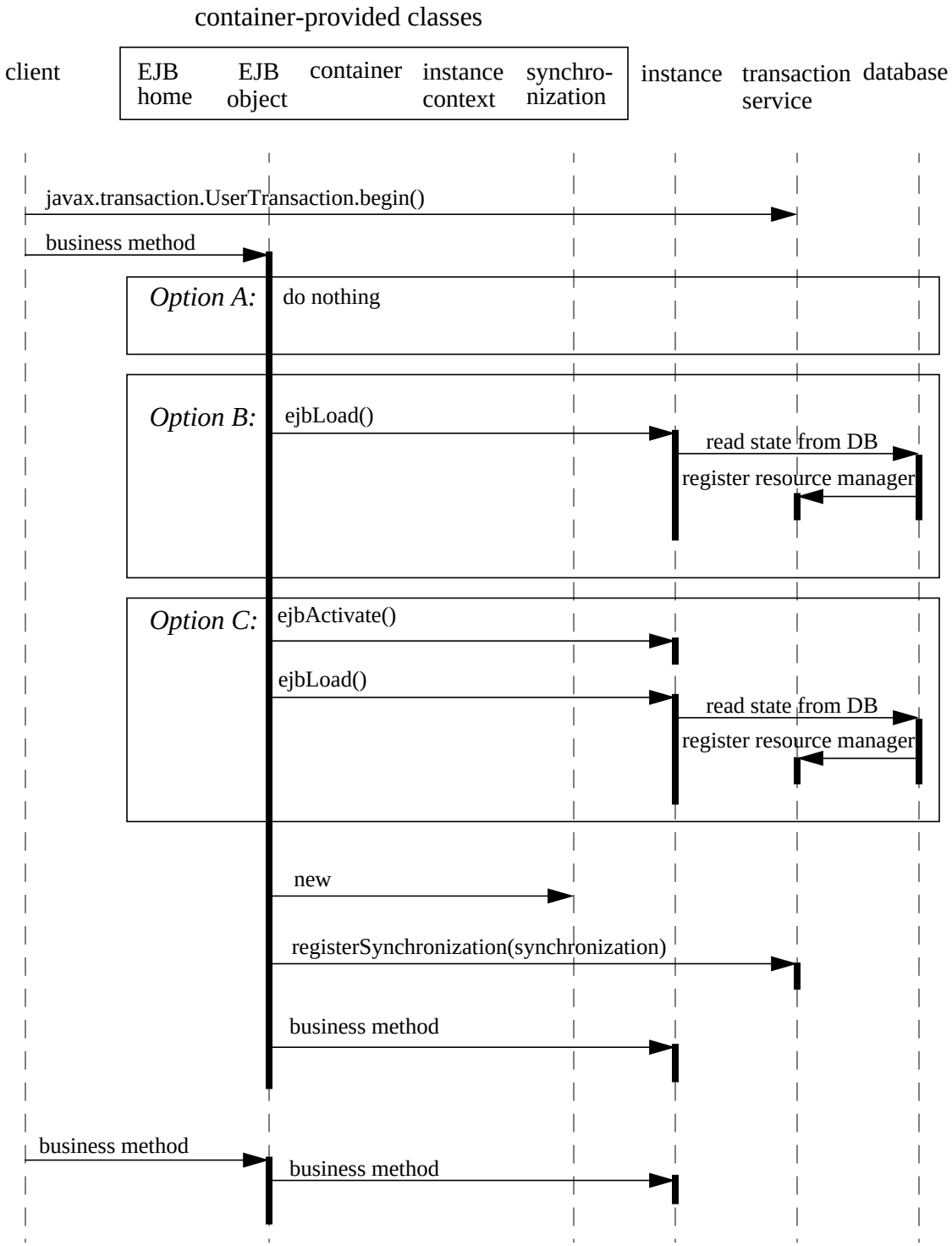
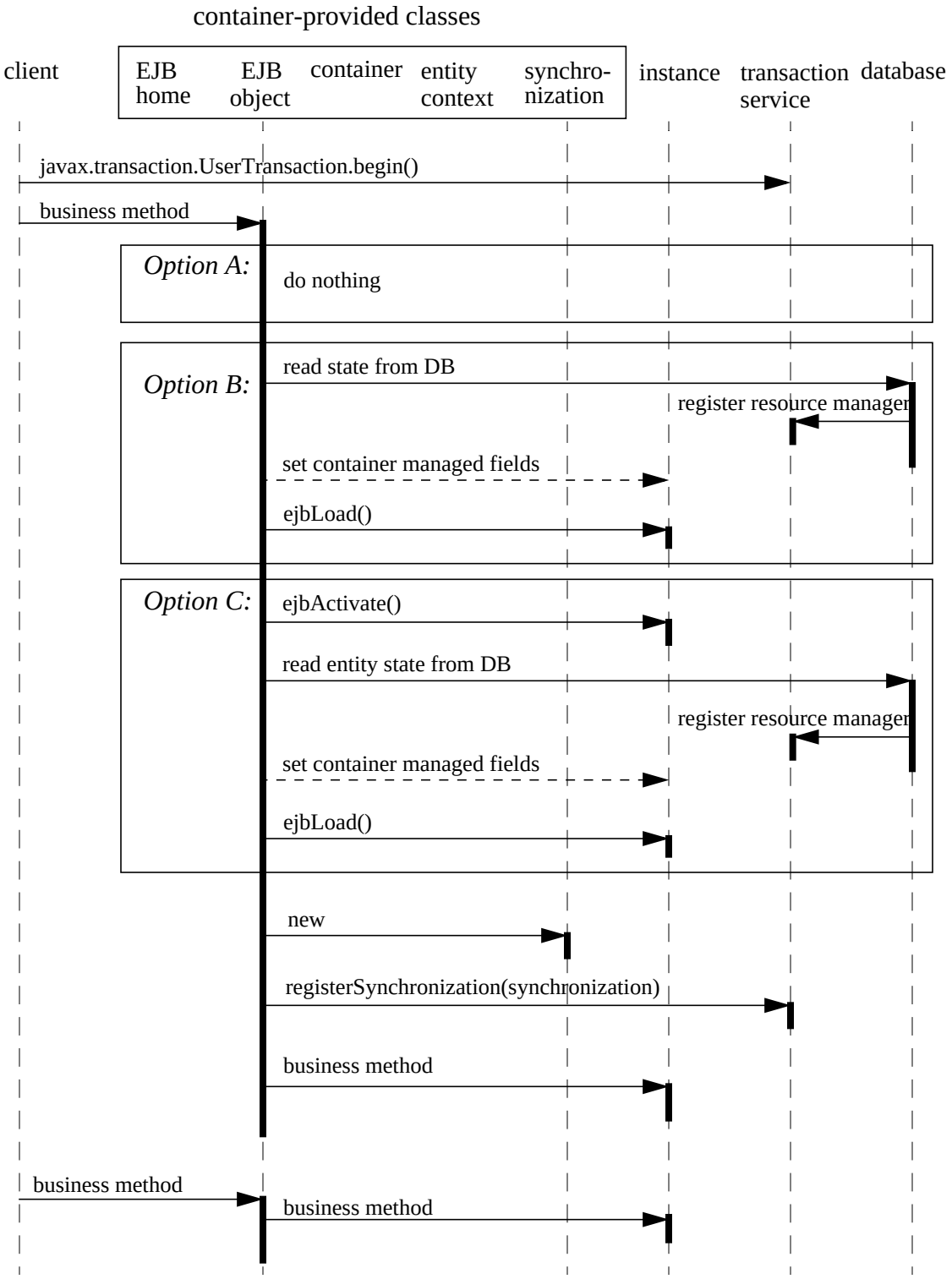


Figure 35 OID of Protocol performed for an EJB with CMP at the beginning of a new transaction.



9.5.6 Removing an entity object

Figure 36 OID of Destruction of an entity EJB with Bean-managed persistence.

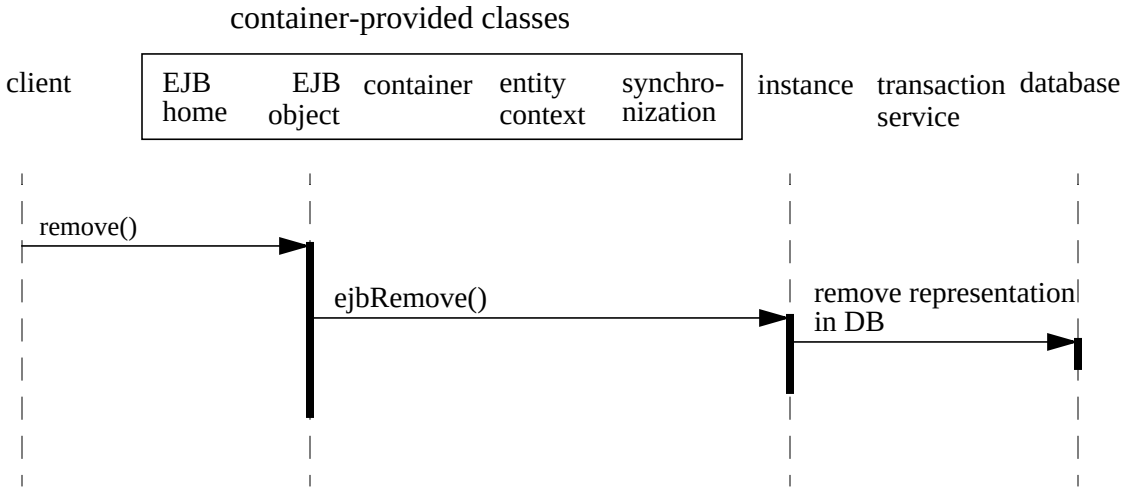
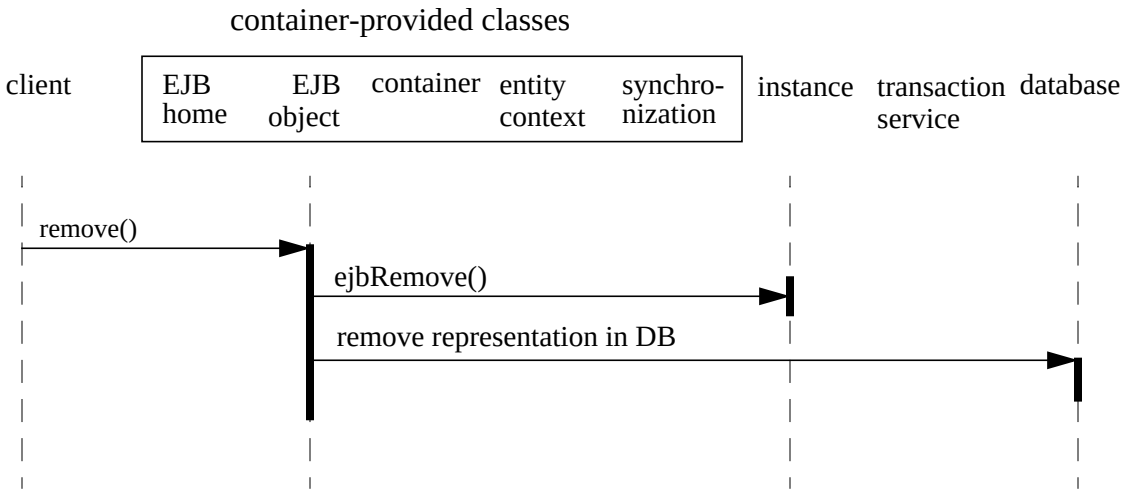


Figure 37 OID of Destruction of an entity EJB with container-managed persistence.



9.5.7 Finding an object

Figure 38 OID of Execution of a finder method on an entity EJB with Bean-managed persistence.

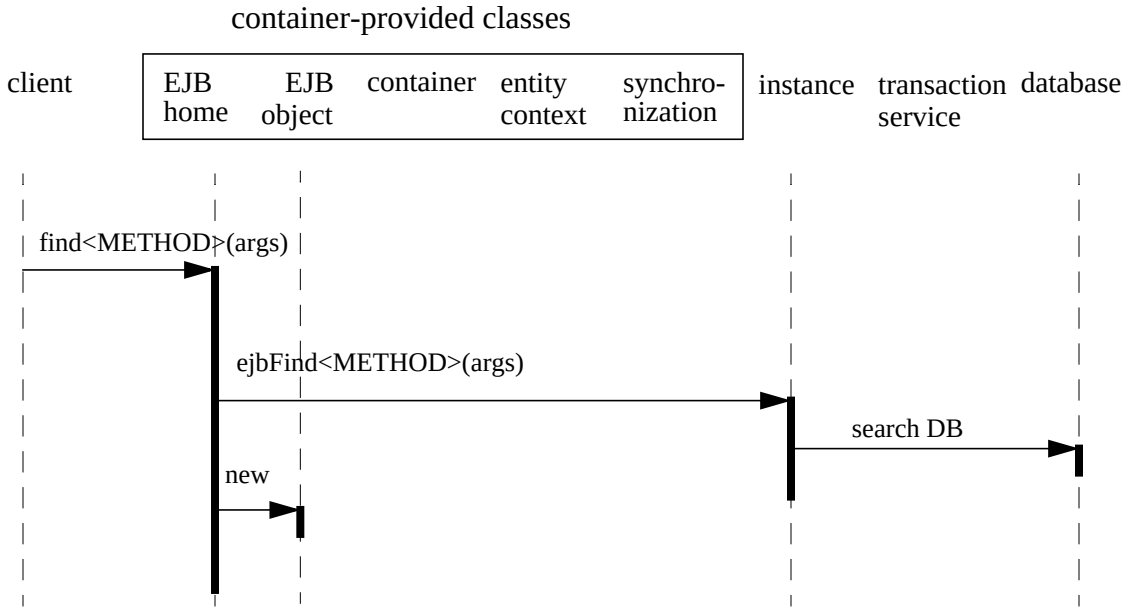
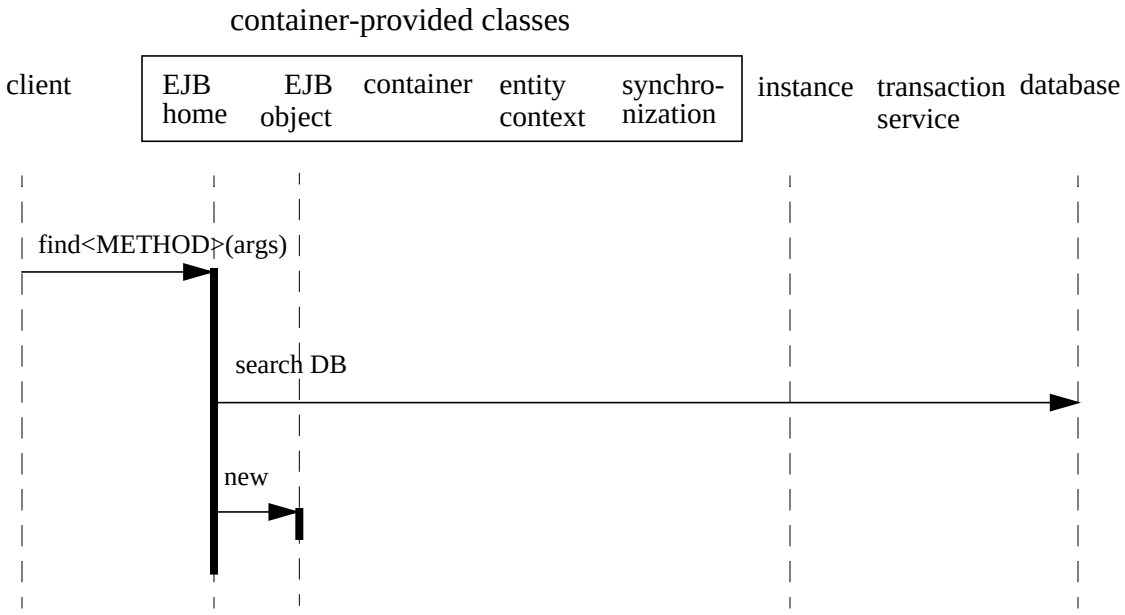


Figure 39 OID of Execution of a finder method on an entity EJB with container-managed persistence.



9.5.8 Adding and removing instance from the pool

The diagrams in Subsections 9.5.2 through 9.5.7 did not show the sequences between the “does not exist” and “pooled” state (see the diagram in Section 9.1.4).

Figure 40 OID of Sequence for a container adding an instance to the pool.

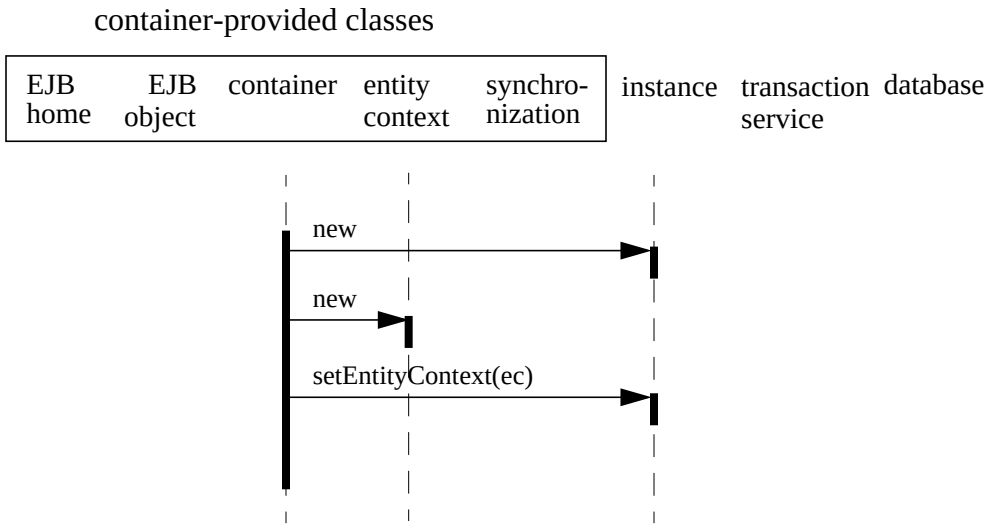
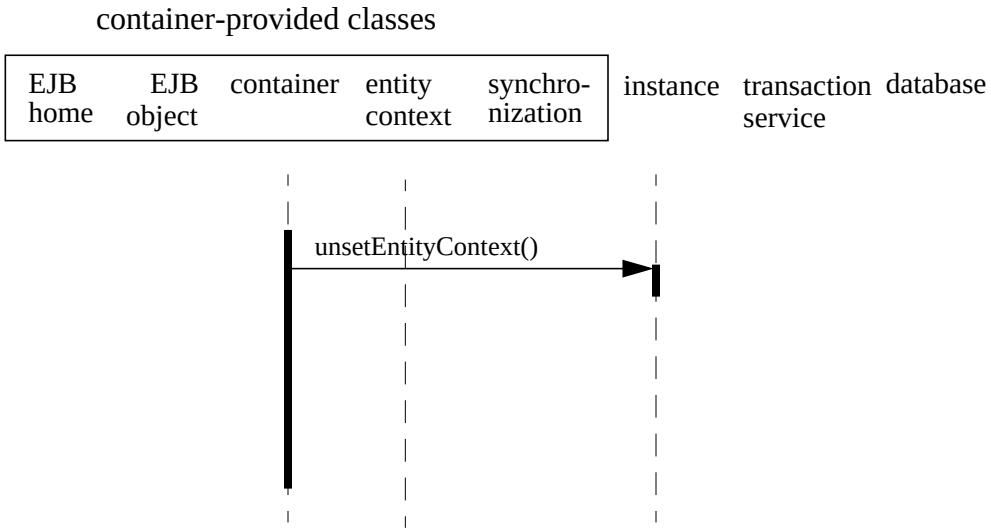


Figure 41 OID of Sequence for a container removing an instance from the pool.



Chapter 10 Example entity scenario

This chapter describes an example development and deployment scenario for an entity enterprise Bean. We use the scenario to explain the responsibilities of the enterprise Bean provider and those of the container provider.

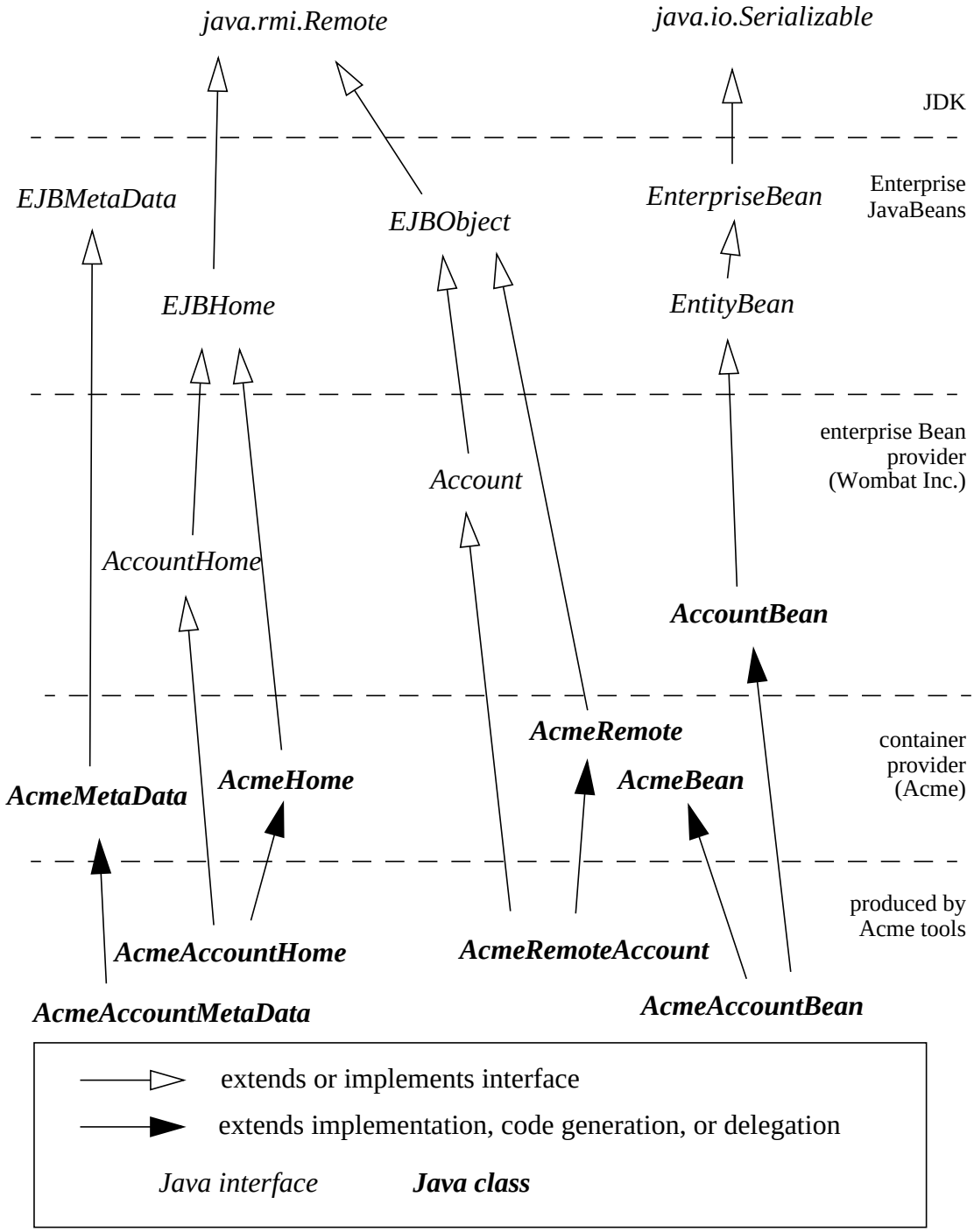
The classes generated by the container provider's tools in this scenario should be considered illustrative rather than prescriptive. Container providers are free to implement the contract between an enterprise Bean and its container in a different way that achieves an equivalent effect (from the perspectives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

10.1 Overview

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

10.2 Inheritance relationship

Figure 42 Example of the inheritance relationship between the interfaces and classes:



10.2.1 What the enterprise Bean provider is responsible for

Wombat Inc. is responsible for providing the following:

- *Define the enterprise 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 enterprise Bean class (AccountBean). The enterprise Bean class may, but is not required to, implement the enterprise Bean's remote interface (Account). The enterprise Bean must implement the javax.ejb.EntityBean interface, and define the ejbCreate(...) methods invoked at an EJB object creation.*

- *Define a home interface (AccountHome) for the enterprise Bean. The home interface defines the EJB class 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 metadata that the enterprise Bean provider wishes to pass with the enterprise Bean to the next stage of the development/deployment workflow.*

10.2.2 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 enterprise 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.*

10.2.3 What the container provider is responsible for

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

- *Generate the remote Bean class (AcmeRemoteAccount) for the enterprise Bean. The remote Bean class is a "wrapper" class for the enterprise Bean and provides the client view of the enterprise Bean. The tools also generate the classes that implement the communication stub and skeleton for the remote Bean class.*

- *Generate the implementation of the enterprise 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 home class (AcmeAccountHome) for the enterprise Bean. The home class implements the enterprise Bean's home interface (AccountHome). The tools also generate the classes that implement the communication stub and skeleton for the home class.*

- *Generate a class (AcmeAccountMetaData) that implements the javax.ejb.EJBMetaData interface for the Account 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 the generated classes most likely will be different from those generated by Acme's tools.

Chapter 11 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.

11.1 Overview

This section provides a brief overview of transactions and illustrates a number of scenarios of transactions in EJB.

11.1.1 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 or declarative transaction demarcation performed automatically by the EJB Container. In the latter case, the Container demarcates transactions per instructions provided by the Application Assembler. In both cases, the burden of implementing transaction management is on the EJB Container and Server Providers. 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, transaction context propagation, and distributed two-phase commit.

11.1.2 Transaction model

The resource access performed by an enterprise bean is done in the scope of a transaction. A transaction can be local or global. A local transaction is managed by the resource manager; a global transaction is managed by a global transaction manager that is part of the EJB Server. The scope of a local transaction is limited to a single resource manager; the scope of a global transaction may include multiple resource managers and may span multiple servers on the network.

In the case of an enterprise bean with bean-managed transactions, it is the Bean Provider who decides whether a local or global transaction is used. In the case of an enterprise bean with container-managed transactions, it is the transaction attribute set by the Application Assembler or Deployer that determines if a local or global transaction is used.

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.*

11.1.3 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.

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 `javax.transaction.UserTransaction` interface defined in JTA, but it does not require the support for the JTA resource manager and application server interfaces.

11.2 Scenarios

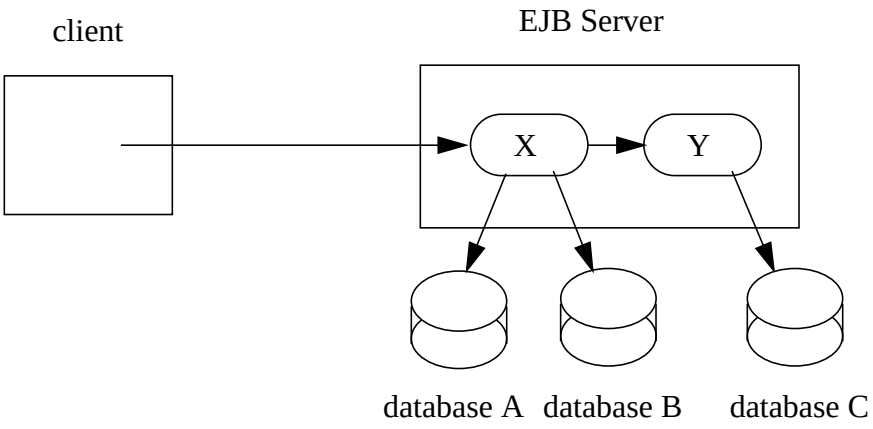
This section describes several scenarios that illustrate the distributed transaction capabilities of the Enterprise JavaBeans architecture.

11.2.1 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. 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. 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 43 Updates to Simultaneous Databases



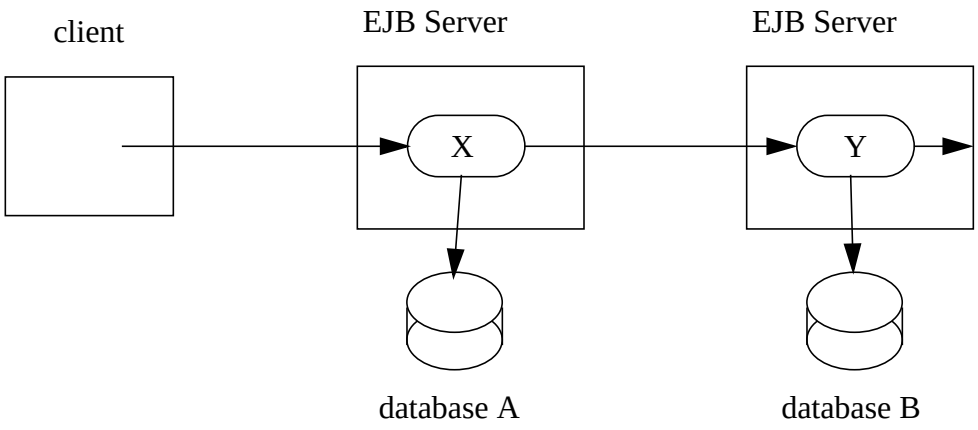
The application programmer does not have to do anything to ensure transactional semantics. The enterprise Beans X and Y perform the database updates using the standard JDBC™ API. Behind the scenes, the EJB Server enlists the database connections as part of the transaction. When the transaction commits, the EJB Server and the database systems perform a two-phase commit protocol to ensure atomic updates across all the three databases.

11.2.2 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. X updates data in database A, and then calls another enterprise Bean Y that is installed in a remote EJB Server. Y updates data in database B. The Enterprise JavaBeans architecture makes it possible to perform the updates to databases A and B as a single transaction.

Figure 44 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.

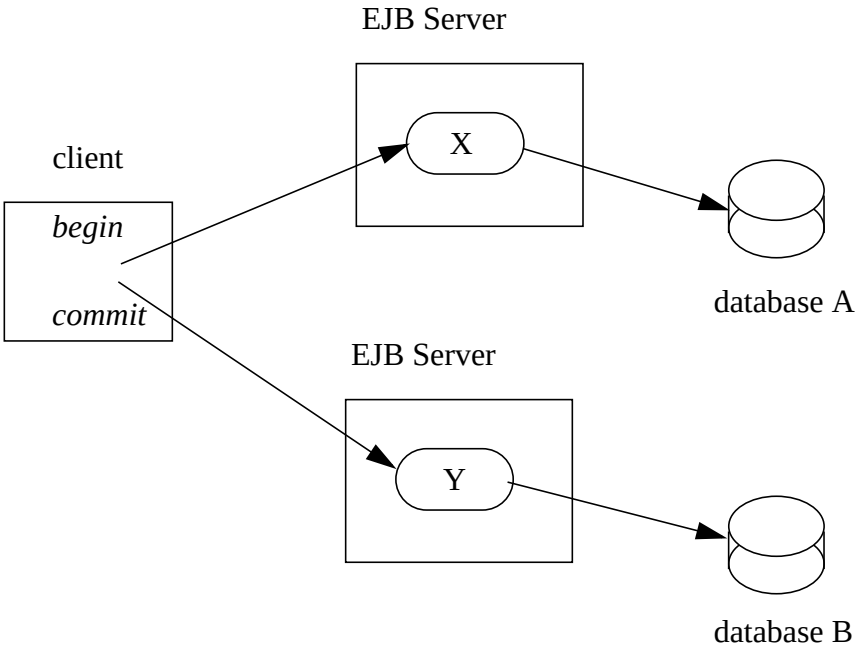
At transaction commit time, the two EJB Servers use a distributed two-phase commit protocol (if the capability exists) to ensure the atomicity of the database updates.

11.2.3 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.User-Transaction` interface using JNDI as defined in the JTA specification [5].

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 45 Updates On Multiples 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.*

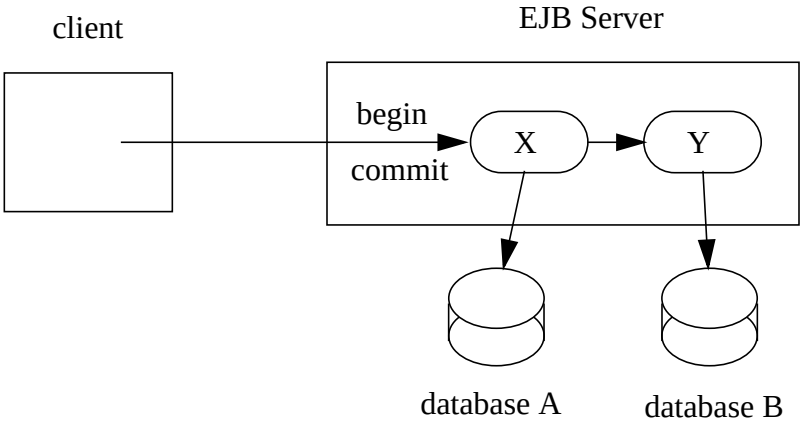
11.2.4 Container-managed demarcation

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** that is set by the Application Assembler. (See [11.4.1] 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 a transaction-enabled enterprise Bean. 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. Since the message from the client does not include a transaction context, the container starts a new transaction before dispatching the remote method on X. 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 46 Update of Multiple Databases From Non-transactional Client



The container automatically commits the transaction at the time X returns a reply to the client.

11.2.5 Bean-managed demarcation

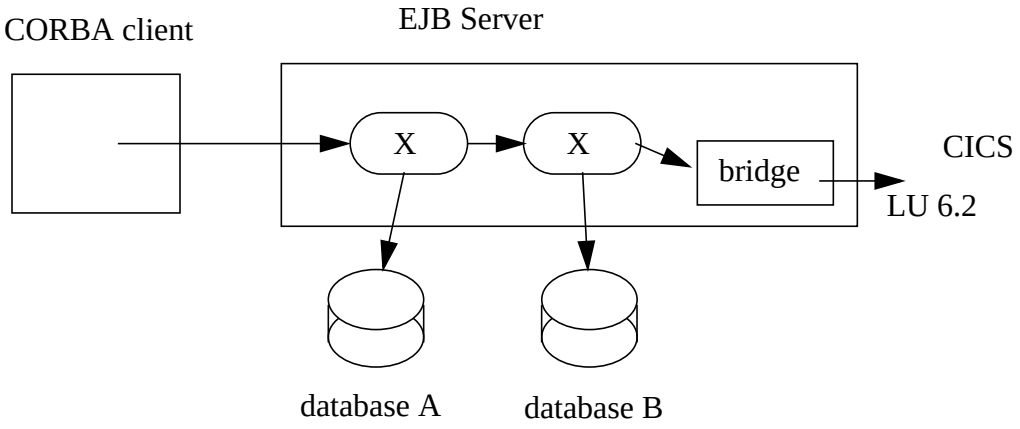
A session Bean can use the `javax.transaction.UserTransaction` interface to programmatically demarcate transactions.

11.2.6 Interoperability with non-Java clients and servers

Although the focus of the Enterprise JavaBeans architecture is the Java API for writing distributed enterprise applications in the Java programming language, it is desirable that such applications are also interoperable with non-Java clients and servers.

A container can make it possible for an enterprise Bean to be invoked from a non-Java client. For example, the CORBA mapping of the Enterprise JavaBeans architecture [8] allows any CORBA client to invoke any enterprise Bean object on a CORBA-enabled server using the standard CORBA API.

Figure 47 Interoperating with Non-Java Clients and/or Servers



Providing connectivity to existing server applications is also important. An EJB Server may choose to provide access to existing enterprise applications, such as applications running under CICS on a mainframe. For example, an EJB Server may provide a bridge that makes existing CICS programs accessible to enterprise Beans. The bridge can make the CICS programs visible to the Java programming language-based developer as if the CICS programs were other enterprise Beans installed in some container on the EJB Server.

> *Note: It is beyond the scope of the Enterprise JavaBeans specification to define the bridging protocols that would enable such interoperability.*

11.3 Bean Provider's responsibilities

This section describes the Bean Provider's view of transactions and defines his responsibilities.

11.3.1 Bean-managed versus container-managed demarcation

As part of the design of an enterprise bean, the Bean Provider must make a decision whether the enterprise bean will demarcate transactions programmatically in the business methods, or whether the transaction demarcation is to be performed by the Container based on the Application Assembler's instructions. (See [11.3.6] for more information.)

If the enterprise bean performs transaction demarcation programmatically, the enterprise bean is referred to as an enterprise bean using *bean-managed transaction*. If the enterprise bean relies on the Container to perform transaction demarcation based on the Application Assembler's instructions, the enterprise bean is referred to as an enterprise bean using *container-managed transaction*.

A Session Bean can be designed for bean-managed transactions or for container-managed transactions. (But it cannot be both at the same time.)

An Entity Bean must always be designed for container-managed transactions.

11.3.2 Local versus global transaction

A *local* transaction is managed by the resource manager itself. A local transaction ensures the ACID properties for multiple updates to the same resource manager. Local transactions cannot ensure the atomicity of updates to multiple resource managers, for the following reason: If an enterprise bean performs updates to multiple resource managers, each using its own local transaction, every resource manager commits or rolls back its transaction independently from the other resource managers' transactions.

A *global* transaction is managed by a global transaction manager, which is typically part of the EJB Server. A global transaction manager ensures the ACID properties for multiple updates to multiple resource managers by performing a two-phase commit protocol across the multiple resource managers enlisted in a transaction.

An enterprise bean using a bean-managed transaction manages local transactions by using the transaction demarcation API specific to each resource manager type. For example, an enterprise bean using JDBC to access a database uses the transaction-related methods of the `java.sql.Connection` interface (i.e. `commit()`, `rollback()`, and `setAutoCommit(...)`) to demarcate transactions on the connection.

An enterprise bean using bean-managed transaction demarcates global transactions by using the `javax.transaction.UserTransaction` interface. All updates to the resource managers between the `UserTransaction.begin()` and `UserTransaction.commit`() methods are performed in a global transaction.

For an enterprise bean using container-managed transaction, the transaction attribute specified in the deployment descriptor by the Application Assembler determines if the container should include the invocation of the enterprise bean's business method as part of a global transaction, or whether the business method is executed using local transaction(s). See Subsection 11.6.2 for the rules that specify how the Container deals with container-managed transactions.

11.3.3 Isolation levels

Transactions not only for make completion of a unit of work atomic, but 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 by a transaction is isolated from the access to the resource 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 resources, the Bean Provider may specify the same or different isolation level for each resource. This means, for example, that if an enterprise bean accesses multiple resources in a transaction, access to each resource 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 levels. 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 using bean-managed transactions, 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 session beans with container-managed transactions and entity beans with bean-managed persistence, the Bean Provider can specify the desirable isolation level programmatically in the enterprise bean's methods, using the resource manager-specific API. The Bean Provider must ensure that the management of the isolation levels performed by the bean's code will not result in conflicting isolation level requests for a resource within a transaction.
- For entity beans using container-managed persistence, transaction isolation is managed by the data access classes that are generated by the container 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 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.

11.3.4 Enterprise beans using bean-managed transaction

This subsection describes the requirements for the Bean Provider of an enterprise bean using bean-managed transaction.

The enterprise bean using bean-managed transaction must be a Session bean.

An instance of an enterprise bean using bean-managed transaction can perform both global and local transactions. An instance must not start a global transaction until it completes all the local transactions that it has previously started. An instance that started a global transaction must complete the transaction before it starts a new global transaction or a local transaction.

To demarcate local transactions, the Bean Provider must use a resource manager-specific API (e.g. JDBC).

To demarcate global transactions, the Bean Provider must use the *bean-managed transaction*. All updates to the resource managers between the `UserTransaction.begin()` and `UserTransaction.commit`() calls are performed in a global transaction. While an instance is in a global transaction, the instance must not attempt to demarcate transaction boundaries using the resource-specific demarcation API (e.g. it must not call the `commit()` or `rollback()` method on the `java.sql.Connection` interface).

A stateful Session Bean instance may, but is not required to, commit a started global transaction or any started local transactions before a business method returns. If a global 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. If a local transaction (or multiple transactions if the instance has accessed multiple resource managers) has not been completed by the end of a business method, the resource (e.g. a JDBC connection) used by the instance automatically retains the local transaction association across multiple client calls.

The bean-managed transaction 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 instance must commit all transactions, local and global, before a business method returns.

The following example illustrates a business method that performs local transactions on two database connections.

```
public class MySessionEJB implements SessionBean {
    EJBContext ejbContext;

    public void someMethod(...) {
        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();

        // start local transaction on con1 and do some work
        stmt1.executeQuery(...);
        stmt1.executeUpdate(...);

        // start local transaction on con2 and do some work
        stmt2.executeQuery(...);

        // interleave some work on con1
        stmt1.executeUpdate(...);

        // commit local transaction on con2
        con2.commit();

        stmt1.executeUpdate(...);

        // commit local transaction on con1
        con1.commit();


        // release resource
        con1.close();
        con2.close();
    }
    ...
}
```

The following example illustrates a business method that performs a global transaction on 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/jdbcDatabase1");
        con1 = ds1.getConnection();

        stmt1 = con1.createStatement();

        // obtain con2 object and set it up for transactions
        ds2 = (javax.sql.DataSource)
                 initCtx.lookup("java:comp/env/jdbcDatabase2");
        con2 = ds2.getConnection();

        stmt2 = con2.createStatement();

        //
        // Now do a global 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 global
        // transaction.
        stmt1.executeQuery(...);
        stmt1.executeUpdate(...);
        stmt2.executeQuery(...);
        stmt2.executeUpdate(...);
        stmt1.executeUpdate(...);
        stmt2.executeUpdate(...);

        // commit the global transaction
        ut.commit();

        // release resource
        stmt1.close();
        stmt2.close();
        con1.close();
        con2.close();
    }
```

```
        ...
}
```

The following example illustrates a stateful Session Bean that retains a global 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 global transaction
          ut.begin();

          // make some updates on con1
          ds1 = (javax.sql.DataSource)
                    initCtx.lookup("java:comp/env/jdbcDatabase1");
          con1 = ds1.getConnection();
          stmt = con1.createStatement();
          stmt.executeUpdate(...);
          stmt.executeUpdate(...);

          //
          // The Container retains the global 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/jdbcDatabase2");
          con2 = ds2.getConnection();
          stmt = con2.createStatement();
          stmt.executeUpdate(...);
          stmt.executeUpdate(...);

          // The Container retains the global 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 global transaction
            ut.commit();

            // release resources
            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 global 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/jdbcDatabase");
          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 global transaction
          ut.commit();
     }
     ...
}
```

11.3.4.1 `getRollbackOnly()` and `setRollbackOnly()` method

An enterprise bean with bean-managed transactions must not use the `getRollbackOnly()` and `setRollbackOnly()` methods of the `EJBContext` interface.

There is no need for an enterprise bean with bean-managed transactions to use these methods, for these reasons:

- An enterprise bean using bean-managed transactions can obtain the status of a global transaction by using the `getStatus()` method of the `javax.transaction.UserTrasaction` interface.
- An enterprise bean using bean-managed transactions can rollback a global transaction using the `rollback()` method of the `javax.transaction.UserTrasaction` interface.

11.3.5 Enterprise beans using container-managed transaction

This subsection describes the requirements for the Bean Provider of an enterprise bean using container-managed transaction.

The enterprise bean's business methods must not use any resource-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()`.

The enterprise bean's business methods 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 that uses container-managed transaction. The business method updates two databases using JDBC™ connections. Transaction demarcation is provided by the Container 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(...);
    }
    ...
}
```

11.3.5.1 `javax.ejb.SessionSynchronization` interface

A stateful Session Bean using container-managed transaction can optionally implement the `javax.ejb.SessionSynchronization` interface. The use of the `SessionSynchronization` interface is described in Subsection 6.5.2.

11.3.5.2 `javax.ejb.EJBContext.setRollbackOnly()` method

An enterprise bean using container-managed transactions can use the `setRollbackOnly()` method of its `EJBContext` object to mark the transaction such that the transaction can never commit. Typically, a bean to marks a transaction for rollback in order 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.

11.3.5.3 `javax.ejb.EJBContext.getRollbackOnly()` method

An enterprise bean using container-managed transactions 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.

11.3.6 Declaration in deployment descriptor

The Bean Provider of a Session Bean must use the `transaction-type` element to declare whether the Session Bean is of the bean-managed or container-managed transaction type. (See [16] 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.

11.4 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 using bean-managed transactions. The Application Assembler must not define transaction attributes for an enterprise bean that is declared as using bean-managed transaction.

The Application Assembler can use the *transaction attribute* mechanism described below to manage transaction demarcation for enterprise beans using container-managed transaction.

11.4.1 Transaction attributes

A transaction attribute is a value associated with a method of an enterprise bean's remote or home interface. The transaction attribute specifies how the Container must manage transactions for a business method when a client invokes the business method via the enterprise bean home or remote interface.

The transaction attribute should be specified only for the following remote and home interface methods:

- For a session bean, the transaction attribute should be specified only for the user defined business methods in the remote interface. It should not be specified for the `remove` and the `create` methods because the delegated `ejbRemove` and `ejbCreate` methods always run in the `NotSupported` mode.

- For an entity bean, the transaction attribute should be specified for the user defined business methods in the remote interface, and for the `remove`, `create`, and `find` methods.

If the deployment descriptor specifies a transaction attribute for methods other than those listed above, the Container should ignore the attribute.

Enterprise JavaBeans defines the following values for the transaction attribute:

- `NotSupported`
- `Required`
- `Supports`
- `RequiresNew`
- `Mandatory`
- `Never`

Refer to Subsection 11.6.2 for the specification of how the value of the transaction attribute affects the transaction management performed by the Container.

Providing the transaction attributes for an enterprise bean is an optional requirement for the Application Assembler in the sense that the Application Assembler must either specify a value of the transaction attribute for **all** the methods of the remote and home interfaces of a given enterprise bean, or specify **none**.

If the Application Assembler does not specify the transaction attributes for an enterprise bean, the Deployer must perform this task.

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 global transaction. If the bean were invoked without a global transaction, the Container would not be able to send the transaction synchronization calls because the Container does not have control of the local transactions.

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 enterprise beans' remote and home interfaces. (See [16.6] 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 the enterprise beans' remote and home interfaces. 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-args` 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-param>PARAMETER_1</method-param>
     ...
     <method-param>PARAMETER_N</method-param>
</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 `RequiresNew`; all other methods of the `EmployeeRecord` bean are assigned the attribute `Requires`. All the methods of the enterprise bean `AardvarkPayroll` are assigned the attribute `Mandatory`.

```
<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>
```

11.5 Deployer's responsibilities

The Deployer is responsible for ensuring that the methods of the deployed enterprise beans with container-managed transactions 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.

11.6 Container Provider responsibilities

This section defines the responsibilities of the Container Provider.

Every client method invocation on an enterprise Bean object via the bean's remote and home interface is interposed by the Container, and every resource 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 access performed by the enterprise bean. Typically, the Container interposes only the resource factory (e.g. a JDBC data source) JNDI look up by registering container-specific implementation of the resource factory object. The resource factory object allows the Container to obtain the 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.

11.6.1 Bean-managed transactions

This subsection defines the Container's responsibilities for the transaction management of enterprise beans that use bean-managed transaction.

Note that only Session beans can be used with bean-managed transaction. A Bean Provider is not allowed to provide an Entity bean that uses bean-managed transaction.

The Container must manage client invocations to an enterprise bean instance with bean-managed transaction 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 global 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 via the `javax.ejb.EJBContext` interface. When an instance uses the `javax.transaction.UserTransaction` interface to perform a global transaction, the Container must enlist all the resources used by the instance between the `begin()` and `commit()` (or `rollback()`) methods with the global transaction. When the instance attempts to commit the transaction, the Container is responsible for the global coordination of the transaction commit[9].

In the case of a *stateful* session bean, it is possible that the business method that started a global 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.

[9] 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 resources.

If a *stateless* session bean instance starts a transaction in a business method, it must commit the transaction before the business method returns (this applies to both global and local transactions). The Container must detect the case in which a global 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.

Note that the Container typically cannot ensure that a stateless session bean instance has completed the local transactions that it started in the business method.

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 6 Container's actions for methods of beans with bean-managed transaction

Client's transaction	Global transaction currently associated with instance	Global transaction associated with the method
none	none	none
T1	none	none
none	T2	T2
T1	T2	T2

The following items each entry in the table:

- If the client request is not associated with a transaction and the instance is not associated with a transaction, the container invokes the instance with no 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

no transaction context. The container resumes the client's transaction association (T1) when the method completes.

- 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.

- 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.

The Container must allow the enterprise bean instance to perform serially several global transactions in a method.

When an instance attempts to start a global transaction using the `begin()` method of the `javax.transaction.UserTransaction` interface while the instance has not committed the previous global transaction, the Container must throw the `javax.transaction.NotSupportedException` in the `begin()` method.

Note that in addition to the above rule, the Bean Provider must not start a global transaction while there are any uncommitted local transactions. However, the Container cannot easily enforce this rule. Violation of this rule may be detected by some resource managers, but the ability to detect the violation is not required by the EJB specification. Most likely, the Container will get an exception when trying to enlist the resource with the global transaction.

The Container must throw the `java.lang.IllegalStateException` if an instance of a bean with bean-managed transactions attempts to invoke the `setRollbackOnly()` or `getRollbackOnly()` method of the `javax.ejb.EJBContext` interface.

11.6.2 Container-managed transactions

The Container is responsible for providing the transaction demarcation for the enterprise beans that the Bean Provider declared as using container-managed transactions. For these enterprise beans, the Container must demarcate transactions as specified in the deployment descriptor by the Application Assembler. (See [16] 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.

11.6.2.1 `NotSupported`

The Container must invoke an enterprise Bean method whose transaction attribute is set to `NotSupported` without a global 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 resources or other enterprise Bean objects that are invoked from the business method.

The EJB specification has no specific requirements for the transactional semantics of the `NotSupported` case. If the enterprise bean's method accesses resources, the access to each resource is typically performed using local transactions. The Container is free to use local transactions in a container and resource specific way to implement the semantics of the `NotSupported` transaction attribute. For example, the Container may wrap all access to a resource within a business method in a single local transaction, or set up the resource such that each individual resource access is a single transaction.

If the business method invokes other enterprise beans, the Container passes no transaction context with the invocation.

11.6.2.2 `Required`

The Container must invoke an enterprise Bean method whose transaction attribute is set to `Required` with a global 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 global transaction before delegating a method call to the enterprise Bean business method. The Container automatically enlists all the resources accessed by the business method with the global 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.

11.6.2.3 `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. Typically, the* Supports *transaction attribute is used instead of the* Required *attribute as a performance optimization to avoid the overhead of a global transaction for the case of a non-transactional caller. However, this optimization should be used only if the transactional semantics of the* NotSupported *case are sufficiently strong. (This would be the case when the bean uses only a single resource and when container can ensure that all accesses to the resource from the business method are done in a single local transaction.) If the transaction semantics of the* NotSupported *case are weak (e.g. when the resource manager cannot combine multiple access calls into a single local transaction), then the* Required *attribute should be used instead.*

11.6.2.4 `RequiresNew`

The Container must invoke an enterprise Bean method whose transaction attribute is set to `RequiresNew` with a new global 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 global transaction before delegating a method call to the enterprise Bean business method. The Container automatically enlists all the resources accessed by the business method with the global 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 has been completed.

11.6.2.5 `Mandatory`

The Container must invoke an enterprise Bean method whose transaction attribute is set to `Mandatory` in a client's transaction context (which is always a global transaction context). The client is required to call with a global 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.

11.6.2.6 `Never`

The Container must not invoke an enterprise Bean method whose transaction attribute is set to `Never` in a global transaction context. 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.

11.6.2.7 Transaction attribute summary

The following table provides a summary of the transaction context that the Container passes to the business method and resources 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, T2 is a transaction initiated by the Container.

Table 7 Transaction attribute summary

Transaction attribute	Client's transaction	Transaction associated with business method	Transaction associated with resources
NotSupported	none	none	local
	T1	none	local
Required	none	T2	T2
	T1	T1	T1
Supports	none	none	local
	T1	T1	T1
RequiresNew	none	T2	T2
	T1	T2	T2
Mandatory	none	**error**	N/A
	T1	T1	T1
Never	none	none	local
	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.

11.6.2.8 Handling of `setRollbackOnly()` method

If an enterprise bean instance invokes the `setRollbackOnly()` method of the `EJBContext` interface, the Container has two responsibilities:

- 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 (global or local) 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 second requirement is not redundant because while the first requirement ensures the rollback of global transactions, it does not apply to local transactions. The requirement for the Container to note that an instance has invoked setRollbackOnly() is also needed for achieving reasonable semantics of interactions between application exceptions and transactions. See Chapter 12 for exception handling.

11.6.2.9 Handling of `getUserTransaction()` method

If an instance with container-managed transactions attempts to invoke the `getUserTransaction()` method of the `EJBContext` interface, the Container must throw the `java.lang.IllegalStateException`.

11.6.2.10 `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 `beforeCompletion()` method is the last chance given to the enterprise bean instance to cause the transaction to rollback. The instance may cause the transaction to roll back by invoking the `EJBContext.setRollbackOnly()` method.

Chapter 12 Exception handling

12.1 Overview and Concepts

12.1.1 Application exceptions

An *application exception* is an exception defined in the throws clause of a method of the enterprise Bean's home and remote interface, other than the `java.rmi.RemoteException`.

Application exceptions are used by the enterprise bean business methods 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 t that the* 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 Subsection 9.1.9). These exceptions are covered by the rules on application exceptions that are defined in this chapter.

12.1.2 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.

12.2 Bean Provider's responsibilities

This section describes the view and responsibilities of the Bean Provider with respect to exception handling.

12.2.1 Application exceptions

The Bean Provider defines the application exceptions in the throws clauses of the methods of the remote and home interface. 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.

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 Subsection 9.1.9.

12.2.2 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 business method or a container callback method (e.g. `ejbLoad`).

The enterprise bean business method and container callback methods may encounter various exceptions or errors that prevent the method from successful completion. 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[10], unexpected `RuntimeException`, JVM errors, etc.

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 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`.

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 throws the `java.rmi.RemoteException` (or subclass thereof) to the client. The Bean Provider can rely on the Container to perform to 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.

[10] Note that the enterprise bean business method may attempt to recover from a RemoteException. The text in this subsection applies only to the case the business method does not wish to recover from the RemoteException.

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.

12.2.2.1 `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.

12.3 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 enterprise bean's business methods.
- Exceptions from container-invoked callbacks on the enterprise bean.
- Exceptions from management of container-managed transactions.

12.3.1 Exceptions from an enterprise 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 methods: `ejbCreate(...)`, `ejbPostCreate(...)`, `ejbRemove()`, and the `ejbFind<METHOD>` methods.

Table 8 specifies how the Container must handle the exceptions thrown by the business methods for beans with container-managed transactions. The table specifies the Container's action as a function of the condition under which the business method executes and the exception thrown by business method. The table also illustrates the exception that the client will receive and how the client can recover from the exception. (Section 12.4 describes the client's view of exceptions in detail.)

Table 8 Handling of exceptions thrown by a business method of a bean with container-managed transactions.

Method condition	Method exception	Container's action	Client's view
Bean method runs in the context of the caller's transaction [Note A].	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 [Note D].	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.

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.

[D] This case also applies when the Container invokes the business method in the context of a local transaction.

Table 9 specifies how the Container must handle the exceptions thrown by the business methods for beans with bean-managed transactions[11]. The table specifies the Container's action as a function of the condition under which the business method executes and the exception thrown by business method. The table also illustrates the exception that the client will receive and how the client can recover from the exception. (Section 12.4 describes the client's view of exceptions in detail.)

Table 9 Handling of exceptions thrown by a business method of a session with bean-managed transactions.

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 global transaction that has been started, but not yet completed, by the instance. Discard instance. Throw `RemoteException`.	Receives `RemoteException`.

12.3.2 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 `afterBegin(), beforeCompletion()` and `afterCompletion(boolean)` methods of the `SessionSynchronization` interface.

[11] Note that the EJB specification allows only Session beans to use bean-managed transactions.

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 global transaction, mark the transaction for rollback. Rollback any local transactions started by the instance.
- 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.)

12.3.3 `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 12.3.1 and 12.3.2.

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`).

12.3.4 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`).

12.3.5 Exceptions from the management of container-managed transactions

The container is responsible for starting and committing the container-managed transactions, as described in Subsection 11.6.2. 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.

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.*

12.3.6 Release of resources

When the Container is discarding an instance because of a system exception, the Container should release all the resources held by the instance.

Resources held by the instance in the context of a global transaction will be automatically released at the transaction rollback that is forced by the Container. Typically, the Container does not have to take any additional measures other than mark the transaction for rollback.

Resources accessed in the context of a local transactions should be explicitly released by the Container. This is optional for the Container because for some resource managers the Container may not be capable of releasing the resources held by the instance.

12.3.7 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` is deprecated in EJB 1.1—enterprise beans written for the EJB 1.1 specification should use the `javax.ejb.EJBException` instead.

The EJB 1.1 specification requires 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.*

12.4 Client's view of exceptions

This section describes the client's view of exceptions received from 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.

12.4.1 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 using container-managed transactions 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.

12.4.2 java.rmi.RemoteException

The client receives the `java.rmi.RemoteException` as an indication of a failure to invoke the 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 transactions 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 transactions, 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.

12.4.2.1 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 sure 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.

12.4.2.2 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.

12.4.2.3 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.

12.5 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.

12.6 Differences from EJB 1.0

The EJB 1.1 specification of exception handling preserved 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 is 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 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 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 13 Support for Distribution

13.1 Overview

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.

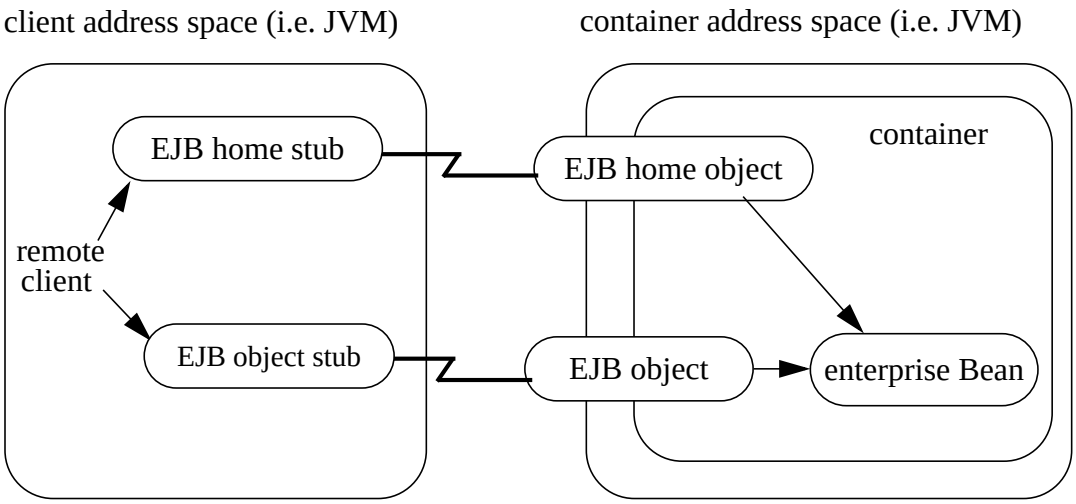
The EJB specification further constrains the Java RMI types that can be used by enterprise beans to the legal RMI-IIOP types [7]. This makes it possible for the EJB Container implementors to use RMI-IIOP as the object distribution protocol.

Note: The EJB 1.1 specification does not require Container vendors to use RMI-IIOP. A later release of the J2EE platform is likely to require a J2EE platform implementor to implement the RMI-IIOP protocol for EJB interoperability in heterogeneous server environments.

13.2 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 48 Location of EJB Client Stubs.



The communication stubs used on the client side are artifacts generated at enterprise Bean's deployment time by the EJB Container provider tools. The stubs used on the client are standard if the Container uses RMI-IIOP as the distribution protocol; the stubs are Container-specific otherwise.

13.3 Standard distribution protocol

The standard mapping of the Enterprise JavaBeans architecture to CORBA is defined in [8].

The mapping enables the following interoperability:

- A client using an ORB from one vendor can access enterprise Beans residing on an EJB Server provided by another vendor.
- Enterprise Beans in one EJB Server can access enterprise Beans in another EJB Server.
- A non-Java CORBA client can access any enterprise Bean object.

Chapter 14 Enterprise bean environment

This chapter specifies the interfaces for accessing the enterprise bean environment.

14.1 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 resources and external information. The key issue is how enterprise beans can locate the external information without the 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 14.2 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 14.3 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 14.4 defines the interfaces for obtaining a resource factory using a *resource factory reference*. An resource factory reference is a special entry in the enterprise bean's environment.

14.2 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. 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.

14.2.1 Bean Provider's responsibilities

This section describes the Bean Provider's view of the enterprise bean's environment, and defines his or her responsibilities.

14.2.1.1 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 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 > maxExemptions ||
            numberOfExemptions < minExemptions)
                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");
        ...
    }
}
```

14.2.1.2 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.

The environment entry values may be one of the following Java types: `String`, `Integer`, `Boolean`, `Double`, and `Float`.

If the Bean Provider provides a value for an environment entry, the value can be changed later by the Application Assembler or Deployer. The value must be a literal that is valid for a Java assignment to a variable of the specified type. This means, for example, that a value of an environment entry of the type `String` must be enclosed in double quotes.

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>
...
```

14.2.2 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.

14.2.3 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.

14.2.4 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.

14.3 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 should be bound to the home of the specified target enterprise bean.

14.3.1 Bean Provider's responsibilities

This subsection describes the Bean Provider's view and responsibilities with respect to EJB references.

14.3.1.1 EJB reference programming interfaces

The Bean Provider must use EJB references to locate the home interfaces of other enterprise bean as follows.

- Assign an entry in the enterprise bean's environment to the reference. (See subsection 14.3.1.2 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.

14.3.1.2 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.

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>
...
```

14.3.2 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 the same ejb-jar file, or in another ejb-jar in the same J2EE application unit as the referencing enterprise bean.

- 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.

14.3.3 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.

14.3.4 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 enterprise 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 enterprise bean.

14.4 Resource factory references

A resource is a Java object that encapsulates access to a resource manager. A resource factory is an object that is used to create resources. For example, an object that implements the `java.sql.Connection` interface is a resource that provides access to a database management system, and an object that implements the `javax.sql.DataSource` interface is a resource factory.

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 factory references*. The resource factory references are special entries in the enterprise bean's environment. The Deployer binds the resource factory references to the actual resource factories that exist in the target operational environment.

14.4.1 Bean Provider's responsibilities

This subsection describes the Bean Provider's view of locating resource factories and defines his responsibilities.

14.4.1.1 Programming interfaces for resource factory references

The Bean Provider must use resource factory references to obtain resources as follows.

- Assign an entry in the enterprise bean's environment to the resource factory reference. (See subsection 14.4.1.2 for information on how resource factory references are declared in the deployment descriptor.)

- *The EJB specification recommends, but does not require, that all resource 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.*

- Lookup the resource factory object in the enterprise bean's environment using the JNDI interface.

- Invoke the appropriate method on the resource factory method to obtain a resource. The factory method is specific to the resource type. It is possible to obtain multiple resource objects by calling the factory object multiple times.

The Bean Provider has two choices with respect to dealing with associating a principal with the resource access:

- Allow the Deployer to set up principal mapping or resource sign-on information. In this case, the enterprise bean code invokes a resource factory method that has no security-related parameters.

- Sign on to the resource from the bean code. In this case, the enterprise bean invokes the appropriate resource 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 authentication approaches is used.

We expect that the first form (i.e. letting the Deployer to set up the resource sign-on information) will be the approach used by most enterprise beans.

The following code sample illustrates obtaining a resource.

```
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 factory
          javax.sql.DataSource ds = (javax.sql.DataSource)
              initCtx.lookup("java:comp/env/jdbc/EmployeeAppDB");

          // Invoke factory to obtain a resource. The security
          // principal for the resource is not given, and therefore
          // it will be configured by the Deployer.
          java.sql.Connection con = ds.getConnection();
          ...
     }
}
```

14.4.1.2 Declaration of resource factory references in deployment descriptor

Although a resource 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 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 factory references used by an enterprise bean.

Each `resource-ref` element describes a single resource factory reference. The `resource-ref` element consists of the `description` element; and the mandatory `res-ref-name`, `res-type`, and `res-auth` elements. The `res-ref-name` element contains the name of the environment entry used in the enterprise bean's code. The `res-type` element contains the Java type of the resource factory that the enterprise bean code expects. The `res-auth` element indicates whether the enterprise bean code performs resource sign-on programmatically, or whether the Container signs on to the resource based on 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 `Bean` or `Container`.

The type declaration allows the Deployer to identify the type of the resource 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 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>
          </resource-ref>
          ...
     </session>
</enterprise-beans>
...
```

14.4.1.3 Standard resource factory types

The Bean Provider must use the `javax.sql.DataSource` resource factory type for obtaining JDBC connections, and the `javax.jms.QueueConnectionFactory` or the `javax.jms.TopicConnectionFactory` for obtaining JMS 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.

Note: A future EJB specification will add the "connector" mechanism that will allow an enterprise bean to use the API described in this section to obtain resource objects that provide access to additional back-end systems.

14.4.2 Deployer's responsibility

The Deployer uses deployment tools to bind the resource factory references to the actual resource factories configured in the target operational environment.

The Deployer must perform the following tasks for each resource factory reference declared in the deployment descriptor:

- Bind the resource factory reference to a resource factory that exists in the operational environment. The Deployer may use, for example, the JNDI `LinkRef` mechanism to create a sym-

bolic link to the actual JNDI name of the resource factory. The resource 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 resources. 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. 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 manner specific to the EJB Container and resource manager; it is beyond the scope of the current EJB specification.

14.4.3 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 factory classes.

- If the Bean Provider set the `res-auth` of a resource reference to `Bean`, the Container must allow the bean to perform explicit programmatic sign-on using the resource manager's API.

- The Container must provide tools that allow the Deployer to set up resource 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 factory reference declared by the enterprise bean, and the Container must be able to use the user/password combination for user authentication when obtaining a resource by invoking the resource 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 resources 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 resources for the enterprise beans and otherwise manage the use of resources by the Container. The pooling must be transparent to the enterprise beans.

14.4.4 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.

14.5 Deprecated `EJBContext.getEnvironment()` method

The *environment naming context* introduced in EJB 1.1 replaces the EJB 1.0 concept of *environment properties*.

An 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 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 = getProperty("foo");
          String barValue = getProperty("bar");
     }
     ...

}
```

Chapter 15 Security management

This chapter defines the EJB support for security management.

The deployment aspect of security management has changed significantly since EJB 1.0. These changes were made primarily to support ISV enterprise beans, which are usually written without the knowledge of the target security domain.

15.1 Overview

We set the following goals for the security management in the EJB architecture:

- *Shift the burden of securing the application from the application developer (i.e. from the Bean Provider) to 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 Deployer at deployment time 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 many cases, the enterprise bean's business method should not contain any security-related logic.

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 presents 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 Deployer is responsible for assigning principals or groups of principals defined in the target operational environment to the security roles defined for the enterprise beans in the deployment descriptor by the Application Assembler. The Deployer is also 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.

15.2 Bean Provider's responsibilities

This section defines the Bean Provider's perspective of the EJB architecture support for security, and defines his responsibilities.

15.2.1 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 neither programmatic nor deployment descriptor interfaces for the invoking enterprise bean to control the principal passed to the invoked enterprise bean.

The management of caller principals passed on 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. The default principal management (in the absence of other deployment instructions) is to propagate the caller principal 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.)

15.2.2 Resource access

Section 14.4 defines the protocol for accessing resources, including the requirements for security management.

15.2.3 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 15.6.8 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.

15.2.4 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.

15.2.5 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 64, Table 3 on page 74, and Table 4 on page 116.

The `getCallerIdentity()` and `isCallerInRole(Identity role)` methods are deprecated in EJB 1.1. The Bean Provider must use the `getCallerPrincipal()` and `isCallerInRole(String roleName)` methods for new enterprise beans.

An EJB 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());
}
```

15.2.5.1 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.

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 company over the Internet may be identified by an 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).

15.2.5.2 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 15.3.1), and are assigned to principals or principal groups that exist in the operational environment by the Deployer.

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 caller's
          // who have the security role "payroll"
          if (info.salary != oldInfo.salary &&
               !ejbContext.isCallerInRole("payroll")) {
                    throw new SecurityException(...);
          }
          ...
     }
     ...
}
```

15.2.5.3 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. This requirement allows the Application Assembler or Deployer to link the names of the security roles used in the code to the roles defined for an assembled application using 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.

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.

15.3 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 Applications 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 roles references using the `security-role-ref` elements, the Application Assembler must link the 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 15.3.3.

15.3.1 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. The Deployer maps the security roles to the user groups and/or user accounts defined in the operational environment.

Defining the security roles in the deployment descriptor is optional[12]l for the Application Assembler. Their omission in the deployment descriptor means that the Application Assembler chose not to pass any client authorization instructions to the Deployer, or that the instructions are passed through some other means. (For example, they may be included in a deployment manual.)

The Application Assembler is responsible for the following:

- Use a `security-role` element to define each security role.
- 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.

[12] 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>
```

15.3.2 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 the enterprise beans, including their superinterfaces. 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.

The method permissions relation is not required to associate every method of the enterprise bean's home or remote interface with a security role. This happens, for example, if none of the security roles defined in the deployment descriptor needs access to the methods. The Deployer should configure the enterprise bean's security such that all access to the methods that not associated with at least one security role is denied.

This case can happen when a generic enterprise bean that implements broad functionality is used in an application that uses only a subset of the beans' methods. For example, an application that uses only the getBalance() method of the Account bean would not need to define method permissions for the remaining methods of the Account bean, especially for those with potential security ramifications (i.e. withdrawFunds()).

The `method` element uses the `ejb-jar`, `method-name`, and `method-args` 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-param>PARAMETER_1</method-param>
      ...
      <method-param>PARAMETER_N</method-param>
</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>
...
```

15.3.3 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.

The following deployment descriptor example shows how to link of 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>
...
```

15.4 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.

15.4.1 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.

15.4.2 Assignment of security roles

The Deployer must assign the security roles defined in the `security-role` elements of the deployment descriptor to the principals and/or groups of principals (such as individual users or user groups) used for managing security in the operational environment.

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 15.3.3 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.)

15.4.3 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 `description` elements of the deployment descriptor, or in a deployment manual).

The default mode is to propagate the caller principal 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). In the absence of instructions from the Application Assembler, the Deployer should configure the enterprise beans such that this "caller propagation" mode is used when one enterprise bean calls another. This ensures that the returned value of `getCallerPrincipal()` will be the same for all the enterprise beans involved in a call chain.

15.4.4 Security management of resource access

The Deployer's responsibilities with respect to securing resource managers access are defined in subsection 14.4.2.

15.4.5 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.

15.5 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.

15.6 EJB Container Provider's responsibilities

This section describes the responsibilities of the EJB Container and Server Provider.

15.6.1 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 15.4.

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.

15.6.2 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 of 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.

15.6.3 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 access.*
- *Secure communication with remote clients (privacy, integrity, etc.).*

15.6.4 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 minimal requirement is that the EJB Container must be capable of allowing the Deployer to specify that 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.

15.6.5 Security methods in `javax.ejbEJBContext`

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 information during the execution of a business method invoked via the enterprise bean's remote or home interface.

The EJB specification does not specify the EJB Container behavior if the enterprise bean invokes the `getCallerPrincipal()` or `isCallerInRole(String roleName)` method in an improper context (e.g. from within the `ejbLoad()` method). The EJB Container can, for example, throw a `java.lang.RuntimeException`.

15.6.6 Secure access to resource managers

The EJB Container Provider is responsible for providing secure access to resource managers as described in Subsection 14.4.3.

15.6.7 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.

15.6.8 System principal

The EJB 1.1 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.

15.6.9 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 operation 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 resources 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[13]. The Container must allow multiple-deployed enterprise beans to co-exist at runtime.

15.6.10 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.

[13] The enterprise bean is installed each time using a different JNDI name.

15.7 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.

15.7.1 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.

15.7.2 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.

15.7.3 Audit trail review

If the EJB Container provides an audit trail facility, the System Administrator is responsible for its management.

Chapter 16 Deployment descriptor

This chapter defines the deployment descriptor that is part of the ejb-jar file. Section 16.1 provides an overview of the deployment descriptor. Sections 16.2 through 16.5 describe the information in the deployment descriptor from the perspective of the EJB Roles responsible for providing the information. Section 16.6 defines the deployment descriptor's XML DTD. Section 16.7 provides a complete example of a deployment descriptor of an assembled application.

16.1 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.

16.2 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.

- **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.

- **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 `primkey-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-fields` elements.

- **Environment entries**. The Bean Provider must declare all the enterprise bean's environment entries as specified in Subsection 14.2.1.

- **Resource factory references**.The Bean Provider must declare all the enterprise bean's resource factory references as specified in Subsection 14.4.1.

- **EJB references**. The Bean Provider must declare all the enterprise bean's references to the homes of other enterprise beans as specified in Subsection 14.3.1.

- **Security role references**. The Bean Provider must declare all the enterprise bean's references to security roles as specified in Subsection 15.2.5.3.

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 16.6. 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">
```

16.3 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 16.2 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 permissions 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 16.6. 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">
```

16.4 Deployer's responsibilities

16.5 Container Provider's responsibilities

16.6 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.

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 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 transactions.

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. Typically, only the ejb-jar
files that contain enterprise beans assembled into a larger applica-
tion unit will include an assembly-descriptor.

Used in: ejb-jar
-->
<!ELEMENT assembly-descriptor (security-role*, method-permission*,
        container-transaction*)>

<!--
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: container-transaction, ejb-jar, entity, env-entry, field,
method, method-permission, resource-ref, security-role, 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 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, and optional application-assembly descriptor.
-->
<!ELEMENT ejb-jar (description?, display-name?, small-icon?,
         large-icon?, enterprise-beans, assembly-descriptor?)>

<!--
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 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, primkey-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, 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 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 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: field

Example:
      <field-name>firstName</fieldName>
-->
<!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-param elements identify
a single method among multiple methods with an overloaded method name.

There are three possible styles of the method element syntax:

1. <ejb-name>EJBNAME</ejb-name><method-name>*</method-name>

   This style is used to refer to all the methods of the specified
   enterprise bean's home and remote interfaces.

2. <ejb-name>EJBNAME</ejb-name><method-name>METHOD</method-name>

   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. <ejb-name>EJBNAME</ejb-name>

   <method-name>METHOD</method-name>
   <method-param>PARAM-1</method-param>
   <method-param>PARAM-2</method-param>
         ...
   <method-param>PARAM-n</method-param>

   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. 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-param>java.lang.String</method-param>
   <method-param>java.lang.String</method-param>
</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-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>
```

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-param>java.lang.String</method-param>
      <method-param>java.lang.String</method-param>
</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-param>java.lang.String</method-param>
            <method-param>java.lang.String</method-param>
      </method>

-->
<!ELEMENT method (description?, ejb-name, method-intf?, method-name,
             method-param*)>

<!--
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
-->
<!ELEMENT method-param (#PCDATA)>

<!--
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 primkey-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 primkey-class element should specify java.lang.Object.

Used in: entity

Examples:
   <primkey-class>java.lang.String</primkey-class>
   <primkey-class>com.wombat.empl.EmployeeID</primkey-class>
   <primkey-class>java.lang.Object</primkey-class>

-->
<!ELEMENT primkey-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 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>Bean</res-auth>
         <res-auth>Container</res-auth>
-->
<!ELEMENT res-auth (#PCDATA)>

<!--
The res-ref-name element specifies the name of a resource factory ref-
erence.

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 reference name, the indication of the resource's
data source 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>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 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)>
```

16.7 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.

```
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems Inc.//DTD Enterprise
JavaBeans 1.1//EN">
<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>String</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>
            <primkey-class>com.wombat.empl.EmployeeID</primkey-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>AaardvarkPayroll</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>
            <primkey-class>com.aardvark.payroll.Accoun-
tID</primkey-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>
```

Chapter 17 Ejb-jar file

The ejb-jar file is the standard format for 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).

17.1 Overview

The ejb-jar file format is the contract between the Bean Provider and 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.

The current EJB specification does not specify the deployment descriptor support that would allow enterprise beans contained in ***multiple*** *ejb-jar files to be assembled into a larger application deployment unit.*

17.2 Deployment descriptor

The ejb-jar file must contain the deployment descriptor in the format defined in Chapter 16. The deployment descriptor must be stored with the name `META-INF/ejb-jar.xml` in the ejb-jar file.

17.3 Class files

For each enterprise bean, the ejb-jar file must include the class files of the following:

- The enterprise bean class.
- The enterprise bean home and remote interface.
- The primary key class if the bean is an entity bean.

The ejb-jar file must contain also the class files for all the classes and interfaces that the enterprise bean class, and the remote and home interfaces depend on. This includes their superclasses and superinterfaces, and the classes and interfaces used as method parameters, results, and exceptions.

17.4 Deprecated in EJB 1.1

This section describes the deployment information that was defined in EJB 1.0, and is deprecated in EJB 1.1.

17.4.1 ejb-jar Manifest

The JAR Manifest file is not used by the EJB architecture.

EJB 1.0 used the Manifest file to identify the individual enterprise beans that were included in the ejb-jar file. In EJB 1.1, the enterprise beans are identified in the deployment descriptor, so the information in the Manifest is no longer needed.

17.4.2 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 18 Runtime environment

This chapter defines the application programming interfaces (APIs) that a compliant EJB 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 Containers.

The chapter also defines the restrictions that the EJB 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.

18.1 Bean Provider's responsibilities

This section describes the view and responsibilities of the Bean Provider.

18.1.1 APIs provided by Container

The EJB Provider can rely on the EJB Container Provider to provide the following APIs:

- JDK 1.1.x or Java 2
- EJB 1.1 Standard Extension
- JDBC 2.0 Standard Extension
- JNDI 1.2 Standard Extension
- JTA 1.0 Standard Extension (the `UserTransaction` interface only)
- JavaMail 1.1 Standard Extension (for sending mail only)

The Bean Provider must take into consideration that while some Containers will provide JDK 1.1.x APIs, other Containers may provide the Java 2 (i.e. JDK 1.2) APIs. This means that the Bean Providers that want to deploy their enterprise beans in all Containers must restrict the APIs used by the enterprise beans to those that are available in JDK 1.1 and the above listed standard extensions.

18.1.2 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 Container. The restrictions apply to the implementation of the business methods. Section 18.2, 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 in 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.

To guarantee portability of the enterprise bean's implementation across all compliant EJB Containers, the Bean Provider should test the enterprise bean using a Container with the security settings defined in Tables 10 and 11. The tables define the minimal functionality that a compliant EJB Container must provide to the enterprise bean instances at runtime.

18.2 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 minimal requirement; a Container may choose to provide additional functionality that is not required by the EJB specification.

18.2.1 Java 2 based Container

A Java™ 2 based EJB Container must make the following APIs available to the enterprise bean instances at runtime:

- Java 2 APIs.
- EJB 1.1 APIs.
- JNDI 1.2
- JTA 1.0, the `UserTransaction` interface only
- JDBC™ 2.0 extension
- JavaMail 1.1, sending mail only

The following subsections describes the requirements in more detail.

18.2.1.1 Java 2 APIs requirements

The Container must provide the full set of Java 2 APIs. The Container is not allowed to subset the Java 2 APIs.

The EJB Container is allowed to make certain Java 2 functionality unavailable to the enterprise bean instances by using the Java 2 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 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 10 Java 2 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	deny
java.lang.reflect.ReflectPermission	deny

Table 10 Java 2 Security policy for a standard EJB Container

Permission name	EJB Container policy
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 Java 2.

Some Containers may allow the Deployer to grant more, or fewer, permissions to the enterprise bean instances than specified in Table 10. 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.

18.2.1.2 EJB 1.1 requirements

The container must implement the EJB 1.1 interfaces as defined in this documentation.

18.2.1.3 JNDI 1.2 requirements

At the minimum, the EJB Container must provide a JNDI 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 of resources 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 name space. However, all the instances of the same enterprise bean must be presented with the same JNDI name space.

18.2.1.4 JTA 1.0 requirements

The EJB Container must include the JTA 1.0 extension, and it must provide the `javax.transaction.UserTransaction` interface to enterprise beans with bean-managed persistence via `javax.ejb.the EJBContext` interface.

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.

18.2.1.5 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.

18.2.2 JDK™ 1.1 based Container

A JDK 1.1 based EJB Container must make the following APIs available to the enterprise bean instances at runtime:

- JDK 1.1 or higher
- EJB 1.1 APIs.
- JNDI 1.2
- JTA 1.0, the `UserTransaction` interface only
- JDBC™ 2.0 extension
- JavaMail 1.1, sending mail only

The following subsections describes the requirements in more detail.

18.2.2.1 JDK 1.1 APIs requirements

The Container must provide the full set of JDK 1.1 APIs. The Container is not allowed to subset the JDK 1.1 APIs.

The EJB Container is allowed to make certain JDK 1.1 functionality unavailable to the enterprise bean instances by using the JDK security manager 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 JDK 1.1 security manager checks that the EJB Container must allow to succeed when the check is invoked from an enterprise bean instance.

Table 11 JDK 1.1 Security manager checks for a standard EJB Container

Security manager check	EJB Container's security manager policy
checkAccept(String, int)	throw SecurityException
checkAccess(Thread)	throw SecurityException
checkAccess(ThreadGroup)	throw SecurityException
checkAwtEventQueueuAccess()	throw SecurityException
checkConnect(String, int)	allow
checkConnect(String, int, Object)	allow
checkCreateClassLoader()	throw SecurityException
checkDelete(String)	throw SecurityException
checkExec(String)	throw SecurityException
checkExit(int)	throw SecurityException
checkListen(int)	throw SecurityException
checkMemberAccess(Class, int)	throw SecurityException
checkMulticast(InetAddress)	throw SecurityException
checkMulticast(InetAddress, byte)	throw SecurityException
checkPackageAccess(String)	throw SecurityException
checkPackageDefinition(String)	throw SecurityException
checkPrintJobAccess()	allow
checkPropertiesAccess()	throw SecurityException
checkPropertyAccess(String)	throw SecurityException
checkRead(FileDescriptor)	throw SecurityException
checkRead(String)	throw SecurityException
checkRead(String, Object)	throw SecurityException
checkSecurityAccess(String)	throw SecurityException
checkSetFactory()	throw SecurityException
checkSystemClipboardAccess()	throw SecurityException

Table 11 JDK 1.1 Security manager checks for a standard EJB Container

Security manager check	EJB Container's security manager policy
checkTopLevelWindow(Object)	throw SecurityException
checkWrite(FileDescriptor)	throw SecurityException
checkWrite(String)	throw SecurityException

Some Containers may allow the Deployer to grant more, or fewer, permissions to the enterprise bean instances than specified in Table 10. 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.

18.2.2.2 EJB 1.1 requirements

The container must implement the EJB 1.1 interfaces as defined in this documentation.

18.2.2.3 JNDI 1.2 requirements

Same as defined in Subsection 18.2.1.3.

18.2.2.4 JTA 1.0 requirements

Same as defined in Subsection 18.2.1.4.

18.2.2.5 JDBC 2.0 extension requirements

Same as defined in Subsection 18.2.1.5, with the following exception: The EJB Container is not required to provide the support for the RowSet functionality.

This exception was made because the RowSet functionality requires the Java 2 Collections.

18.2.3 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 19 Responsibilities of EJB Roles

This chapter provides the summary of the responsibilities of each EJB Role.

19.1 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.

19.1.1 API requirements

The enterprise beans must meet all the API requirements defined in the individual chapters of this document.

19.1.2 Packaging requirements

The Bean Provider is responsible for packaging the enterprise beans in an ejb-jar file in the format described in Chapter 17.

The deployment descriptor must include the *structural* information described in Section 16.2.

The deployment descriptor may optionally include any of the *application assembly* information as described in Section 16.3.

19.2 Application Assembler's responsibilities

The requirements for the Application Assembler are in defined in Section 16.3.

19.3 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.

19.4 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.

19.5 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.

19.6 Client Programmer's responsibilities

The EJB client programmer writes applications that access enterprise beans via their home and remote interfaces.

Chapter 20

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 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 is 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 compliant Containers.

We intend to a tool which will help convert an EJB 1.0 deployment descriptor to the EJB 1.1 XML-based format. The `javax.ejb.deployment` package will be provided only as part of this tool.

The Javadoc specification of the EJB interface is included in a ZIP file distributed with this document.

Chapter 21 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 to IDL Mapping. OMG TC Document TC orbos/98-07-19.

[8] Enterprise JavaBeans to CORBA Mapping. *http://java.sun.com/products/ejb/docs.html.*

[9] OMG Object Transaction Service. *http://www.omg.org/corba/sectrans.htm#trans*.

Appendix A Features deferred to future releases

We plan to provide an SPI-level interface for attaching a resource manager (such as a JDBC driver) to the EJB Container as a separate Connector API.

We plan to enhance the support for Entities in the next major release (EJB 2.0). We are looking into the area of using of UML for the design and analysis of enterprise beans applications.

We plan to provide integration of EJB with JMS as part of EJB 2.0.

Appendix B Frequently asked questions

This Appendix provides the answers to a number of frequently asked questions.

B.1 Client-demarcated transactions

The EJB 1.0 specification did not explain how a client other than another enterprise bean can obtain a the `javax.transaction.UserTransaction` interface.

The EJB 1.1 specification refers to the Java Transaction API (JTA) [5] which specifies how a client obtains the `javax.transaction.UserTransaction` interface.

The following is an example of how a Java application can obtain the `javax.transaction.UserTransaction` interface.

```
//
// Obtain the JNDI name using an application-type specific
// configuration mechanism. (This example shows the use of
// system properties which is applicable to stand-alone
// applications, but the JTA specification does not prescribe
// the use of System properties for application configuration)
//
String utxPropVal = System.getProperty(“jta.UserTransaction”);

//
// Obtain the UserTransaction interface from JNDI. Note that
// the InitialContext is created using the default (no-arg)
// constructor.
//
Context ctx = new InitialContext();
UserTransaction utx = (UserTransaction)ctx.lookup(utsPropVal);

//
// Perform calls to enterprise beans in a transaction.
//
utx.begin();
... call one or more enterprise beans
utx.commit();
```

B.2 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.

B.3 Entities and relationship

The current EJB architecture does not specify how one Entity bean should store an object reference of another Entity bean. The desirable strategy is application-dependent. The enterprise bean (if the bean uses bean-managed persistence) or the Container (if the bean uses container-managed persistence) can use any of the following strategies for maintaining persistently a relationship between entities (the list is not inclusive of all possible strategies):

- Object's primary key. This is applicable if the target object's Home is known and fixed.
- Home name and object's primary key.
- Home object reference and object's primary key.
- Object's handle.

We plan to describe these strategies in more detail in a future release of the specification.

B.4 Finder methods for entities with container-managed persistence

The EJB specification does not provide a *formal* mechanism for the Bean Provider of a bean with container-managed persistence to specify the criteria for the finder methods.

The current mechanism is that Bean Provider describes the finders in a description of the Entity Bean. The current EJB specification does not provide any syntax for describing the finders.

We plan to address this issue in a future release of the specification.

B.5 JDK 1.1 or Java 2

Chapter 18 describes the issue of using JDK 1.1 versus Java 2 in detail.

In summary, the Bean Provider can produce enterprise beans that will run in both JDK 1.1 and Java 2 based Containers. The Container Provider can use either JDK 1.1 or Java 2 as the basis for the implementation of the Container.

B.6 javax.transaction.UserTransaction versus javax.jts.UserTransaction

The correct spelling is `javax.transaction.UserTransaction`.

The use of `javax.jts.UserTransaction` is deprecated in EJB 1.1.

B.7 How to obtain database connections

Section 14.4 specifies how an enterprise bean should obtain resources such as JDBC connections. The resource acquisition protocol uses resource 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 factory
          javax.sql.DataSource ds = (javax.sql.DataSource)
              initCtx.lookup("java:comp/env/jdbc/EmployeeAppDB");

          // Invoke factory to obtain a resource. The security
          // principal for the resource is not given, and therefore
          // it will be configured by the Deployer.
          java.sql.Connection con = ds.getConnection();
          ...
     }
}
```

B.8 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.

B.9 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 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 C Revision History

C.1 Changes since Release 0.8

Removed `java.ejb.BeanPermission` from the API. This file was incorrectly included in the 0.8 specification.

Renamed packages to `java.ejb` and `javax.ejb.deployment`. The Enterprise JavaBeans API is packaged as a standard extension, and standard extensions should be prefixed with `javax`. Also renamed `java.jts` to `javax.jts`.

Made clear that a container can support multiple EJB classes. We renamed the `javax.ejb.Container` to `javax.ejb.EJBHome.` Some reviewers pointed out that the use of the term "Container" for the interface that describes the life cycle operations of an EJB class as seen by a client was confusing.

Folded the factory and finder methods into the enterprise bean's **home interface**. This reduces the number of Java classes per EJB class and the number of round-trips between a client and the container required to create or find an EJB object. It also simplifies the client-view API.

Removed the PINNED mode of a Session Bean. Many reviewers considered this mode to be "dangerous" since it could prevent the container from efficiently managing its memory resources.

Clarified the life cycle of a stateless Session Bean.

Added a chapter with the specification for exception handling.

We have renamed the contract between a component and its container to **component contract**. The previously used term **container contract** confused several reviewers.

Added description of finder methods.

Modified the entity create protocol by breaking the `ejbCreate` method into two: `ejbCreate` and `ejbPostCreate`. This provides a cleaner separation of the discrete steps involved in creating an entity in a database and its associated middle-tier object.

Added more clarification to the description of the entity component protocol.

Added more information about the responsibilities of the enterprise bean provider and container provider.

Renamed `SessionSynchronization.beginTransaction()` to `SessionSynchronization.afterBegin()` to avoid confusion with `UserTransaction.begin()`.

Added the specification of isolation levels for container-managed Entity Beans.

C.2 Changes since Release 0.9

Renamed `javax.ejb.InstanceContext` to `javax.ejb.EJBContext`.

Fixed bugs in the javadoc of the `javax.ejb.EntityContext` interface.

Combined the state diagrams for non-transactional and transactional Session Beans into a single diagram.

Added the definition of the restrictions on using transaction scopes with a Session Bean (a Session Bean can be only in a single transaction at a time).

Allowed the enterprise bean's class to implement the enterprise bean's remote interface. This change was requested by reviewers to facilitate migration of existing Java code to Enterprise JavaBeans.

Removed the `javax.ejb.EJBException` from the specification, and replaced its use by the standard `java.rmi.RemoteException`. This change was necessary because of the previous change that allows the enterprise bean class to implement its remote interface.

Changed some rules regarding exception handling.

Renamed to the `javax.jts.CurrentTransaction` interface to `javax.jts.UserTransaction` to avoid confusion with the `org.omg.CosTransactions.Current` interface. The `javax.jts.UserTransaction` interface defines the subset of operations that are "safe" to use at the application-level, and can be supported by the majority of the transaction managers used by existing platforms.

Added specification for TX_BEAN_MANAGED transactions.

Made the isolation levels supplied in the deployment descriptor applicable also to Session Beans and entities with bean-managed persistence.

Renamed the `destroy()` methods to `remove()`. This change was requested by several reviewers who pointed out the potential for name space collisions in their implementations.

Added the create arguments to the `ejbPostCreate` method. This simplifies the programming of an Entity Bean that needs the create arguments in the `ejbPostCreate` method (previously, the bean would have to save these arguments in the `ejbCreate` method).

Added restrictions on the use of per-method deployment attributes.

Added `javax.ejb.EJBMetaData` to the examples, and added the generation of the class that implements this interface as a requirements for the container tools.

Added the `getRollbackOnly` method to the `javax.ejb.EJBContext` interface. This method allows an instance to test if the current transaction has been marked for rollback. The test may help the enterprise bean to avoid fruitless computation after it caught an exception.

We removed the placeholder Appendix for examples. We will provide examples on the Enterprise JavaBeans Web site rather than in this document.

C.3 Changes since Release 0.95

Allowed a container-managed field to be of any Java Serializable type.

Clarified the bean provider responsibilities for the `ejbFind<METHOD>` methods Entity Beans with container-managed persistence.

Added two rules to Subsection xxx on exception handling and transaction management. The new rules are for the TX_BEAN_MANAGED beans.

Use the `javax.rmi.PortableRemoteObject.narrow(...)` method to perform the narrow operations after a JNDI lookup in the code samples used in the specification. While some JNDI providers may return from the `lookup(...)` method the exact stub for the home interface making it possible to for the client application to use a Java cast, other providers may return a wider type that requires an explicit narrow to the home interface type. The `javax.rmi.PortableRemoteObject.narrow(...)` method is the standard Java RMI way to perform the explicit narrow operation.

Changed several deployment descriptor method names.

C.4 Changes since 1.0

This sections lists the changes since EJB 1.0.

Specified the behavior of `EJBObject.getPrimaryKey()`, `EJBMetaData.getPrimaryKeyClass()`, `EJBHome.remove(Object primaryKey,)` and `isIdentical(Object other)` for Session Beans. As Session Bean do not have client-accessible primary keys, these operations result in exceptions.

Disallowed TX_BEAN_MANAGED for Entity Beans.

Disallowed use of `SessionSynchronization` for `TX_BEAN_MANAGED` sessions.

Allowed using java.lang.String as a primary key type.

Allowed deferring the specification of the primary key class for entities with container-managed persistence to the deployment time.

Clarified that a matching ejbPostCreate is **required** for each ejbCreate.

Added requirement for hashCode and equals for the primary key class.

Deprecated the package `javax.ejb.deployment` by replacing the JavaBeans-based deployment descriptor with an XML-based deployment descriptor.

Improved the information in the deployment descriptor by clearly separating structural information from application assembly information, and by removing support for information that should be supplied by the Deployer rather than by the ejb-jar producer (i.e. ISV). The EJB 1.0 deployment descriptor mixed all this information together, making it hard for people to understand the division of responsibility for setting the various values, and it was not clear what values can be changed at application assembly and/or deployment.

Added the requirement for the Bean Provider to specify whether the enterprise bean uses a bean-managed or container-managed transaction.

Added `Never` the list of possible values of the transaction attributes to allow specification of the case in which an enterprise bean must never be called from a transactional client.

Removed the Appendix describing the `javax.transaction` package. Inclusion of this package in the EJB document is no longer needed because the JTA documentation is publicly available.

Tightened the specification of the responsibilities for transaction management.

Tighten the rules for the runtime environment that the Bean Provider can expect and the EJB Container Provider must provide. See Chapter 18.

C.5 Changes since 1.1 Draft 1

This sections lists the changes since EJB 1.1 Draft 1.

Allow use of the Java 2 `java.util.Collection` interfaces for the result of entity finder methods.

Defining the FinderException in the finder methods of the home interface is mandatory now.

Clean up of the exception specification, including minor changes from EJB 1.0 summarized in Section 12.6.

The scope of the EJB specification for managing transaction isolation levels was reduced to sessions with bean-managed transactions. The current EJB specification does not have any API for managing transaction isolation for beans using container-managed transactions (note that all Entity beans fall into this category).

Eliminated the `stateless-session` element in the XML DTD. Now the `session` element is used to describe both the stateful and stateless session beans.

Added an optional `description` element to the `method` element. The intention is to allow tools to display the description of the method.

Clarified that the enterprise bean class may have superclasses, and that the business methods and the various container callbacks can be implemented in the enterprise bean class, or in any of its superclasses.

Fixed the example that illustrates the use of handles for session objects. Serialized handles are not guaranteed to be deserializable in a different system, and therefore they cannot be emailed.

Updated the Overview chapter.

Allowed deferring the specification of the primary key class for all entities (not only for those with container-managed persistence as it was the case in Draft 1).

Allow enterprise beans to print. The Container must grant the permission to the enterprise beans to queue printer job.

The `setRollbackOnly()` and `getRollbackOnly()` methods of the `EJBContext` object must not be used by enterprise beans with bean-managed transactions. There is no need for these beans to use these methods.

C.6 Changes since 1.1 Draft 2

Fix an error in the requirement for how a Container must deal with inter-EJB invocations when both the calling and called bean are in the same JVM. The correct requirement is that the RMI semantics must be ensured, and therefore the Container must not pass non-remote objects by reference.

Clarified the requirements for serialization of the session objects.

Specified that an EJB Compliant Container may always return a null from the deprecated `getCallerIdentity()` method.

Added a section on distributed transaction scenarios involving access to the same entity from multiple clients in the same transaction.

Changed the specification of the return value type of the `ejbCreate(...)` methods for entities with container-managed persistence. The previous specification required that the `ejbCreate` methods are defined as returning void. The new requirement is that the `ejbCreate` methods be defined as returning the primary key class type. The implementation of the `ejbCreate` method should return null. This change is to allow tools, if they wish, to create an entity bean with bean-managed persistence by subclassing an original entity bean with container-managed persistence.

For compatibility with EJB 1.0, added the support for the `java.rmi.RemoteException` to be thrown from the enterprise bean class methods. This is needed to allow an EJB 1.1 Container to support enterprise beans written to the EJB 1.0 specification. The use of the `java.rmi.RemoteException` in the enterprise bean class methods is deprecated, and new applications should throw the `javax.ejb.EJBException` instead.

Removed the deprecated package `javax.ejb.deployment` from the EJB interfaces. The the deprecated package `javax.ejb.deployment` will be distributed only with the deployment descriptor conversion tool.

Updated the examples in the transaction chapter by removing the `setAutoCommit` and `setTransactionIsolation` calls. These calls are not typically done by the enterprise bean.

Added the `<method-intf>` element to allow a method element to differentiate between a method with same signature when defined in both the remote and home interfaces.

Specified the behavior of the `getUserTransaction()`, `setRollbackOnly()`, and `getRollbackOnly()` methods for the cases when the methods are invoked by beans that are not allowed to use these methods. The Container will throw the `java.lang.IllegalException` in these situations.

Specified that `PortableRemoteObject.narrow(...)` must be used by a client to convert the result of `Handle.getEJBObject()` to the remote interface type.

Required portable enterprise bean clients to use the `PortableRemoteObject.narrow(...)`.

Clarified the minimal lifetime for handles.

Clarified that the caller must have **at least one** security role (not **all**) associated with the method permission in order to be allowed to invoke the method.

Support for entities has been made mandatory for the Container Provider.

Added a section to the Exception chapter dealing with the release of resources held by the instance when the instance is being discarded because of a system exception.

Added the `res-auth` element to the deployment descriptor for the Bean Provider to indicate whether the bean code performs an explicit sign-on to a resource manager, or whether the Bean relies on the Container to perform sign-on based on the information supplied by Deployer.

Added `java.io.Serializable` as a superinterface of `javax.ejb.Handle`. The EJB 1.0 spec required that the implementation class implements the `java.io.Serializable` interface, this change expresses the requirement syntactically.

Added the interface `javax.ejb.HomeHandle` to provide support for handles for home objects.

Allowed a Session bean instance to be removed upon a timeout while the instance is in the passivated state.

Add the `javax.ejb.NoSuchEntityException` exception to the API. Added requirements for throwing the `java.rmi.NoSuchObjectExcetion` to the chapter on exceptions.

C.7 Changes since EJB 1.1 Draft 3

Replaced the support for environment properties with the JNDI-based environment entries. The EJB 1.0 style of environment properties access is deprecated in EJB 1.1.

Removed the `finalize()` method from the state diagrams. Specified that an enterprise bean must not define the `finalize()` method in the enterprise bean class. This is because it cannot be guaranteed that the method is called at all in some Container implementations.

Made clear that the result of comparing two object reference using the Java "==" operator or the `equals()` method is undefined.

Added Tables 2, 3, and 4 that specify which operations are allowed in the enterprise bean methods.

Clarified what “proper transaction context” means in the Chapter on entities.

Flattened the DTD hierarchy by removing the elements that grouped entries of the same type.

Relaxed the rules for the primary key class. An entity with bean-managed persistence can use any RMI-IIOP Value Type as its primary key type; the primary key type of an entity with container-managed persistence is more constrained.

Added the `isStatelessSession()` method to the `EJBMetaData` interface.

Updated the chapter in distribution to simply reference RMI-IIOP. The original chapter had been written before RMI-IIOP was completed.