Sun Microsystems

Enterprise JavaBeans™

This is the specification of the Enterprise JavaBeans™ 1.0 architecture. The Enterprise JavaBeans architecture is a component architecture for the development and deployment of object-oriented distributed enterprise-level 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 specification to:

ejb-spec-comments@sun.com

Please send product and business questions to:

ejb-marketing@sun.com

Copyright © 1998 by Sun Microsystems Inc.
901 San Antonio Road, Palo Alto, CA 94303
All rights reserved.

Version 1.0

Vlada Matena & Mark Hapner

March 21, 1998 8:54 am

RESTRICTED RIGHTS: Use, duplication or disclosure by the government is subject to the restrictions as set forth in subparagraph (c) (1) (ii) of the Rights in Technical Data and Computer Software Clause as DFARS 252.227-7013 and FAR 52.227-19.

Sun, Sun Microsystems, the Sun logo, Java, and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the Unites States and other countries.

THIS PUBLICATION IS PROVIDED “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MECHANTABILITY, FITNESS FOR A PARTICULAR USE, OR NON-INFRINGEMENT.

THIS PUBLICATION COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS. CHANGES ARE PERIODICALLY ADDED TO THE INFORMATION HEREIN; THESE CHANGES WILL BE INCORPORATED IN NEW EDITIONS OF THE PUBLICATION. SUN MICROSYSTEMS, INC., MAY MAKE NEW IMPROVEMENTS AND/OR CHANGES IN THE PRODUCT(S) AND/OR THE PROGRAM(S) DESCRIBED IN THIS PUBLICATION AT ANY TIME.

1. Introduction 6

1.1 Target audience 6
1.2 Acknowledgments 6
1.3 Organization 6

2. Goals 7

2.1 Overall goals 7
2.2 Goals for Release 1.0 7

3. Roles and scenarios 9

3.1 Roles 9
3.2 Scenario 1: Development, deployment, assembly 11

4. Fundamentals 12

4.1 Enterprise Beans as components 12
4.2 Enterprise JavaBeans contracts 13
4.3 Session and entity objects 15
4.4 Standard CORBA mapping 16

5. Client view of a session Bean 18

5.1 Overview 18
5.2 EJB container 18
5.3 Home interface 19
5.4 EJB object 20
5.5 Session object identity 21
5.6 Client's view of session Bean's life cycle 21
5.7 Creating and using a session Bean 22

6. Session Bean component contract 24

6.1 Overview 24
6.2 Goals 24
6.3 A container's management of its working set 24
6.4 Conversational state 25
6.5 The protocol between a session Bean and its container 26
6.6 STATEFUL session Bean state diagram 29
6.7 Sequence diagrams for a STATEFUL session Bean 31
6.8 Stateless session Beans 36
6.9 Sequence diagrams for a STATELESS session Bean 38
6.10 The responsibilities of the enterprise Bean provider 40
6.11 The responsibilities of the container provider 42

7. Example session scenario 45

7.1 Overview 45
7.2 Inheritance relationship 46

8. Client view of an entity 49

8.1 Overview 49

8.2 EJB container 50
8.3 Enterprise Bean's home interface 51
8.4 Entity EJB object life cycle 53
8.5 Primary key and object identity. 55
8.6 Enterprise Bean's remote interface 56
8.7 Enterprise Bean's handle 57

9. Entity Bean component contract 58

9.1 The runtime execution model 58
9.2 Entity persistence. 59
9.3 Instance life cycle. 61
9.4 The entity Bean component contract. 63
9.5 Concurrent access from multiple transactions. 68
9.6 Non-reentrant and re-entrant instances 69
9.7 The responsibilities of the enterprise Bean provider. 70
9.8 The responsibilities of the container provider 73
9.9 Miscellaneous. 74
9.10 Container-managed entity Beans. 75
9.11 Sequence diagrams 78

10. Example entity scenario. 91

10.1 Overview 91
10.2 Inheritance relationship 92

11. Support for transactions 95

11.1 Transaction model 95
11.2 Relationship to JTS. 95
11.3 Scenarios 96
11.4 Declarative transaction management 100
11.5 TX_BEAN_MANAGED transactions 103
11.6 Transaction isolation levels. 104
11.7 Deployment descriptor restrictions 105
11.8 Transaction management and exceptions 106

12. Exception handling 107

12.1 Client's view of exceptions. 107
12.2 Rules for the enterprise Bean developer. 108
12.3 Rules for the container provider 108

13. Support for distribution 111

13.1 Overview 111
13.2 Client-side objects 111
13.3 Interoperability via network protocol. 111

14. Support for security 113

14.1 Package java.security 113
14.2 Security-related methods in EJBContext. 113
14.3 Security-related deployment descriptor properties 114

15. Ejb-jar file 116

15.1 ejb-jar file 116
15.2 Deployment descriptor 116
15.3 ejb-jar Manifest. 116

16. Enterprise Bean provider responsibilities 117

16.1 Classes and interfaces 117
16.2 Environment properties 117
16.3 Deployment descriptor 117
16.4 Programming restrictions 117
16.5 Component packaging responsibilities. 118

17. Container provider responsibilities 119

17.1 Enterprise Bean deployment tools. 119
17.2 Runtime infrastructure 120
17.3 Runtime management tools 120
17.4 Evolution management tools 120

18. Enterprise JavaBeans API Reference. 121

19. Related documents 165

Appendix A. Features deferred to future releases 166

Appendix B. package javax.jts. 167

Appendix C. Revision history 179

C.1 Changes since Release 0.8 179
C.2 Changes since Release 0.9 179
C.3 Changes since Release 0.95 180

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 provide support for Enterprise JavaBeans in their products.

Many concepts described in this document are system-level issues that are transparent to the Enterprise JavaBeans application programmer. Since the main goal of Enterprise JavaBeans is to hide these complex system level issues from the application programmer, we plan to provide a separate Enterprise JavaBean programmer's primer.

1.2 Acknowledgments

Rick Cattell, Shel Finkelstein, Graham Hamilton, Li Gong, Rohit Garg, Susan Cheung, Sanjeev Krishnan, Anil Vijendran, and Larry Cable have provided invaluable input to the design of Enterprise JavaBeans.

Enterprise JavaBeans 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.

1.3 Organization

Chapter 2, "Goals"discusses the advantages of Enterprise JavaBeans. Chapter 3, "Roles and Scenarios" discusses the responsibilities of Bean providers, application assemblers, application deployers, server providers, container providers, and system administrators with respect to Enterprise JavaBeans. 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; the client view, component contract, and an example scenario are presented in each of the chapters, respectively.

Chapters 11 through 15 discuss transactions, exceptions, distribution, security, and packaging, respectively, in the context of Enterprise JavaBeans.

Chapters 16 and 17 summarize enterprise Bean provider responsibilities and container provider responsibilities, respectively.

Chapter 18 is the Enterprise JavaBeans API Reference.

Chapter 19 provides a list of related documents and an Enterprise JavaBeans specification change history.

2 Goals

2.1 Overall goals

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

- Enterprise JavaBeans will be the standard component architecture for building distributed object-oriented business applications in the Java programming language. Enterprise JavaBeans will make it possible to build distributed applications by combining components developed using tools from different vendors.
- Enterprise JavaBeans 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. However, an expert-level programmer will be allowed to gain direct access to the 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 CORBA.

2.2 Goals for Release 1.0

In Release 1.0, we want to focus on the following aspects:

- Define the distinct "roles" that are assumed by the component architecture.
- Define the client's 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.

3 Roles and scenarios

3.1 Roles

The Enterprise JavaBeans architecture defines six distinct roles in the application development and deployment workflow. Each role may be performed by a different party. Enterprise JavaBeans specifies the contracts that ensure that the product of each role is compatible with the product of the other roles.

In some scenarios, a single party may perform several roles. For example, the container provider and the EJB server provider may be the same vendor. Or a single programmer may perform the role of the enterprise Bean provider and the role of the application assembler.

The following sections define the six EJB roles.

3.1.1 Enterprise Bean provider

An enterprise Bean provider is typically an application domain expert. An enterprise Bean provider develops reusable components called enterprise Beans. An enterprise Bean implements a business task, or a business entity.

An enterprise Bean provider is not an expert at system-level programming. Therefore, an enterprise Bean provider usually does not program transactions, concurrency, security, distribution and other services into the enterprise Beans. An enterprise Bean provider relies on an EJB container provider for these services.

The output of an enterprise Bean provider is an ejb-jar file that contains enterprise Beans. Each Bean includes its classes for the Java programming language, its remote and home interfaces, its deployment descriptor, and its environment properties. The enterprise Beans must conform to the Enterprise JavaBeans component contract to ensure that they can be installed into any compliant EJB container.

3.1.2 Application assembler

An application assembler is a domain expert who composes applications that use enterprise Beans. The application assembler works with 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.

The output of the application assembler can be new enterprise Beans, or applications that are not enterprise Beans (for example, servlets, applets, or scripts). The assembler may also write a GUI for the applications.

The application assembly can be done before or after the deployment of the enterprise Beans into an operational environment.

3.1.3 Deployer

A deployer is an expert at a specific operational environment, and is responsible for the deployment of enterprise Beans and their containers. A deployer typically uses tools

provided by the container provider to adapt enterprise Beans to a specific operational environment.

For example, a deployer is responsible for mapping the security roles assumed by the enterprise Beans to those required by the organization that will be using the enterprise Beans. A deployer typically reads the attribute settings in the enterprise Beans' deployment descriptors and modifies the values of the enterprise Beans' environment properties.

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

An 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.

Typically, the EJB server provider will provide a container that implements the EJB *session container*[4.2.2] contract, and may also provide an *entity container*[4.2.2] for one or more data sources supported on the EJB server.

An EJB server provider will typically publish its lower-level interfaces to allow third parties to develop containers. These interfaces are not currently specified by Enterprise JavaBeans and are vendor specific.

A later release of Enterprise JavaBeans may standardize the interfaces between a container and an EJB server.

3.1.5 EJB container provider

The expertise of a 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 system. 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 enterprise 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 may be an expert in the area of providing access to an enterprise's existing data sources or packaged application systems.

A container provider typically provides support for versioning of 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.

Enterprise JavaBeans defines the component contract that must be supported by every compliant EJB container. Enterprise JavaBeans allows container vendors to develop specialized containers that extend this contract. Examples of specialized containers include a container that supports an application-domain specific framework, a container that bridges the EJB environment with an existing application system (such a container allows modeling of the existing applications as enterprise Beans), a container that implements an Object/Relational mapping, or a container that is built on top of an object-oriented database system.

3.1.6 System administrator

The role of a system administrator is to oversee the well-being of a running system. The system administrator typically uses runtime monitoring and management tools provided by the EJB server and container providers to accomplish this task.

3.2 Scenario 1: Development, deployment, assembly

Wombat Inc. is an enterprise Bean provider that specializes in the development of software components for the banking sector. Wombat Inc. has developed the *AccountBean* and *TellerBean* enterprise Beans, and packages them in an *ejb-jar* file.

Wombat sells the enterprise Beans to banks that may use containers and EJB servers from multiple vendors. One of the banks uses a container from the Acme Corporation. Acme's tools that are part of Acme's container product facilitate the deployment of enterprise Beans from any provider, including Wombat Inc. The deployment process results in generating additional classes used internally by the Acme container. The additional classes allow the Acme container to manage enterprise Bean objects at runtime, as defined by the EJB component contract.

Since the *AccountBean* and *TellerBean* enterprise Beans by themselves are not a complete application, the bank MIS department may use Acme's tools to assemble the *AccountBean* and *TellerBean* enterprise Beans with other enterprise Beans (possibly from another vendor) and possibly with some non-EJB existing software, into a complete application. The MIS department takes on both the EJB deployer and application assembler roles.

4 Fundamentals

This chapter defines the scope 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's instances are created and managed at runtime by a container.
- An enterprise Bean can be customized at deployment time by editing its environment properties.
- Various metadata, such as a transaction mode and security attributes, are separated out from the enterprise Bean class. This allows the metadata to be manipulated using the container's tools at design and deployment time.
- Client access is mediated by the container and the EJB server on which the enterprise Bean is deployed.
- If an enterprise Bean uses only the standard container 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 a composite application without requiring source code changes or recompilation of the enterprise Bean.
- A client's view of an enterprise Bean is defined by the Bean developer. Its view is unaffected by the container and server the Bean is deployed in. This ensures that both Beans and their 100% Pure Java™ clients are write-once-run-anywhere.

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.

- A persistent entity object that is shared among multiple clients.

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 describes the Enterprise JavaBeans Release 1.0 contracts.

4.2.1 Client's view contract

This is a contract between a client and a container. The client's view contract provides a uniform development model for applications using enterprise Beans as components. This uniform model enables using higher level development tools, and will allow greater reuse of components.

Both the enterprise Bean provider and the container provider have obligations to fulfill the contract. This contract includes:

- Object identity.
- Method invocation.
- Home interface.

A client expects that an enterprise Bean object has a unique identifier. The container provider is responsible for generating a unique identifier for each session EJB object. For entity enterprise Beans (See Subsection 4.3.2), the Bean provider is responsible for supplying a unique primary key that the container embeds into the EJB object's identifier. The Bean provider supplies the primary key at EJB object creation time, and then uses the primary key of the EJB object at object activation and/or load time.

A client locates an enterprise Bean home interface through the standard Java Naming and Directory Interface™ (JNDI). Within a home, the primary key is used to identify each EJB object.

An enterprise Bean and its container cooperate to implement the create, find, and remove operations callable by clients.

An enterprise Bean provider defines a remote interface that defines the business methods callable by a client. The enterprise Bean provider is also responsible for writing the implementation of the business methods in the enterprise Bean class. The container is responsible for allowing the clients to invoke an enterprise Bean through its associated remote interface. The container delegates the invocation of a business method to its implementation in the enterprise Bean class.

An enterprise Bean provider is responsible for supplying an enterprise Bean's *home interface*. The enterprise Bean's home interface extends the *javax.ejb.EJBHome* inter-

face. A home interface defines zero or more *create(...)* methods, one for each way to create an EJB object. A home interface for entity Beans defines zero or more *finder* methods, one for each way to lookup an EJB object, or a collection of EJB objects of a particular type.

The enterprise Bean provider is responsible for the implementation of the *ejbCreate(...)* methods in the enterprise Bean class, whose signature must match those of the *create(...)* methods defined in the Bean's home interface. The container is responsible for delegating a client-invoked *create(...)* method to the matching *ejbCreate(...)* method on an enterprise Bean instance.

The enterprise entity Bean provider is responsible for the implementation of the *ejbFind<METHOD>(...)* methods in the enterprise Bean class, whose signature must match those of the *find<METHOD>(...)* finder methods defined in the Bean's home interface. The container is responsible for delegating a client-invoked *find<METHOD>(...)* method to the matching *ejbFind<METHOD>(...)* method on an enterprise Bean instance.

4.2.2 Component contract

This is a contract between an enterprise Bean and its container. This contract includes:

- An enterprise Bean class instance's view of its life cycle. For a session enterprise Bean, this includes the state management callbacks defined by the *javax.ejb.SessionBean* and *javax.ejb.SessionSynchronization* interfaces. For an entity enterprise Bean, this includes the state management callbacks defined by the *javax.ejb.EntityBean* interface. The container invokes the callback methods defined by these interfaces at the appropriate times to notify the instance of the important events in its life cycle.
- The *javax.ejb.SessionContext* interface that a container passes to a session enterprise Bean instance at instance creation. The instance uses the *SessionContext* interface to obtain various information and services from its container. Similarly, an entity instance uses the *javax.ejb.EntityContext* interface to communicate with its container.
- The environment *java.util.Properties* that a container makes available to an enterprise Bean.
- A list of services that every container must provide for its enterprise Beans.

4.2.3 Ejb-jar file

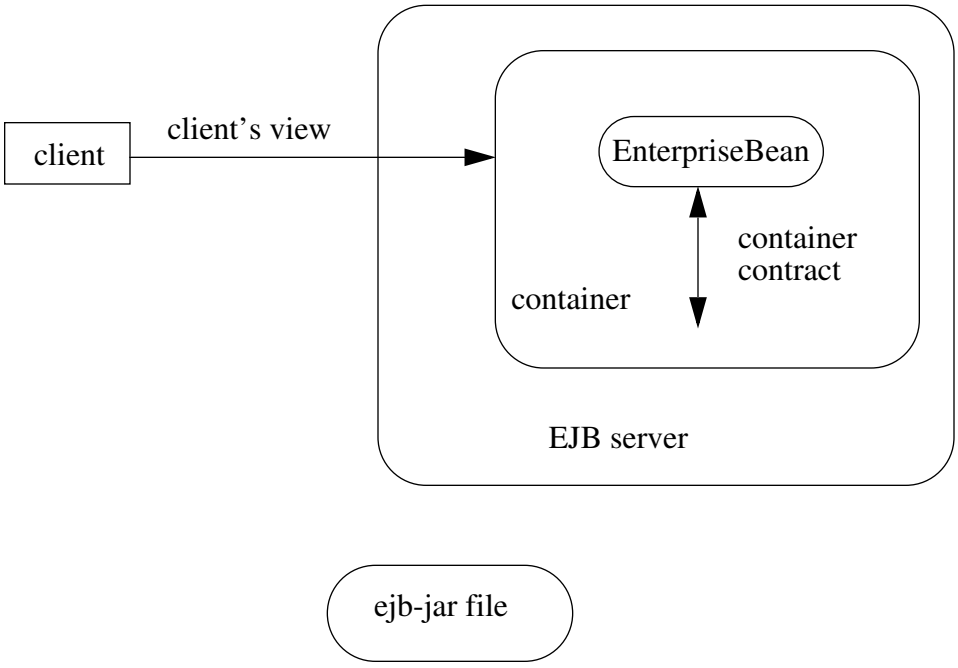
An ejb-jar file is a standard format used by EJB tools for packaging enterprise Beans with their declarative deployment information. All EJB tools must support ejb-jar files.

The ejb-jar contract includes:

- JAR file manifest entries that describe the contents of the ejb-jar file.
- Java class files for the enterprise Beans.

- Enterprise Bean deployment descriptors. A deployment descriptor includes the declarative attributes associated with an enterprise Bean. The attributes instruct the container how to manage the enterprise Bean objects.
- Enterprise Bean environment properties that the enterprise Bean requires at runtime.

The following figure illustrates the Enterprise JavaBeans contracts that are defined in Release 1.0.



Note that while the figure illustrates only a remote client running outside of the container, the client-side API is also applicable to clients who themselves are enterprise Beans installed in an EJB container.

4.3 Session and entity objects

Enterprise JavaBeans 1.0 defines two types of enterprise Beans:

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

Support for session objects is mandatory for an EJB 1.0 compliant container. Support for entity objects is optional for an EJB 1.0 compliant container, but it will become mandatory for EJB 2.0 compliant containers.

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 server crashes. The client has to re-establish a new session object to continue computation.

A typical EJB server and container provide a scalable runtime environment to execute a large number of session objects concurrently.

4.3.2 Entity objects

A typical entity object has the following characteristics:

- Represents data in the database.
- Is transactional.
- Allows shared access from multiple users.
- Can be long-lived (lives as long as the data in the database).
- Survives crashes of the EJB server. A crash is transparent to the client.

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

4.4 Standard CORBA mapping

To ensure interoperability for multi-vendor EJB environments, we define a standard mapping of the Enterprise JavaBeans client's view contract to CORBA.

The mapping to CORBA covers:

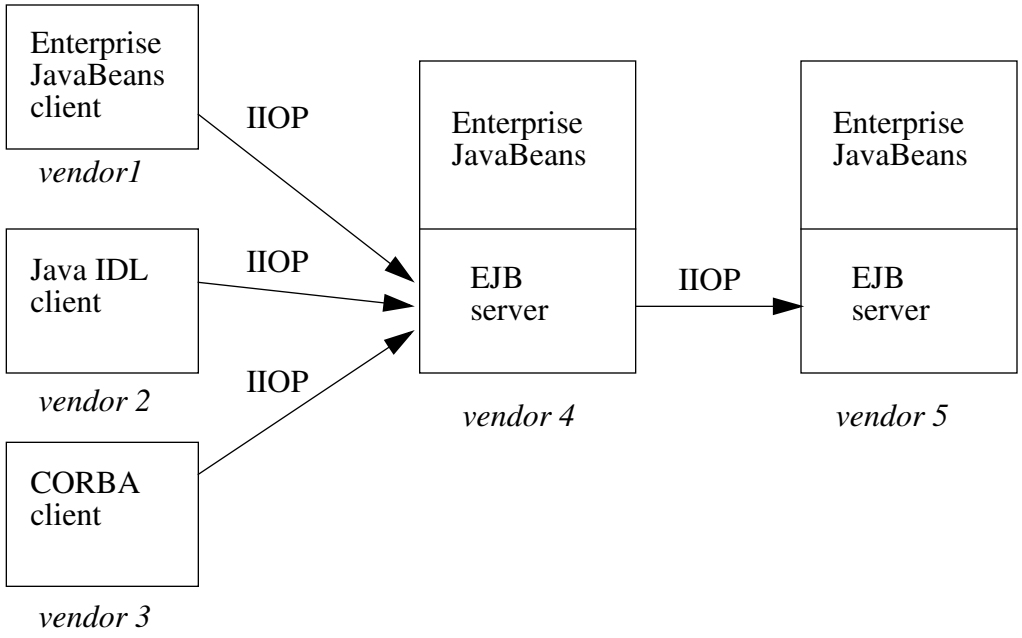
1. Mapping of the EJB client interfaces to CORBA IDL.
2. Propagation of transaction context.
3. Propagation of security context.

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

The Enterprise JavaBeans to CORBA mapping relies 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 [6].

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



5 Client view of a session Bean

This chapter describes the client's view of a session enterprise Bean. The session Bean itself implements the Bean's business logic. All the functionality for remote access, security, concurrency, transactions, etc. is provided by the Bean's container.

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's 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.

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's 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

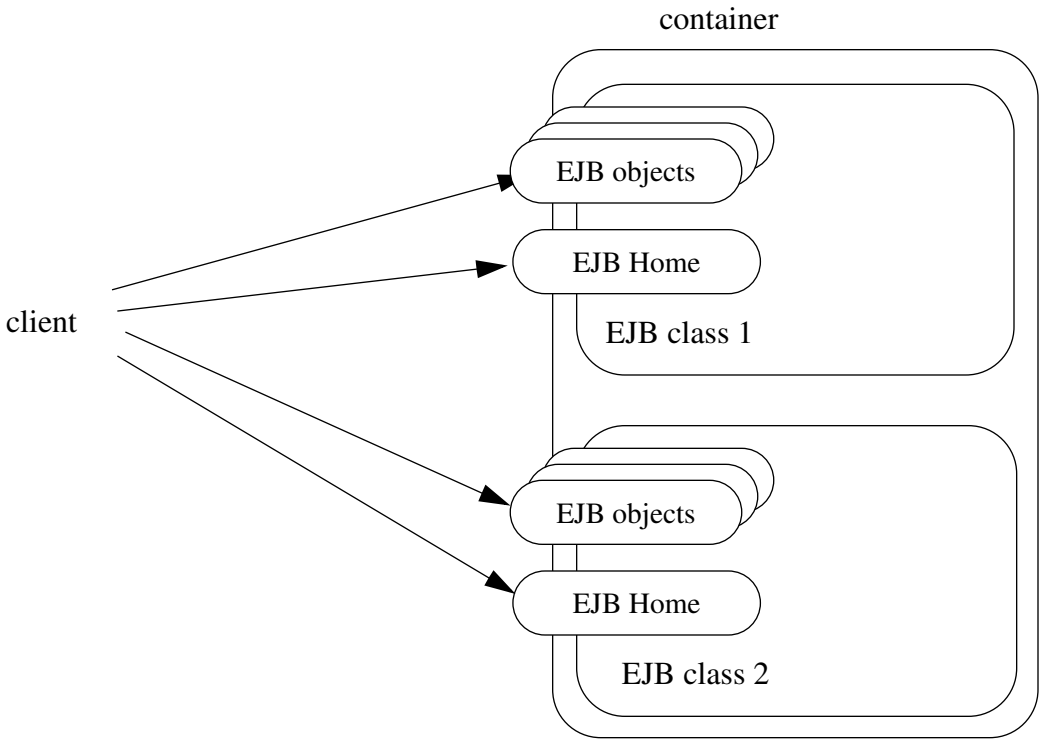
A client locates an enterprise Bean's home interface using JNDI. For example, a container for *Cart* EJB objects can be located using the following code segment:

```
Context initialContext = new InitialContext();
CartHome cartHome = 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.

5.2.2 What a container provides

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



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.

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

The *javax.ejb.EJBHome* interface defines several methods that allow a client to remove an EJB object. In addition, a client may remove an EJB object using the *remove()* method on the *javax.ejb.EJBObject* interface.

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 container.
  - 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

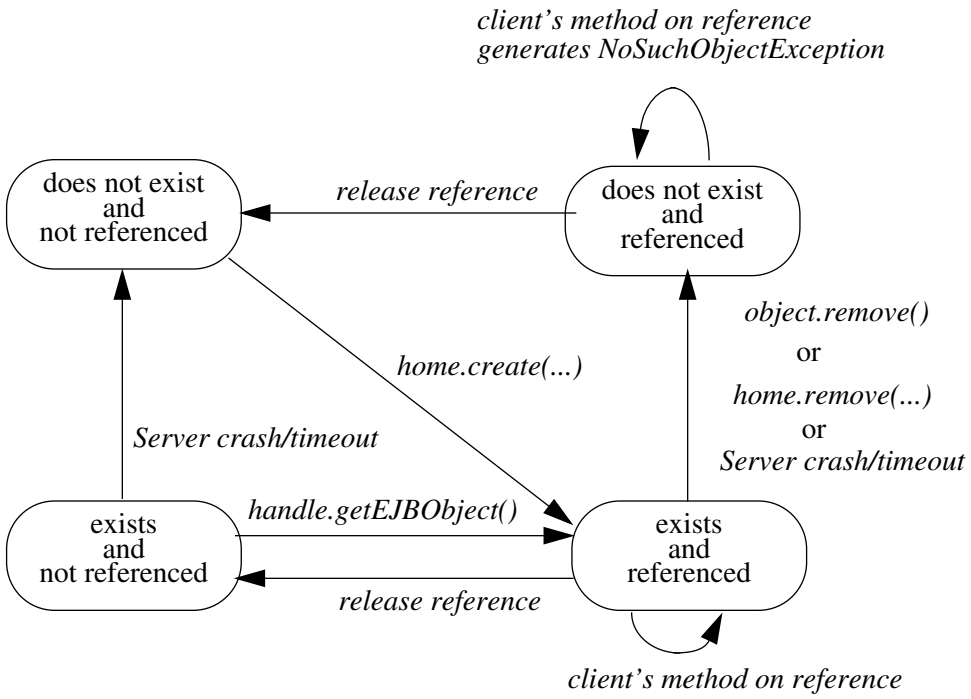
Session objects are intended to be private resources used only by the client that created them. For this reason, session EJB objects, from the clients perspective, appear anonymous. In contrast to entity EJB objects which expose their identity as a primary key, session objects hide their identity.

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).

5.6 Client's view of session Bean's life cycle

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



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:

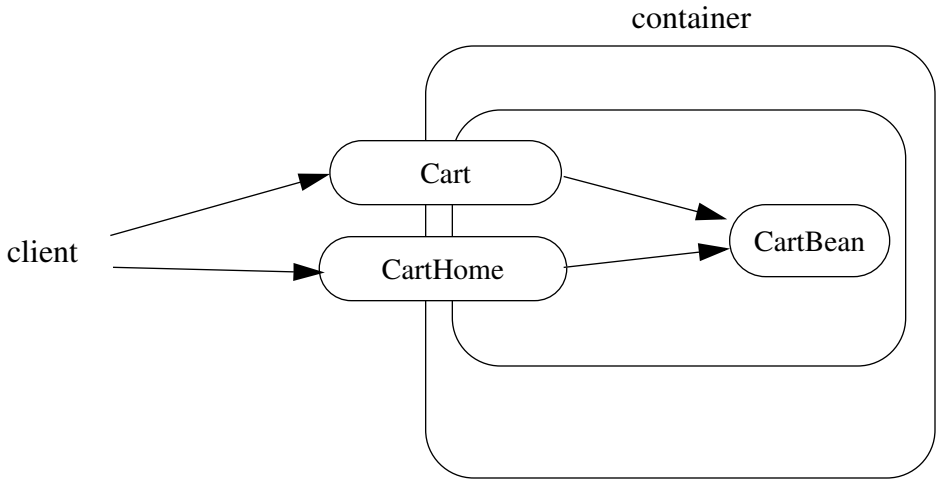
- Invoke application methods on the object through the session Bean's remote interface.

- 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:



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.

If the client wishes to start his shopping session on his work machine and later complete this session from his home machine, this can be done 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 = 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 home so we serialize a handle to this cart session and mail it home:

```
Handle cartHandle = cart.getHandle();
serialize cartHandle, attach to message and mail it home...
```

Finally we deserialize the handle at home and purchase the content of the shopping cart:

```
Handle cartHandle = deserialize from mail attachment...
Cart cart = (Cart) cartHandle.getEJBObject();
cart.purchase();
cart.remove();
```

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 *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 session Bean 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 the Bean implementation scales to 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 reachable from the session Bean's fields.

The transitive closure of a session Bean instance is defined in terms of the standard Serialization protocol for the Java programming language —the fields that would be stored by serializing the enterprise Bean instance are considered part of the enterprise Bean state.

In advanced cases, a session Bean's conversational state may contain open resources. Examples of this are: open files, open sockets, and open database cursors. It is not possible for a container to retain 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 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 enterprise Bean developer must ensure that the instance's state is serializable after *ejbPassivate* completes. The container may destroy an instance if the instance is not serializable after *ejbPassivate*.

An instance may hold EJB object references to other EJB objects (sessions or entities). When the container passivates the instance after *ejbPassivate*, it must store the EJB object references with the passivated instance, and reconstruct these object references when it loads the instance's state before *ejbActivate*.

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[1]. 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.

---

1. This is to allow containers that use specialized JVM to swap out an instance's state without performing the actual Serialization protocol for the Java programming language on the instance.

If a rollback could result in a session Bean's conversational state being inconsistent with the state of the underlying database, the Bean 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.

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. Since most session Beans don't have any database or resource state to clean up, the implementation of this method is typically left empty.

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 *getHome* method returns the home interface for the instance's EJB class.
- The *getEnvironment* method returns the environment properties list that the Bean was deployed with.
- The *getCallerIdentity* method returns the identity of the current 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.
- The *getRollbackOnly* method allows the instance to test if the current transaction has been marked for rollback.

- The *getUserTransaction* method returns the *javax.jts.UserTransaction* interface that the Bean can use for explicit transaction demarcation[1].

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. At this point, the instance is already in the transaction and 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. This is the time when the instance should write out any database updates it has cached. The instance can cause the transaction to rollback 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 the transaction committed; a status of *false* indicates a rollback occurred. Since a session instance's conversational state is not transactional, it may need to manually reset its state if a rollback occurred.

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 implementation 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 Bean's container.

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.

1.The container makes the *UserTransaction* interface available only to the Beans deployed with the *TX_BEAN_MANAGED* transaction attribute. The *getUserTransaction* interface will fail if invoked by a Bean that is deployed with a different value of the transaction attributes.

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 personalize 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 method calls that a container serializes includes those delivered via an instance's EJB object as well as the service callbacks made by the container itself (for example, the container must not invoke the *ejbPassivate* method on an instance while the instance executes a business method).

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

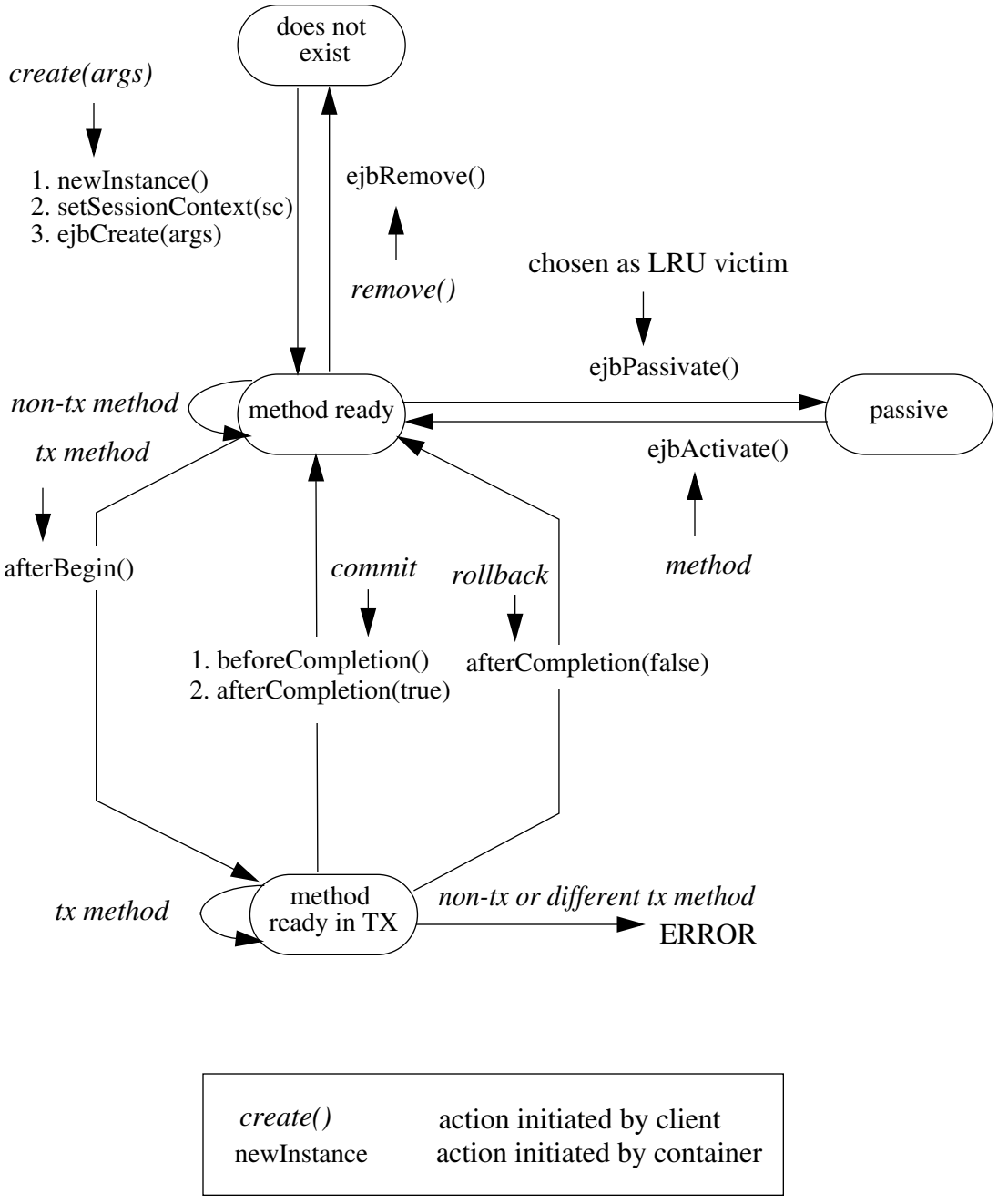
A session Bean's *afterBegin* and *beforeCompletion* methods are always called with the proper transaction context (if the Bean is transactional).

A session Bean's *newInstance, setSessionContext*, *ejbCreate*, *ejbRemove*, *ejbPassivate*, *finalize()*, *ejbActivate,* and *afterCompletion* methods are always called without a transaction. So, for example, it would usually be wrong to make database updates within a session Bean's *ejbCreate* or *ejbRemove* method.

A session Bean's deployment descriptor determines whether or not its business methods are called with a transaction.

6.6 STATEFUL session Bean state diagram

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



The following is a walk-through of the life cycle of a STATEFUL transactional session Bean instance:

- A session Bean's life starts when a client invokes a c*reate(...)* 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()* followed by *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 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. It is an error if a client attempts to invoke a method on the Bean and the deployment descriptor for the method would require that the container invokes 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, prior to actually committing the transaction, the transaction service notifies the container, and the container issues *beforeCompletion* on the instance. At this time, 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, transaction rollback had been requested, 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 (not in a transaction).

- 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 result in throwing the *java.rmi.NoSuchObjectException* (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 result in the container's throwing the *javax.ejb.RemoveException* to the client.

6.6.1 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 Sequence diagrams for a STATEFUL session Bean

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

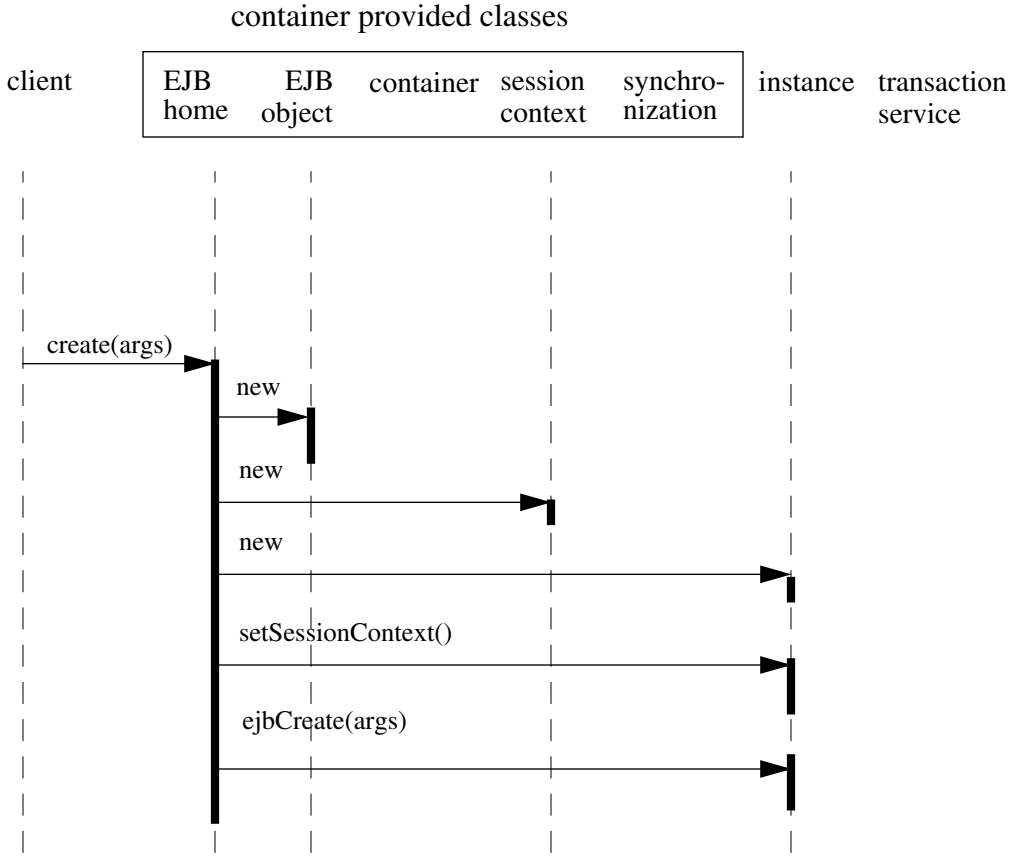
6.7.1 Notes

The sequence 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 a prescriptive one.

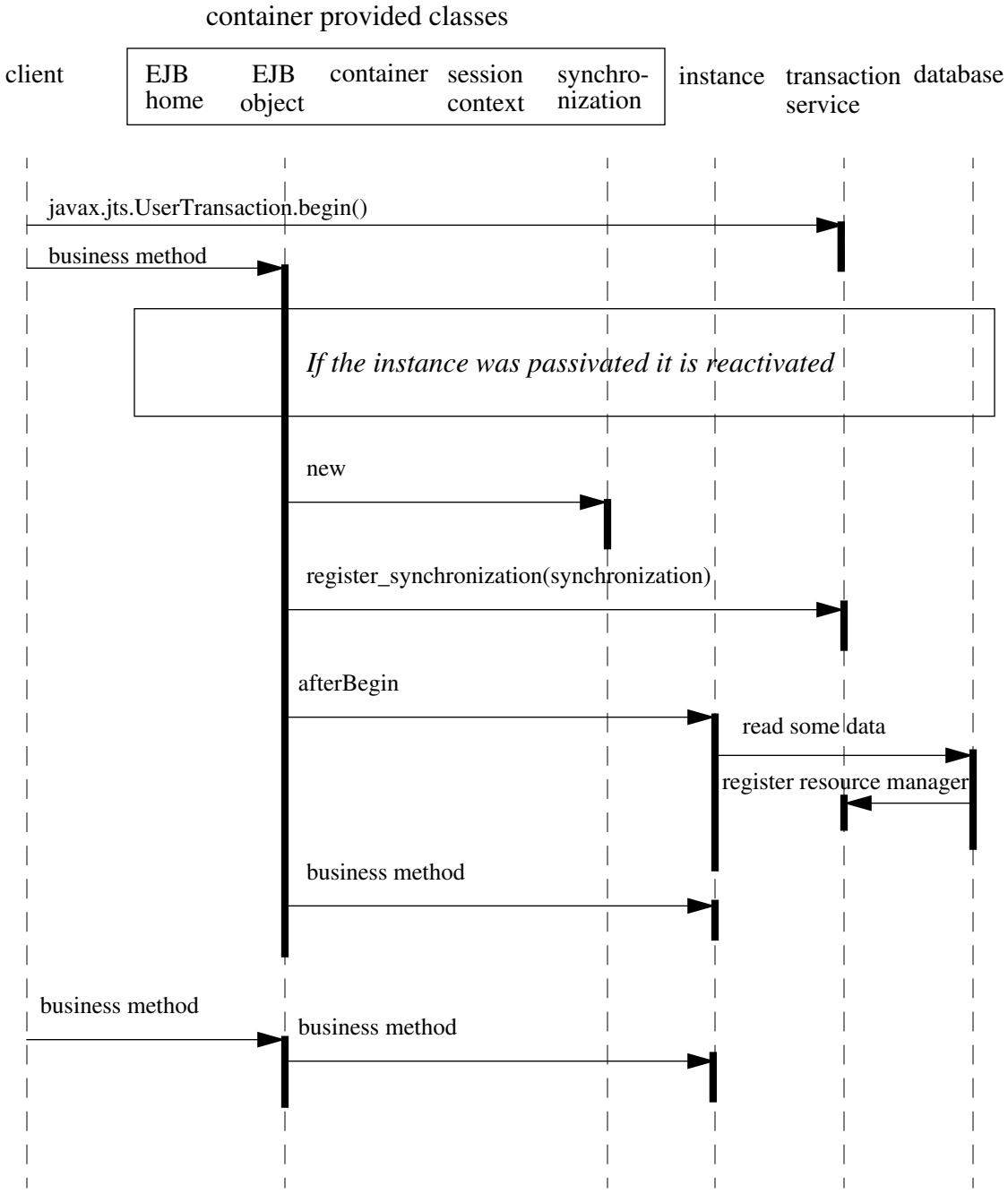
6.7.2 Creating a session object

The following diagram illustrates the creation of a transactional session enterprise Bean.



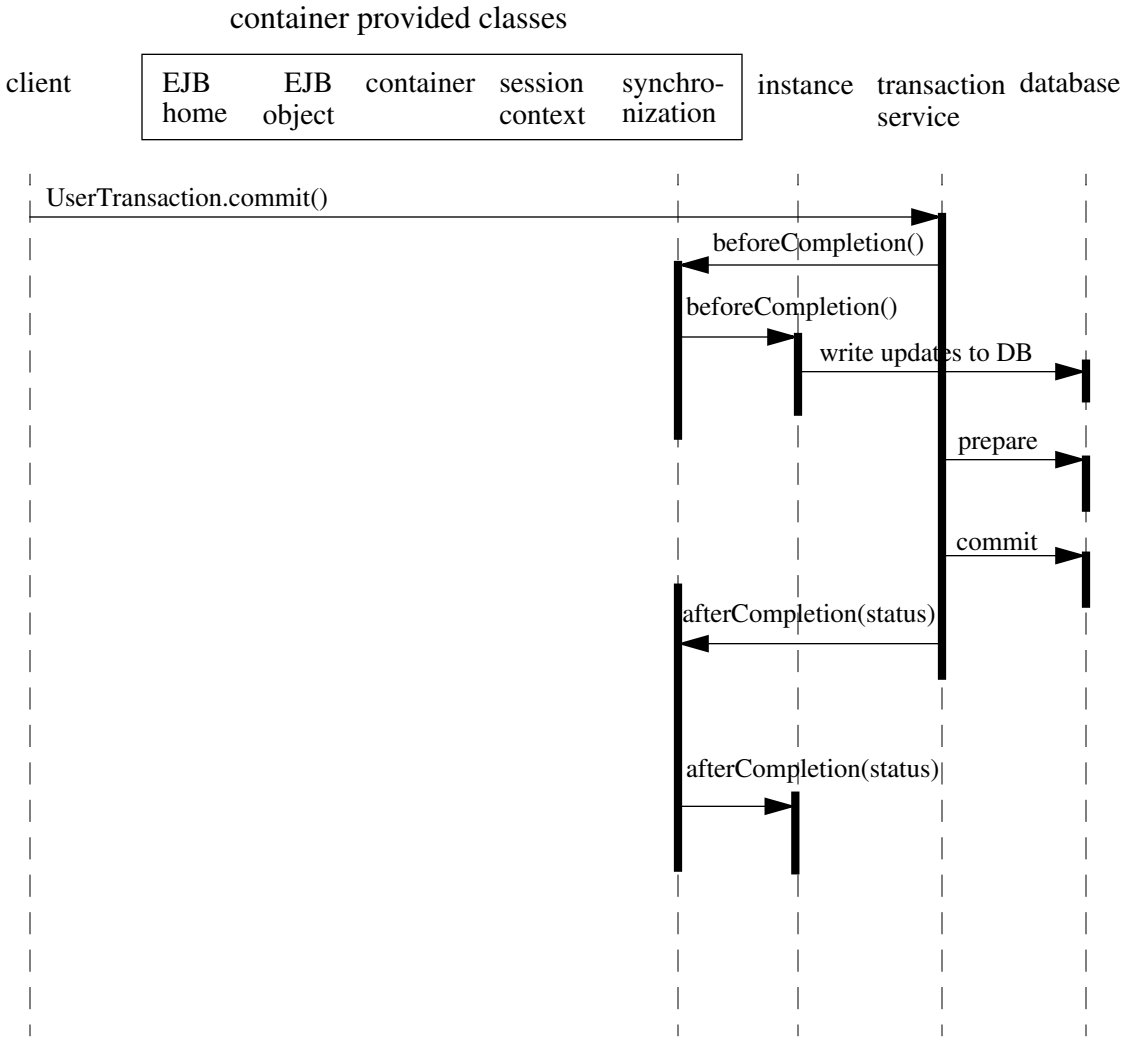
6.7.3 Starting a transaction

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



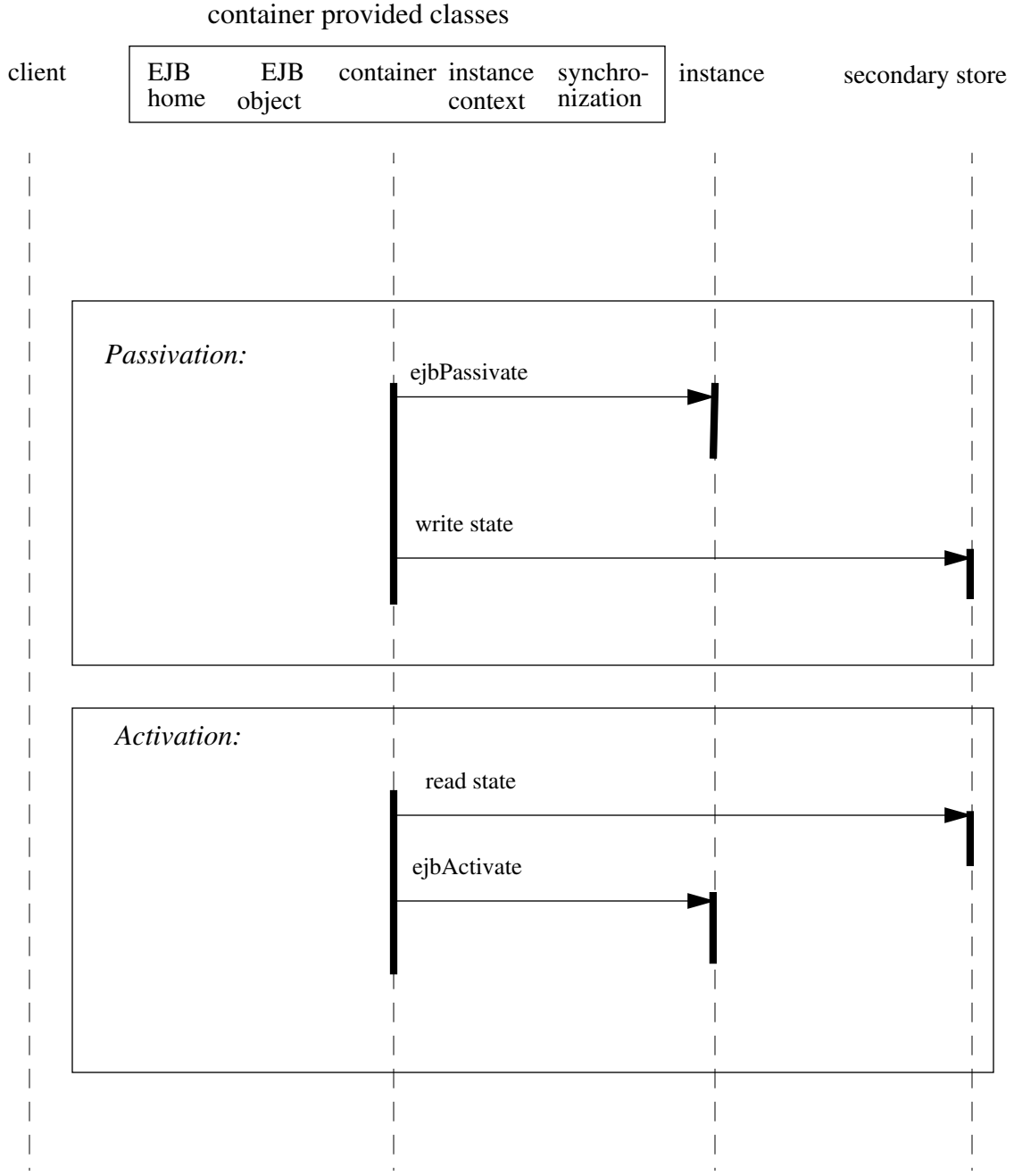
6.7.4 Committing a transaction

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



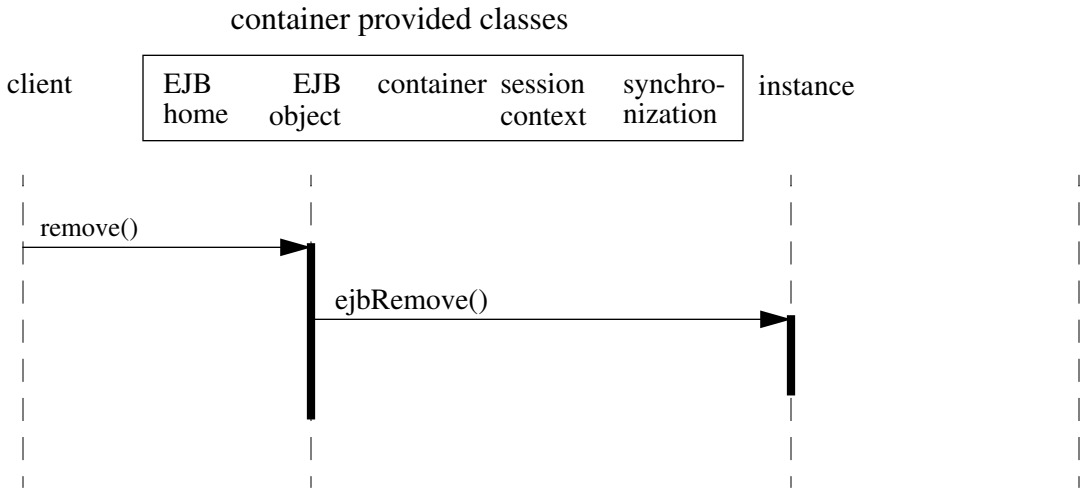
6.7.5 Passivating and activating an instance between transactions

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.



6.7.6 Removing a session object

The following diagram illustrates the destruction of a session Bean.



6.8 Stateless session Beans

Stateless session Beans are session Beans with no conversational state. This means that each Bean instance is identical, when it is not involved in serving a client-invoked method.

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. This *ejbCreate* method must take 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 it needs 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* method on the home interface of a stateless session Bean in the same way as they do on a stateful session Bean. Although the client thinks

it is controlling the life cycle of an EJB instance, the container is handling the *create* and *remove* calls without necessarily creating and removing an EJB instance.

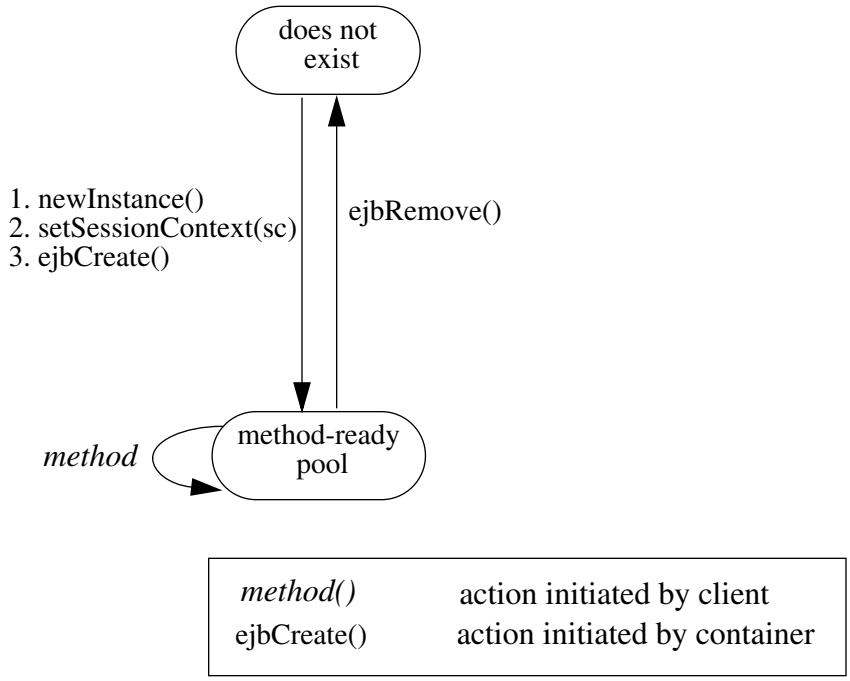
There is no fixed mapping between clients and stateless instances. The container simply delegates client work to any available instance that is method-ready.

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.



The following is a walk-through 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 method call from any client.

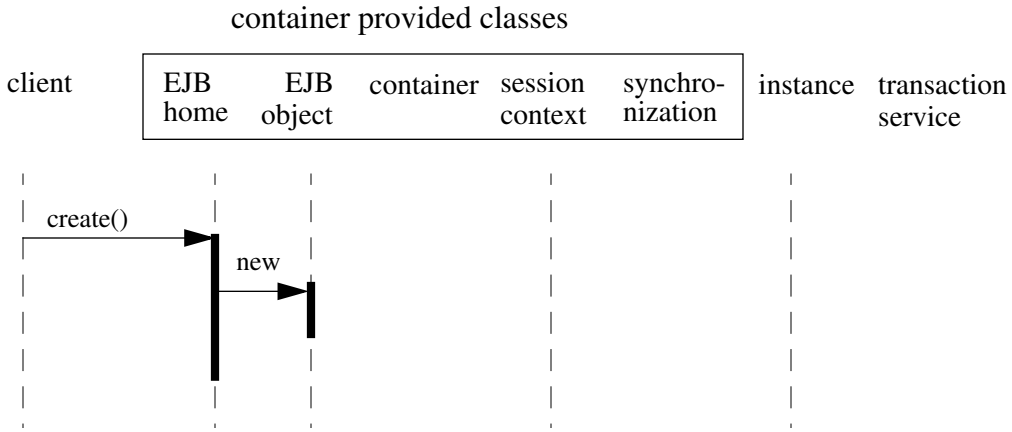
- When the container no longer needs the instance (this usually happens when the container wants to reduce the number of instances in the method-ready pool), the container invokes *ejbRemove()* on it. This ends the life of the stateless session Bean instance.

6.9 Sequence diagrams for a STATELESS session Bean

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

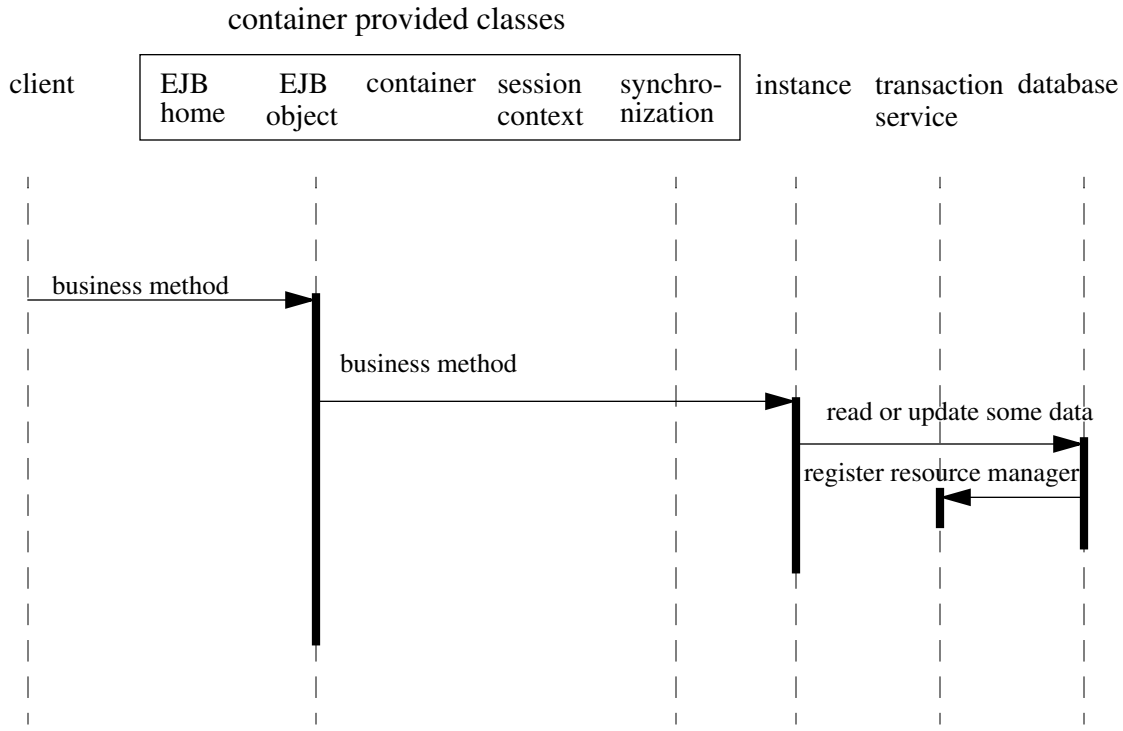
6.9.1 Client-invoked *create()*

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



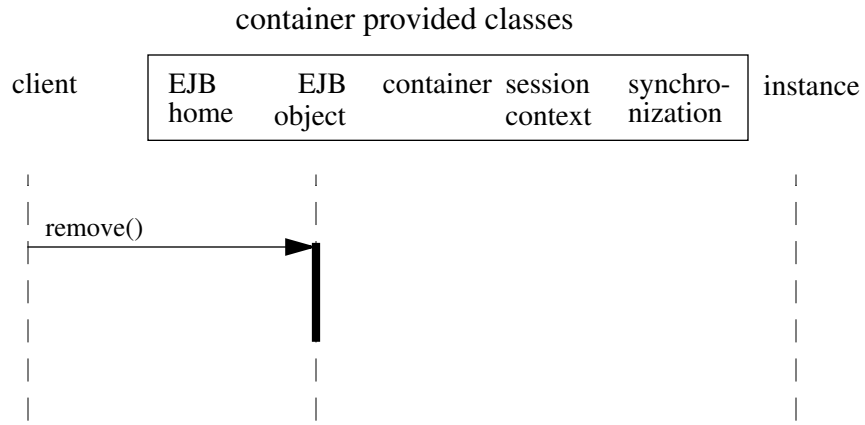
6.9.2 Business method invocation

The following diagram illustrates the invocation of a business method.



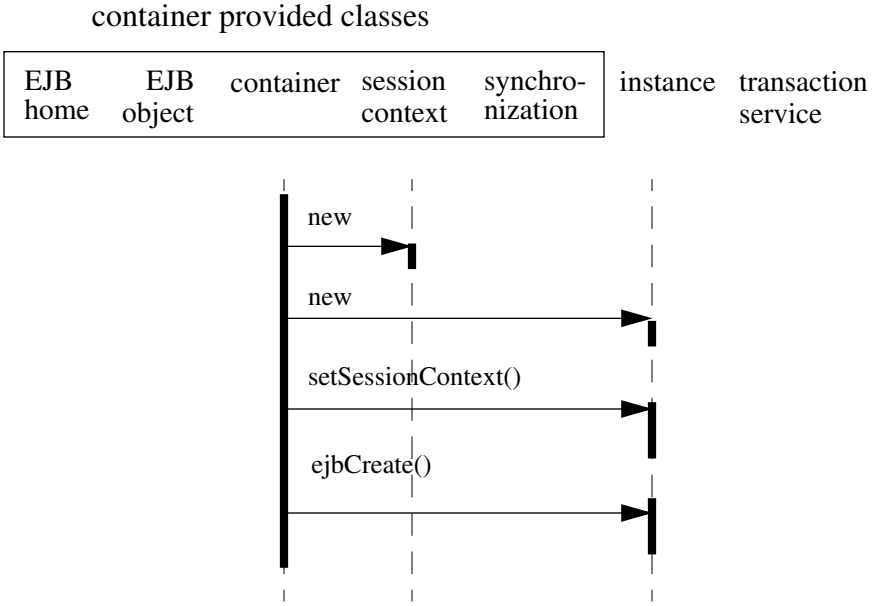
6.9.3 Client-invoked *remove()*

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

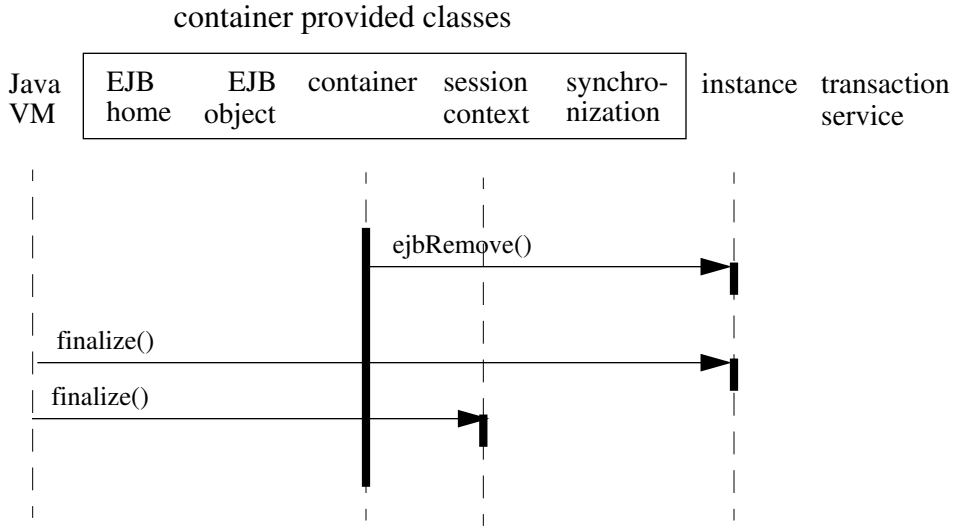


6.9.4 Adding instance to the pool

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



The following diagram illustrates the sequence for a container removing an instance from the method-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 the *javax.ejb.SessionBean* interface.

The class must be defined as *public*, and must not be *abstract*.

The class may, but is not required to, implement the enterprise Bean's remote interface[1].

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

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

6.10.3 *ejbCreate* methods

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

The method name must be *ejbCreate*.

The method must be declared as *public*.

The return type must be *void*.

The methods arguments must be legal types for Java RMI.

The throws clause may define arbitrary application specific exceptions.

The throws clause may include *java.rmi.RemoteException.*

The throws clause may include *javax.ejb.CreateException.*

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 methods arguments and return value types must be legal types for Java RMI.

The throws clause may define arbitrary application specific exceptions.

1.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 throws clause may include *java.rmi.RemoteException.*

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 Java RMI. This means that their arguments and return values must be of valid types for Java RMI, and their throws clause must include the *java.rmi.RemoteException*.

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.

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 Java RMI. This means that their arguments and return values must be of valid types for Java RMI, and their throws clause must include the *java.rmi.RemoteException*.

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 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 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.

6.11.1 Generation of implementation classes

The tools provided by the container are responsible for the generation of additional classes at enterprise Bean deployment time. The tools obtains the information that they need for generation of the additional classes by introspecting the classes and interfaces

provided by the enterprise Bean provider and from the information obtained from the Bean's deployment descriptor.

The container 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 container 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 container's tools may also allow 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 is a container generated class that implements the enterprise Bean's home interface. The class implements the methods of the *javax.ejb.EJBHome* interface, and the *create* methods specific to the enterprise Bean.

The implementation of each *create(...)* methods invokes a matching *ejbCreate(...)*.

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 is a container generated class that 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 container is responsible for implementing the handle class for the enterprise Bean. The handle class must be serializable by the Serialization protocol for the Java programming language.

6.11.5 Meta-data class

The container is responsible for implementing the class that provides meta-data to the client's view contract. The class must be a valid RMI/Value class and implement the *javax.ejb.EJBMetaData* interface.

6.11.6 Non-reentrant instances

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

> *Note that a session 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 is not possible for an application to 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 described in Chapters 11, 14, and 12.

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.

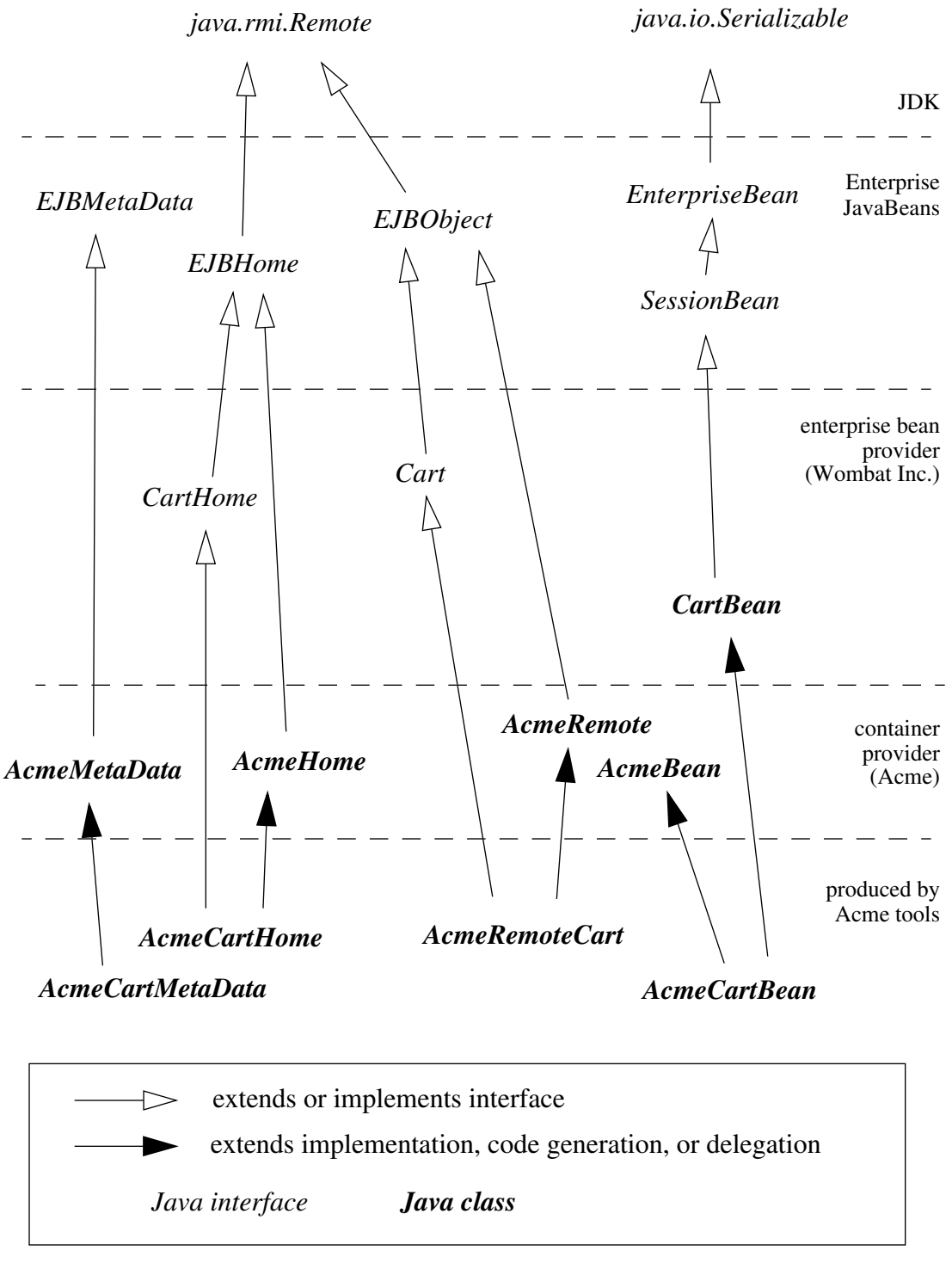
The classes generated by the container provider's tools in this scenario should be considered illustrative rather than prescriptive. Container providers are free to implement the contract between a session enterprise Bean and its container in a different way that achieves an equivalent effect (from the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



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 Java RMI 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 Java RMI remote interfaces.
- Specify the environment properties that the session Bean needs at runtime. The environment properties is a standard *java.util.Properties* file.
- 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's 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, and the generated classes most likely will be different from those generated by Acme's tools.

8 Client view of an entity

Note: Container support for entity enterprise Beans is an optional feature for EJB 1.0 compliance. Container support for entity enterprise Beans will become mandatory in EJB 2.0.

This chapter describes the client's 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, with only the business methods supplied by the enterprise Bean itself.

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

8.1 Overview

For a client, an entity enterprise Bean is a persistent object that represents an object view of an entity stored in a persistent storage (for example, in 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 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's 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.

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's 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

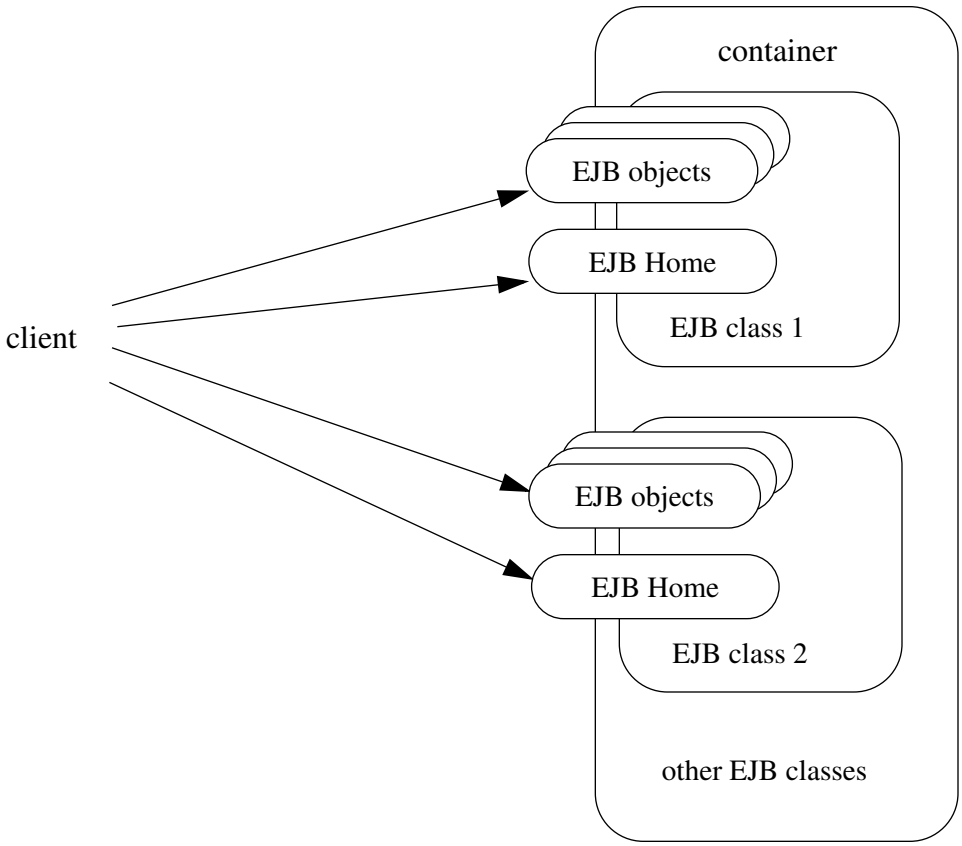
A client locates an enterprise Bean’s home interface using JNDI. For example, the home interface for the Account enterprise Bean can be located using the following code segment:

```
Context initialContext = new InitialContext();
AccountHome accountHome = 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.



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.

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 also 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[1], one for each way to look up an EJB object, or collection of EJB objects of a particular type. The names 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.

The throws clause of every finder method must include the *java.rmi.RemoteException*. The throws clause typically includes also 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.

1.The *findByPrimaryKey(primaryKey)* method is mandatory for all entity Beans.

The following is an example of 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

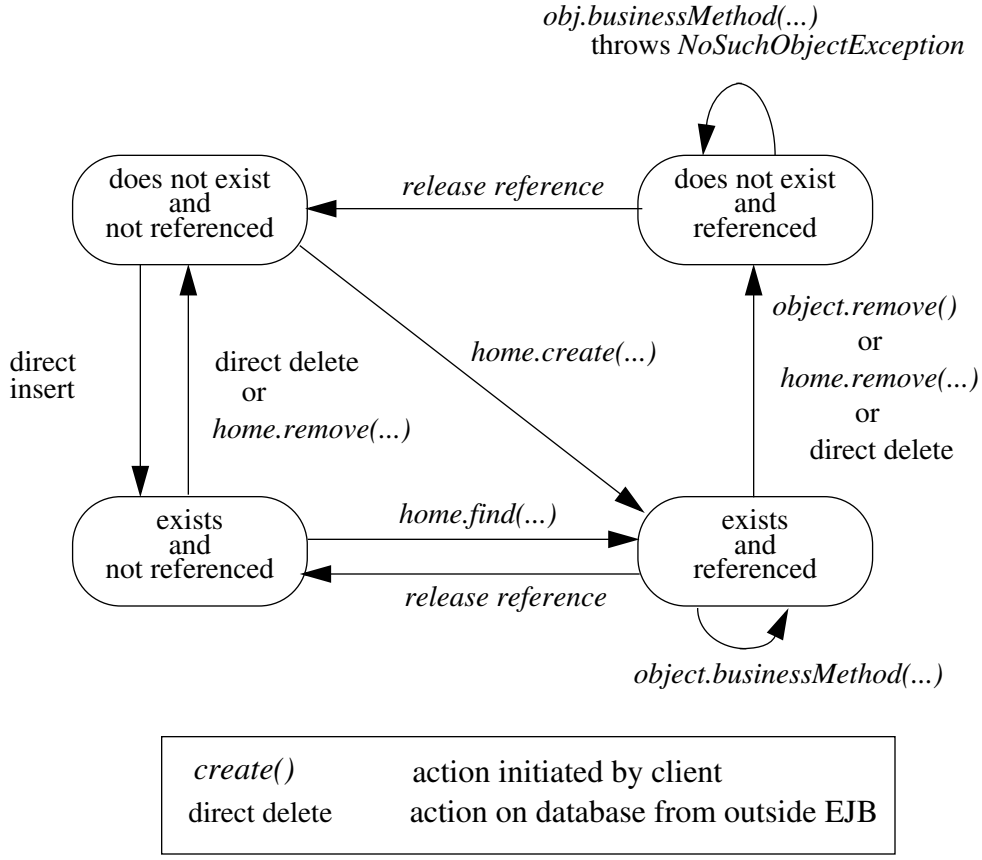
The *javax.ejb.EJBHome* interface defines several methods that allow the client to remove EJB objects.

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

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).



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 a 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 (handles are described later in Section 8.7).

A client that has a reference to an object can then 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.

Enterprise JavaBeans allows a primary key object to be any *java.io.Serializable* class. 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.

A client can test whether two EJB object references refer to the same entity by any of the following methods:

- Invoke the *isIdentical(object)* method on one of the references and pass the other reference as the method's argument.

- Obtain the entity objects' primary keys, and compare the keys using the Java programming language equality operator. This method works only if the two EJB references refer to EJB objects with the same home.

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 Enterprise JavaBeans does not specify "object equality" 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 Enterprise 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 do 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 introduced by 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 Enterprise 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 the handle's class must implement the *java.io.Serializable* interface, a client may serialize it. The client may use the serialized handle later, possibly in a different process, to re-obtain a reference to the EJB object identified by the handle.

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) handle.getEJBObject();
account.debit(100.00);
```

9 Entity Bean component contract

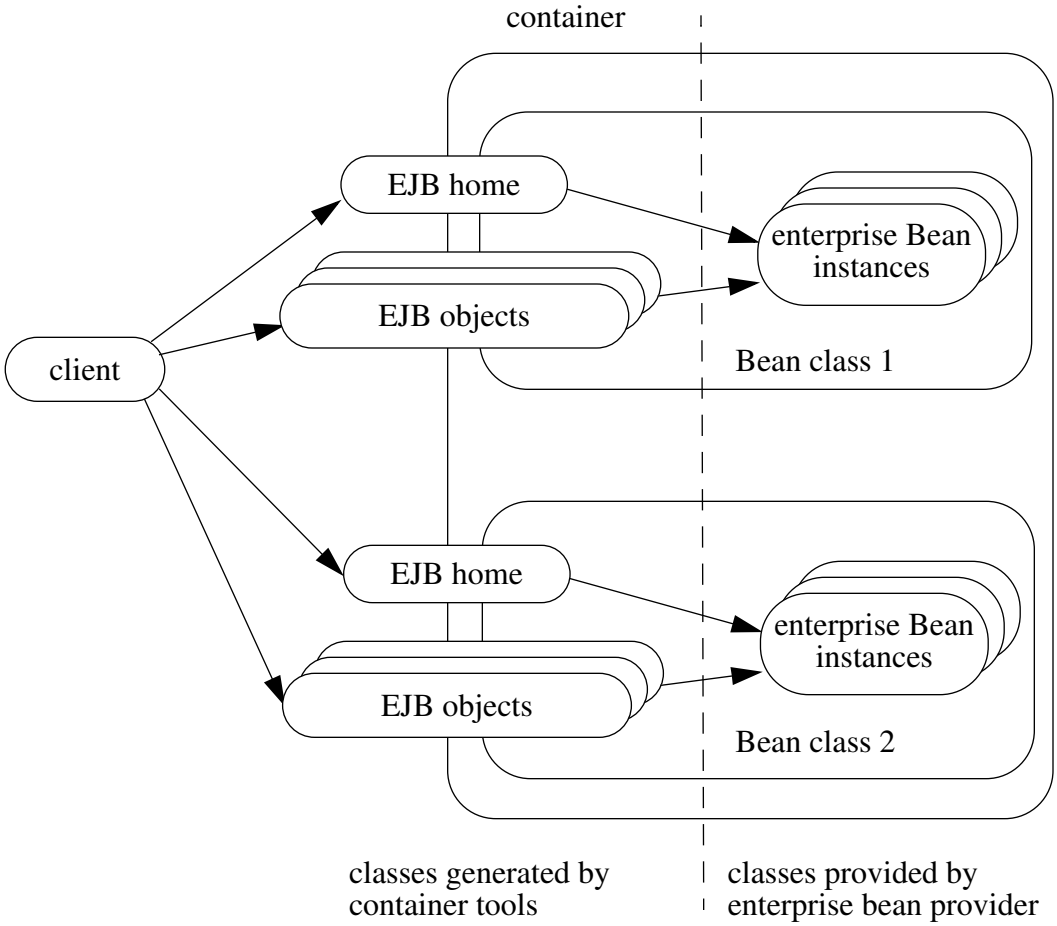
Note: Container support for entity enterprise Beans is an optional feature for EJB 1.0 compliance. Container support for entity enterprise Beans will become mandatory in EJB 2.0.

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 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 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.



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 EJB provider.

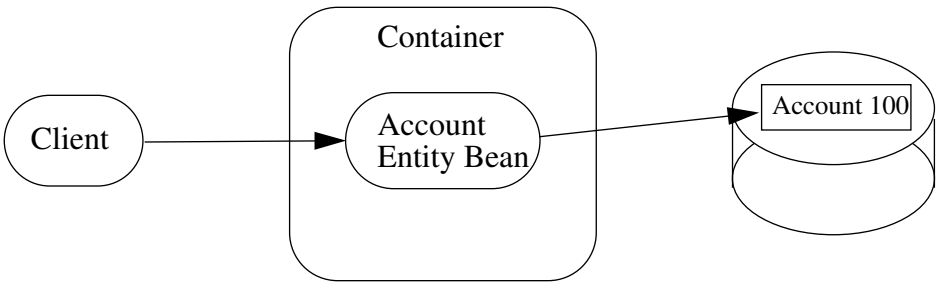
9.2 Entity persistence

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 protocol for transferring the state of the entity between the instance variables of an enterprise Bean instance and the underlying database is referred to as object *persistence*.

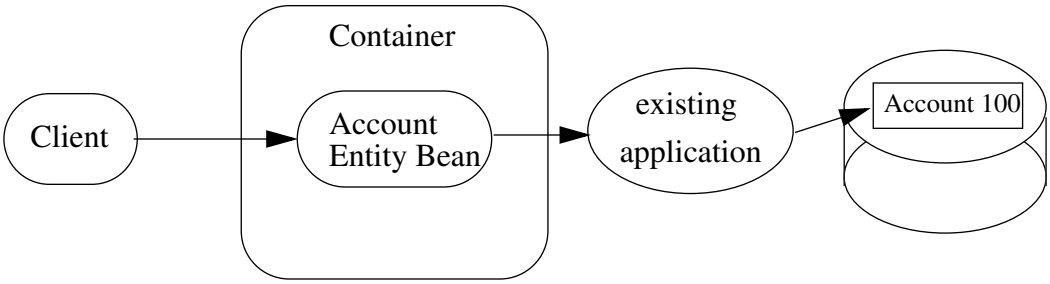
The entity component protocol allows the enterprise Bean provider either to implement the enterprise Bean's persistence directly in the enterprise Bean class (we call this Bean-managed persistence), or delegate the enterprise Bean's persistence to the container (we call this container-managed persistence).

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

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



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



9.2.1 Bean-managed persistence

In the Bean-managed case, the enterprise Bean provider writes database access calls (e.g. using JDBC™ or JSQL) directly in the methods of the enterprise Bean class. The database access calls are performed in the *ejbCreate(...)*, *ejbRemove()*, *ejbFind<METHOD>*(), *ejbLoad()*, and *ejbStore()* enterprise Bean callback methods.

The advantage of using Bean-managed persistence is that the enterprise Bean can be installed into a container without the container having to generate database calls that implement the enterprise Bean's persistence. The main disadvantage is that the persistence is hard-coded into the enterprise Bean class, which makes it hard to adapt the enterprise Bean to a different data source.

9.2.2 Container-managed persistence

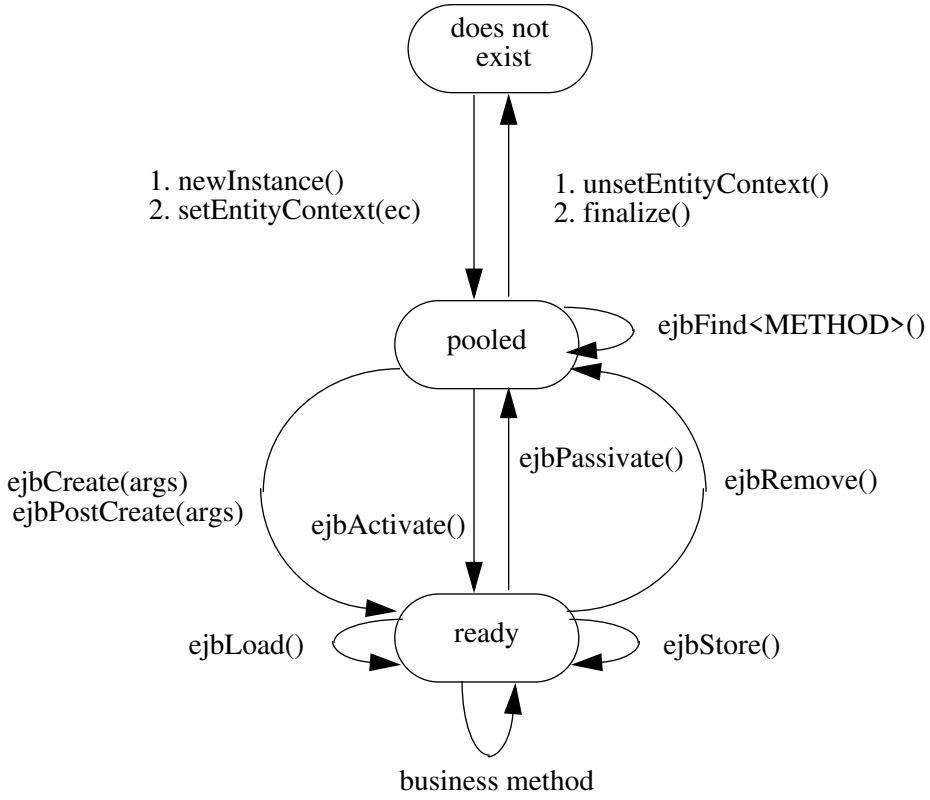
In the container-managed case, the Bean developer does not write the database access calls in the enterprise Bean. Instead, the container provider's tools generate the database access calls at enterprise Bean's deployment time (i.e. when the enterprise Bean class is installed into a container). The enterprise Bean provider must specify the *containerManagedFields* deployment descriptor property to specify 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 is independent from the data source in which the entity is stored. The container tools can generate classes that use JDBC or JSQL to access the entity state in a relational database, or classes that implement access to a non-relational data source, such as IMS databases, 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.

9.3 Instance life cycle

The following diagram illustrates the 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 is a walk-through of the life cycle of an entity enterprise Bean instance:

- An enterprise Bean's instance 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 obtain the information about the caller of a client-invoked method[1].

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 getPrimaryKey() method might be different each time an instance moves from the pooled state to the ready state.

- 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).
- 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 for which there is no instance in the ready state in the proper transaction context. 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 the instance needs to be activated to service an invocation on an existing EJB object.
- 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, at anytime. 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.
- 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. The Java application environment runtime will eventually invoke the *finalize()* 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 *getPrimaryKey()* method might be different each time an instance moves from the pooled state to the ready state.

9.4 The entity Bean component contract

This section specifies the contract between an entity Bean and its container. The contract is specified here assuming Bean-managed persistence. The differences in the contract for a container-managed persistence are defined in Section 9.10.

9.4.1 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:

- *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(EntityContext ic);*

  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(ec)* method to free any resources that are held by the instance (these resources typically had been allocated by the *setEntityContext()* method).

- *public void ejbCreate(...);*

There are zero[1] 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 proper transaction context.

- *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 proper transaction context.

- *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 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 put the instance back into the pool of available 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).

---

1. 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.

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 effective transaction context of the client's *remove* 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 advise the instance that it must synchronize 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.

  An instance should refresh its state in the *ejbLoad()* method by reading the entity state from the database.

  This method executes in the proper transaction context.

- *public void ejbStore();*

  The container invokes this method on an instance to advise the instance that the instance must synchronize the entity state in the database with 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 should write its state to the database in the *ejbStore()* method.

  This method executes in the proper 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 proper transaction context.

The implementation of an *ejbFind<METHOD>* method should use 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.

9.4.2 Container's view:

The following describes the container's side of the state management contract. The container must call the following methods as indicated below:

- *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 is not allowed to reuse this instance, and therefore it should drop any references to the instance to allow the Java programming language garbage collector to eventually invoke the *finalize()* method on the instance.

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

- *public void 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 this method in the proper transaction context.

- *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 of the client's *remove* method. 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.

- *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 container invokes this method in the proper transaction context.

- *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.

  The container invokes this method in the proper transaction context.

- *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.

The container must invoke the *ejbFind<METHOD>(...)* method in the proper transaction context.

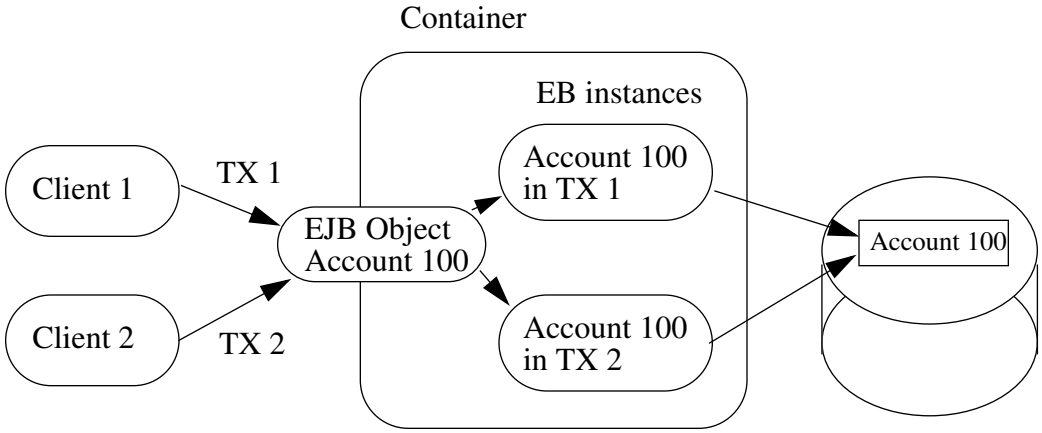
If the *ejbFind<METHOD>* method is declared to return a single primary key, the container creates and EJB object reference for the primary key and returns it to the client. If the *ejbFind<METHOD>* method is declared to return a collection of primary keys, the container creates a collection of EJB objects for the primary keys returned from *ejbFind<METHOD>*, and returns the collection to the client.

9.5 Concurrent access from multiple transactions

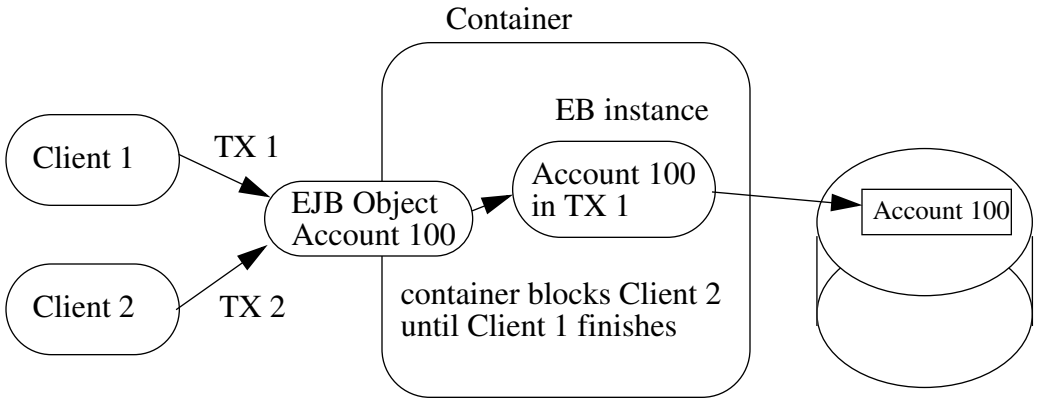
The enterprise Bean developer does not have to worry about concurrent access from multiple transactions when writing the business methods. The enterprise Bean developer writes the methods assuming that the container will ensure appropriate synchronization for entity Beans that are accessed concurrently from multiple transactions.

The entity container typically uses one of two 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 9.11.4 is applicable to this type of container.



- The container acquires an exclusive lock on 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 9.11.4 is applicable to this type of container.



9.6 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 container will allow loopbacks if the Bean's deployment descriptor specifies that the Bean is re-entrant. 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.

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 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.

9.7 The 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.

9.7.1 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.

9.7.2 Enterprise Bean class

The following are the requirements for an entity enterprise Bean class:

The class must implement the *javax.ejb.EntityBean* interface.

The class must be defined as *public*, and must not be *abstract*.

The class may, but is not required to, implement the enterprise Bean's remote interface[1].

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

9.7.3 *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 return type must be the primary key type.

The methods arguments and return value types must be legal types for Java RMI.

The throws clause may define arbitrary application specific exceptions.

The throws clause may include *java.rmi.RemoteException.*

The throws clause may include *javax.ejb.CreateException*.

The return type of an *ejbCreate* method must be either the primary key type, or a collection (See 9.9.1).

9.7.4 *ejbCreate* methods

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

1.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 *ejbPostCreate*.

The method must be declared as *public*.

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.

The throws clause may include *java.rmi.RemoteException.*

The throws clause may include *javax.ejb.CreateException*.

9.7.5 *ejbFind* methods

The enterprise Bean class must define the *ejbFindByPrimaryKey* method.

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 methods arguments and return value types must be legal types for Java RMI.

The return type of a finder method must be the enterprise Bean's primary key type, or a collection of objects of the primary key type (See Subsection 9.9.1).

The throws clause may define arbitrary application specific exceptions.

The throws clause may include *java.rmi.RemoteException.*

The throws clause may include *javax.ejb.FinderException*.

Each entity enterprise Bean class must define the *ejbFindByPrimaryKey* method.

9.7.6 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 methods arguments and return value types must be legal types for Java RMI.

The throws clause may define arbitrary application specific exceptions.

The throws clause may also include the *java.rmi.RemoteException*.

9.7.7 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 Java RMI. This means that their arguments and return values must be of valid types for Java RMI, and their throws clause must include the *java.rmi.RemoteException*.

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.

9.7.8 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 Java RMI. This means that their arguments and return values must be of valid types for Java RMI, and their throws clause must include the *java.rmi.RemoteException*.

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>"* (i.e. *findLargeAccounts*)*,* and it must match one of the *ejbFind<METHOD>* methods defined in the enterprise Bean class (i.e. *ejbFindLargeAccounts*). The matching *ejbFind<METHOD>* method must have the same number and types of its arguments (note that the return type may be different).

The return type for a *finder* method must be the enterprise Bean's remote interface type, or a collection of thereof.

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 *create* method of the remote interface.

The throws clause of a *finder* method must include the *javax.ejb.FindException*.

9.7.9 Enterprise Bean's primary key class

The Bean provider must define a primary key class. The class must be serializable by the Java programming language Serialization protocol.

9.8 The responsibilities of the container provider

This section describes the responsibilities of the container provider to support an entity Bean.

9.8.1 Generation of implementation classes

The tools provided by the container are responsible for the generation of additional classes at enterprise Bean deployment time. 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 from the information obtained from the Bean's deployment descriptor.

The container 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 container 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 container's 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.

9.8.2 EJB Home class

The EJB home class is a container generated class that implements the enterprise Bean's home interface. The class 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.

9.8.3 EJB Object class

The EJB Object class is a container generated class that 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 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.

9.8.4 Handle class

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

9.8.5 Meta-data class

The container is responsible for implementing the class that provides meta-data to the client's view contract. The class must be a valid RMI/Value class and implement the *javax.ejb.EJBMetaData* interface.

9.8.6 Instance's re-entrance

The container must enforce the rules defined in Section 9.6.

9.8.7 Transaction scoping, security, exceptions

The container must follow the rules with respect to transaction scoping, security checking, and exception handling described in Chapters 11, 14, and 12.

9.9 Miscellaneous

9.9.1 Collections

The return type of an entity finder method can be either a single EJB object reference or a collection of EJB object references. If there is the possibility that the finder method may find more than one EJB object, the Bean developer should define the return type of the *ejbFind<METHOD>(...)* and *find<METHOD>(...)* method to be a collection.

The JDK 1.1.x type for a collection is the *java.util.Enumeration* interface, and therefore a finder method that returns a collection of EJB objects must define the return type to be *java.util.Enumeration*.

JDK 1.2 will provide better support for collections. A future release of EJB will extend the allowed types for finders to use the JDK 1.2 collections, in addition to the java.util.Enumeration type.

The following is an example of a finder method defined in a home interface:

```
public AccountHome {
        ...
        java.util.Enumeration findLargeAccounts(double limit)
                throws FinderException, RemoteException;
        ...
}
```

The following is an example of a finder method implemented in the enterprise Bean's class:

```
public AccountBean {
        ...
        public java.util.Enumeration ejbFindLargeAccounts(
                double limit)
                throws FinderException, RemoteException
        {
                ...
        }
        ...
}
```

9.10 Container-managed entity Beans

Sections 9.3 through 9.9 describe the component contract for entity Beans with Bean-managed persistence. This section specifies the contract for the entity Beans with container-managed persistence.

We define here the differences in the contract for entities with Bean-managed persistence.

9.10.1 *containerManagedFields* deployment descriptor property

The container determines that an entity Bean is of the container-managed persistence type by examining the *containerManagedFields* property of the deployment descriptor. If the *containerManagedFields* property is defined in the deployment descriptor, the entity Bean is of the container-managed persistence type.

The value of the *containerManagedFields* property is a list of instance fields that the enterprise Bean provider expects the container to manage by loading and storing from a database. The enterprise Bean code should not contain any database access calls—the database access calls will be generated by the container tools at deployment time.

The containers that specialize in providing support for container-managed persistence will typically provided rich deployment time tools to allow the enterprise Bean deployer to establish the mapping of the instance fields to the underlying data source. Such containers are likely to be specialized for a particular legacy data source. It is expected that although the mapping process is made easy by the container provider's tools, the Bean deployer may be involved in the mapping process (i.e. the mapping process is not fully automatic).

The container moves data between the Bean's instance variables and the underlying data source before or after the execution of the *ejbCreate*, *ejbRemove*, *ejbLoad*, and *ejbStore*, as described in the below subsections. The container is also responsible for the implementation of the finder methods.

The enterprise bean provider must declare the container-managed fields as *public* to allow the container tools to generate the additional classes that transfer data between the instance's fields and the data source. A container-managed field must be of a Java Serializable type[1].

9.10.2 ejbCreate, ejbPostCreate

While in the case of Bean-managed persistence the enterprise Bean developer is responsible for writing the code that inserts a record into the database in the *ejbCreate(...)* methods, in the case of container-managed persistence it is the container who 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 return value of an *ejbCreate(...)* method must be void for enterprise Beans with container-managed persistence.

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.

Then the container invokes the matching *ejbPostCreate(...)* method on the instance. The instance can discover the primary key by calling *getPrimaryKey()* on its session context object.

The container must perform *ejbCreate*, database insert operation, and *ejbPostCreate* in the proper transaction context.

---

1. This does not imply that the container must use Java Serialization to extract the field. For example, if a field is of a primitive type, or a reference to an instance of a class whose fields are all declared as *public*, the container may extract the fields directly.

9.10.3 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 proper transaction context.

9.10.4 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 example, to perform some computation on the values of the fields that were read by the container (for example, perform uncompression of text fields).

9.10.5 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.

9.10.6 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 their description supplied by the bean provider. The Enterprise JavaBeans architecture does not specify the format of the finder method description.

9.10.7 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 these rules:

- The primary key type must be *public*.
- All fields in the primary key class must be declared as *public*.
- The class must have a *public* default constructor.
- The names of the fields in the primary key class must be a subset of the names of the container-managed fields (this allows the container to extract the primary key fields from an instance's container-managed fields, and vice versa).

9.11 Sequence diagrams

This section uses sequence diagrams to illustrate the interactions between an entity Bean instance and its container.

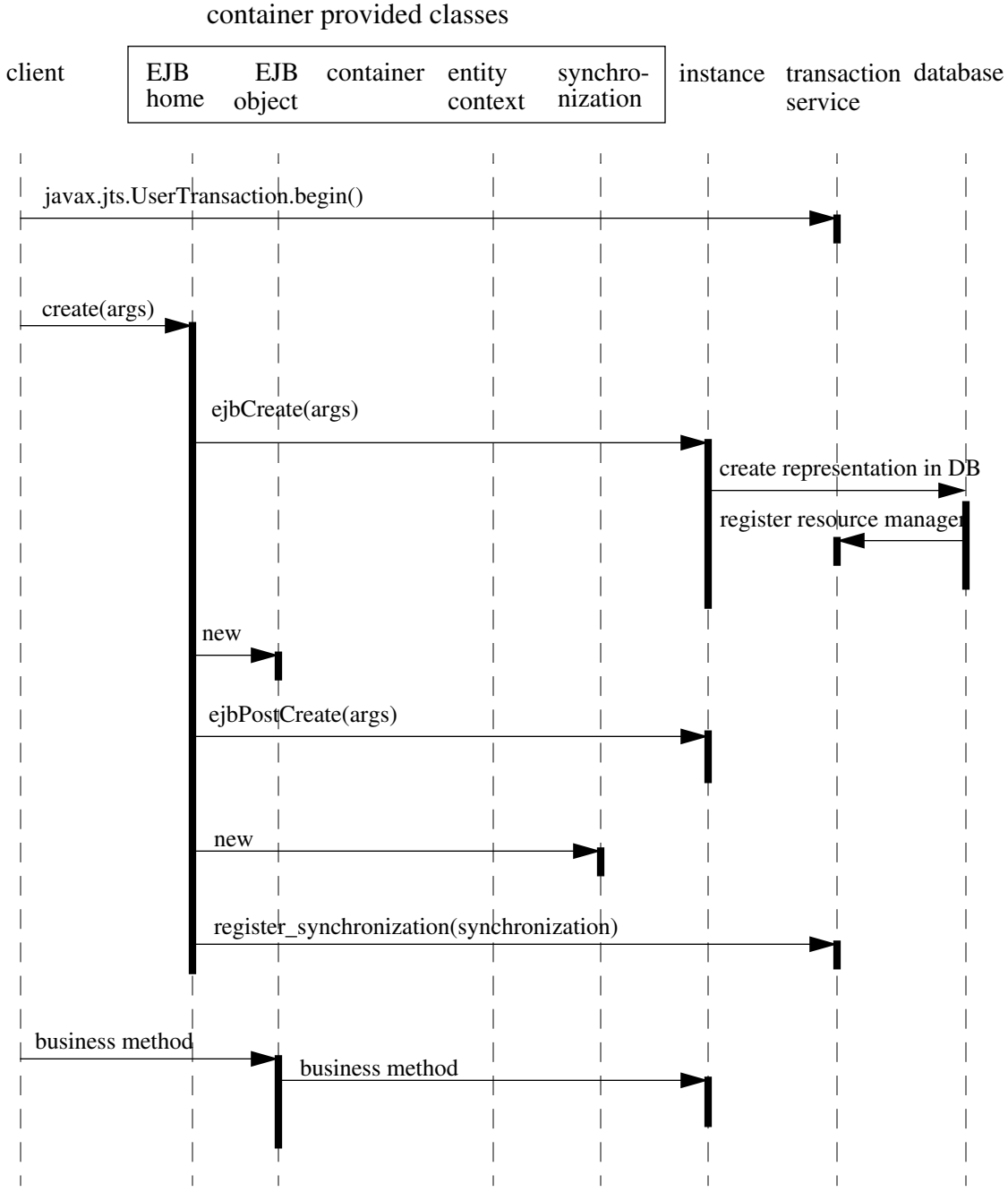
9.11.1 Notes

The sequence 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.

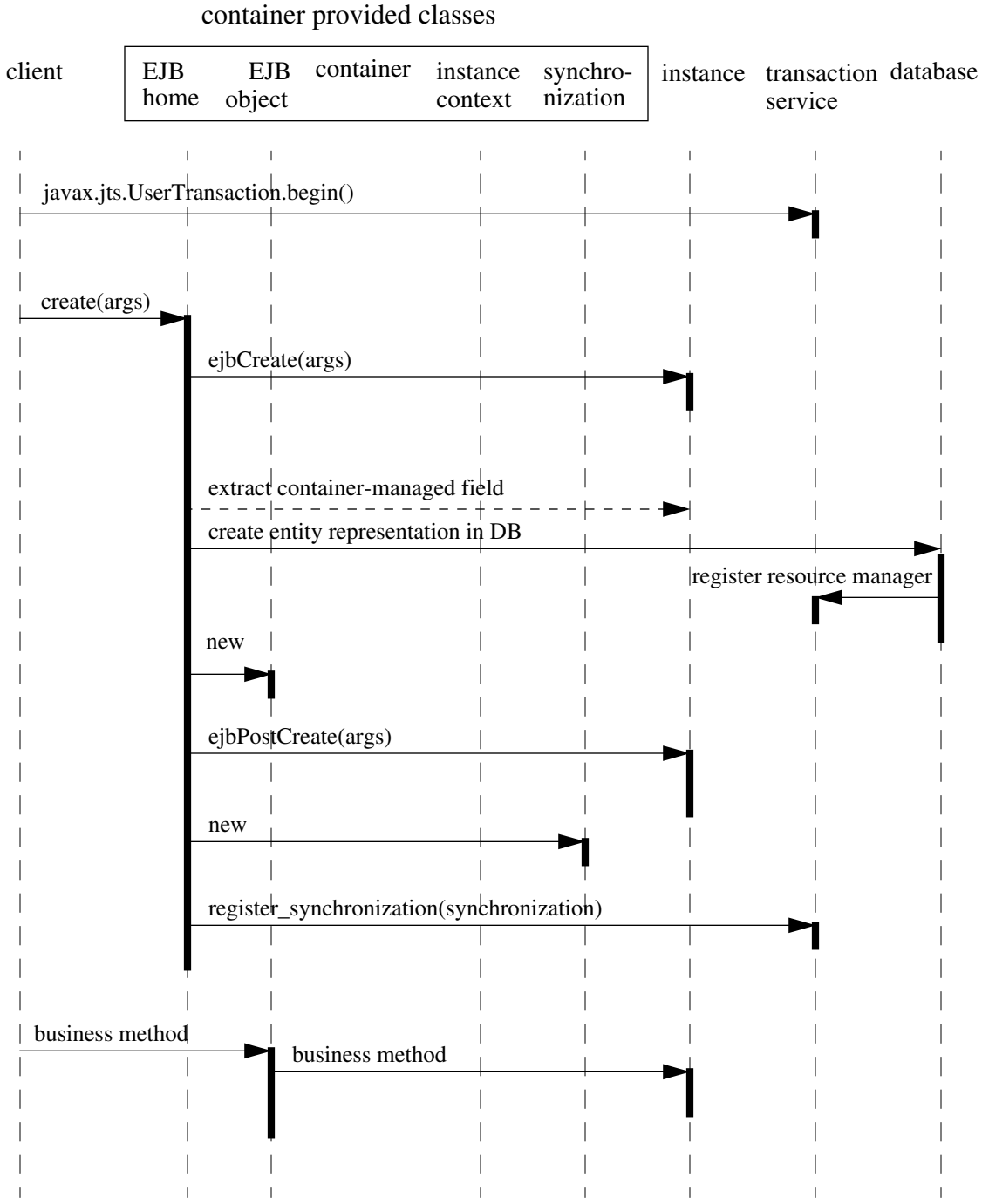
The classes shown in the diagrams should be considered as an illustrative implementation rather than a prescriptive one

9.11.2 Creating an entity object

The following diagram illustrates the creation of an enterprise Bean with Bean-managed persistence.

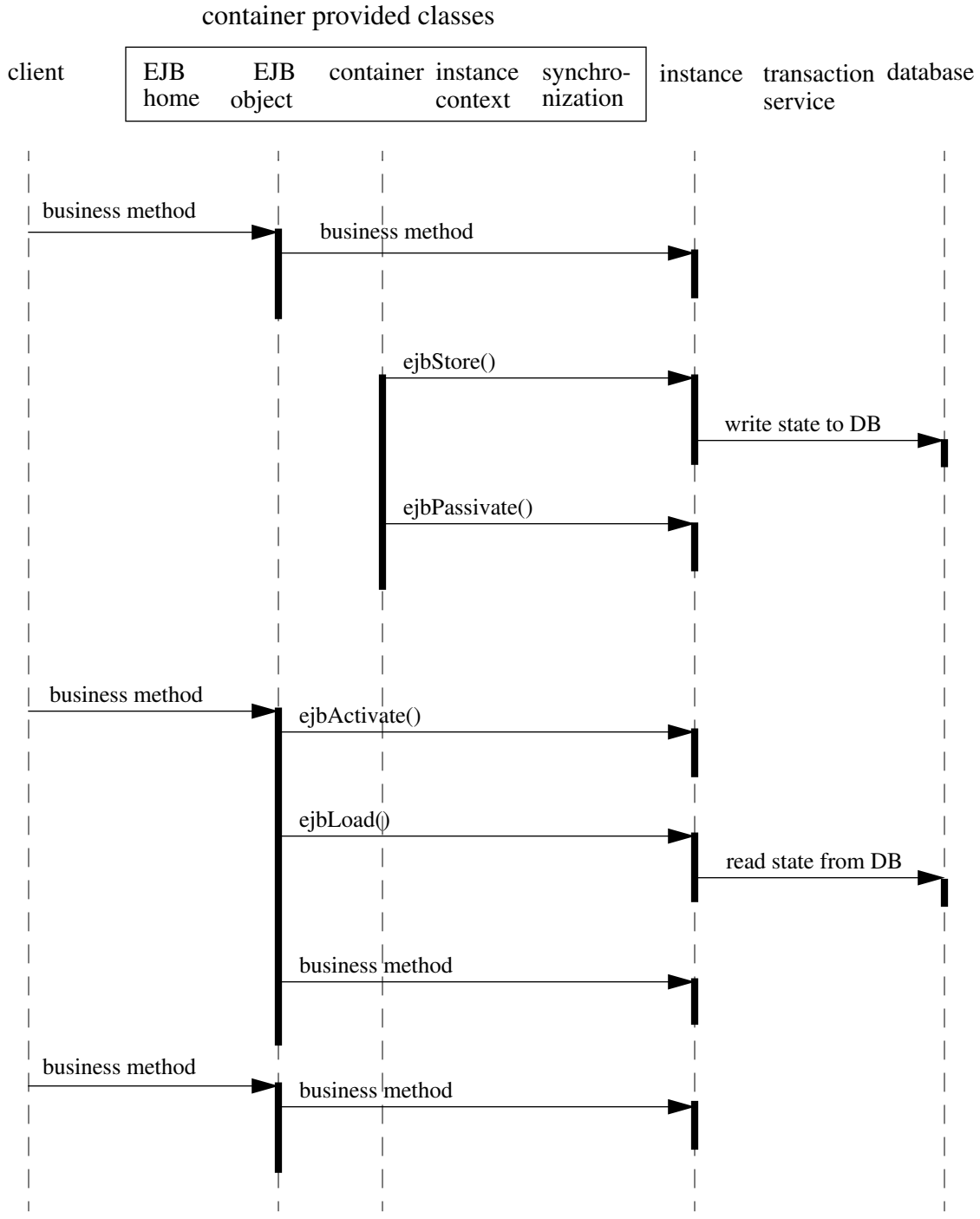


The following diagram illustrates the creation of an enterprise Bean with container-managed persistence:

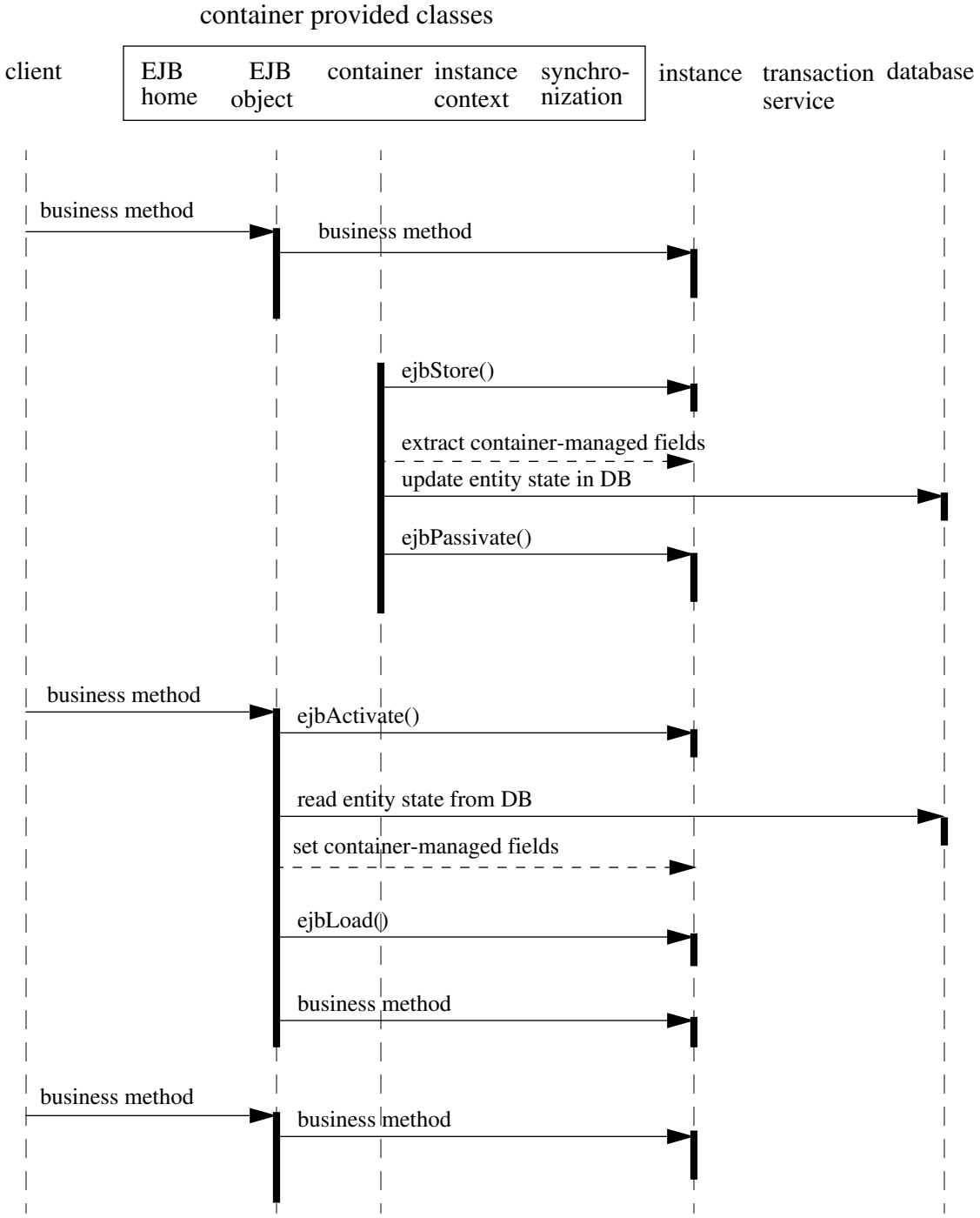


9.11.3 Passivating and activating an instance in a transaction

The following diagram illustrates the passivation and reactivation of an enterprise Bean instance with Bean-managed persistence.



The following diagram illustrates the passivation and reactivation of an enterprise Bean instance with container-managed persistence.



9.11.4 Committing a transaction

This section describes the sequence during transaction commit.

The entity Bean protocol is designed to allow a container the flexibility to select the disposition of an instance at transaction commit time. The sequence diagrams in this section illustrate three alternative commit options with respect to the instance state. The selection of the commit option is transparent to the entity Bean—the entity Bean will work correctly regardless of the option chosen by the container.

The three options are:

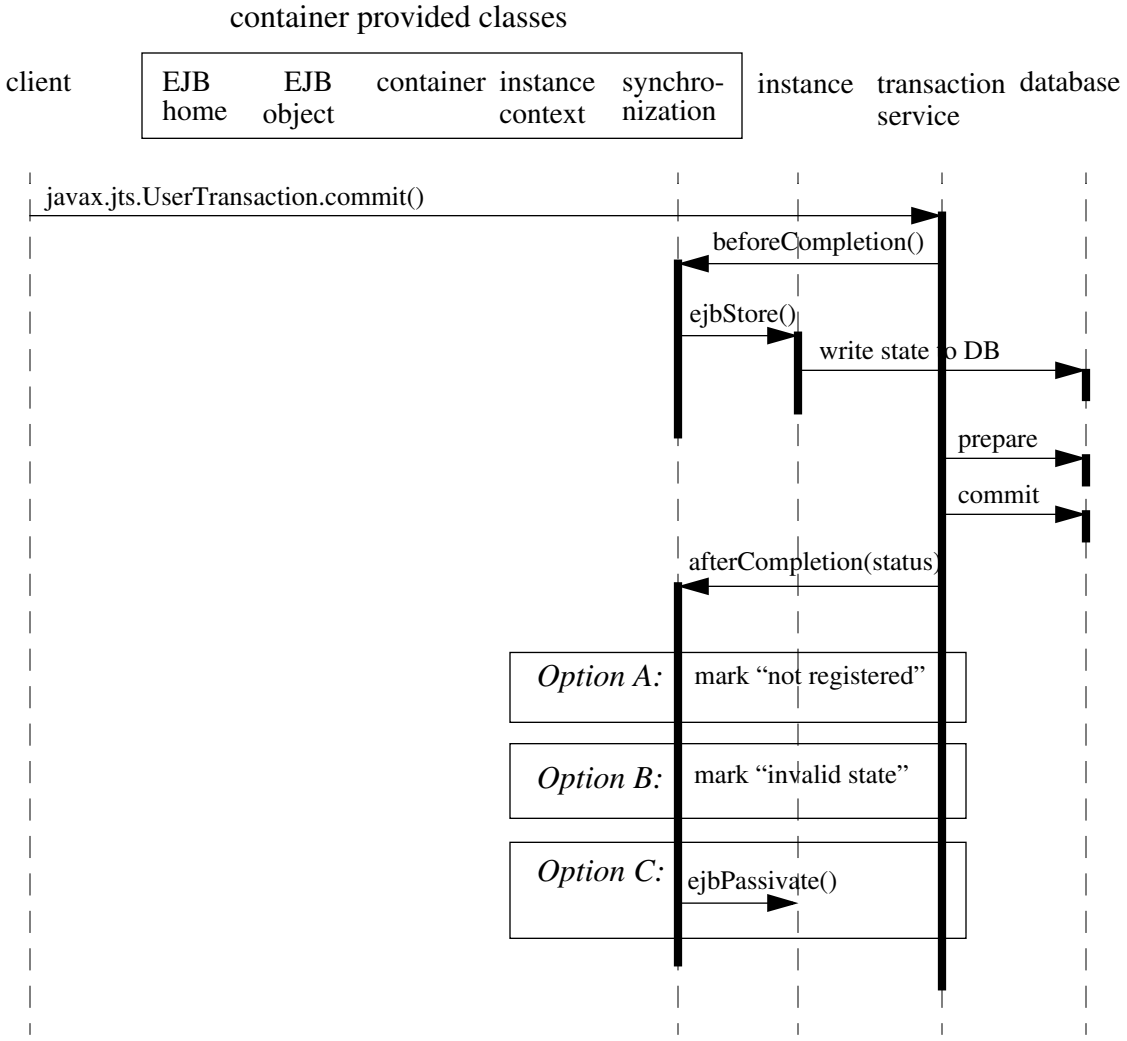
- Option A: The container caches a "ready" instance between transactions. The instance has exclusive access to the state of the object in the persistent storage, and therefore the container does not have to synchronize the instance's state from the persistent storage at the beginning of the next transaction.
- Option B: The container caches a "ready" instance between transactions. Unlike in Option A, the instance does not have exclusive access to the state of the object in the persistent storage, and 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. An instance is returned to the pool of available instances after a transaction has completed.

Note that the container must synchronize the instance's state with the persistent storage at transaction commit for all the three options.

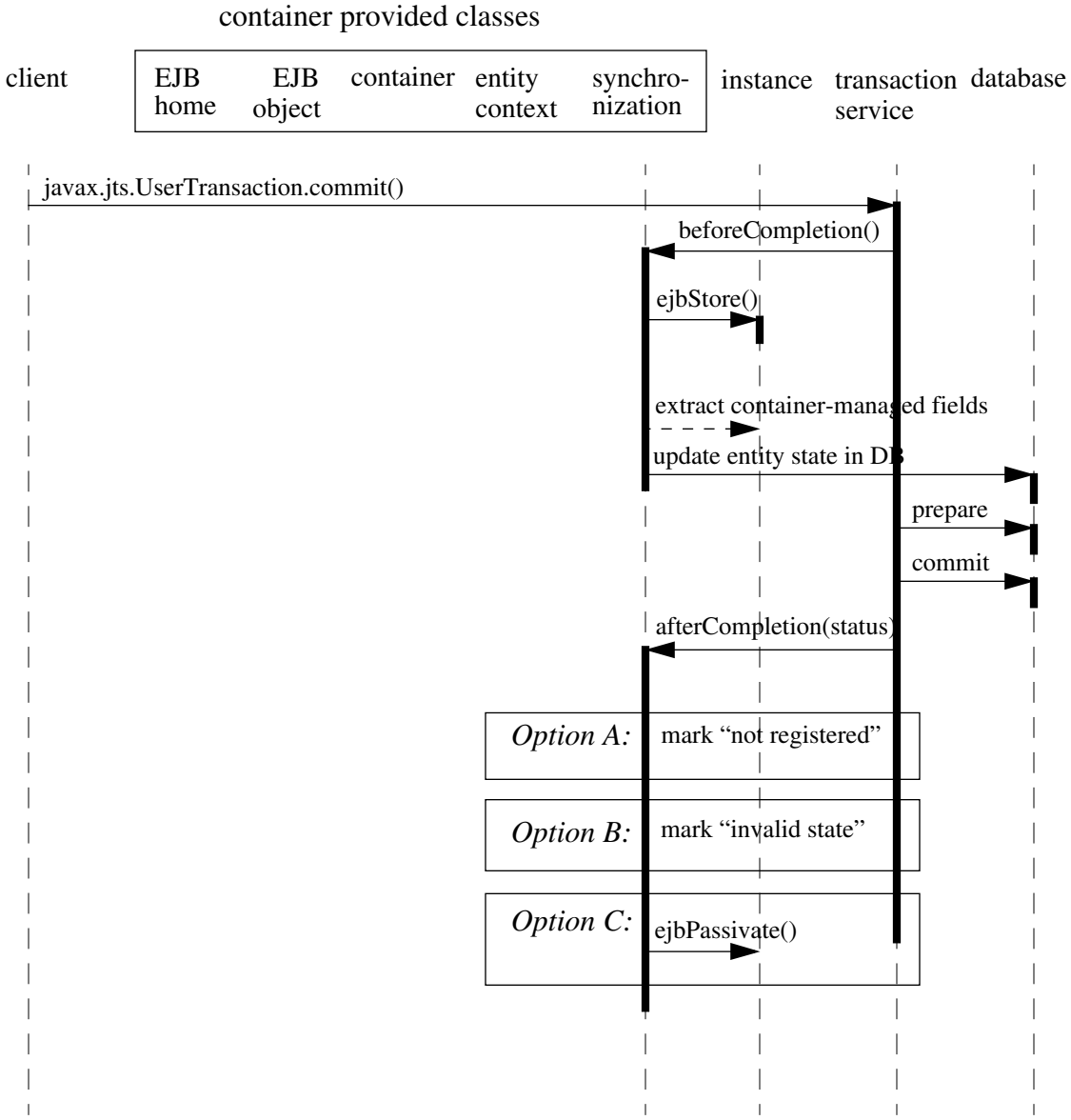
Table 1: Summary of commit-time options

	Write instance state to database	Instance stays ready	Instance state remains valid
Option A	Yes	Yes	Yes
Option B	Yes	Yes	No
Option C	Yes	No	No

The following diagram illustrates the transaction commit protocol that involves an enterprise Bean instance with Bean-managed persistence.

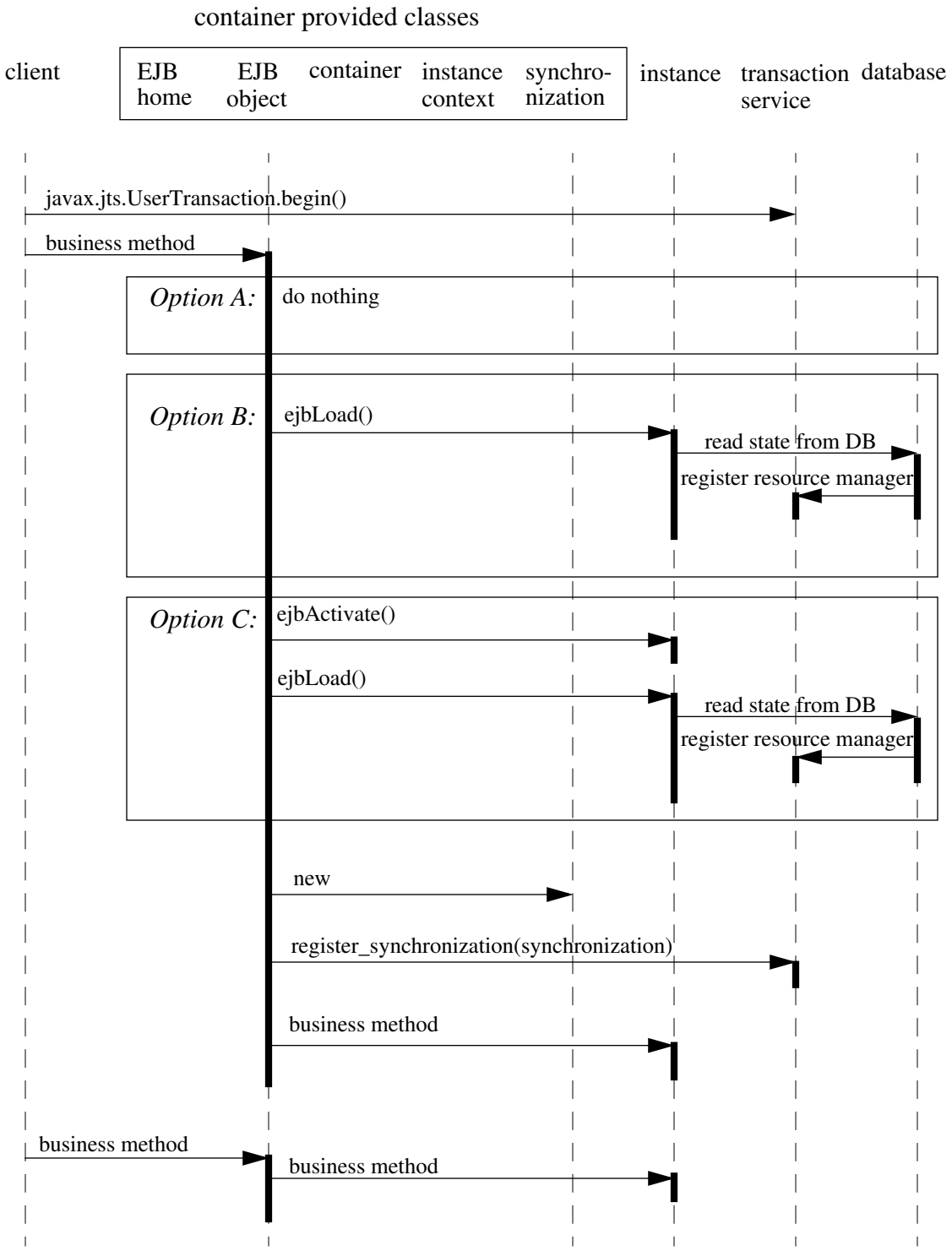


The following diagram illustrates the transaction commit protocol for an enterprise Bean instance with container-managed persistence.

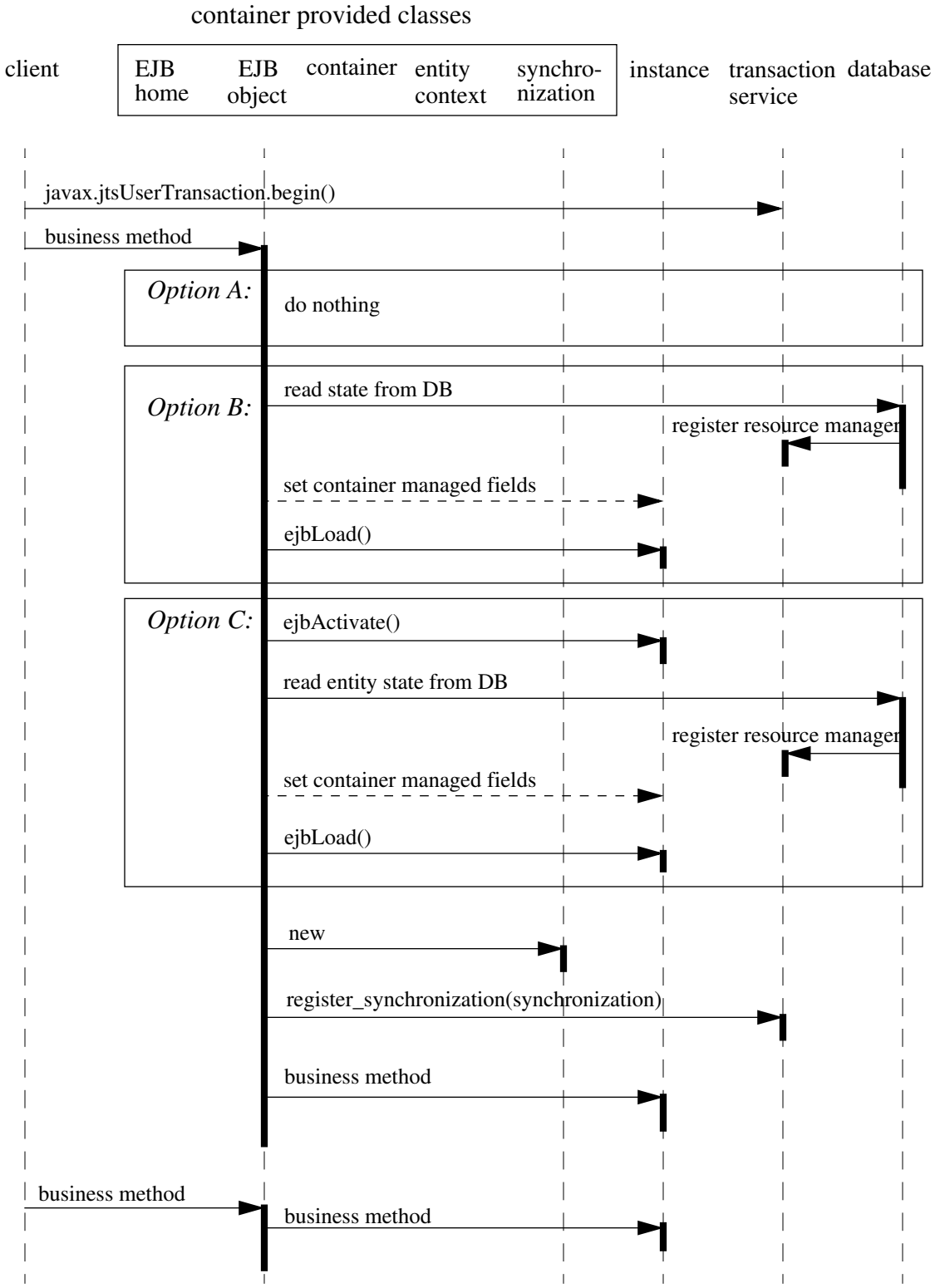


9.11.5 Starting the next transaction

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.

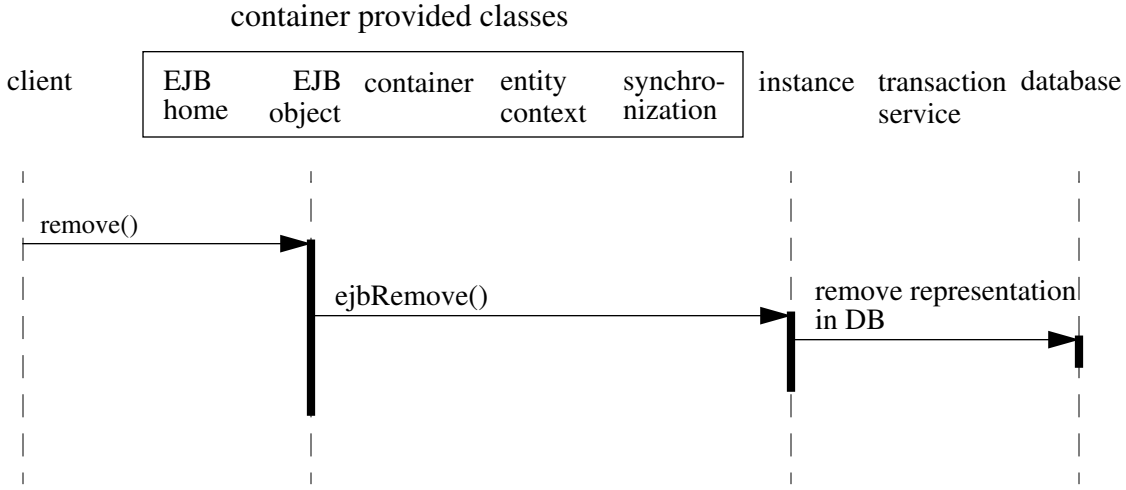


The following diagram illustrates the protocol performed for a Bean with container-managed persistence at the beginning of a new transaction.

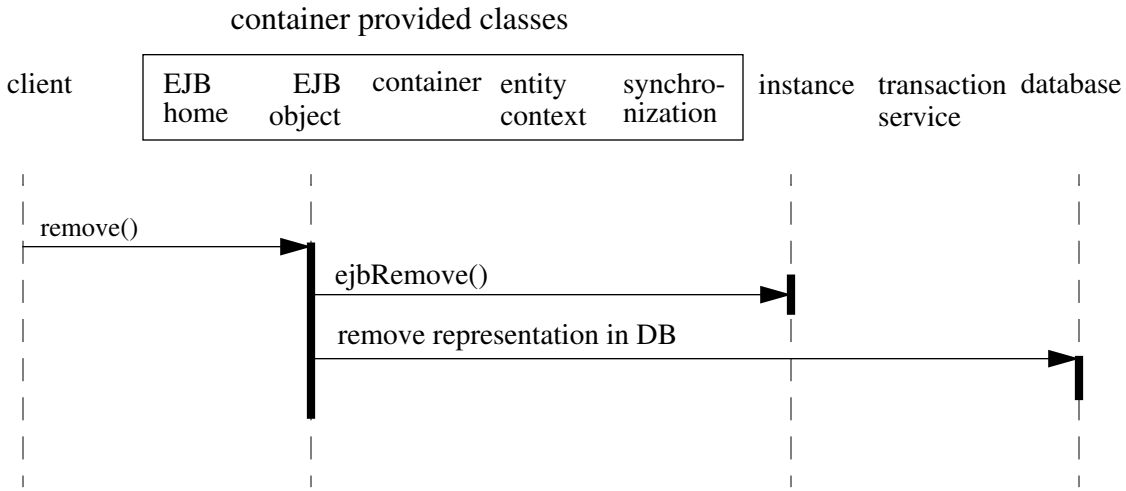


9.11.6 Removing an entity object

The following diagram illustrates the destruction of an entity enterprise Bean with Bean-managed persistence.

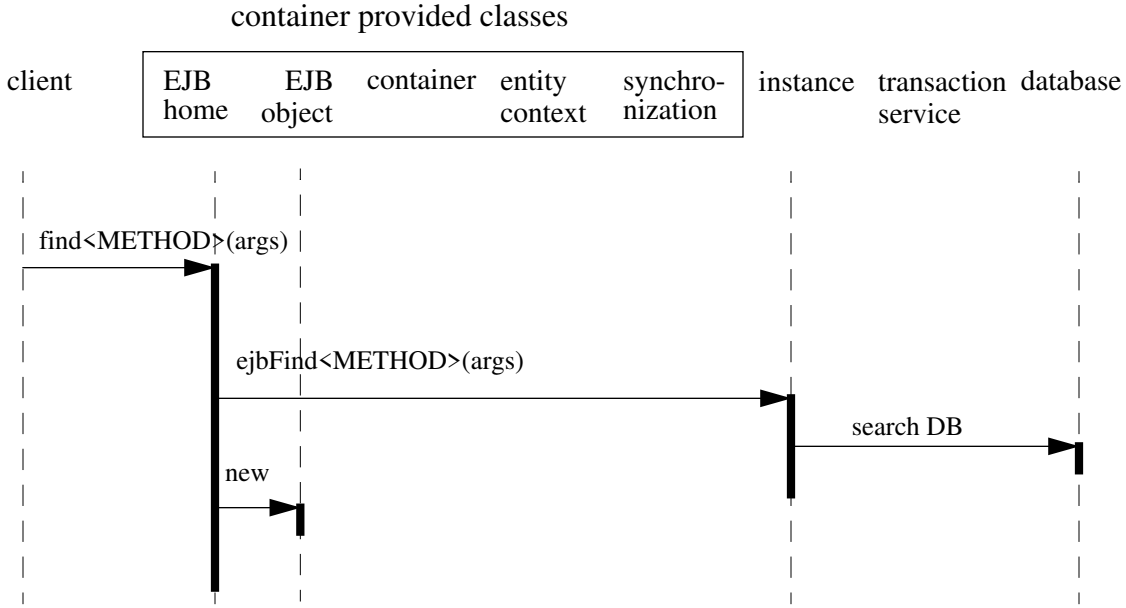


The following diagram illustrates the destruction of an entity enterprise Bean with container-managed persistence.

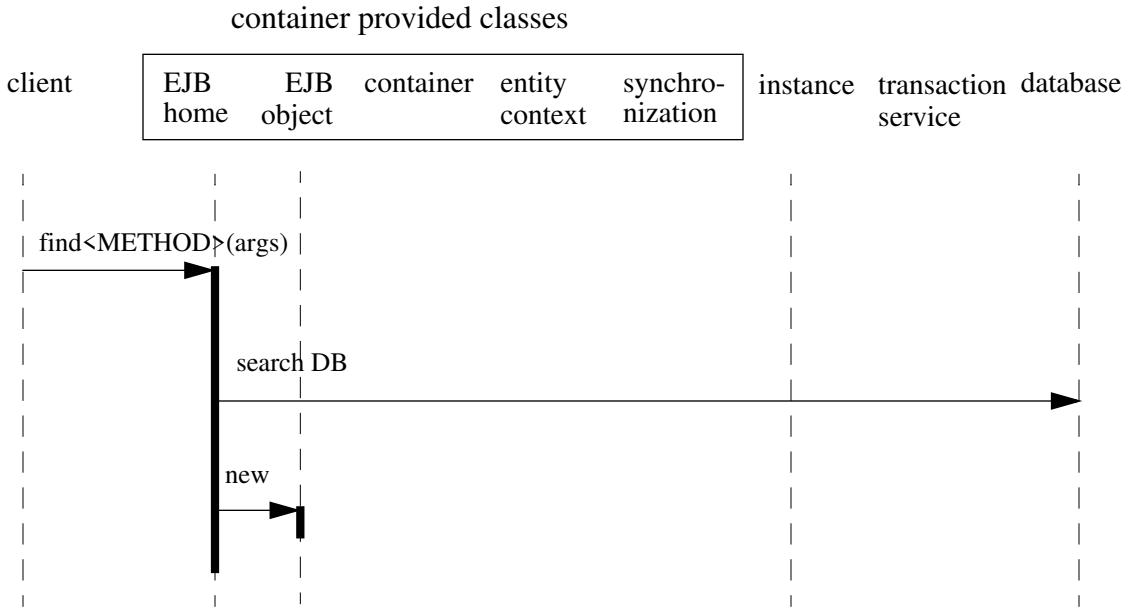


9.11.7 Finding an object

The following diagram illustrates the execution of a finder method on an entity enterprise Bean with Bean-managed persistence.



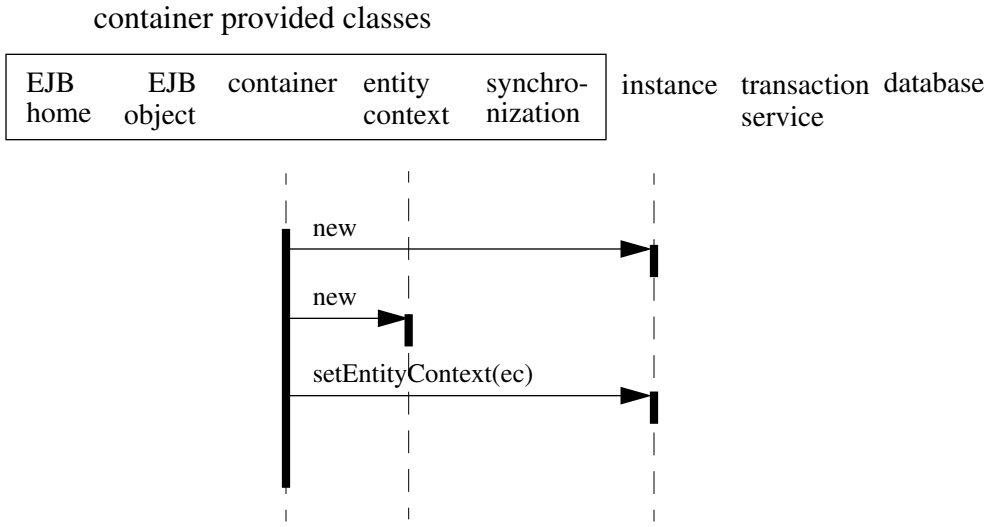
The following diagram illustrates the execution of a finder method on an entity enterprise Bean with container-managed persistence.



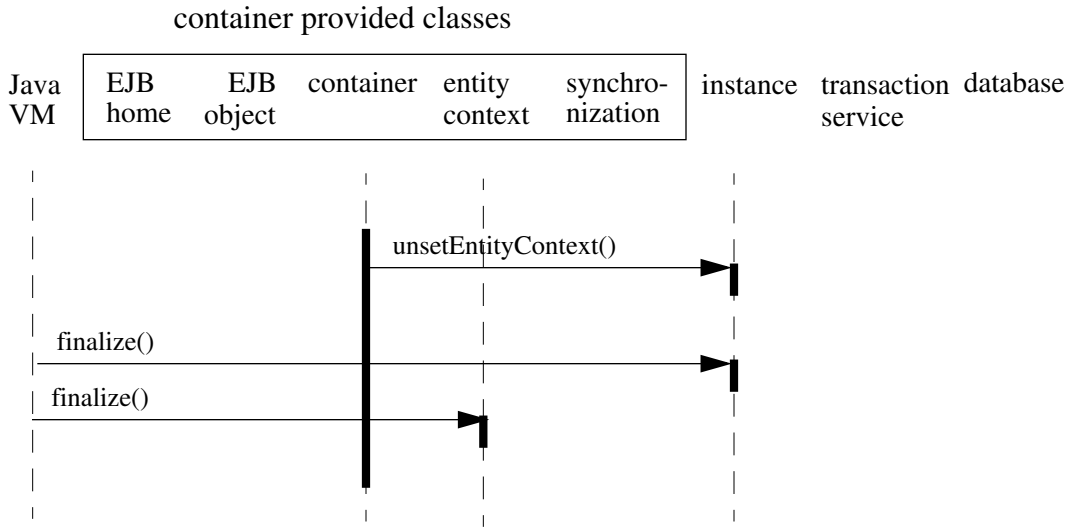
9.11.8 Adding and removing instance from the pool

The diagrams in Subsections 9.11.2 through 9.11.7 did not show the sequences between the “does not exist” and “pooled” state (See the diagram in Section 9.3).

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



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



10 Example entity scenario

Note: Container support for entity enterprise Beans is an optional feature for EJB 1.0 compliance. Container support for entity enterprise Beans will become mandatory in EJB 2.0.

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.

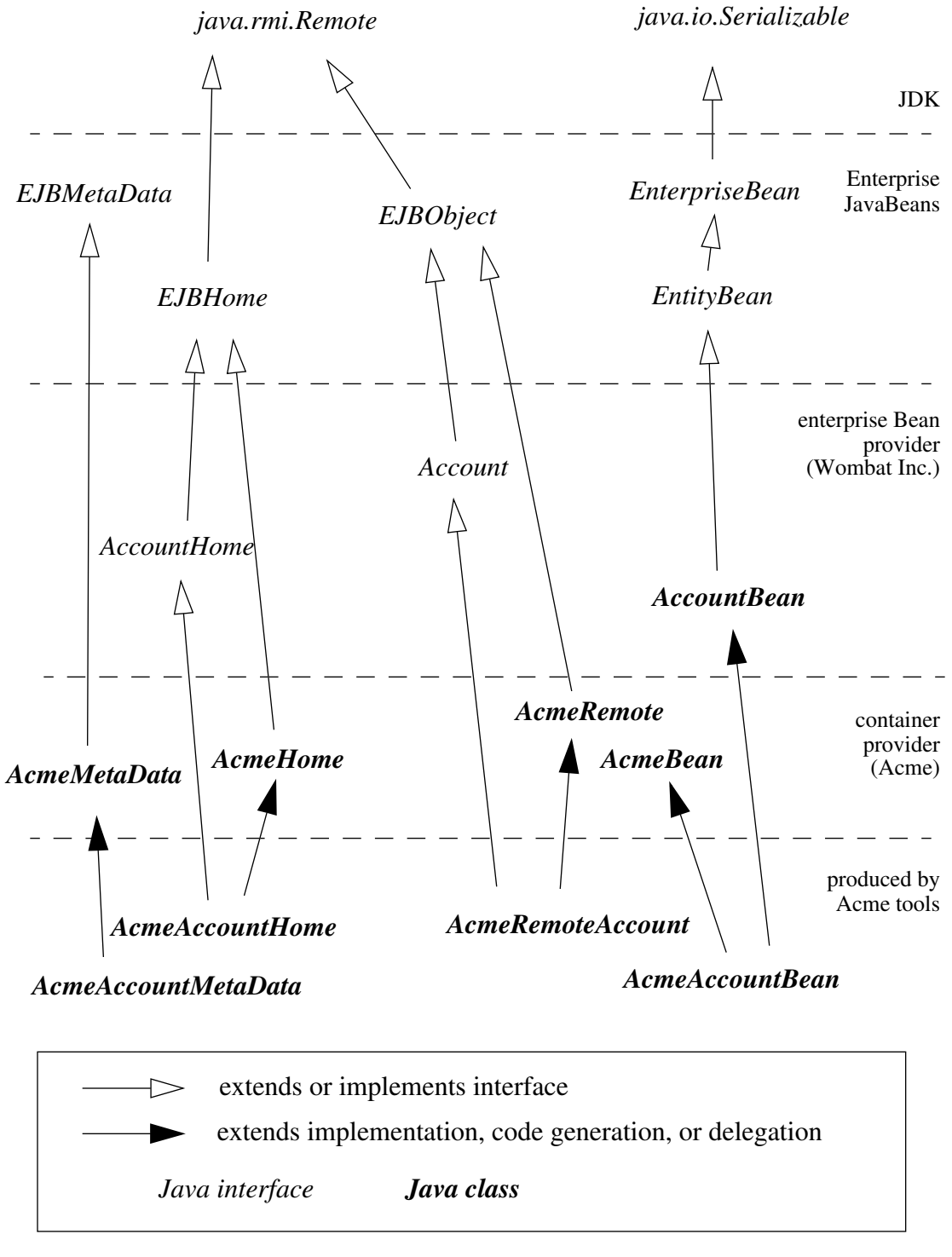
The classes generated by the container provider's tools in this scenario should be considered illustrative rather than prescriptive. Container providers are free to implement the contract between an enterprise Bean and its container in a different way that achieves an equivalent effect (from the perspectives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

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



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 Java RMI 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 Java RMI remote interfaces.
- Specify the environment properties that an enterprise Bean requires at runtime. The environment properties is a standard *java.util.Properties* file.
- 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's 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.

11 Support for transactions

One of the key features of Enterprise JavaBeans is support for distributed transactions. Enterprise JavaBeans allows an application developer to write an application that atomically updates data in multiple databases which are possibly distributed across multiple sites. The sites may use EJB servers and containers from different vendors.

No distinction is made between session and entity Beans in this section. This section applies equally to both.

An enterprise Bean developer or client programmer is not exposed to the complexity of distributed transactions. The burden of managing transactions is shifted to the container and EJB server providers. A container implements the declarative transaction scopes defined later in this chapter. The EJB server implements 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 Transaction model

Enterprise JavaBeans supports flat transactions, modeled after the OMG Object Transaction Service 1.1 (OTS). An enterprise Bean object that is *transaction-enabled* corresponds to the *TransactionalObject* described in OTS (a future release may allow an enterprise Bean to act as a recoverable object).

> *Note: The decision not to support nested transactions was intended to allow 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.*

11.2 Relationship to JTS

Enterprise JavaBeans is a high-level component framework that attempts to hide system complexity from the application developer. Therefore, most enterprise Beans and their clients do not need to access transaction management programmatically. The clients and Beans that have to programmatically control transaction scopes should use the *javax.jts.UserTransaction* interface that is defined as part of the Java Transaction Service (JTS) API. The *javax.jts.UserTransaction* interface is the only JTS interface that the EJB container provider must implement in order to support EJB.

Java Transaction Service (JTS) API defines all the Java programming language interfaces related to transaction management on the Java platform. Currently, JTS has the following parts:

- The package *javax.jts* defines the application-level demarcation interface. This packaged is intended to be used in the EJB environment, both by the Bean implementor (for TX_BEAN_MANAGED Beans) and by a client in the Java

programming language who wants to explicitly demarcate transaction boundaries. The *javax.jts* package defines interfaces that can be easily implemented by any existing transaction manager.

- A Java programming language mapping of the OTS 1.1 API. This is a Java programming language mapping of the OMG specification using the Java IDL mapping. This API is intended to be used by a CORBA programmer using the Java programming language to implement CORBA objects. Support for this API is not required in the EJB environment. Note that a vendor may choose to use this API as part of the interface between the EJB server and container, but the vendor is free to use another API as well.
- A Java programming language mapping of the standard X/Open XA interface. This is an API for attaching a resource manager (such as JDBC driver) to an external transaction manager. An EJB server and/or container could use this API to interface to the database drivers. This API is being proposed and reviewed as part of the JDBC 2.0 specification.

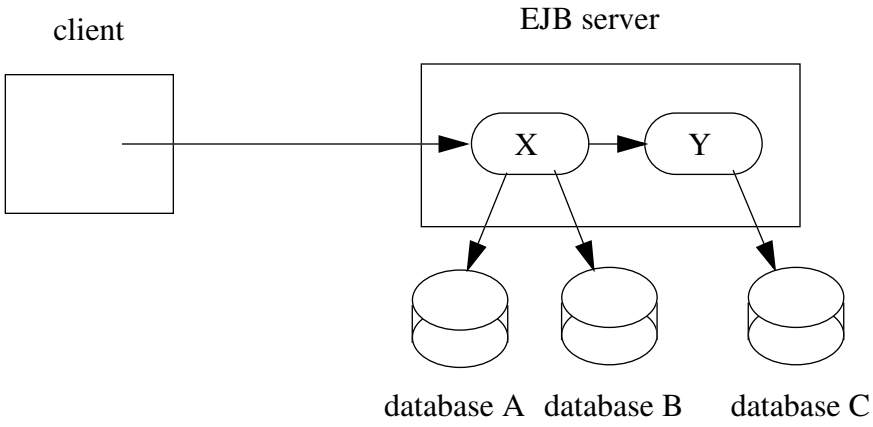
11.3 Scenarios

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

11.3.1 Update of multiple databases

Enterprise JavaBeans 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 in two 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.



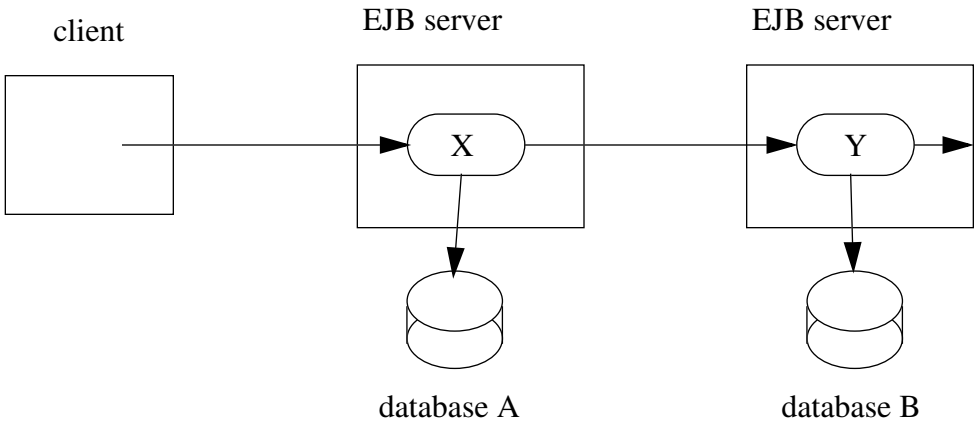
The application programmer does not have to do anything to ensure transactional semantics. The enterprise Beans X and Y perform the database updates using the standard

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.

11.3.2 Update of databases via multiple EJB servers

Enterprise JavaBeans 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. Enterprise JavaBeans makes it possible to perform the updates to databases A and B as a single transaction.



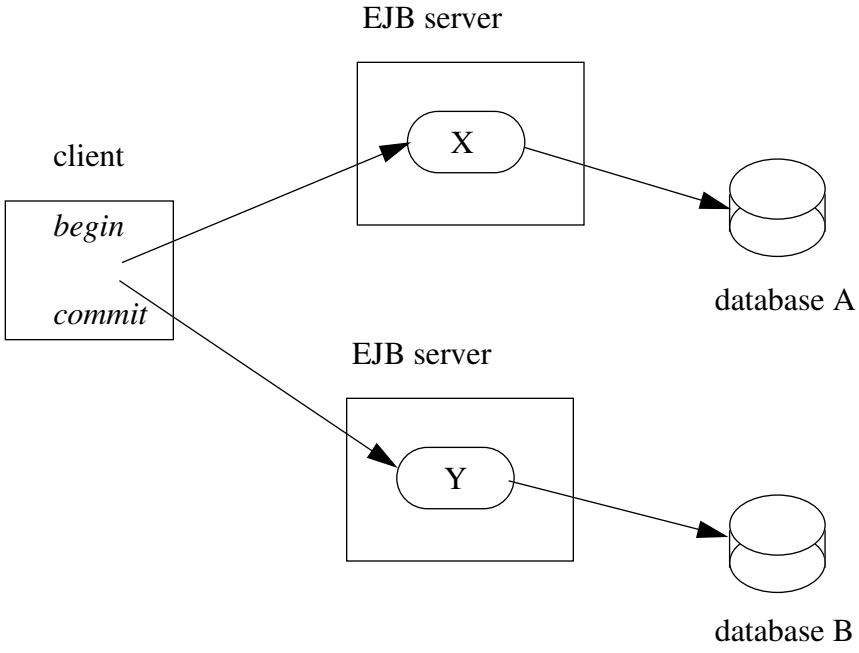
When X invokes Y, the two EJB servers cooperate to propagate the transaction context from X to Y. This transaction context propagation is transparent to the application-level code.

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.

11.3.3 Client-managed demarcation

A client or a non-transaction enterprise Bean object can use the *javax.jts.UserTransaction* interface to explicitly demarcate transaction boundaries.

A client program using explicit transaction demarcation may perform atomic updates across multiple databases residing at multiple transaction servers, as illustrated in the following figure.



The application programmer demarcates the transaction with *begin* and *commit* calls, and the EJB server ensures that the updates to databases A and B are transactional. A proxy of a transaction service on the client automatically propagates the transaction context to the two EJB servers. When the client program calls commit, the two EJB servers perform the two-phase commit protocol.

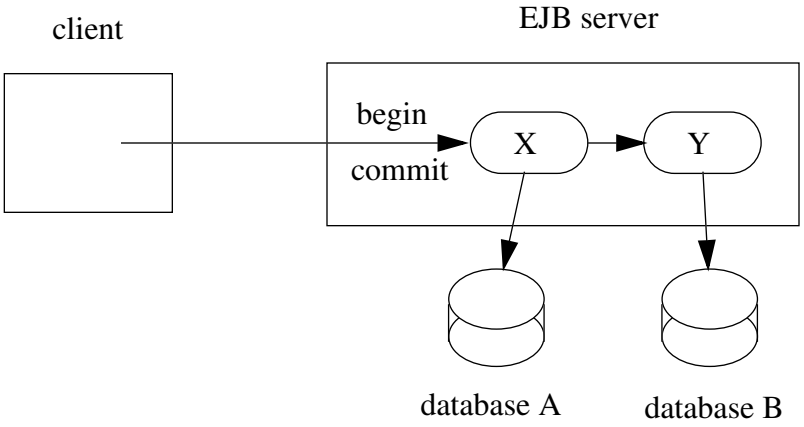
11.3.4 Container-managed demarcation

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*.

For example, if an enterprise Bean is deployed with the *TX_REQUIRED* transaction attribute, the container automatically initiates a transaction whenever a client invokes a transaction-enabled enterprise Bean while the client is not associated with a transaction context.

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 au-

tomatically included in the transaction (subject to the transaction attribute of the other enterprise Bean).



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

11.3.5 Bean-managed demarcation

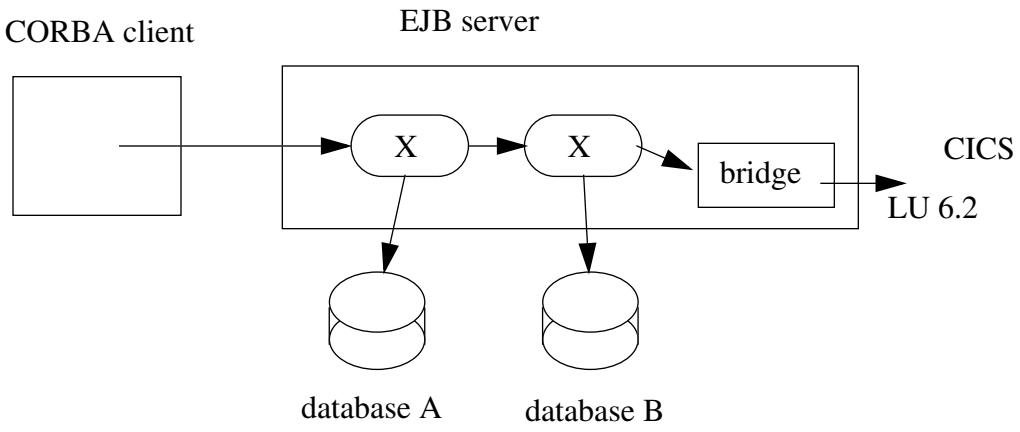
An enterprise Bean with the *TX_BEAN_MANAGED* transaction attribute can use the *javax.jts.UserTransaction* interface to demarcate transactions.

11.3.6 Interoperability with non-Java clients and servers

Although the focus of Enterprise JavaBeans is the Java API for writing distributed enterprise applications in Java, 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 Enterprise JavaBeans [6] allows any

CORBA client to invoke any enterprise Bean object on a CORBA-enabled server using the standard CORBA API.



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. Such bridges will be a value added by some EJB servers.

11.4 Declarative transaction management

Every client method invocation on an enterprise Bean object is interposed by the container. The interposition allows for delegating the transaction management responsibilities to the container.

The declarative transaction management is controlled by a *transaction attribute* associated with each enterprise Bean's home container. The container provider's tools can be used to set and change the values of transaction attributes.

Enterprise JavaBeans defines the following values for the transaction attribute:

- *TX_NOT_SUPPORTED*
- *TX_BEAN_MANAGED*
- *TX_REQUIRED*
- *TX_SUPPORTS*
- *TX_REQUIRES_NEW*

- *TX_MANDATORY*

The transaction attribute is specified in the enterprise Bean's deployment descriptor. A transaction attribute can be associated with the entire Bean (to apply to all methods), or it can be associated with an individual method. If method-level descriptors are used, they must follow the restrictions defined in Section 15.2.

11.4.1 TX_NOT_SUPPORTED

A container must always invoke an enterprise Bean that has the *TX_NOT_SUPPORTED* transaction attribute without a transaction scope. If a client calls with a transaction scope, the container suspends the association of the transaction scope with the current thread before delegating the method call to the enterprise Bean object. The container resumes the suspended association when the method call on the enterprise Bean object has completed.

The suspended transaction context of the client is not passed to resources or other enterprise Bean objects that are invoked from the enterprise Bean object.

11.4.2 TX_BEAN_MANAGED

An enterprise Bean with the *TX_BEAN_MANAGED* attribute can use the *javax.jts.UserTransaction* interface to demarcate transaction boundaries. The rules for TX_BEAN_MANAGED transactions are defined in Section 11.5.

11.4.3 TX_REQUIRED

If a client invokes an enterprise Bean object that has the *TX_REQUIRED* transaction attribute 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 object while the client is not associated with a transaction context, the container automatically starts a new transaction before delegating a method call to the enterprise Bean object, and attempts to commit the transaction when the method call on the enterprise Bean object has completed. The container performs the commit protocol before the method result is sent to the client.

The transaction context is passed to the resources or other enterprise Bean objects that are invoked from the enterprise Bean object.

11.4.4 TX_SUPPORTS

An enterprise Bean object that has the *TX_SUPPORTS* transaction attribute is invoked in the client's transaction scope. If the client does not have a transaction scope, the enterprise Bean is also invoked without a transaction scope.

The transaction context (if any) is passed to the resources or other enterprise Bean objects that are invoked from the enterprise Bean object.

11.4.5 TX_REQUIRES_NEW

An enterprise Bean that has the *TX_REQUIRES_NEW* transaction attribute is always invoked in the scope of a new transaction. The container starts a new transaction before

delegating a method call to the enterprise Bean object, and attempts to commit the transaction when the method call on the enterprise Bean object has completed. The container performs the commit protocol before the method result is sent to the client.

If the client request is associated with a transaction, the association is suspended before the new transaction is started and is resumed when the new transaction has completed.

The new transaction context is passed to the resources or other enterprise Bean objects that are invoked from the enterprise Bean object.

11.4.6 TX_MANDATORY

An enterprise Bean object that has the *TX_MANDATORY* attribute is always invoked in the scope of the client's transaction. If the client attempts to invoke the enterprise Bean without a transaction context, the container throws the *TransactionRequired* exception to the client.

The client's transaction context is passed to the resources or other enterprise Bean objects that are invoked from the enterprise Bean object.

11.4.7 Transaction attribute summary

The following table provides a summary of the transaction scopes under which a method on an enterprise Bean object method executes, as a function of the transaction attribute and client's transaction context.

Table 2: Effect of the declarative transaction attribute

Transaction attribute	Client's transaction	Transaction associated with enterprise Bean's method
TX_NOT_SUPPORTED	-	-
	T1	-
TX_REQUIRED	-	T2
	T1	T1
TX_SUPPORTS	-	-
	T1	T1
TX_REQUIRES_NEW	-	T2
	T1	T2
TX_MANDATORY	-	error
	T1	T1

A dash indicates that "no global transaction context exists or will be propagated". The container can execute the method outside of any transaction, or as a local transaction.

11.5 TX_BEAN_MANAGED transactions

An enterprise Bean with the *TX_BEAN_MANAGED* attribute is allowed to use the *javax.jts.UserTransaction* interface to demarcate transaction boundaries.

The container makes the *javax.jts.UserTransaction* interface available to the enterprise Bean though the *EJBContext.getUserTransaction()* method, as illustrated in the following example.

```
import javax.jts.UserTransaction;
...
EJBContext ic = ...;
...
UserTransaction tx = ic.getUserTransaction();
tx.begin();
...      // do work
tx.commit();
```

Enterprise Beans deployed with a transaction attribute other than *TX_BEAN_MANAGED* are not allowed to access directly the underlying transaction manager.The container makes the *javax.jts.UserTransaction* interface unavailable to these enterprise Beans.

11.5.1 Specification of TX_BEAN_MANAGED for stateful session Beans

The container must manage transactions on a TX_BEAN_MANAGED Bean as follows. When a client invokes a stateful TX_BEAN_MANAGED Bean, the container suspends any incoming transaction. The container allows the session instance to initiate a transaction using the *begin javax.jts.UserTransaction* interface. The instance becomes associated with the transaction and remains associated until the transaction terminates. When a Bean-initiated transaction is associated with the instance, methods on the instances run under that transaction.

It is possible that a business method that initiated the transaction completes without committing or rolling back the transaction. The container must retain the association between the transaction and the instance across multiple client calls until the transaction terminates.

The actions performed by the container are summarized by the following table.

Table 3: TX_BEAN_MANAGED session Bean

Client's transaction	Transaction currently associated with instance	Transaction associated with the method
-	-	-
T1	-	-
-	T3	T3
T1	T3	T3

The following explains the entries in the table in detail.

- 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, 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 when the method completes.
- If the client request is not associated with a transaction and the instance is already associated with a transaction, the container invokes the instance with the transaction that is associated with the instance.
- If the client is associated with a transaction, and the instance is already associated with a transaction, the container suspends the client's transaction association and invokes the method with the transaction context that is associated with the instance. The container resumes the client's transaction association when the method completes.

In all cases, if the instance is associated with a transaction when the method completes (i.e. the instance has no yet completed the transaction), the container suspends the transaction before returning control to the client.

It is legal for a Bean to perform serially several transactions in a method.

It is illegal for an instance to attempt to start a new transaction before the current one has completed.

11.5.2 Specification of TX_BEAN_MANAGED for stateless sessions and entity Beans

The specification of the behavior of TX_BEAN_MANAGED transactions for stateless session Beans and all entity Beans is identical to that for stateful session Beans, with one important exception. An instance of a stateless session Bean or an entity Bean is not allowed to retain an association with a transaction across multiple calls from a client. This means that if a business method initiates a transaction, the method must complete (commit or rollback) the transaction before it returns.

Associating a transaction with an instance across multiple calls from the client makes sense for a stateful session Bean because an instance of a session Bean is permanently associated with a single client. Retaining transaction association across multiple client calls would not make sense for entity Beans or stateless session Beans because their instances are, in general, shared across multiple clients.

11.6 Transaction isolation levels

The enterprise Bean provider must specify the requested transaction isolation level in the deployment descriptor. The possible isolation levels are:

- TRANSACTION_READ_UNCOMMITTED

- TRANSACTION_READ_COMMITTED
- TRANSACTION_REPEATABLE_READ
- TRANSACTION_SERIALIZABLE

These isolation levels correspond to the JDBC isolation levels[1], and are defined in the *javax.ejb.deployment.ControlDescriptor* class.

The container uses the transaction isolation level information provided in the deployment descriptor in the following way:

- For sessions Beans and entity Beans with Bean-managed persistence, the container ensures that the specified transaction isolation level is set on the database connections used by the Bean at the start of each transaction.
- For entity Beans with container-managed persistence, the database access calls generated by the container tools must achieve the specified isolation level.

11.7 Deployment descriptor restrictions

A transaction attribute and isolation level can be specified for the entire enterprise Bean, and for individual enterprise Bean's methods. If a value is specified at a method-level, the value takes precedence over the value specified at the Bean level.

This sections defines the restrictions on the use of the transaction attribute and isolation level at the method level.

11.7.1 TX_BEAN_MANAGED

The TX_BEAN_MANAGED transaction attribute value must not be mixed with the other values of the transaction attributes. This means that if the Bean-level descriptor or one of the method-level descriptors specifies the TX_BEAN_MANAGED attribute, then all method-level descriptors, if there are any, must specify TX_BEAN_MANAGED.

A container should detect this error at deployment time and alert the deployer.

11.7.2 Session Bean

If a client performs explicit transaction demarcation, the client must not invoke a sequence of methods on an instance that would lead to the situation that:

- the instance is associated with a transaction, and
- the container, using the rules for the transaction attribute described in this chapter, would be required to invoke a method on the Bean in a different transaction, or no transaction.

The container must detect such an attempt and throw the *java.rmi.RemoteException* to the client, and should log the error to alert the system administrator.

1.Note that EJB does not allow the JDBC TRANSACTION_NONE isolation level.

For example, it is an error for a client to begin a transaction, and then invoke *method1* and *method2* on a session Bean if *method1* is deployed with the TX_REQUIRED attribute and *method2* with TX_REQUIRES_NEW.

11.7.3 Isolation level

If method-level descriptors are used, the isolation level specified in the descriptors must be used consistently so that a method does not require a different isolation level than one that is associated with an instance. The container must detect such a condition and throw the *java.rmi.RemoteException* to the client. The container should log the error to alert the system administrator.

For example, assuming that an enterprise Bean's transaction attribute is TX_REQUIRED, it is an error for a client to begin a transaction, and invoke *method1* and *method2* on an enterprise Bean if the isolation levels specified for *method1* and *method2* are different.

This rule applies to both session and entity enterprise Beans.

11.8 Transaction management and exceptions

The EJB server and EJB container may throw the *TransactionRolledbackException*, *TransactionRequiredException*, and *InvalidTransactionException* exceptions in the situations defined in the JTS specification. See Appendix B for the reference pages of these exceptions.

12 Exception handling

This chapter describes the rules for exception handling.

12.1 Client's view of exceptions

A client accesses an enterprise Bean through the enterprise Bean's remote and home interfaces. Both of these are Java RMI interfaces. Therefore, the throws clause of every method of these interfaces includes the mandatory *java.rmi.RemoteException.* This exception is thrown to the client as an indication of a system-level failure.

The *java.rmi.RemoteException* may be thrown by the communication subsystem between the client and the container; by the container; or by the enterprise Bean. The container throws this exception to the client if it cannot complete a client's request because of an unexpected condition when delegating the client invocation to the enterprise Bean. The Bean throws this exception to indicate a system-level error (e.g. database error).

In addition to the mandatory *java.rmi.RemoteException* exception, the throws clause of the methods may include any number of application specific exceptions. These exceptions are thrown by the enterprise Bean, and passed unchanged by the container to the client.

The *javax.ejb.CreateException*, *javax.ejb.RemoveException*, and *javax.ejb.FindException* are standard application-level exceptions to report errors to the client from the *create*, *remove*, and *finder* methods.

12.1.1 Exceptions and transactions

If a client running in a transaction scope invokes an enterprise Bean business method, a *create* method, a *remove* method, or a *finder* method, and the method returns with an exception other than *javax.jts.TransactionRolledbackException*, the client can assume that the transaction has not been automatically marked for rollback. The client may attempt to recover the transaction, for example, by calling the enterprise Bean method again with different arguments, or by calling a different enterprise Bean.

The client can assume that the transaction has been marked for rollback if the exception is *javax.jts.TransactionRolledbackException* . It is fruitless for the client to continue the transaction because the transaction can never commit.

If the client receives the *java.rmi.RemoteException* exception other than the *javax.jts.TransactionRolledbackException* (note that *javax.jts.TransactionRolledbackException* is a subclass of *java.rmi.RemoteException*), the client, in general, does not know if the enterprise Bean's method has completed or not. Therefore, if a transactional client receives the *java.rmi.RemoteException* exception, the client should roll back the current transaction to prevent inconsistent data. Only expert-level clients should attempt to recover an *java.rmi.RemoteException* within a transaction.

Note that an enterprise Bean who is also a client of another enterprise Bean can use the *getRollbackOnly* method to test if the current transaction has been marked for rollback.

12.2 Rules for the enterprise Bean developer

12.2.1 Application-level exceptions

The throws clauses of an enterprise Bean's business methods, the *ejbCreate* methods, *ejbRemove* methods, and *ejbFind<METHOD>* methods, may include arbitrary application-level exceptions.

The *javax.ejb.CreateException*, *javax.ejb.RemoveException*, and *javax.ejb.FindException* are considered also application-level exceptions, and may be used in the throws clauses of the create, remove, and finder methods of the enterprise Bean class.

The application-level exceptions are meant to be thrown to the client to indicate an application-specific error condition (for example, exceeding a bank-imposed withdrawal limit on a checking account, or an attempt to create a duplicate account).

The enterprise Bean developer may assume that the container passes these exceptions unchanged to the client.

12.2.2 System-level exceptions

The enterprise Bean developer should throw the *java.rmi.RemoteException* from any of its method (business method, *ejbCreate*, *ejbRemove*, and any of the other container callback methods) to indicate an unexpected system-level failure (e.g. failure to open database connection).

> *EJB 1.0 does not standardize the system-level exceptions that a Bean instance can throw to its container.*

12.2.3 Marking a transaction for rollback

The enterprise Bean developer must assume that the container does not automatically rollback the client's transaction if the Bean throws an exception (this applies to both the application and system-level exceptions). Therefore, the enterprise Bean developer needs to ensure that the Bean is in a consistent state before throwing an exception.

If the Bean cannot ensure that is in an consistent state at the time it throws an exception, the Bean should use the *setRollbackOnly* method on the *EJBContext* interface to mark a transaction for rollback before it throws the exception. This mechanism allows the Bean to make an explicit decision whether a transaction should be rolled back or not.

12.3 Rules for the container provider

The container must handle exceptions thrown by the enterprise Bean's methods as follows.

12.3.1 Application-level exceptions

The container must pass all the application-level exceptions (i.e. all exceptions defined in a method's throws clause other than the *java.rmi.RemoteException*) thrown by the enterprise Bean's business methods, *ejbCreate*, *ejbPostCreate*, *ejbRemove*, and *ejbFind<METHOD>* methods to the client. This means, for example, that the container must not convert an application-level exception into the *java.rmi.RemoteException*.

12.3.2 System-level exceptions

The container should catch all *java.rmi.RemoteException* exceptions thrown by the enterprise Bean's methods. The container should log the exception to alert the system administrator of the problem. If the error prevents successful completion of a client's request, the container must throw the *java.rmi.RemoteException* to the client. The container may rethrow the exception thrown originally by the Bean, or throw a different exception (which must be *java.rmi.RemoteException* or a subclass thereof).

12.3.3 Unchecked exceptions

The container must catch all unchecked exceptions thrown by the enterprise Bean's methods. The container should log the exception, and must throw the *java.rmi.RemoteException* to the client. The container must assume that the instance is in an undefined state, and must not use the instance for further requests. If the instance participated in a transaction, the container must roll back the transaction, or mark it for rollback.

12.3.4 Exceptions and transactions

In general, a checked exception (both application and system-level) thrown by an enterprise Bean method should not cause the container to automatically rollback a transaction or mark the transaction for rollback. This allows the client to recover from the exception.

If the container decides for any reason to mark a transaction for rollback, it should throw the *javax.jts.TransactionRolledbackException* to the client. The *javax.jts.TransactionRolledbackException* is a subclass of the *java.rmi.RemoteException*, and it informs the client that any attempted recovery of the exception within the transaction would be fruitless since the transaction cannot commit.

There are four cases that require the container to rollback a transaction after an instance method execution resulted in an exception:

- If the enterprise Bean's method that threw the exception executed in a transaction that was automatically started by the container before dispatching the method (Section 11.4 explains when the container automatically starts a transaction before calling an enterprise Bean's method), the container must rollback the transaction when it catches any exception (including application-level exceptions) from the enterprise Bean's method before it throws an exception to the client (the exception thrown to the client is determined using the rules in the previous subsection). If the client is associated with a transaction, the client's transaction is not marked for rollback because the instance executed in a different transaction (note that this case can happen only if the Bean is deployed with the TX_REQUIRES_NEW transaction attribute).
- If an instance of a TX_BEAN_MANAGED entity bean or stateless session bean throws an exception while the instance is associated with a transaction, the container must roll back the transaction before throwing the appropriate exception to the client.

- If an instance has thrown an unchecked exception while executing in a client's transaction context, the container must mark the transaction for rollback and throw *javax.jts.TransactionRolledbackException* to the client.
- If a TX_BEAN_MANAGED bean instance throws an unchecked exception while the instance is associated with a transaction, the container must roll back the transaction, and throw the *javax.jts.RemoteException* to the client.

13 Support for distribution

13.1 Overview

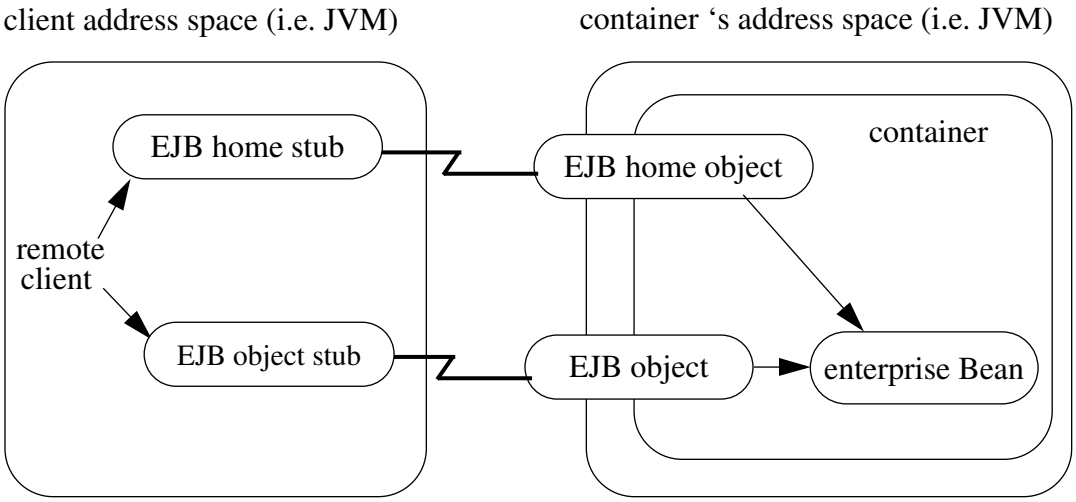
Support for remote client access to an enterprise Bean object is through the standard Java API for remote method invocation (Java RMI) [3]. This API allows a client to invoke an enterprise Bean object using any distributed object protocol, including the industry standard IIOP protocol, as defined in the OMG Java to IDL Mapping specification [5].

The Java RMI API makes access to an enterprise Bean object *location transparent* to a client programmer.

13.2 Client-side objects

The following objects are present in the client's JVM:

- A local for the EJB object.
- A stub for the enterprise Bean's home object.



The EJB home object, and the EJB object are Java RMI remote objects. The Java RMI specification [3] and the OMG Java to IDL Mapping specification [5] define the stubs for the factory, container, and EJB objects, and the communication between the stubs and the objects on the server.

The communication stubs and skeletons are generated at enterprise Bean's deployment time by the EJB container provider tools.

13.3 Interoperability via network protocol

13.3.1 Mapping to CORBA

The standard mapping of Enterprise JavaBeans to CORBA is defined in [6].

The mapping enables the following interoperability:

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

13.3.2 Support for other protocols

Other forms of distributions are possible. For example, a client may use HTTP to invoke a servlet that invokes an enterprise Bean object through the EJB object and home interfaces.

A container may also provide additional client's view API for the installed enterprise Beans. For example, a container may choose to expose the installed enterprise Beans to OLE Automation clients, such as Visual Basic or Visual Basic Scripting engine. The mapping to protocols other than IIOP is not covered by the current EJB specification.

14 Support for security

The Enterprise JavaBeans architecture makes it possible to shift most of the burden of implementing security management from the enterprise Bean to the EJB container and server.

Support for security in Enterprise JavaBeans includes the following components:

- Existing Java programming language security APIs defined in the core package *java.security.*
- Security-related methods in the *javax.ejb.EJBContext* interface.
- Security-related attributes in the deployment descriptor.

The following sections describe the support for security in more detail.

14.1 Package *java.security*

The package *java.security* provides the generic Java programming language security-related interfaces. The Enterprise JavaBeans architecture uses the applicable existing Java programming language security APIs.

More specifically, the Enterprise JavaBeans architecture uses the *java.security.Identity* class as the API to describe a user identity for security purposes. An instance of *java.security.Identity* can describe a specific user, or a security role.

Please refer to the JDK reference page for the description of the *java.security.Identity* class.

The EJB container is responsible for mapping the instances of the *java.security.Identity* class to the user accounts and/or roles of the EJB server (i.e. user accounts and roles defined and managed by the underlying platform). This mapping is done in a platform-specific way.

14.2 Security-related methods in *EJBContext*

The Enterprise JavaBeans API allows an enterprise Bean instance to obtain the identity of the client that invoked the current method. For this purpose, the *javax.ejb.EJBContext* interface contains the following methods:

- *getCallerIdentity()*
- *isCallerInRole(Identity ident)*

Please refer to reference page of *javax.ejb.EJBContext* for the description of these methods.

The following examples illustrates how an enterprise Bean instance can obtain the identity of the client that invoked the current method:

```
/* Obtain the security identity of the client. */
Identity caller = EJBContext.getCallerIdentity();

/* getName returns a printable representation of identity. */
```

```
String clientAccount = caller.getName();
```

The following examples illustrates how an enterprise Bean instance can test whether the invoker of the method has a specified security role.

```
/*
* Check if the client has the "vip-account" role
*/
Identity vipAccount = new Identity("vip-account");

if (EJBContext.isCallerInRole(vipAccount)) {
        do something;
} else {
        do something else;
}
```

14.3 Security-related deployment descriptor properties

This section describes the declarative security information passed in the deployment descriptor in the ejb-jar file. This information is set by the enterprise Bean provider, and read and interpreted by the container tools at deployment time.

14.3.1 Access control entries

An enterprise Bean's deployment descriptor includes access control entries that allow the container to perform runtime security management on behalf of the enterprise Bean.

The enterprise Bean descriptor allows the Bean provider to specify an *AccessControlEntry* for an individual method and/or for the entire Bean. If an *AccessControlEntry is* specified for the entire Bean, it applies to all the methods that do not have an individual *AccessControlEntry.*

An *AccessControlEntry* associates a method with a list of entries of the type *java.security.Identity*. These security identities are users or roles that are allowed to invoke the method.

The container imports the *AccessControlEntries* from the deployment descriptor at deployment time and enforces them at runtime. The container typically allows the deployer and system administrator to modify the *AccessControlEntries* using container-provided tools.

14.3.2 *RunAsMode* and *RunAsIdentity*

An enterprise Bean's deployment descriptor allows the Bean provider to specify the security identity to be associated with the execution of the enterprise Bean's methods. At runtime, when an instance of the enterprise Bean makes a call to an underlying resource manager (e.g. a database access call) or invokes another enterprise Bean, this security identity will be associated with the call.

The identity is referred to as "RunAs" security identity. It consists of two parts:

- *RunAsMode* that specifies whether a method should execute with the security identity of the client (CLIENT_IDENTITY), the identity of a privileged system account (SYSTEM_IDENTITY), or with the identity of a specified user account (SPECIFIED_IDENTITY).
- *RunAsIdentity* that specifies the user identity if the value of *RunAsMode* is equal to SPECIFIED_IDENTITY. This value is ignored if the value of *RunAsMode* is CLIENT_IDENTITY or SYSTEM_IDENTITY.

The EJB container maps the SYSTEM_IDENTITY to a privileged account of the underlying platform in a platform specific way.

The deployment descriptor allows the Bean provider to specify the RunAs security identity at the level of the entire Bean, or at the level of individual methods. The value specified at the Bean-level applies to all the methods that do not have a method-level security identity specified.

While the method-level RunAs security identity may be different for different methods of the same Bean, the following restrictions must be observed in their use.

- For a stateful session Bean, the RunAs security identity associated with an instance is determined at instance creation time. This security identity remains associated with the instance for the session lifetime. This means that the security identity associated with the *ejbCreate(...)* method will apply not only to the *ejbCreate(...)* method itself, but also to all the methods invoked subsequently on the instance. It is required that the invoked methods do not specify a conflicting RunAs security identity.
- For stateless session Beans and entity Beans, it is required that all methods invoked on a Bean instance in the same transaction be executed with the same RunAs identity. This means that the specified RunAs identity for all the methods that are executed in the same transaction must be the same.

These restrictions must be observed by the Bean provider and the application assembler. If an EJB container detects violation of the rules, it should throw the *java.rmi.RemoteException* to the client, and log the error to alert the system administrator.

15 Ejb-jar file

Enterprise JavaBeans defines the format for the packaging of enterprise Beans. The packaging format can be used both for distribution of individual enterprise Beans as components, and for distribution of an entire server-side application built of multiple enterprise Beans.

15.1 ejb-jar file

Enterprise Beans are packaged for deployment in a standard Java programming language Archive File called an *ejb-jar* file.

An ejb-jar file contains the enterprise Beans' class files and their deployment descriptors. The ejb-jar file's manifest file identifies the enterprise Beans that are included in the file.

15.2 Deployment descriptor

An enterprise Bean provider must include a deployment descriptor for each enterprise Bean. A deployment descriptor is a serialized instance of a *javax.ejb.deployment.EntityDescriptor* or *javax.ejb.deployment.SessionDescriptor* object. Please refer to the reference pages for information on deployment descriptors.

15.3 ejb-jar Manifest

An ejb-jar file must include a *manifest file*. The manifest file identifies the enterprise Beans included in the ejb-jar file.

The manifest file must be named "META-INF/MANIFEST.MF".

The manifest file is organized as a sequence of *sections*. Sections are separated by empty lines. Each section contains one or more *headers*, each of the form *<tag>: <value>*. The sections that provide information on enterprise Beans in the archive use headers with the following *<tags>*:

- **Name**, whose *<value>* is the relative name of the enterprise Bean's serialized deployment descriptor.
- **Enterprise-Bean**, whose *<value>* is **True**.

Every enterprise Bean must have a section in the manifest file. The headers with the **Name** and **Enterprise-Bean** *<tags>* are mandatory for all enterprise Beans.

For example, two relevant sections of an ejb-jar manifest might be:

```
Name: bank/AccountDeployment.ser
Enterprise-Bean: True

Name: quotes/QuoteServerDeployment.ser
Enterprise-Bean: True
```

16 Enterprise Bean provider responsibilities

16.1 Classes and interfaces

The enterprise Bean provider is responsible for the following classes and interfaces:

- The enterprise Bean class.
- The enterprise Bean's remote interface.
- The enterprise Bean's home interface.

The requirements for these classes and interfaces are specified in Sections 6.9, 9.7, and 9.10.

Furthermore, the Java programming language types used for the arguments, return value, and exceptions of the enterprise Bean's remote interface and enterprise Bean's home interface must be valid types in the Java to IDL Mapping specification [5].

16.2 Environment properties

If the enterprise Bean depends on certain environment properties, the enterprise Bean provider must provide the environment properties for the Bean. Environment properties are defined as a standard *java.util.Properties* object.

The enterprise Bean provider must define all the *key:value* pairs that the enterprise Bean's instances will require at runtime. The values are typically edited at deployment time by the container provider tools.

16.3 Deployment descriptor

The enterprise Bean provider must provide a deployment descriptor for every enterprise Bean. The format of a deployment descriptor is described in Section 15.2.

16.4 Programming restrictions

NOTE: this is only a partial list of restrictions that the enterprise developer must observe.

The enterprise Bean developer must follow these restriction when implementing the methods of the enterprise Bean's class:

- An enterprise Bean is not allowed to start new threads or attempt to terminate the running thread.
- An enterprise Bean is not allowed to use read/write *static* fields. Using read-only *static* fields is allowed. Therefore, all *static* fields must be declared as *final*.
- An enterprise Bean is not allowed to use thread synchronization primitives.
- An enterprise Bean is not allowed to use the calls to an underlying transaction manager directly. The only exception are enterprise Beans with the *TX_BEAN_MANAGED* transaction attribute which are allowed to use the *javax.jts.UserTransaction* interface to demarcate transactions.

- An enterprise Bean is not allowed to change its *java.security.Identity*. Any such attempt will result in the *java.security.SecurityException* being thrown.
- A transaction-enabled enterprise Bean using JDBC is not allowed to use the *commit* and *rollback* methods. An enterprise Bean that is not transaction-enabled is allowed to use the *commit* and *rollback* methods.

16.5 Component packaging responsibilities

The enterprise Bean provider is responsible for putting the following classes and files in the ejb-jar file:

- The enterprise Bean class with any classes that the enterprise Bean depends on.
- The deployment descriptor file that contains the deployment attributes for the enterprise Bean.
- The enterprise Bean's remote interface with any classes that the interface depends on.
- The enterprise Bean's home interface with any classes that the interface depends on.
- Enterprise Bean's environment properties.
- The Manifest file that identifies the deployment descriptors of all the enterprise Beans in the ejb-jar file.

17 Container provider responsibilities

17.1 Enterprise Bean deployment tools

17.1.1 Tools to read ejb-jar

The container must include tools that support deployment of enterprise Beans packaged in the ejb-jar file format.

The tools must discover all the enterprise Beans that are in the JAR file by reading the ejb-jar Manifest file. The Manifest file provides the relative name of the serialized deployment descriptors (i.e. .ser files).

For each enterprise Bean in the ejb-jar file, the tools must:

- Deserialize the deployment descriptor, and read the information contained in the Bean's deployment descriptor using the getter methods. The information provides the default setting for the enterprise Bean's declarative attributes, such as transaction and security attributes. The deployment descriptor also includes the initial value of the Bean's environment properties, and provides the class names for the enterprise Bean class, remote and home interfaces, and the class name of the primary key type.
- Generate the container-specific classes as specified in the Sections 6.11 and 9.8.
- Generate the classes for stubs and skeletons used by the underlying distributed objects protocol.
- Make the enterprise Bean's home interface available in JNDI for clients to be able to find and access the enterprise Bean.
- Make the enterprise Bean's environment properties available to the Bean instances at runtime.

17.1.2 Tools to manage deployment descriptor attributes

The EJB container may provide tools that allow the EJB deployer to modify the information imported from the enterprise Bean's deployment descriptor. In certain scenarios, the tools may restrict the deployer from changing some or all deployment descriptor attributes. The EJB specification does not specify which attributes can or cannot be changed at deployment time.

17.1.3 Tools to customize business logic

The EJB container may provide tools that allow the EJB deployer to customize business logic of the deployed enterprise Beans. For example, the tools may allow the deployer to write *wrapper* functions for the business methods. To allow maximum freedom for the tool vendors, the EJB specification does not architect the customization.

17.1.4 Tools for container-managed persistence

The EJB containers that support container-managed persistence should provide tools that allow the deployer to map the container-managed fields to an enterprise's existing

data source or application system. These tools are typically specific to the legacy data source or application system.

17.2 Runtime infrastructure

The EJB container must provide the runtime infrastructure that complies with the EJB specification. In particular, the container must:

- Implement the Session protocol described in Chapters 5 and 6.
- Implement the Entity protocol (if the container supports entities) described in Chapters 8 and 9.
- Implement the support for transactions described in Chapter 11.
- Implement the support for security described in Chapter 14.
- Handle exceptions as described in Chapter 12.
- Make the enterprise Bean's environment properties available to the Bean instances at runtime through the *javax.ejb.EJBContext* interface.
- Should implement the EJB to CORBA mapping (if the container uses IIOP as the distributed object protocol) defined in [6].

17.3 Runtime management tools

The container should provide tools that allow runtime management and monitoring of the enterprise Beans running in the container.

17.4 Evolution management tools

The container should provide tools that allow the deployer and system administrator to manage evolution of the enterprise Beans' implementation. The tools should make it possible, for example, to upgrade the business logic implemented by an enterprise Bean by installing a new version of the enterprise Bean class.

18 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 SessionBean
public interface SessionContext
public interface SessionSynchronization
```

Classes:

```
public class CreateException
public class DuplicateKeyException
public class FinderException
public class ObjectNotFoundException
public class RemoveException
```

package *javax.ejb.deployment*:

Classes:

```
public class AccessControlEntry
public class ControlDescriptor
public class DeploymentDescriptor
public class EntityDescriptor
public class SessionDescriptor
```

Interface EJBContext

```
public interface javax.ejb.EJBContext
{
    public abstract Identity
        getCallerIdentity();
    public abstract EJBHome getEJBHome();
    public abstract Properties
        getEnvironment();
    public abstract boolean
        getRollbackOnly();
    public abstract UserTransaction
        getUserTransaction();
    public abstract boolean
        isCallerInRole(Identity role);
    public abstract void setRollbackOnly();
}
```

The EJBContext interface provides an instance with access to the container-provided runtime context of an enterprise Bean instance.

This interface is extended by the SessionContext and EntityContext interface to provide additional methods specific to the enterprise Bean type.

Methods

- **getCallerIdentity**

  ```
  public abstract Identity getCallerIdentity()
  ```

  Obtain the java.security.Identity of the caller.

  Returns:
  The Identity object that identifies the caller.

- **getEJBHome**

  ```
  public abstract EJBHome getEJBHome()
  ```

  Obtain the enterprise bean's home interface.

  Returns:
  The enterprise bean's home interface.

- **getEnvironment**

  ```
  public abstract Properties getEnvironment()
  ```

  Obtain the enterprise bean's environment properties.

  Note: If the enterprise bean has no environment properties this method returns an empty java.util.Properties object. This method never returns null.

  Returns:
  The environment properties for the enterprise bean.

- **`getRollbackOnly`**

  ```
  public abstract boolean getRollbackOnly()
  ```

  Test if the transaction has been marked for rollback only. An enterprise bean instance can use this operation, for example, to test after an exception has been caught, whether it is fruitless to continue computation on behalf of the current transaction.

  Returns:
  True if the current transaction is marked for rollback, false otherwise.

- **`getUserTransaction`**

  ```
  public abstract UserTransaction getUserTransaction()
       throws IllegalStateException
  ```

  Obtain the transaction demarcation interface.

  Returns:
  The UserTransaction interface that the enterprise bean instance can use for transaction demarcation.

  Throws: IllegalStateException
  Thrown if the instance container does not make the UserTransaction interface available to the instance (only the enterprise beans with the TX_BEAN_MANAGED transaction attribute are allowed to use the UserTransaction interface).

- **`isCallerInRole`**

  ```
  public abstract boolean
  isCallerInRole(Identity role)
  ```

  Test if the caller has a given role.

  Parameters:
  `role`
  The java.security.Identity of the role to be tested.

  Returns:
  True if the caller has the specified role.

- **`setRollbackOnly`**

  ```
  public abstract void setRollbackOnly()
  ```

  Mark the current transaction for rollback. The transaction will become permanently marked for rollback. A transaction marked for rollback can never commit.

Interface EJBHome

```
public interface javax.ejb.EJBHome
    extends java.rmi.Remote
{
    public abstract EJBMetaData
        getEJBMetaData();
    public abstract void
        remove(Handle handle);
    public abstract void
        remove(Object primaryKey);
}
```

The EJBHome interface is extended by all enterprise Bean's home interfaces. An enterprise Bean's home interface defines the methods that allow a client to create, find, and remove EJB objects.

Each enterprise Bean has a home interface. The home interface must extend the javax.ejb.EJBHome interface, and define the enterprise Bean type specific create and finder methods (session Beans do not have finders).

The home interface is defined by the enterprise Bean provider and implemented by the enterprise Bean container.

Methods

- **getEJBMetaData**

```
public abstract EJBMetaData getEJBMetaData()
     throws RemoteException
```

Obtain the EJBMetaData interface for the enterprise Bean. The EJBMetaData interface allows the client to obtain information about the enterprise Bean.

The information obtainable via the EJBMetaData interface is intended to be used by tools.

Returns:
The enterprise Bean's EJBMetaData interface.

Throws: RemoteException
Thrown when the method failed due to a system-level failure.

- **remove**

```
public abstract void remove(Handle handle)
     throws RemoteException, RemoveException
```

Remove an EJB object identified by its handle.

Throws: RemoveException
Thrown if the enterprise Bean or the container does not allow the client to remove the object.

Throws: RemoteException
Thrown when the method failed due to a system-level failure.

- **remove**

```
public abstract void remove(Object primaryKey)
     throws RemoteException, RemoveException
```

Remove an EJB object identified by its primary key.

Throws: RemoveException
Thrown if the enterprise Bean or the container does not allow the client to remove the object.

Throws: RemoteException
Thrown when the method failed due to a system-level failure.

Interface EJBMetaData

```
public interface javax.ejb.EJBMetaData
{
    public abstract EJBHome getEJBHome();
    public abstract Class
        getHomeInterfaceClass();
    public abstract Class
        getPrimaryKeyClass();
    public abstract Class
        getRemoteInterfaceClass();
    public abstract boolean isSession();
}
```

The EJBMetaData interface allows a client to obtain the enterprise Bean's meta-data information.

The meta-data is intended for development tools used for building applications that use deployed enterprise Beans, and for clients using a scripting language to access the enterprise Bean.

Note that the EJBMetaData is not a remote interface. The class that implements this interface (this class is typically generated by container tools) must be serializable, and must be a valid RMI/IDL value type.

Methods

- **getEJBHome**

  ```
  public abstract EJBHome getEJBHome()
  ```

  Obtain the home interface of the enterprise Bean.

- **getHomeInterfaceClass**

  ```
  public abstract Class getHomeInterfaceClass()
  ```

  Obtain the Class object for the enterprise Bean's home interface.

- **getPrimaryKeyClass**

  ```
  public abstract Class getPrimaryKeyClass()
  ```

  Obtain the Class object for the enterprise Bean's primary key class.

- **getRemoteInterfaceClass**

  ```
  public abstract Class getRemoteInterfaceClass()
  ```

  Obtain the Class object for the enterprise Bean's remote interface.

- **isSession**

  ```
  public abstract boolean isSession()
  ```

  Test if the enterprise Bean's type is "session".

  Returns:

  True if the type of the enterprise Bean is session.

Interface EJBObject

```
public interface javax.ejb.EJBObject
    extends java.rmi.Remote
{
    public abstract EJBHome getEJBHome();
    public abstract Handle getHandle();
    public abstract Object getPrimaryKey();
    public abstract boolean
        isIdentical(EJBObject obj);
    public abstract void remove();
}
```

The EJBObject interface is extended by all enterprise Bean's remote interface. An enterprise Bean's remote interface provides the client's view of an EJB object. An enterprise Bean's remote interface defines the business methods callable by a client.

Each enterprise Bean has a remote interface. The remote interface must extend the javax.ejb.EJBObject interface, and define the enterprise Bean specific business methods.

The enterprise Bean's remote interface is defined by the enterprise Bean provider and implemented by the enterprise Bean container.

Methods

- **getEJBHome**

```
public abstract EJBHome getEJBHome()
     throws RemoteException
```

Obtain the enterprise Bean's home interface. The home interface defines the enterprise Bean's create, finder, and remove operations.

Returns:
A reference to the enterprise Bean's home interface.

Throws: RemoteException
Thrown when the method failed due to a system-level failure.

- **getHandle**

```
public abstract Handle getHandle()
     throws RemoteException
```

Obtain a handle for the EJB object. The handle can be used at later time to re-obtain a reference to the EJB object, possibly in a different Java Virtual Machine.

Returns:
A handle for the EJB object.

Throws: RemoteException
Thrown when the method failed due to a system-level failure.

- **getPrimaryKey**

```
public abstract Object getPrimaryKey()
     throws RemoteException
```

Obtain the primary key of the EJB object.

Returns:
The EJB object's primary.

- **`isIdentical`**

```
public abstract boolean isIdentical(EJBObject obj)
     throws RemoteException
```

Test if a given EJB object is identical to the invoked EJB object.

Parameters:
`obj`
An object to test for identity with the invoked object.

Returns:
True if the given EJB object is identical to the invoked object, false otherwise.

Throws: RemoteException
Thrown when the method failed due to a system-level failure.

- **`remove`**

```
public abstract void remove()
     throws RemoteException, RemoveException
```

Remove the EJB object.

Throws: RemoteException
Thrown when the method failed due to a system-level failure.

Throws: RemoveException
The enterprise Bean or the container does not allow destruction of the object.

Interface EnterpriseBean

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

The EnterpriseBean interface must be implemented by every enterprise Bean class. It is a common super-interface for the SessionBean and EntityBean interfaces.

Interface EntityBean

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

The EntityBean interface is implemented by every entity enterprise Bean class. The container uses the EntityBean methods to notify the enterprise Bean instances of the instance's life cycle events.

Note: Support for entity enterprise Beans is optional for EJB 1.0 compliant containers. Support for entities will become mandatory for EJB 2.0 compliant containers.

Methods

- **ejbActivate**

```
public abstract void ejbActivate()
     throws RemoteException
```

A container invokes this method when the instance is taken out of the pool of available instances to become associated with a specific EJB object. This method transitions the instance to the ready state.

This method executes in an unspecified transaction context.

Throws: RemoteException
Thrown if the instance could not perform the function requested by the container because of a system-level error.

- **ejbLoad**

```
public abstract void ejbLoad()
     throws RemoteException
```

A container invokes this method to instruct the instance to synchronize its state by loading it state from the underlying database.

This method always executes in the proper transaction context.

Throws: RemoteException
Thrown if the instance could not perform the function requested by the container because of a system-level error.

- **ejbPassivate**

```
public abstract void ejbPassivate()
     throws RemoteException
```

A container invokes this method on an instance before the instance becomes disassociated with a specific

EJB object. After this method completes, the container will place the instance into the pool of available instances.

This method executes in an unspecified transaction context.

Throws: RemoteException
Thrown if the instance could not perform the function requested by the container because of a system-level error.

- **`ejbRemove`**

```
public abstract void ejbRemove()
     throws RemoteException, RemoveException
```

A container invokes this method before it removes the EJB object that is currently associated with the instance. This method is invoked when a client invokes a remove operation on the enterprise Bean's home interface or the EJB object's remote interface. This method transitions the instance from the ready state to the pool of available instances.

This method is called in the transaction context of the remove operation.

Throws: RemoteException
Thrown if the instance could not perform the function requested by the container because of a system-level error.

Throws: RemoveException
The enterprise Bean does not allow destruction of the object.

- **`ejbStore`**

```
public abstract void ejbStore()
     throws RemoteException
```

A container invokes this method to instruct the instance to synchronize its state by storing it to the underlying database.

This method always executes in the proper transaction context.

Throws: RemoteException
Thrown if the instance could not perform the function requested by the container because of a system-level error.

- **`setEntityContext`**

```
public abstract void
setEntityContext(EntityContext ctx)
     throws RemoteException
```

Set the associated entity context. The container invokes this method on an instance after the instance has been created.

This method is called in an unspecified transaction context.

Parameters:
`ctx`
An EntityContext interface for the instance. The instance should store the reference to the context in an instance variable.

Throws: RemoteException
Thrown if the instance could not perform the function requested by the container because of a system-level error.

- **`unsetEntityContext`**

```
public abstract void unsetEntityContext()
     throws RemoteException
```

Unset the associated entity context. The container calls this method before removing the instance.

This is the last method that the container invokes on the instance. The Java garbage collector will eventually invoke the finalize() method on the instance.

This method is called in an unspecified transaction context.

Throws: RemoteException
Thrown if the instance could not perform the function requested by the container because of a system-level error.

Interface EntityContext

```
public interface javax.ejb.EntityContext
    extends javax.ejb.EJBContext
{
    public abstract EJBObject getEJBObject();
    public abstract Object getPrimaryKey();
}
```

The EntityContext interface provides an instance with access to the container-provided runtime context of an entity enterprise Bean instance. The container passes the EntityContext interface to an entity enterprise Bean instance after the instance has been created.

The EntityContext interface remains associated with the instance for the lifetime of the instance. Note that the information that the instance obtains using the EntityContext interface (such as the result of the getPrimaryKey() method) may change, as the container assigns the instance to different EJB objects during the instance's life cycle.

Methods

- **getEJBObject**

```
public abstract EJBObject getEJBObject()
     throws IllegalStateException
```

Obtain a reference to the EJB object that is currently associated with the instance.

An instance of an entity enterprise Bean can call this method only when the instance is associated with an EJB object identity, i.e. in the ejbActivate, ejbPassivate, ejbPostCreate method, ejbRemove, ejbLoad, ejbStore, and business methods.

An instance can use this method, for example, when it wants to pass a reference to itself in a method argument or result.

Returns:
The EJB object currently associated with the instance.

Throws: IllegalStateException
Thrown if the instance invokes this method while the instance is in a state that does not allow the instance to invoke this method.

- **getPrimaryKey**

```
public abstract Object getPrimaryKey()
     throws IllegalStateException
```

Obtain the primary key of the EJB object that is currently associated with this instance.

An instance of an entity enterprise Bean can call this method only when the instance is associated with an EJB object identity, i.e. in the ejbActivate, ejbPassivate, ejbPostCreate method, ejbRemove, ejbLoad, ejbStore, and business methods.

Note: The result of this method is that same as the result of getEJBObject().getPrimaryKey().

Returns:
The EJB object currently associated with the instance.

Throws: IllegalStateException

Thrown if the instance invokes this method while the instance is in a state that does not allow the instance to invoke this method.

Interface Handle

```
public interface javax.ejb.Handle
{
    public abstract EJBObject getEJBObject();
}
```

The Handle interface is implemented by all EJB object handles. A handle is an abstraction of a network reference to an EJB object. A handle is intended to be used as a "robust" persistent reference to an EJB object.

The implementation class for the handle (typically provided by the container) must be java.io.Serializable to allow the client to serialize a handle object.

Methods

- **getEJBObject**

```
public abstract EJBObject getEJBObject()
     throws RemoteException
```

Obtain the EJB object represented by this handle.

Throws: RemoteException
The EJB object could not be obtained because of a system-level failure.

Interface SessionBean

```
public interface javax.ejb.SessionBean
    extends javax.ejb.EnterpriseBean
{
    public abstract void ejbActivate();
    public abstract void ejbPassivate();
    public abstract void ejbRemove();
    public abstract void
        setSessionContext(SessionContext ctx);
}
```

The SessionBean interface is implemented by every session enterprise Bean class. The container uses the SessionBean methods to notify the enterprise Bean instances of the instance's life cycle events.

Methods

- **ejbActivate**

  ```
  public abstract void ejbActivate()
       throws RemoteException
  ```

  The activate method is called when the instance is activated from its "passive" state. The instance should acquire any resource that it has released earlier in the ejbPassivate() method.

  This method is called with no transaction context.

  Throws: RemoteException
  Thrown if the instance could not perform the function requested by the container because of a system-level error.

- **ejbPassivate**

  ```
  public abstract void ejbPassivate()
       throws RemoteException
  ```

  The passivate method is called before the instance enters the "passive" state. The instance should release any resources that it can re-acquire later in the ejbActivate() method.

  After the passivate method completes, the instance must be in a state that allows the container to use the Java Serialization protocol to externalize and store away the instance's state.

  This method is called with no transaction context.

  Throws: RemoteException
  Thrown if the instance could not perform the function requested by the container because of a system-level error.

- **ejbRemove**

  ```
  public abstract void ejbRemove()
       throws RemoteException
  ```

  A container invokes this method before it ends the life of the session object. This happens as a result of a client's invoking a remove operation, or when a container decides to terminate the session object after a timeout.

  This method is called with no transaction context.

Throws: RemoteException
Thrown if the instance could not perform the function requested by the container because of a system-level error.

- **`setSessionContext`**

```
public abstract void
setSessionContext(SessionContext ctx)
     throws RemoteException
```

Set the associated session context. The container calls this method after the instance creation.

The enterprise Bean instance should store the reference to the context object in an instance variable.

This method is called with no transaction context.

Parameters:
`ctx`
A SessionContext interface for the instance.

Throws: RemoteException
Thrown if the instance could not perform the function requested by the container because of a system-level error.

Interface SessionContext

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

The SessionContext interface provides access to the runtime session context that the container provides for a session enterprise Bean instance. The container passes the SessionContext interface to an instance after the instance has been created. The session context remains associated with the instance for the lifetime of the instance.

Methods

- **`getEJBObject`**

```
public abstract EJBObject getEJBObject()
     throws IllegalStateException
```

Obtain a reference to the EJB object that is currently associated with the instance.

An instance of a session enterprise Bean can call this method at anytime between the ejbCreate() and ejbRemove() methods, including from within the ejbCreate() and ejbRemove() methods.

An instance can use this method, for example, when it wants to pass a reference to itself in a method argument or result.

Returns:
The EJB object currently associated with the instance.

Throws: IllegalStateException
Thrown if the instance invokes this method while the instance is in a state that does not allow the instance to invoke this method.

Interface SessionSynchronization

```
public interface javax.ejb.SessionSynchronization
{
    public abstract void afterBegin();
    public abstract void
        afterCompletion(boolean committed);
    public abstract void beforeCompletion();
}
```

The SessionSynchronization interface allows a session Bean instance to be notified by its container of transaction boundaries.

An session Bean class is not required to implement this interface. A session Bean class should implement this interface only if it wishes to synchronize its state with the transactions.

Methods

- **afterBegin**

```
public abstract void afterBegin()
     throws RemoteException
```

The afterBegin method notifies a session Bean instance that a new transaction has started, and that the subsequent business methods on the instance will be invoked in the context of the transaction.

The instance can use this method, for example, to read data from a database and cache the data in the instance fields.

This method executes in the proper transaction context.

Throws: RemoteException
Thrown if the instance could not perform the function requested by the container because of a system-level error.

- **afterCompletion**

```
public abstract void
afterCompletion(boolean committed)
     throws RemoteException
```

The afterCompletion method notifies a session Bean instance that a transaction commit protocol has completed, and tells the instance whether the transaction has been committed or rolled back.

This method executes with no transaction context.

This method executes with no transaction context.

Parameters:
committed
True if the transaction has been committed, false if is has been rolled back.

Throws: RemoteException
Thrown if the instance could not perform the function requested by the container because of a system-level error.

- **beforeCompletion**

```
public abstract void beforeCompletion()
```

```
throws RemoteException
```

The beforeCompletion method notifies a session Bean instance that a transaction is about to be committed. The instance can use this method, for example, to write any cached data to a database.

This method executes in the proper transaction context.

Note: The instance may still cause the container to rollback the transaction by invoking the setRollback-Only() method on the instance context, or by throwing an exception.

Throws: RemoteException

Thrown if the instance could not perform the function requested by the container because of a system-level error.

Class CreateException

```
public class javax.ejb.CreateException
    extends java.lang.Exception
{
    public CreateException();
    public CreateException(String message);
}
```

The CreateException exception must be included in the throws clauses of all create(...) methods define in an enterprise Bean's remote interface.

The exception is used as a standard application-level exception to report a failure to create an entity EJB object.

Constructors

- **`CreateException`**

  `public CreateException()`

  Constructs an CreateException with no detail message.

- **`CreateException`**

  `public CreateException(String message)`

  Constructs an CreateException with the specified detail message.

Class DuplicateKeyException

```
public class javax.ejb.DuplicateKeyException
    extends javax.ejb.CreateException
{
    public DuplicateKeyException();
    public
        DuplicateKeyException(String message);
}
```

The DuplicateKeyException exception is thrown if an entity EJB object cannot be created because an object with the same key already exists. This exception is thrown by the create methods defined in an enterprise Bean's home interface.

Constructors

- **DuplicateKeyException**

  ```
  public DuplicateKeyException()
  ```

  Constructs an DuplicateKeyException with no detail message.

- **DuplicateKeyException**

  ```
  public DuplicateKeyException(String message)
  ```

  Constructs an DuplicateKeyException with the specified detail message.

Class FinderException

```
public class javax.ejb.FinderException
    extends java.lang.Exception
{
    public FinderException();
    public FinderException(String message);
}
```

The FinderException exception must be included in the throws clause of every findMETHOD(...) method of an entity Bean's home interface.

The exception is used as a standard application-level exception to report a failure to find the requested EJB object(s).

Constructors

- **`FinderException`**

  `public FinderException()`

  Constructs an FinderException with no detail message.

- **`FinderException`**

  `public FinderException(String message)`

  Constructs an FinderException with the specified detail message.

Class ObjectNotFoundException

```
public class javax.ejb.ObjectNotFoundException
    extends javax.ejb.FinderException
{
    public ObjectNotFoundException();
    public
        ObjectNotFoundException(String message);
}
```

The ObjectNotFoundException exception is thrown by a finder method to indicate that the specified EJB object does not exist.

Only the finder methods that are declared to return a single EJB object use this exception. This exception should not be thrown by finder methods that return a collection of EJB objects (they should return a null collection instead).

Constructors

- **`ObjectNotFoundException`**

  `public ObjectNotFoundException()`

  Constructs an ObjectNotFoundException with no detail message.

- **`ObjectNotFoundException`**

  `public ObjectNotFoundException(String message)`

  Constructs an ObjectNotFoundException with the specified detail message.

Class RemoveException

```
public class javax.ejb.RemoveException
    extends java.lang.Exception
{
    public RemoveException();
    public RemoveException(String message);
}
```

The RemoveException exception is thrown at an attempt to remove an EJB object when the enterprise Bean or the container does not allow the EJB object to be removed.

Constructors

- **RemoveException**

  ```
  public RemoveException()
  ```

  Constructs an RemoveException with no detail message.

- **RemoveException**

  ```
  public RemoveException(String message)
  ```

  Constructs an RemoveException with the specified detail message.

Class AccessControlEntry

```
public class javax.ejb.deployment.AccessControlEntry
    extends java.lang.Object
    implements java.io.Serializable
{
    public AccessControlEntry();
    public AccessControlEntry(Method method);
    public
        AccessControlEntry(Method method,
                           Identity identities[]);
    public Identity[] getAllowedIdentities();
    public Identity
        getAllowedIdentities(int index);
    public Method getMethod();
    public void
        setAllowedIdentities(Identity values[]);
    public void
        setAllowedIdentities(int index,
                             Identity value);
    public void setMethod(Method value);
}
```

The class AccessControlEntry associates a list of security identities with an enterprise Bean's method. The specified identities are permitted to invoke the enterprise Bean's method.

The Method that is associated with an AccessControlEntry must be a Method of the enterprise Bean class and the method must be one of the following: a business method, an ejbCreate(...) method, a finder method, or the ejbDestroy method.

If the Method used in an AccessControlEntry is null, then the AccessControlEntry is considered to be associated with the entire Bean. A Bean-level AccessControlEntry provides the default value for the methods that do not have a method-level AccessControlEntry.

Constructors

- **AccessControlEntry**

  `public AccessControlEntry()`

  Constructor.

- **AccessControlEntry**

  `public AccessControlEntry(Method method)`

  Construct an AccessControlEntry for the specified enterprise Bean's method. If method is null, the entry is considered to be the default AccessControlEntry for the enterprise Bean.

  Parameters:

  `method`
  : An enterprise Bean's method, or null if this is the default AccessControlEntry for the enterprise Bean.

- **AccessControlEntry**

  `public AccessControlEntry(Method method,`

```
                                    Identity identities[])
```

Construct an AccessControlEntry for the specified enterprise Bean's method. If method is null, the entry is considered to be the default AccessControlEntry for the enterprise Bean.

Parameters:

`method`
: An enterprise Bean's method, or null if this is the default AccessControlEntry for the enterprise Bean.

`identities`
: An array of security Identities that are permitted to invoke this method.

Methods

- **`getAllowedIdentities`**

  ```
  public Identity[] getAllowedIdentities()
  ```

  Get the array of Identities that are permitted to invoke this method.

  Returns:
  An array of security Identities that are permitted to invoke this method.

- **`getAllowedIdentities`**

  ```
  public Identity getAllowedIdentities(int index)
  ```

  Get the Identity at the specified index from the array of Identities that are permitted to invoke this method.

  Parameters:

  `index`
  : The index in the array.

  Returns:
  The Identity at the specified index.

- **`getMethod`**

  ```
  public Method getMethod()
  ```

  Get the method to which this AccessControlEntry applies.

  Returns:
  An enterprise Bean's method to which this AccessControlEntry applies. If the return value is null, this is the default AccessControlEntry for the enterprise Bean.

- **`setAllowedIdentities`**

  ```
  public void setAllowedIdentities(Identity values[])
  ```

  Set the array of Identities that are permitted to invoke this method.

  Parameters:

  `values`
  : An array of security Identities that are permitted to invoke this method.

- **`setAllowedIdentities`**

  ```
  public void
  setAllowedIdentities(int index, Identity value)
  ```

Set the Identity at the specified index in the array of Identities that are permitted to invoke this method.

Parameters:

`index`
: The index in the array.

`identity`
: The Identity to be set at the specified index.

- **`setMethod`**

  ```
  public void setMethod(Method value)
  ```

  Set the method to which this AccessControlEntry applies.

  Parameters:

  `value`
  : An enterprise Bean's method, or null if this is the default AccessControlEntry for the enterprise Bean.

Class ControlDescriptor

```
public class javax.ejb.deployment.ControlDescriptor
    extends java.lang.Object
    implements java.io.Serializable
{
    public final static int CLIENT_IDENTITY;
    public final static int
        SPECIFIED_IDENTITY;
    public final static int SYSTEM_IDENTITY;
    public final static int
        TRANSACTION_READ_COMMITTED;
    public final static int
        TRANSACTION_READ_UNCOMMITTED;
    public final static int
        TRANSACTION_REPEATABLE_READ;
    public final static int
        TRANSACTION_SERIALIZABLE;
    public final static int TX_BEAN_MANAGED;
    public final static int TX_MANDATORY;
    public final static int TX_NOT_SUPPORTED;
    public final static int TX_REQUIRED;
    public final static int TX_REQUIRES_NEW;
    public final static int TX_SUPPORTS;
    public ControlDescriptor();
    public ControlDescriptor(Method method);
    public int getIsolationLevel();
    public Method getMethod();
    public Identity getRunAsIdentity();
    public int getRunAsMode();
    public int getTransactionAttribute();
    public void setIsolationLevel(int value);
    public void setMethod(Method value);
    public void
        setRunAsIdentity(Identity value);
    public void setRunAsMode(int value);
    public void
        setTransactionAttribute(int value);
}
```

The ControlDescriptor defines the transaction and security attributes to be associated with the runtime execution of an enterprise Bean method.

If the Method used in an ControlDescriptor is null, then the ControlDescriptor is considered to be associated with the entire Bean. A Bean-level ControlDescriptor provides the default value for the methods that do not have a method-level ControlDescriptor.

The methods of the ControlDescriptor class conform to the JavaBeans property design pattern.

Variables

- **CLIENT_IDENTITY**

  ```
  public final static int CLIENT_IDENTITY
  ```

Run the enterprise Bean method with the client's security identity.

- **SPECIFIED_IDENTITY**

  ```
  public final static int SPECIFIED_IDENTITY
  ```

  Run the enterprise Bean method with the security identity of a specified user account.

- **SYSTEM_IDENTITY**

  ```
  public final static int SYSTEM_IDENTITY
  ```

  Run the enterprise Bean method with the Identity of a "privileged account". The container maps the abstract notion of a "privileged account" to a suitable privileged account on the underlying platform, such as the database administrator, or the operating system administrator account.

- **TRANSACTION_READ_COMMITTED**

  ```
  public final static int TRANSACTION_READ_COMMITTED
  ```

  Isolation degree equivalent to the JDBC TRANSACTION_READ_COMMITTED level.

- **TRANSACTION_READ_UNCOMMITTED**

  ```
  public final static int TRANSACTION_READ_UNCOMMITTED
  ```

  Isolation degree equivalent to the JDBC TRANSACTION_READ_UNCOMMITTED level.

- **TRANSACTION_REPEATABLE_READ**

  ```
  public final static int TRANSACTION_REPEATABLE_READ
  ```

  Isolation degree equivalent to the JDBC TRANSACTION_REPEATABLE_READ level.

- **TRANSACTION_SERIALIZABLE**

  ```
  public final static int TRANSACTION_SERIALIZABLE
  ```

  Isolation degree equivalent to the JDBC TRANSACTION_SERIALIZABLE level.

- **TX_BEAN_MANAGED**

  ```
  public final static int TX_BEAN_MANAGED
  ```

  The enterprise Bean manages transaction boundaries itself using the javax.jts.CurrentTransaction interface.

- **TX_MANDATORY**

  ```
  public final static int TX_MANDATORY
  ```

  The enterprise Bean requires that the client invocation includes a global transaction scope. The container is responsible for managing transaction boundaries for the enterprise Bean as follow.

  If the caller is associated with a transaction, the execution of the enterprise Bean method will be associated with the caller's transaction.

  If the caller is not associated with a transaction, the container throws the javax.jts.TransactionRequiredException to the caller.

- **TX_NOT_SUPPORTED**

  ```
  public final static int TX_NOT_SUPPORTED
  ```

  The enterprise Bean does not support a global transaction. The container must not invoke the enterprise Bean's method in the scope of a global transaction.

- **TX_REQUIRED**

  ```
  public final static int TX_REQUIRED
  ```

  The enterprise Bean requires that the method be executed in a global transaction.

  The container is responsible for managing transaction boundaries for the enterprise Bean as follow.

  If the caller is associated with a transaction, the execution of the enterprise Bean method will be associated with the caller's transaction.

  If the caller is not associated with a transaction, the container starts a new global transaction, executes the enterprise Bean's method in the scope of the transaction, and commits the transaction when the enterprise Bean's method has completed.

- **TX_REQUIRES_NEW**

  ```
  public final static int TX_REQUIRES_NEW
  ```

  The enterprise Bean requires that a method is executed in a new global transaction scope.

  The container is responsible for managing transaction boundaries for the enterprise Bean as follow.

  The container starts a new transaction, executes the enterprise Bean's method in the scope of the new transaction, and commits the new transaction when the enterprise Bean's method has completed.

  If the caller is associated with a transaction, the association of the current thread with the caller's transaction is suspended during the execution of the enterprise Bean's method, and resumed when the enterprise Bean's method has completed.

- **TX_SUPPORTS**

  ```
  public final static int TX_SUPPORTS
  ```

  The enterprise Bean supports the execution of a method in a global transaction scope. The container is responsible for managing transaction boundaries for the enterprise Bean as follow.

  If the caller is associated with a transaction, the execution of the enterprise Bean method will be associated with the caller's transaction.

  If the caller is not associated with a transaction, the container executes the enterprise Bean's method without a transaction.

Constructors

- **ControlDescriptor**

  ```
  public ControlDescriptor()
  ```

  Construct a Bean-level ControlDescriptor.

- **ControlDescriptor**

  ```
  public ControlDescriptor(Method method)
  ```

  Construct a ControlDescriptor for a specified Method.

  Parameters:

  `method`

  The Method associated with the ControlDescriptor. The Method must be a method of the enterprise Bean class and the Method must be one of the following: a business method, an ejbCreate(...) method, an finder method, or the ejbDestroy method. If method is null, the ControlDescriptor will be a Bean-level one.

Methods

- **`getIsolationLevel`**

  ```
  public int getIsolationLevel()
  ```

  Get the transaction isolation level.

  Returns:

  Transaction isolation level. The value is one of TRANSACTION_READ_UNCOMMITTED, TRANSACTION_READ_COMMITTED, TRANSACTION_REPEATABLE_READ, and TRANSACTION_SERIALIZABLE.

- **`getMethod`**

  ```
  public Method getMethod()
  ```

  Obtain the Method associated with the with this ControlDescriptor.

  Returns:

  A Method associated with the ControlDescriptor. If the return value is null, the ControlDescriptor is a Bean-level one.

- **`getRunAsIdentity`**

  ```
  public Identity getRunAsIdentity()
  ```

  Get the value of the runAsIdentity security attribute. The runAsIdentity attribute tells the container the security identity to associate with the execution of the enterprise Bean method.

  The value of the runAsIdentity is used only if the value of the runAsMode is SPECIFIED_IDENTITY; it is ignored otherwise.

  Returns:

  The Identity to associate with the execution of the enterprise Bean method.

- **`getRunAsMode`**

  ```
  public int getRunAsMode()
  ```

  Get the value of the runAsMode security attribute. The runAsMode attribute tells the container the security identity to associate with the execution of the enterprise Bean method.

  Returns:

  The value of the runAsMode attribute. The value is one of CLIENT_IDENTITY, SPECIFIED_IDENTITY, and SYSTEM_IDENTITY.

- **`getTransactionAttribute`**

  ```
  public int getTransactionAttribute()
  ```

  Get the value of the transaction attribute. The transaction attribute tells the container how to manage transaction scopes before and after the execution of the enterprise Bean method.

  Returns:

  The value of the transaction attribute. It must be one of TX_NOT_SUPPORTED, TX_BEAN_MANAGED, TX_REQUIRED, TX_REQUIRES_NEW, and TX_MANDATORY.

- **`setIsolationLevel`**

  ```
  public void setIsolationLevel(int value)
  ```

Set the transaction isolation level.

Parameters:

`value`

Transaction isolation level. The value must be one of TRANSACTION_READ_UNCOMMITTED, TRANSACTION_READ_COMMITTED, TRANSACTION_REPEATABLE_READ, and TRANSACTION_SERIALIZABLE.

- **`setMethod`**

```
public void setMethod(Method value)
```

Set the method to which this ControlDescriptor applies.

Parameters:

`value`

An enterprise Bean's method, or null if this is the default ControlDescriptor for the enterprise Bean.

- **`setRunAsIdentity`**

```
public void setRunAsIdentity(Identity value)
```

Set the value of the runAsIdentity security attribute. The runAsIdentity attribute tells the container the security identity to associate with the execution of the enterprise Bean method.

The value of the runAsIdentity is used only if the value of the runAsMode is SPECIFIED_IDENTITY; it is ignored otherwise.

Parameters:

`value`

The Identity to associate with the execution of the enterprise Bean method.

- **`setRunAsMode`**

```
public void setRunAsMode(int value)
```

Set the value of the runAsMode security attribute. The runAsMode attribute tells the container the security identity to associate with the execution of the enterprise Bean method.

Parameters:

`value`

The value of the runAsMode attribute. The value must be one of CLIENT_IDENTITY, SPECIFIED_IDENTITY, and SYSTEM_IDENTITY.

- **`setTransactionAttribute`**

```
public void setTransactionAttribute(int value)
```

Set the value of the transaction attribute. The transaction attribute tells the container how to manage transaction scopes before and after the execution of the enterprise Bean method.

Parameters:

`value`

The value of the transaction attribute. It must be one of TX_NOT_SUPPORTED, TX_BEAN_MANAGED, TX_REQUIRED, TX_REQUIRES_NEW, and TX_MANDATORY.

Class DeploymentDescriptor

```
public class javax.ejb.deployment.DeploymentDescriptor
    extends java.lang.Object
    implements java.io.Serializable
{
    protected int versionNumber;
    public DeploymentDescriptor();
    public AccessControlEntry[]
        getAccessControlEntries();
    public AccessControlEntry
        getAccessControlEntries(int index);
    public Name getBeanHomeName();
    public ControlDescriptor[]
        getControlDescriptors();
    public ControlDescriptor
        getControlDescriptors(int index);
    public String
        getEnterpriseBeanClassName();
    public Properties
        getEnvironmentProperties();
    public String
        getHomeInterfaceClassName();
    public boolean getReentrant();
    public String
        getRemoteInterfaceClassName();
    public boolean isReentrant();
    public void
        setAccessControlEntries(AccessControlEntry values[]);
    public void
        setAccessControlEntries(int index,
                                AccessControlEntry value);
    public void setBeanHomeName(Name value);
    public void
        setControlDescriptors(ControlDescriptor value[]);
    public void
        setControlDescriptors(int index,
                              ControlDescriptor value);
    public void
        setEnterpriseBeanClassName(String value);
    public void
        setEnvironmentProperties(Properties value);
    public void
        setHomeInterfaceClassName(String value);
    public void setReentrant(boolean value);
    public void
        setRemoteInterfaceClassName(String value);
}
```

The DeploymentDescriptor class is the common base class for the SessionDescriptor and EntityDescriptor deployment descriptor classes.

The methods of the class conform to the JavaBeans property design pattern.

See Also:

EntityDescriptor , SessionDescriptor .

Variables

- **`versionNumber`**

  ```
  protected int versionNumber
  ```

Constructors

- **`DeploymentDescriptor`**

  ```
  public DeploymentDescriptor()
  ```

  Create an instance of DeploymentDescriptor.

Methods

- **`getAccessControlEntries`**

  ```
  public AccessControlEntry[]
  getAccessControlEntries()
  ```

  Get the AccessControlEntry objects for the enterprise Bean. An AccessControlEntry object associates an enterprise Bean's method with a list of security Identities that are allowed to invoke the method.

  Returns:

  An array of AccessControlEntry objects.

- **`getAccessControlEntries`**

  ```
  public AccessControlEntry
  getAccessControlEntries(int index)
  ```

  Get the AccessControlEntry object at a specified index. An AccessControlEntry object associates an enterprise Bean's method with a list of security Identities that are allowed to invoke the method.

  Parameters:

  `index`

  An index in the array of Identities.

  Returns:

  The AccessControlEntry at the index.

- **`getBeanHomeName`**

  ```
  public Name getBeanHomeName()
  ```

  Get the name to associate with the enterprise Bean in the JNDI name space. The container will bind the enterprise Bean's home interface with a JNDI name that includes this name as its trailing part. This means that the container can prefix the name returned by getBeanName() with an arbitrary JNDI path.

  For example, if getBeanHomeName() returns "bank/Account", the container can bind the Bean's home interface in the JNDI name space with the name "mis/ejb-components/bank/Account".

  Returns:

A JNDI name for this enterprise Bean.

- **getControlDescriptors**

  ```
  public ControlDescriptor[] getControlDescriptors()
  ```

  Get the array of the enterprise Bean's control descriptors.

  Returns:
  An array of enterprise Bean's control descriptors.

- **getControlDescriptors**

  ```
  public ControlDescriptor
  getControlDescriptors(int index)
  ```

  Get the control descriptor at the specified index.

  Parameters:
  `index`
  The index of the control descriptor.

  Returns:
  The control descriptor at the specified index.

- **getEnterpriseBeanClassName**

  ```
  public String getEnterpriseBeanClassName()
  ```

  Get the enterprise Bean's full class name.

  Returns:
  The enterprise Bean's class name.

- **getEnvironmentProperties**

  ```
  public Properties getEnvironmentProperties()
  ```

  Get enterprise Bean's environment properties.

  Returns:
  Enterprise Bean's environment properties.

- **getHomeInterfaceClassName**

  ```
  public String getHomeInterfaceClassName()
  ```

  Get the full name of the enterprise Bean's home interface.

  Returns:
  The name of the enterprise Bean's home interface.

- **getReentrant**

  ```
  public boolean getReentrant()
  ```

  This method returns the same result as isReentrant(). It is included for compatibility with the JavaBeans design-pattern.

- **getRemoteInterfaceClassName**

  ```
  public String getRemoteInterfaceClassName()
  ```

Get the full name of the enterprise Bean's remote interface.

Returns:
The name of the enterprise Bean's remote interface.

- **isReentrant**

```
public boolean isReentrant()
```

Test if the enterprise Bean is re-entrant. Only entity Beans can be defined as re-entrant, and it is an error for a session Bean deployment descriptor to return true.

Returns:
True if the Bean is reentrant, false otherwise.

- **setAccessControlEntries**

```
public void
setAccessControlEntries(AccessControlEntry values[])
```

Set the AccessControlEntry objects for the enterprise Bean. An AccessControlEntry object associates an enterprise Bean's method with a list of security Identities that are allowed to invoke the method.

Parameters:
`values`
An array of AccessControlEntry objects.

- **setAccessControlEntries**

```
public void
setAccessControlEntries(int index,
                        AccessControlEntry value)
```

Set the AccessControlEntry object at a specified index. An AccessControlEntry object associates an enterprise Bean's method with a list of security Identities that are allowed to invoke the method.

Parameters:
`index`
An index in the array of Identities.
`value`
The AccessControlEntry to set at the index.

- **setBeanHomeName**

```
public void setBeanHomeName(Name value)
```

Set the name to associate with the enterprise Bean in the JNDI name space. The container will bind the enterprise Bean's home interface with a JNDI name that includes this name as its trailing part. This means that the container can prefix the name returned by getBeanName() with an arbitrary JNDI path.

Note that using the type java.naming.Name makes the format of the name independent of the syntax used by the actual naming system.

Parameters:
`value`
A JNDI name for this enterprise Bean.

- **setControlDescriptors**

```
public void
setControlDescriptors(ControlDescriptor value[])
```

Set the array of the enterprise Bean's control descriptors.

Parameters:

`value`
An array of the enterprise Bean's control descriptors.

- **`setControlDescriptors`**

```
public void
setControlDescriptors(int index,
                      ControlDescriptor value)
```

Set the control descriptor at the specified index.

Parameters:

`index`
The index of the control descriptor.

`value`
The control descriptor to be set at the specified index.

- **`setEnterpriseBeanClassName`**

```
public void setEnterpriseBeanClassName(String value)
```

Set the enterprise Bean's full class name.

Parameters:

`value`
The enterprise Bean's class name.

- **`setEnvironmentProperties`**

```
public void
setEnvironmentProperties(Properties value)
```

Set enterprise Bean's environment properties.

Parameters:

`value`
Enterprise Bean's environment properties.

- **`setHomeInterfaceClassName`**

```
public void setHomeInterfaceClassName(String value)
```

Set the full name of the enterprise Bean's home interface.

Parameters:

`value`
The name of the enterprise Bean's home interface.

- **`setReentrant`**

```
public void setReentrant(boolean value)
```

Specify that the enterprise Bean is re-entrant. Only entity Beans can be defined as re-entrant, and it is an error for a session Bean deployment descriptor to attempt to specify that the Bean is re-entrant.

Returns:

True if the Bean is reentrant, false otherwise.

- **`setRemoteInterfaceClassName`**

  ```
  public void
  setRemoteInterfaceClassName(String value)
  ```

  Set the full name of the enterprise Bean's remote interface.

  Parameters:

  `value`
  The name of the enterprise Bean's remote interface.

Class EntityDescriptor

```
public class javax.ejb.deployment.EntityDescriptor
    extends javax.ejb.deployment.DeploymentDescriptor
{
    public EntityDescriptor();
    public Field[]
        getContainerManagedFields();
    public Field
        getContainerManagedFields(int index);
    public String getPrimaryKeyClassName();
    public void
        setContainerManagedFields(Field values[]);
    public void
        setContainerManagedFields(int index,
                                  Field value);
    public void
        setPrimaryKeyClassName(String value);
}
```

The EntityDescriptor class defines the deployment descriptor for an entity enterprise Bean.

A serialized instance of the EntityDescriptor class is used as the standard format for passing the entity enterprise Bean's declarative deployment attributes in the ejb-jar file.

The Bean provider tools use the setter functions to initialize an instance of the deployment descriptor. The Bean provider tools then serialize the instance into the ejb-jar file.

The getter functions are used by the container tools at deployment time. The tools deserialize the instance from the ejb-jar file, and use the getter functions to obtain information about the enterprise Bean.

Note that the Enterprise JavaBeans architecture does not prescribe whether the actual deployment descriptor class is used by the container at runtime. Therefore, the container is allowed to import the information from the deployment descriptor at deployment time, and store the information in a container-specific format. The container is however required to enforce the declarative attributes at runtime, as specified by the Enterprise JavaBeans specification.

After an enterprise Bean has been installed into a container, the container tools can be then used to view and change the values of the deployment attributes. As changing the values of the deployment descriptor attributes may alter the semantics of a deployed application, the container may restrict changes to certain attributes.

The methods of the EntityDescriptor class conform to the JavaBeans property design pattern.

Constructors

- **`EntityDescriptor`**

  ```
  public EntityDescriptor()
  ```

  Create an instance of EntityDescriptor.

Methods

- **`getContainerManagedFields`**

  ```
  public Field[] getContainerManagedFields()
  ```

  Get the array of the container-managed fields.

  Returns:
  The array of the container-managed fields.

- **`getContainerManagedFields`**

  ```
  public Field getContainerManagedFields(int index)
  ```

  Get the name of field at the given index in the array of container-managed fields.

  Parameters:
  `index`
  The index in the array.

  Returns:
  The container-managed field at the specified index.

- **`getPrimaryKeyClassName`**

  ```
  public String getPrimaryKeyClassName()
  ```

  Get the full class name of the enterprise Bean's primary key.

  Returns:
  The primary key class name.

- **`setContainerManagedFields`**

  ```
  public void
  setContainerManagedFields(Field values[])
  ```

  Set the array of the names of the container-managed fields.

  Parameters:
  `value`
  The array of the names of the container-managed fields.

- **`setContainerManagedFields`**

  ```
  public void
  setContainerManagedFields(int index, Field value)
  ```

  Set the field at the given index in the array of container-managed fields.

  Parameters:
  `index`
  The index in the array.
  `value`
  The container-managed field to be set at the index.

- **`setPrimaryKeyClassName`**

  ```
  public void setPrimaryKeyClassName(String value)
  ```

Set the full class name of the enterprise Bean's primary key.

Returns:

The primary key class name.

Class SessionDescriptor

```
public class javax.ejb.deployment.SessionDescriptor
    extends javax.ejb.deployment.DeploymentDescriptor
{
    public final static int STATEFUL_SESSION;
    public final static int
        STATELESS_SESSION;
    public SessionDescriptor();
    public int getSessionTimeout();
    public int getStateManagementType();
    public void setSessionTimeout(int value);
    public void
        setStateManagementType(int value);
}
```

The SessionDescriptor class defines the deployment descriptor for a session enterprise Bean.

A serialized instance of the SessionDescriptor class is used as the standard format for passing the session enterprise Bean's declarative deployment attributes in the ejb-jar file.

The Bean provider tools use the setter functions to initialize an instance of the deployment descriptor. The Bean provider tools then serialize the instance into the ejb-jar file.

The getter functions are used by the container tools at deployment time. The tools deserialize the instance from the ejb-jar file, and use the getter functions to obtain information about the enterprise Bean.

Note that the Enterprise JavaBeans architecture does not prescribe whether the actual deployment descriptor class is used by the container at runtime. Therefore, the container is allowed to import the information from the deployment descriptor at deployment time, and store the information in a container-specific format. The container is however required to enforce the declarative attributes at runtime, as specified by the Enterprise JavaBeans specification.

After an enterprise Bean has been installed into a container, the container tools can be used to view and change the values of the deployment attributes. As changing the values of the deployment descriptor attributes may alter the semantics of a deployed application, the container may restrict changes to certain attributes.

The methods of the SessionDescriptor class conform to the JavaBeans property design pattern.

Variables

- **STATEFUL_SESSION**

  ```
  public final static int STATEFUL_SESSION
  ```

  The session Bean is stateful. An instance of a stateful session Bean remains associated with a session EJB object for its lifetime.

- **STATELESS_SESSION**

  ```
  public final static int STATELESS_SESSION
  ```

  The session Bean is stateless. An instance of a stateless Bean can be reused for multiple session EJB objects.

Constructors

- **`SessionDescriptor`**

  ```
  public SessionDescriptor()
  ```

  Create an instance of SessionDescriptor.

Methods

- **`getSessionTimeout`**

  ```
  public int getSessionTimeout()
  ```

  Get the session timeout value in seconds. A zero value means that the container should use a container-specific default value.

  Returns:
  The timeout value in seconds.

- **`getStateManagementType`**

  ```
  public int getStateManagementType()
  ```

  Get the session Bean's state management type.

  Returns:
  The session Bean's state management type. Its value must be either STATEFUL_SESSION or STATELESS_SESSION.

- **`setSessionTimeout`**

  ```
  public void setSessionTimeout(int value)
  ```

  Set the session timeout value in seconds. A zero value means that the container should use a container-specific default value.

  Parameters:
  `value`
  The timeout value in seconds.

- **`setStateManagementType`**

  ```
  public void setStateManagementType(int value)
  ```

  Set the session Bean's state management type;.

  Returns:
  The session Bean's state management type. Its value must be either STATEFUL_SESSION or STATELESS_SESSION.

19 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 to IDL Mapping. Joint Initial Submission. OMG TC Document TC orbos/98-02-01.

[6] Enterprise JavaBeans to CORBA Mapping. Unpublished JavaSoft document available to the Enterprise JavaBeans reviewers.

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

[8] ORB Portability Submission, OMG document orbos/97-04-14.

Appendix A: Features deferred to future releases

The focus of Release 1.0 is to define the basic component model for session and entity enterprise beans. The model includes: the distributed object model; enterprise bean application programming model; state and transaction management protocols.

Given the broad scope of the Enterprise JavaBeans specification, we defer to future releases the features that introduce an advanced programming style. This conservative approach reduces the chance of our having to make a backward incompatible change in a future release.

Examples of the features that we would like to consider for a later release are listed below.

- Programmatic access to security. We would like to allow expert-level enterprise beans to manage their security Identity.
- Allow a serialized bean prototype. In EJB 1.0, an enterprise bean can be only a Java class, not a serialized Java object. We want to investigate if there would be a value in allowing a serialized object to qualify as an enterprise bean.
- Add the capability for a client to obtain a URL string from an enterprise bean object reference.
- A standard API between the EJB server and EJB container.

Appendix B: package javax.jts

This Appendix provides the documentation of the classes and interfaces that are part of the package *javax.jts* that are relevant to Enterprise JavaBeans. Note that the package *javax.jts* may include other classes and interfaces that are not shown here.

```
interface UserTransaction

class HeuristicCommitException
class HeuristicException
class HeuristicMixedException
class HeuristicRollbackException
class TransactionRequiredException
class TransactionRolledbackException
class InvalidTransactionException
```

Interface UserTransaction

```
public interface javax.jts.UserTransaction
{
    public final static int STATUS_ACTIVE;
    public final static int STATUS_COMMITTED;
    public final static int
        STATUS_COMMITTING;
    public final static int
        STATUS_MARKED_ROLLBACK;
    public final static int
        STATUS_NO_TRANSACTION;
    public final static int STATUS_PREPARED;
    public final static int STATUS_PREPARING;
    public final static int
        STATUS_ROLLEDBACK;
    public final static int
        STATUS_ROLLING_BACK;
    public final static int STATUS_UNKNOWN;
    public abstract void begin();
    public abstract void commit();
    public abstract int getStatus();
    public abstract void rollback();
    public abstract void setRollbackOnly();
    public abstract void
        setTransactionTimeout(int seconds);
}
```

The UserTransaction interface defines the methods that allow an application to explicitly manage transaction boundaries.

Variables

- **STATUS_ACTIVE**

  ```
  public final static int STATUS_ACTIVE
  ```

  A transaction is associated with the target object and it is in the active state. An implementation returns this status after a transaction has been started and prior to a Coordinator issuing any prepares unless the transaction has been marked for rollback.

- **STATUS_COMMITTED**

  ```
  public final static int STATUS_COMMITTED
  ```

  A transaction is associated with the target object and it has been committed. It is likely that heuristics exists, otherwise the transaction would have been destroyed and NoTransaction returned.

- **STATUS_COMMITTING**

  ```
  public final static int STATUS_COMMITTING
  ```

  A transaction is associated with the target object and it is in the process of committing. An implementation returns this status if it has decided to commit, but has not yet completed the process, probably because it is waiting for responses from one or more Resources.

- **STATUS_MARKED_ROLLBACK**

  ```
  public final static int STATUS_MARKED_ROLLBACK
  ```

  A transaction is associated with the target object and it has been marked for rollback, perhaps as a result of a setRollbackOnly operation.

- **STATUS_NO_TRANSACTION**

  ```
  public final static int STATUS_NO_TRANSACTION
  ```

  No transaction is currently associated with the target object. This will occur after a transaction has completed.

- **STATUS_PREPARED**

  ```
  public final static int STATUS_PREPARED
  ```

  A transaction is associated with the target object and it has been prepared, i.e. all subordinates have responded Vote.Commit. The target object may be waiting for a superior's instruction as how to proceed.

- **STATUS_PREPARING**

  ```
  public final static int STATUS_PREPARING
  ```

  A transaction is associated with the target object and it is in the process of preparing. An implementation returns this status if it has started preparing, but has not yet completed the process, probably because it is waiting for responses to prepare from one or more Resources.

- **STATUS_ROLLEDBACK**

  ```
  public final static int STATUS_ROLLEDBACK
  ```

  A transaction is associated with the target object and the outcome has been determined as rollback. It is likely that heuristics exist, otherwise the transaction would have been destroyed and NoTransaction returned.

- **STATUS_ROLLING_BACK**

  ```
  public final static int STATUS_ROLLING_BACK
  ```

  A transaction is associated with the target object and it is in the process of rolling back. An implementation returns this status if it has decided to rollback, but has not yet completed the process, probably because it is waiting for responses from one or more Resources.

- **STATUS_UNKNOWN**

  ```
  public final static int STATUS_UNKNOWN
  ```

  A transaction is associated with the target object but its current status cannot be determined. This is a transient condition and a subsequent invocation will ultimately return a different status.

Methods

- **begin**

  ```
  public abstract void begin()
       throws IllegalStateException
  ```

  Create a new transaction and associate it with the current thread.

  Throws: IllegalStateException

Thrown if the thread is already associated with a transaction.

- **commit**

```
public abstract void commit()
     throws TransactionRolledbackException, HeuristicMixedException,
HeuristicRollbackException, SecurityException, IllegalStateException
```

Complete the transaction associated with the current thread. When this method completes, the thread becomes associated with no transaction.

Throws: TransactionRolledbackException
Thrown to indicate that the transaction has been rolled back rather than committed.

Throws: HeuristicMixedException
Thrown to indicate that a heuristic decision was made and that some relevant updates have been committed while others have been rolled back.

Throws: HeuristicRollbackException
Thrown to indicate that a heuristic decision was made and that some relevant updates have been rolled back.

Throws: SecurityException
Thrown to indicate that the thread is not allowed to commit the transaction.

Throws: IllegalStateException
Thrown if the current thread is not associated with a transaction.

- **getStatus**

```
public abstract int getStatus()
```

Obtain the status of the transaction associated with the current thread.

Returns:
The transaction status. If no transaction is associated with the current thread, this method returns the Status.NoTransaction value.

- **rollback**

```
public abstract void rollback()
     throws IllegalStateException, SecurityException
```

Roll back the transaction associated with the current thread. When this method completes, the thread becomes associated with no transaction.

Throws: SecurityException
Thrown to indicate that the thread is not allowed to roll back the transaction.

Throws: IllegalStateException
Thrown if the current thread is not associated with a transaction.

- **setRollbackOnly**

```
public abstract void setRollbackOnly()
     throws IllegalStateException
```

Modify the transaction associated with the current thread such that the only possible outcome of the transaction is to roll back the transaction.

Throws: IllegalStateException

Thrown if the current thread is not associated with a transaction.

- **`setTransactionTimeout`**

  ```
  public abstract void
  setTransactionTimeout(int seconds)
  ```

  Modify the value of the timeout value that is associated with the transactions started by the current thread with the begin method.

  If an application has not called this method, the transaction service uses some default value for the transaction timeout.

  Parameters:

  `seconds`
  The value of the timeout in seconds. If the value is zero, the transaction service restores the default value.

Class HeuristicCommitException

```
public class javax.jts.HeuristicCommitException
    extends java.rmi.RemoteException
{
    public HeuristicCommitException();
    public
        HeuristicCommitException(String msg);
}
```

This exception is thrown by the rollback operation on a resource to report that a heuristic decision was made and that all relevant updates have been committed.

Constructors

- **HeuristicCommitException**

  ```
  public HeuristicCommitException()
  ```

- **HeuristicCommitException**

  ```
  public HeuristicCommitException(String msg)
  ```

Class HeuristicException

```
public class javax.jts.HeuristicException
    extends java.rmi.RemoteException
{
    public HeuristicException();
    public HeuristicException(String msg);
}
```

This exception indicates that one or more participants in a transaction has made a unilateral decision to commit or roll back updates without first obtaining the outcome determined by the transaction service.

Heuristic decisions are normally made only in unusual circumstances, such as communication failures, that prevent normal processing. When a participant makes a heuristic decision, there is a risk that the decision will differ from the consensus outcome, potentially resulting in loss of data integrity.

The subclasses of this exception provide more specific reporting of the incorrect heuristic decision or the possibility of incorrect heuristic decision.

Constructors

- **HeuristicException**

  ```
  public HeuristicException()
  ```

- **HeuristicException**

  ```
  public HeuristicException(String msg)
  ```

Class HeuristicMixedException

```
public class javax.jts.HeuristicMixedException
    extends java.rmi.RemoteException
{
    public HeuristicMixedException();
    public
        HeuristicMixedException(String msg);
}
```

This exception is thrown to report that a heuristic decision was made and that some relevant updates have been committed and others have been rolled back.

Constructors

- **HeuristicMixedException**

  ```
  public HeuristicMixedException()
  ```

- **HeuristicMixedException**

  ```
  public HeuristicMixedException(String msg)
  ```

Class HeuristicRollbackException

```
public class javax.jts.HeuristicRollbackException
    extends java.rmi.RemoteException
{
    public HeuristicRollbackException();
    public
        HeuristicRollbackException(String msg);
}
```

This exception is thrown by the commit operation to report that a heuristic decision was made and that all relevant updates have been rolled back.

Constructors

- **HeuristicRollbackException**

  ```
  public HeuristicRollbackException()
  ```

- **HeuristicRollbackException**

  ```
  public HeuristicRollbackException(String msg)
  ```

Class TransactionRequiredException

```
public class javax.jts.TransactionRequiredException
    extends java.rmi.RemoteException
{
    public TransactionRequiredException();
    public
        TransactionRequiredException(String msg);
}
```

This exception indicates that a request carried a null transaction context, but the target object requires an activate transaction.

Constructors

- **TransactionRequiredException**

  ```
  public TransactionRequiredException()
  ```

- **TransactionRequiredException**

  ```
  public TransactionRequiredException(String msg)
  ```

Class TransactionRolledbackException

```
public class javax.jts.TransactionRolledbackException
    extends java.rmi.RemoteException
{
    public TransactionRolledbackException();
    public
        TransactionRolledbackException(String msg);
}
```

This exception indicates that the transaction associated with processing of the request has been rolled back, or marked to roll back. Thus the requested operation either could not be performed or was not performed because further computation on behalf of the transaction would be fruitless

Constructors

- **TransactionRolledbackException**

  ```
  public TransactionRolledbackException()
  ```

- **TransactionRolledbackException**

  ```
  public TransactionRolledbackException(String msg)
  ```

Class InvalidTransactionException

```
public class javax.jts.InvalidTransactionException
    extends java.rmi.RemoteException
{
    public InvalidTransactionException();
    public
        InvalidTransactionException(String msg);
}
```

This exception indicates that the request carried an invalid transaction context. For example, this exception could be raised if an error occurred when trying to register a resource.

Constructors

- **InvalidTransactionException**

  ```
  public InvalidTransactionException()
  ```

- **InvalidTransactionException**

  ```
  public InvalidTransactionException(String msg)
  ```

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 12.3.4 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.