Sun Microsystems

Enterprise JavaBeans™

This is a draft specification of the Enterprise JavaBeans architecture for public review. We expect to finalize this specification after 60 - 90 days of public review.

Enterprise JavaBeans is a component architecture for development and deployment of object-oriented distributed enterprise-level Java applications. Applications written using Enterprise JavaBeans are scalable, transactional, and multi-user secure. These applications can be written once, and then deployed on any Java enabled server platform.

Please send technical comments on this specification to:

ejb-spec-comments@sun.com

Please send product and business questions to:

ejb-marketing@sun.com

Version 0.9

Vlada Matena & Mark Hapner

February 4, 1998 4:46 pm

Contents

1 Introduction

1.1 Preface

We would like to thank all the reviewers who sent us feedback on Release 0.8 of the Enterprise JavaBeans specification. We have read and incorporated the feedback and comments from as many reviewers as we could, and will try to incorporate the remaining comments into the next revision of the specification.

We thank the reviewers for their time and effort in helping us to improve the specification.

1.2 Target audience

The target audience 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 the application programmer using Enterprise JavaBeans will not have to deal with. 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.

1.3 Acknowledgments

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

Enterprise JavaBeans is a broad effort that includes contribution 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.

1.4 Organization

TODO - describe document organization

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 Java. 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 deal with low-level details of transaction and state management; 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 Java. 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 APIs.
- The Enterprise JavaBeans architecture will provide interoperability between enterprise beans and non-Java 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 five 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 five 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.

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 Java classes, its deployment descriptor, and its environment properties.

3.1.2 Application assembler

An application assembler is a domain expert who composes applications out of enterprise beans. An application assembler produces an ejb-jar file that contains enterprise beans with their deployment descriptors and environment properties. The ejb-jar file might include additional files that are part of the assembled application but whose definition is outside of Enterprise JavaBeans.

The assembler may also write a GUI client-side for the applications.

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.

3.1.4 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 part of this specification).

For enterprise entity beans with container-managed persistence, the entity container is typically 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 middle-tier application. The container provider may be an expert in the area of providing access to enterprise's existing data sources or applications.

A container provider is responsible for the evolution of enterprise beans. For example, the container provider should allow enterprise bean classes to be upgraded without invalidating existing clients or losing existing enterprise bean objects.

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 server environment with existing applications (such a container can make the existing applications appear as regular enterprise beans), a container that implements an Object/Relational mapping, or container that is built on top of an object-oriented database system.

3.1.5 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.

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

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 is contained 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 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 EJB object. For entity enterprise beans (See Subsection 4.3.2), the EJB provider is responsible for supplying a unique primary key that the container embeds into the EJB object's identifier.

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

An enterprise bean and its container cooperate to implement the create, find, and destroy 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* interface. A home interface defines zero or more *create(...)* methods, one for each way to create an EJB object. A home interface for entity beans may define 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 *ejbFindMETHOD(...)* methods in the enterprise bean class, whose signature must match those of the *findMETHOD(...)* finder methods defined in the bean's home interface. The container is responsible for delegating a client-invoked *findMETHOD(...)* method to the matching *ejbFindMETHOD(...)* 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.

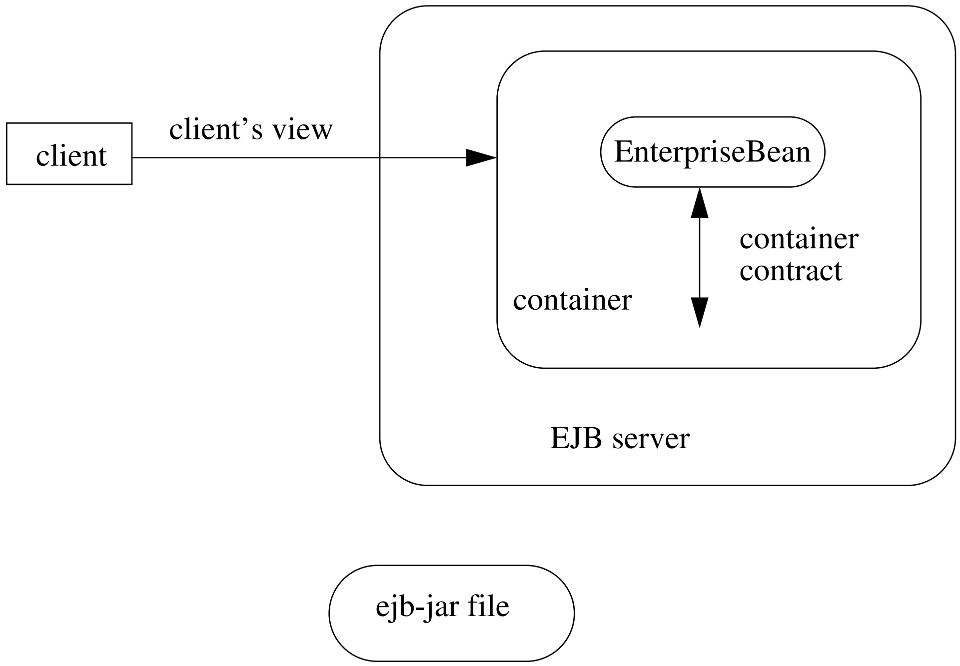
4.2.3 Ejb-jar file

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 content of the ejb-jar file.
- Java class files for the enterprise beans.
- Enterprise beans' 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 beans' 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.

The support for session objects is mandatory for an EJB 1.0 compliant container. The 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 data in an underlying database.

- Relatively short-lived.
- Is destroyed when the EJB server crashes. The client has to re-establish a new session object to continue computation.
- Does not represent data that should be stored in a database.

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.
- Transactional.
- 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:

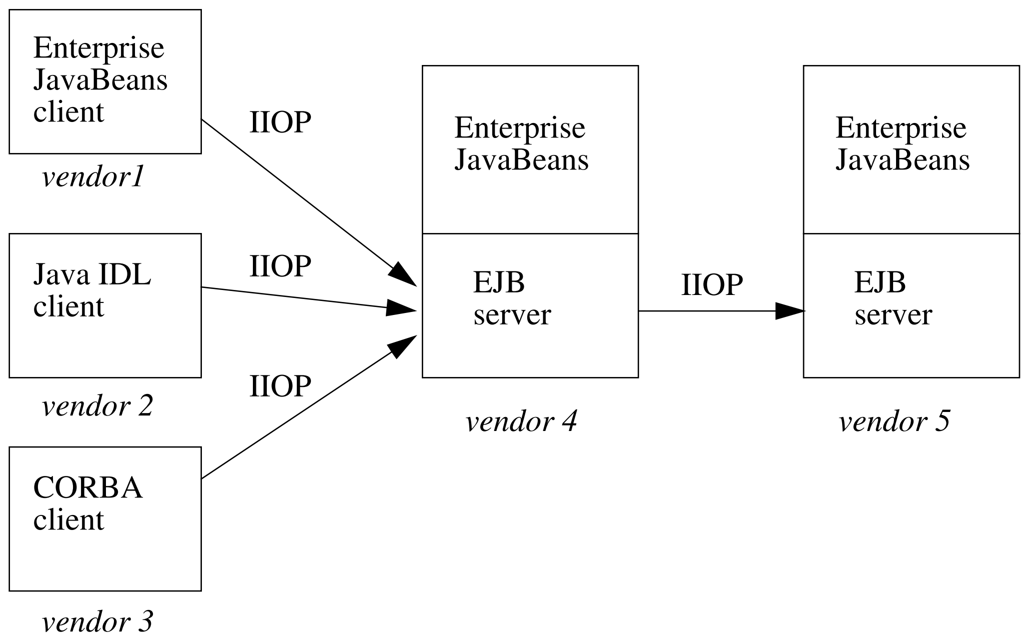
1. Mapping of the EJB client interfaces to CORBA IDL.
2. Propagation of transaction context.
3. Propagation of security context.

The Enterprise JavaBeans to CORBA mapping not only enables on-the-wire interoperability among multiple vendors' implementations of an EJB server, but it 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 the remote interface is called an *EJB object.* An EJB object is a remote Java 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 destroy 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

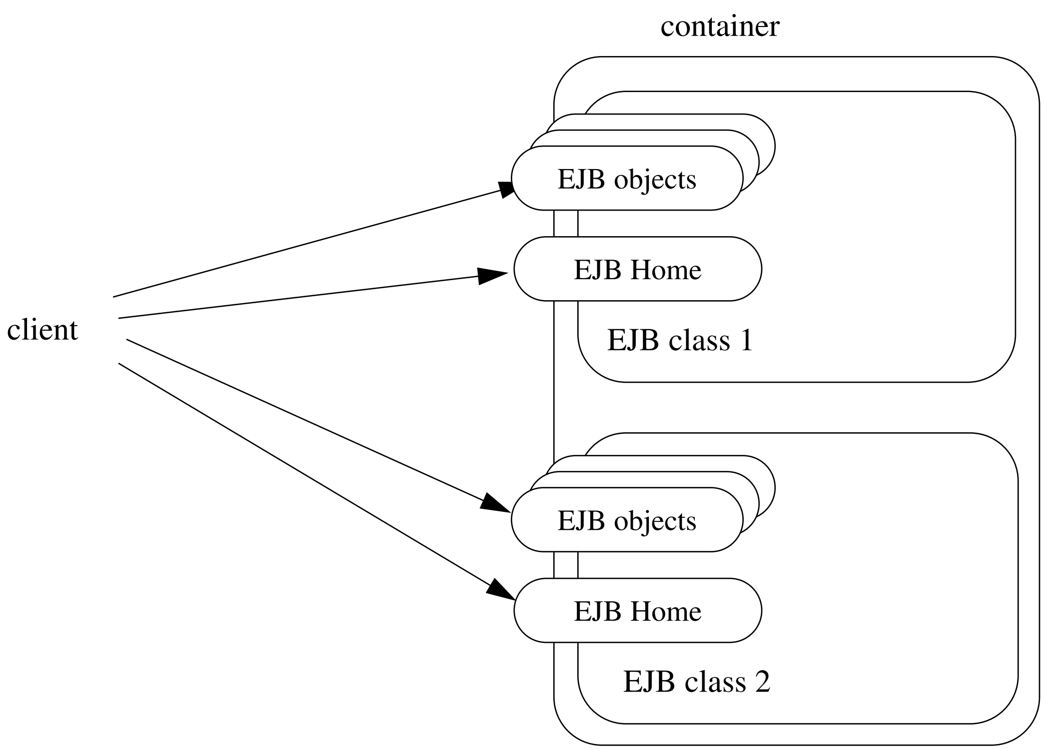
A client locates an enterprise bean's home interface using JNDI. For example, a container for *Cart* EJB objects can be located using the following code segment:

```
Context initialContext = new InitialContext();
CartHome cartHome = (CartHome)
        initialContext.lookup("applications/mall/freds-carts");
```

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 new EJB objects of a given class in the container.

- Destroy an EJB object of a given class.
- 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;
}
```

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 Destroying an EJB object

The *javax.ejb.EJBHome* interface defines several methods that allow a client to destroy an EJB object. In addition, a client may destroy an EJB object using the *destroy()* 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. *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.
  - Destroy 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

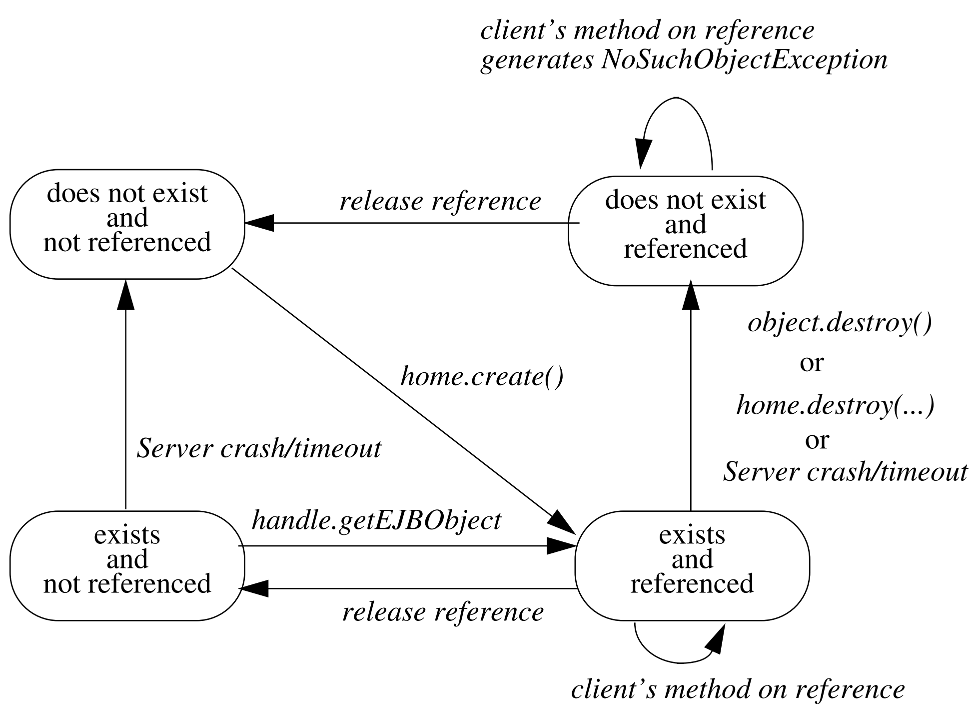
Session objects are intended to be private resources used only by the client that created them. For this reason, session EJB objects, from the clients perspective, appear anonymous. In contrast to entity EJB objects which expose their identity as a primary key, session objects hide their identity.

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 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:

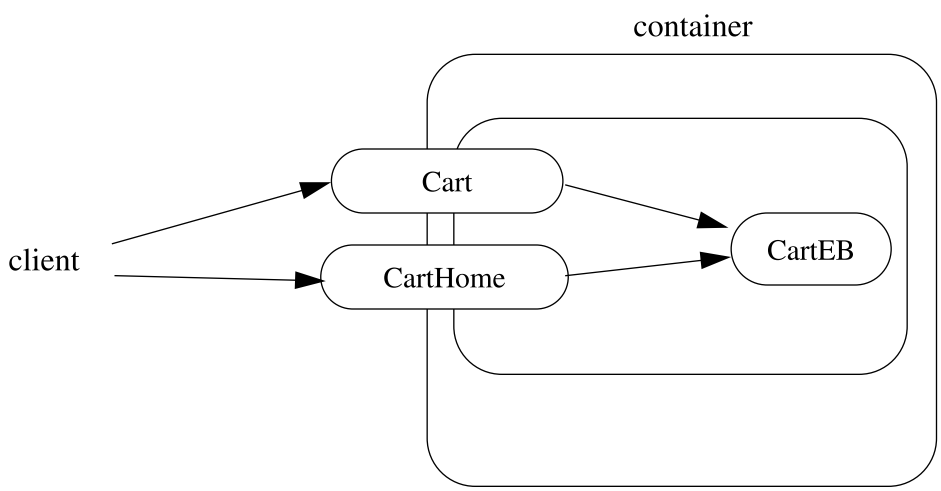
- Invoke application methods on the object through the session bean's remote interface.
- Get a reference to the object's home interface.
- Get a handle for the object

- Pass the object as a parameter or return value within the scope of the client.
- Destroy the object. A container may also destroy 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.ejb.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 = (CartHome) lookup(...);
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.destroy();
```

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 database updates it is caching prior to the bean's transaction completion, and it must refresh any potentially stale database data it contains 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 Java Serialization protocol—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 Serialization (or an equivalent of Java Serialization) 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 Java Serialization protocol 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 strongly discouraged to avoid having to deal with the indeterministic state of the transient variables after an instance has been passivated and re-activated.

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 Java Serialization protocol 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 *ejbDestroy* notification signals that the instance is in the process of being destroyed 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 when it is reactivated.

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 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 immediate 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 to roll it back.
- The *getCurrentTransaction* method returns the *javax.jts.CurrentTransaction* 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 provides 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.

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.

1.The container makes the *CurrentTransaction* interface available only to the beans deployed with the *BEAN_MANAGED* transaction attribute. The *getCurrentTransaction* interface will fail if invoked by a bean that is deployed with any of the other transaction attributes.

6.5.6 Serializing session bean methods

A container serializes calls to each of its bean instances. Most containers will support many instances of a bean executing concurrently; however, each instance sees only a serialized sequence of method calls.

The method calls a container serializes includes those delivered via an instance's EJB object as well as the service callbacks made by the container itself.

6.5.7 Transaction context of session bean methods

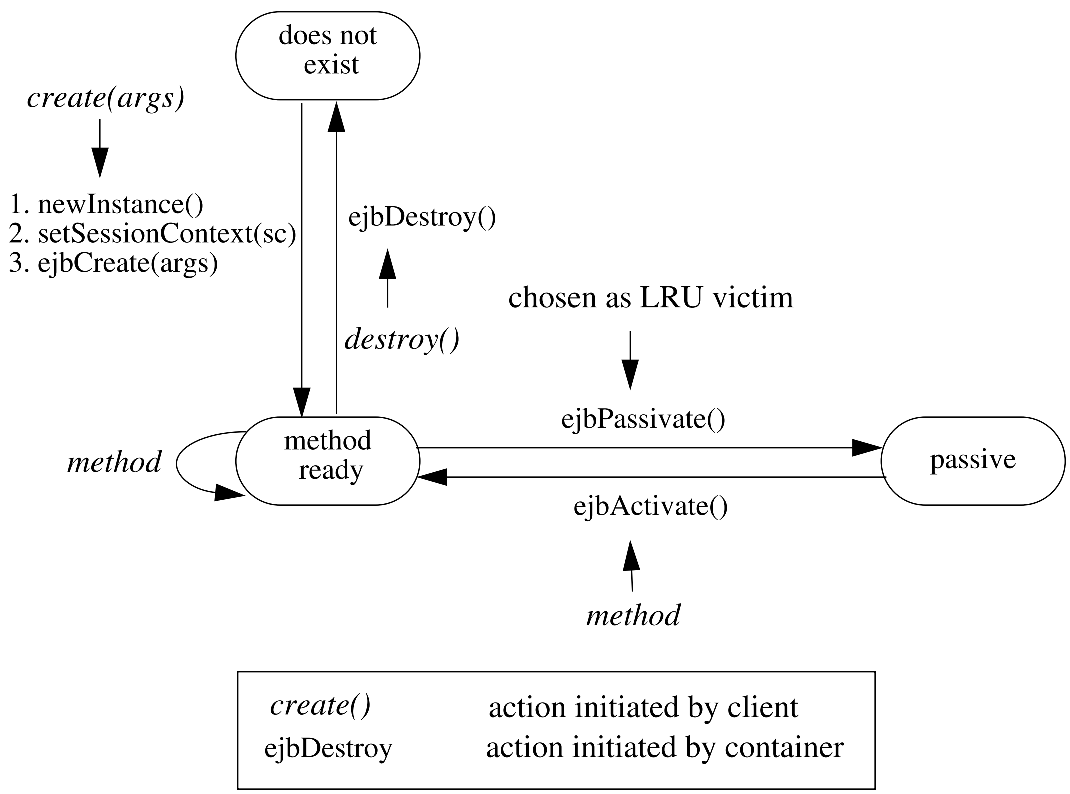
A session bean's *afterBegin* and *beforeCompletion* methods are always called with the proper transaction context (if the bean is transactional).

A session bean's *newInstance, setSessionContext*, *ejbCreate*, *ejbDestroy*, *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 *ejbDestroy* method.

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

6.6 STATEFUL non-transactional session bean state diagram

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



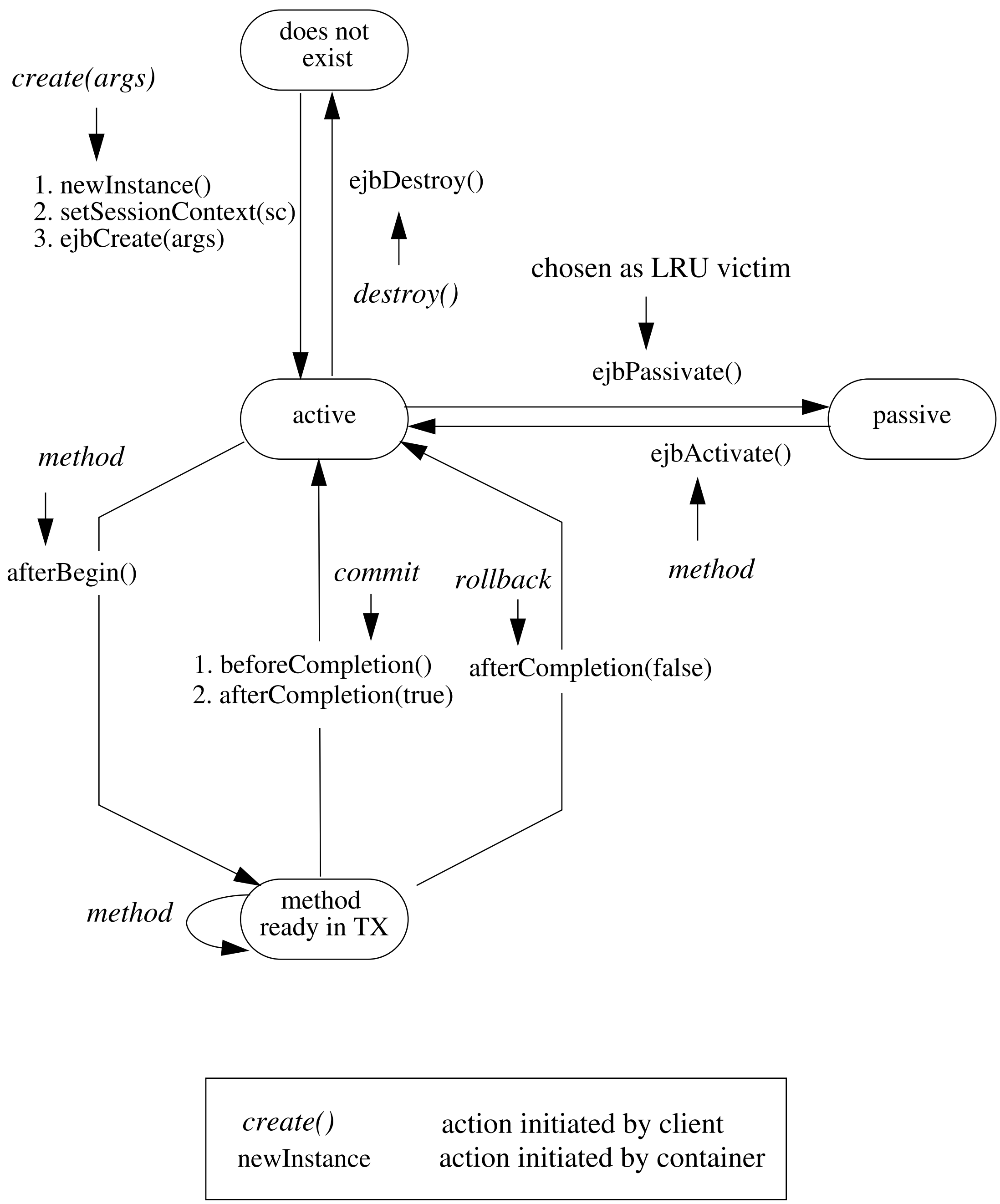
The following is a walk-through the lifecyle of a session bean instance:

- A non-transactional session bean's life starts when a client invokes a c*reate(...)* method on the 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 creates a session context for the instance, and calls *setSessionContext()* followed by *ejbCreate(...)* on the instance, and returns an EJB object to the client.
- The bean instance is now ready for client methods.
- 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 will issue *ejbPassivate()* on the instance. After this completes, the container must save the instance's state to secondary storage.

- If a client invokes a method on this passivated instance, the container activates the session instance prior to delegating the method invocation to it. To activate the session instance, the container restores the instance's state from secondary storage and issues *ejbActivate()* on it.
- The enterprise bean is again ready for client methods.
- When the client calls *destroy()* on the EJB object, this causes the container to issue *ejbDestroy()* 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 call the *destroy()* method on the EJB object after the lifetime of the EJB object has expired.

6.7 STATEFUL transactional session bean state diagram

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



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

- A transactional 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 to the client.
- The bean instance is now ready to be included in a transaction.
- After the bean instance is included in a transaction and before any of its other methods are executed within the transaction, the container issues *afterBegin* on it.
- Bean methods invoked by the client in this transaction can now be delegated to the bean instance.
- If a transaction commit has been requested, prior to actually committing the transaction, the container issues *beforeCompletion* on the instance. This is when the instance should write any cached updates to the database.
- The container 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.
- If this passivated instance's EJB object is included in a transaction, the container will activate the session instance. To activate the session instance, the container restores the instance's state from secondary storage and issues *ejbActivate()* on it.
- The enterprise bean is again ready to be included in a transaction.
- When the client calls *destroy()* on the EJB object, this causes the container to issue *ejbDestroy()* 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 *destroy()* method on the instance after the lifetime of the EJB object has expired.

6.8 Sequence diagrams for a STATEFUL session bean

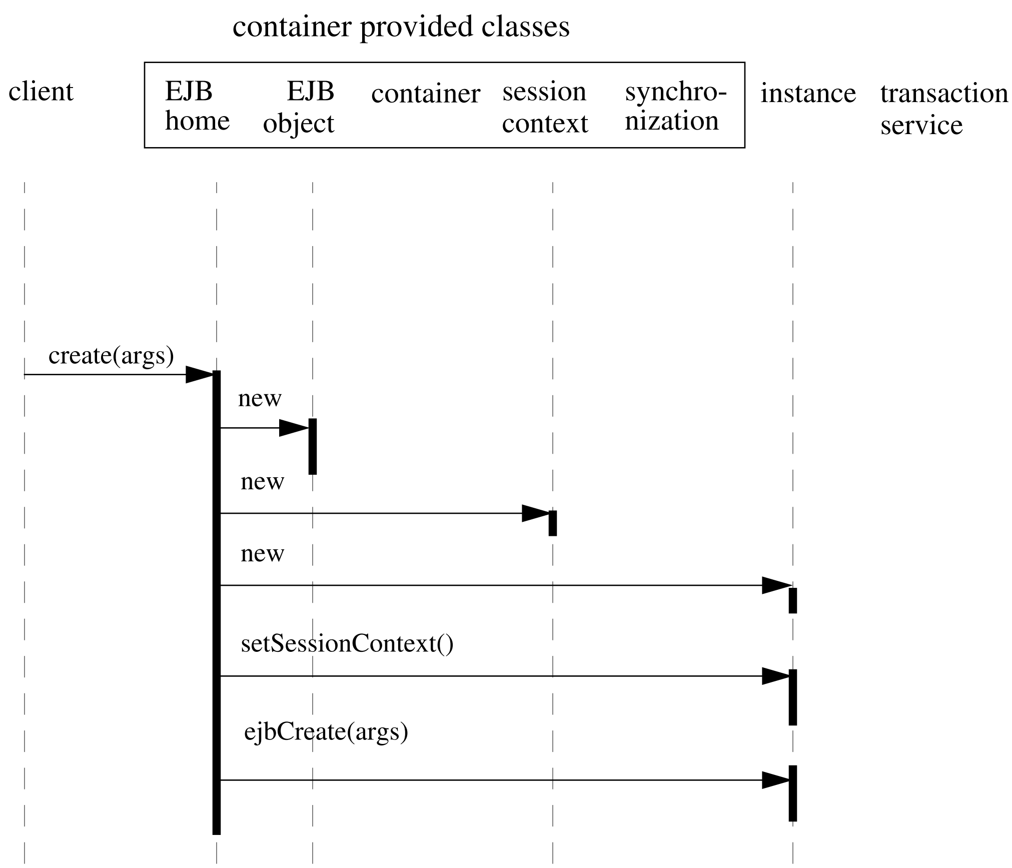
This section contains sequence diagrams that illustrates the interaction of the classes.

6.8.1 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.

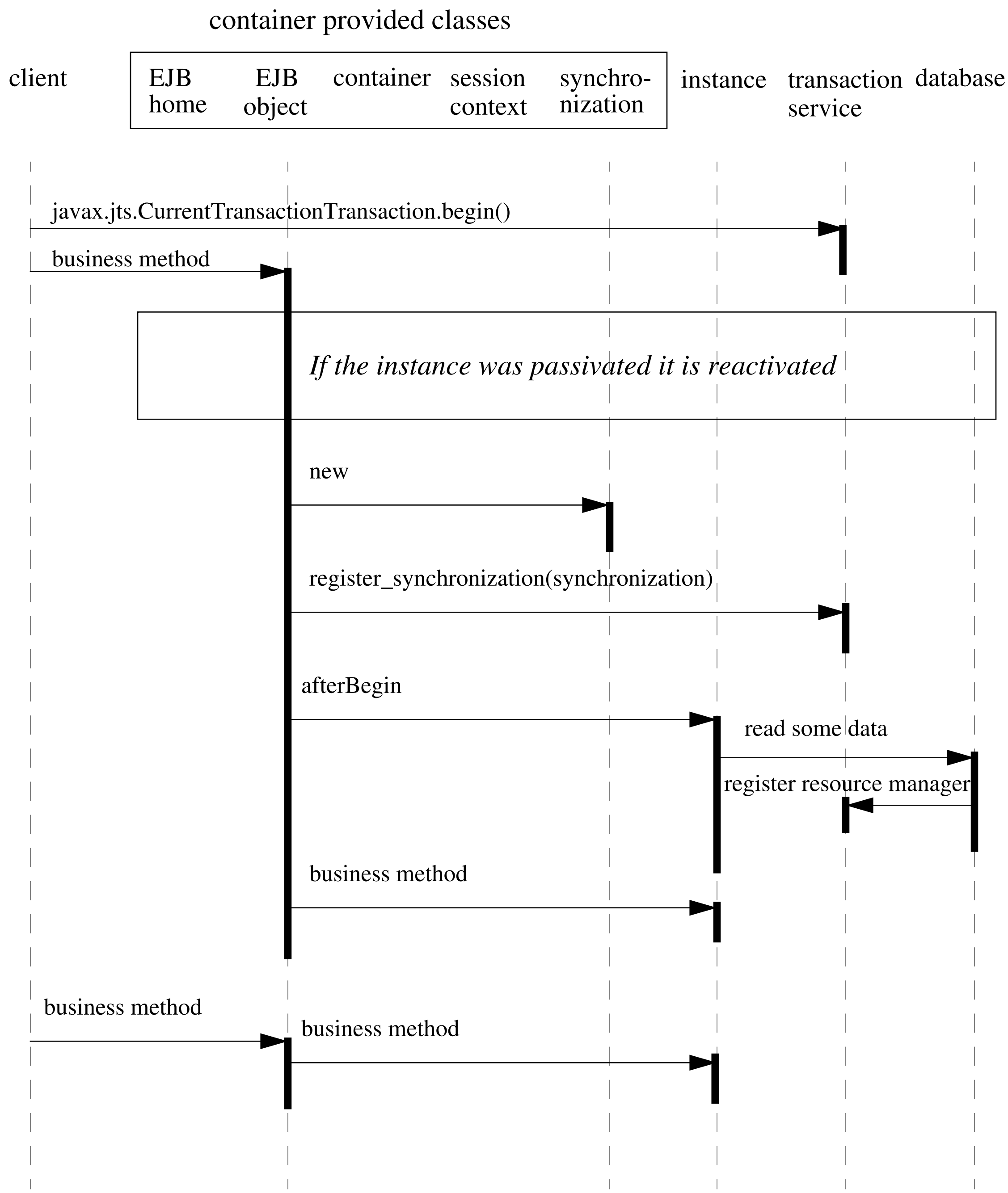
6.8.2 Creating a session object

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



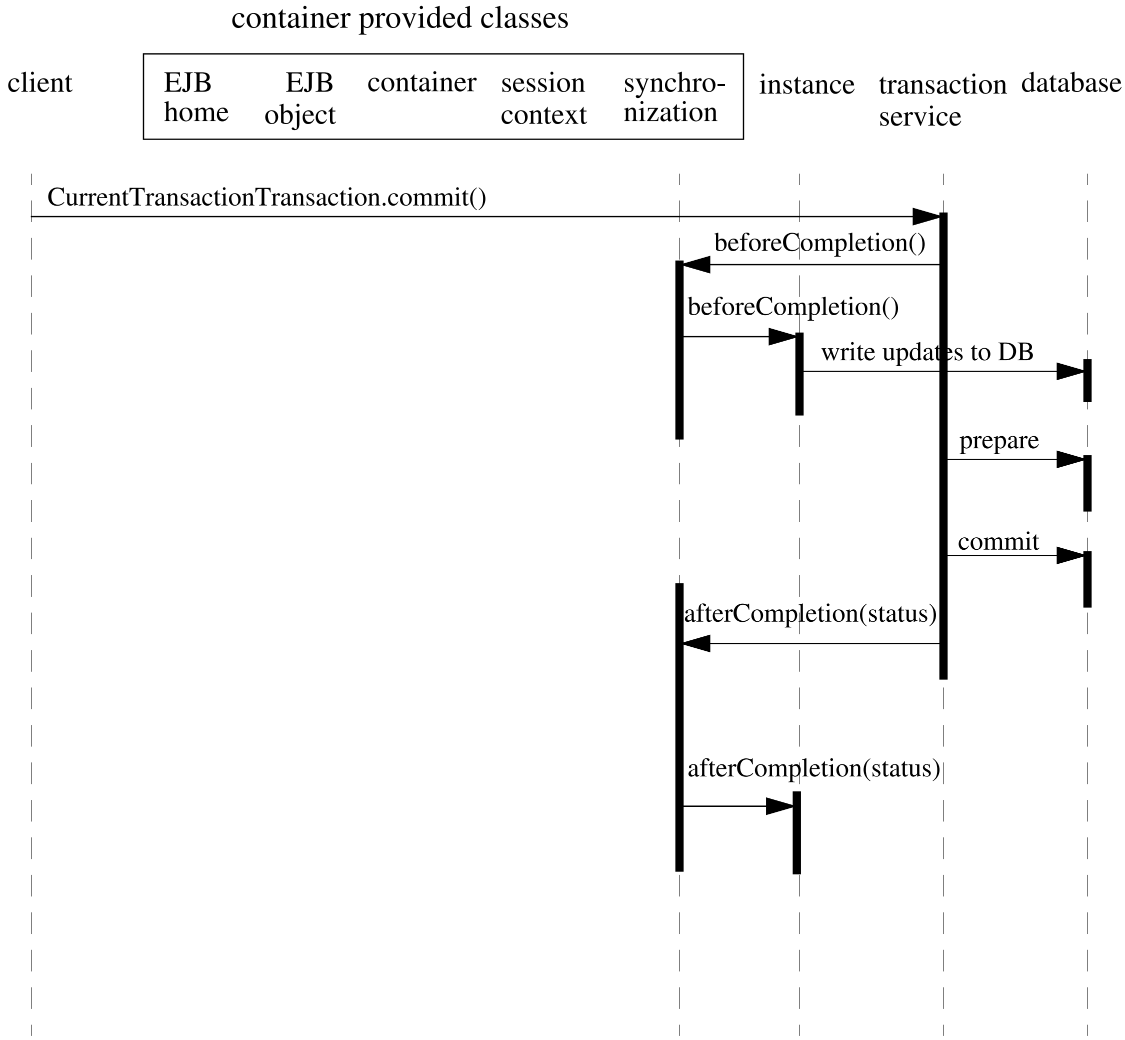
6.8.3 Starting a transaction

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



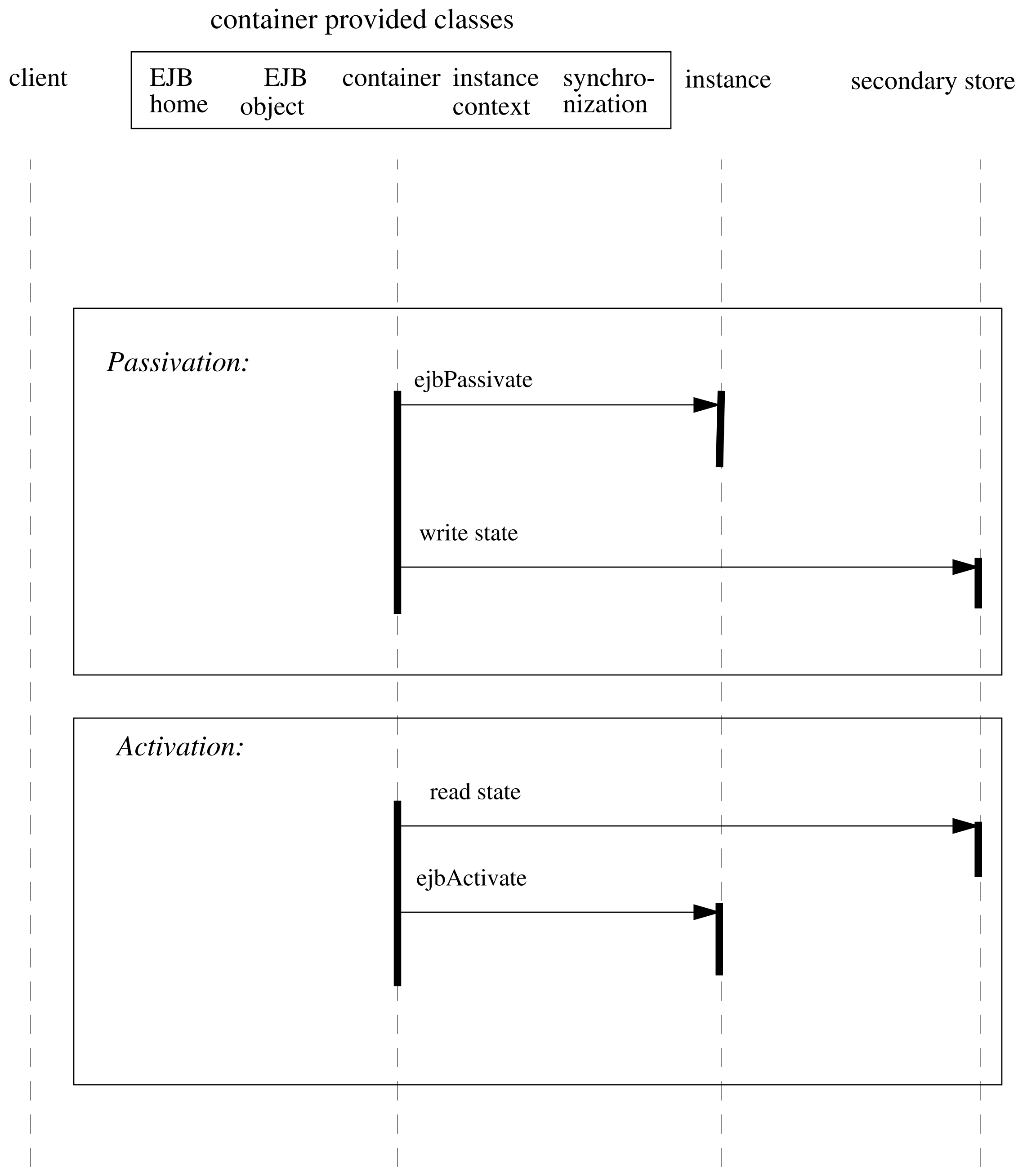
6.8.4 Committing a transaction

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



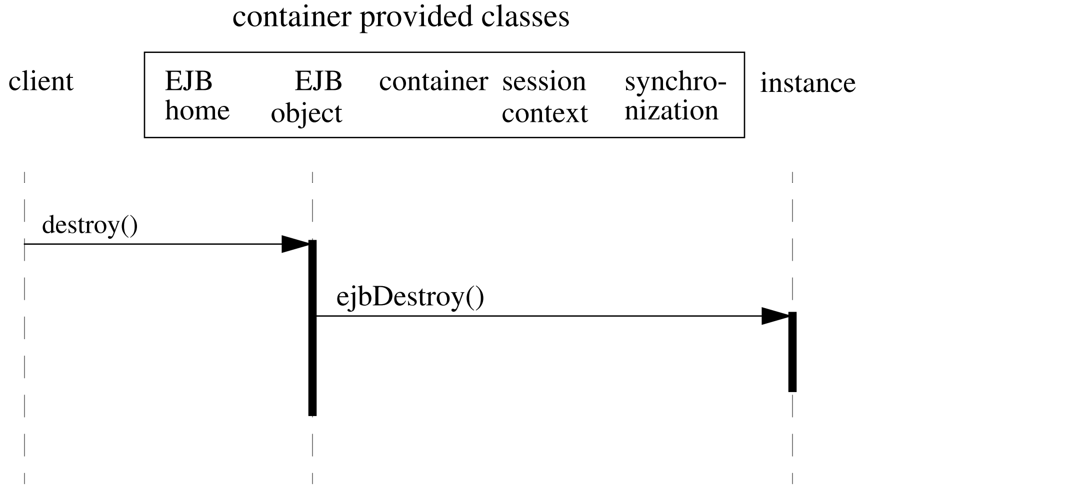
6.8.5 Passivating and activating an instance between transactions

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.



6.8.6 Destroying a session object

The following diagram illustrates the destruction of a session bean.



6.9 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.

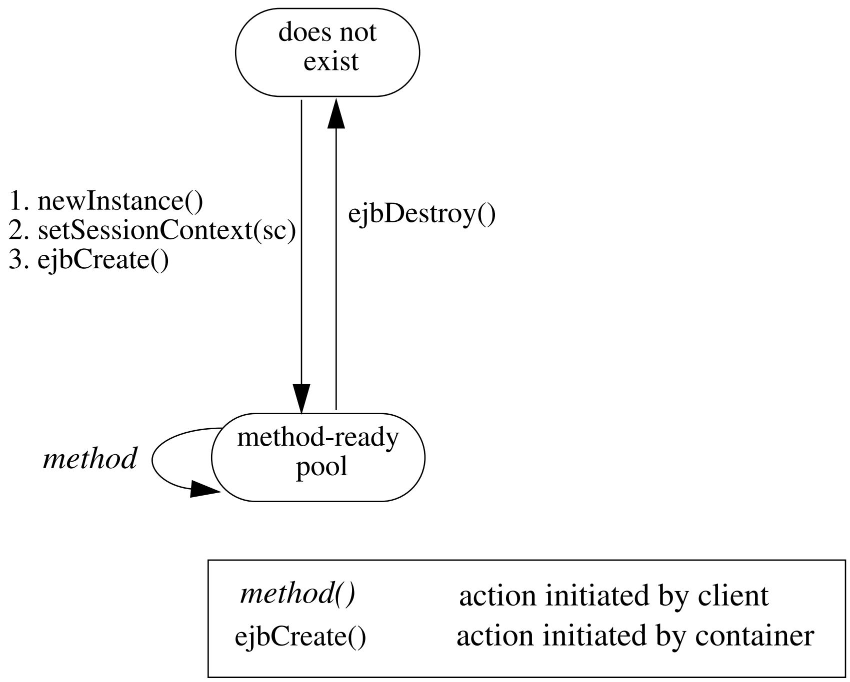
Clients use the *create* and *destroy* method on the home interface of a stateless session bean just like they do on a stateful session bean. Although the client thinks it is controlling the life cycle of an EJB instance, the container is handling the *create* and *destroy* calls without necessarily creating and destroying 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.

6.9.1 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:

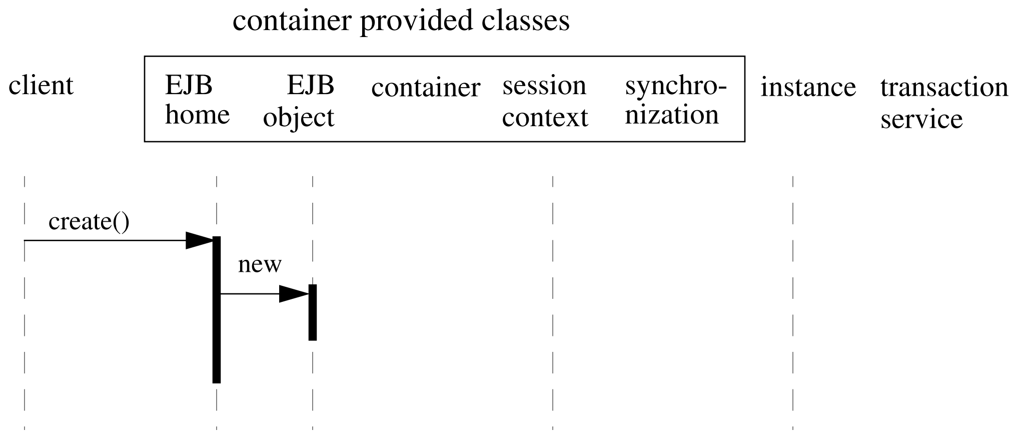
- A non-transactional session bean's life starts when the container invokes *newInstance()* on the bean class to create a new memory object for the enterprise bean. Next, the container calls *setSessionContext()* followed by *ejbCreate()* on the instance. The container can perform the instance creation 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 *ejbDestroy()* on it. This ends the life of the stateless session bean instance.

6.10 Sequence diagrams for a STATELESS session bean

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

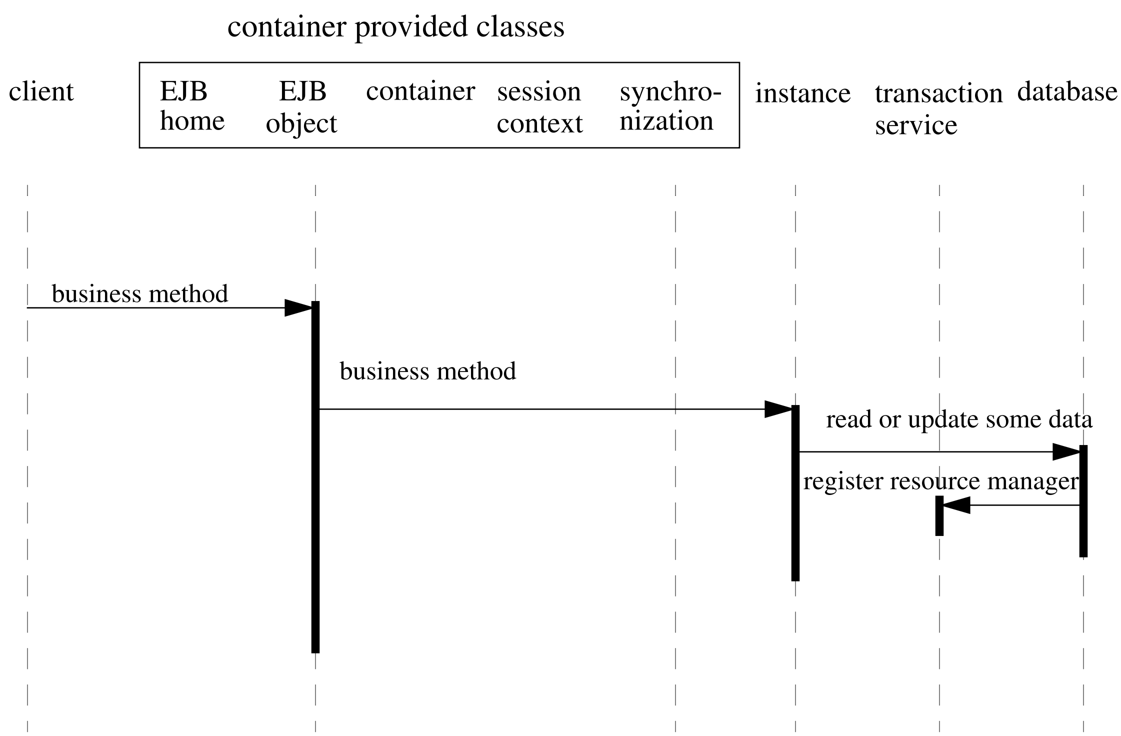
6.10.1 Client-invoked *create()*

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



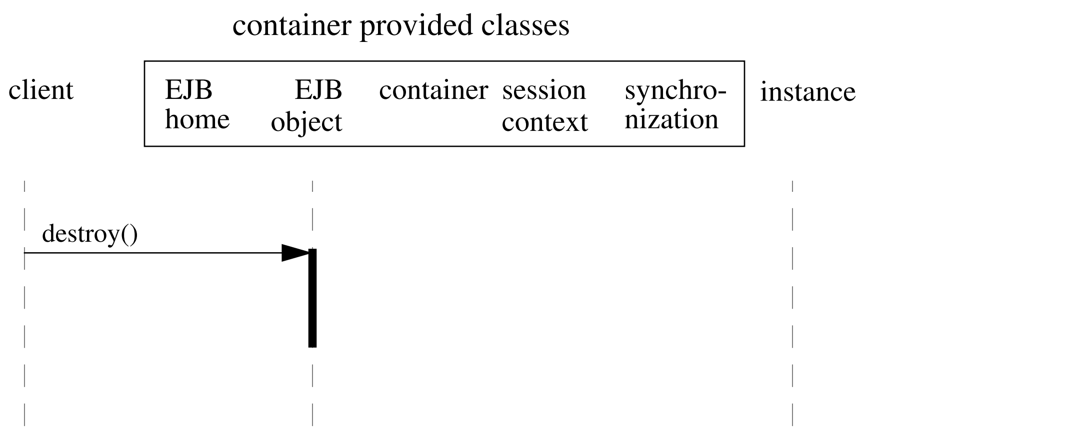
6.10.2 Business method invocation

The following diagram illustrates the invocation of a business method.



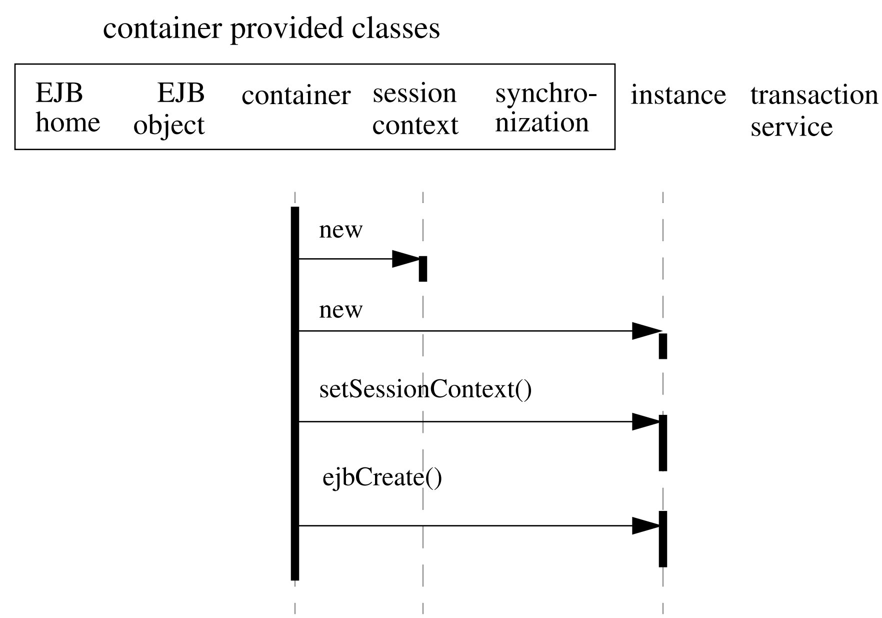
6.10.3 Client-invoked *destroy()*

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

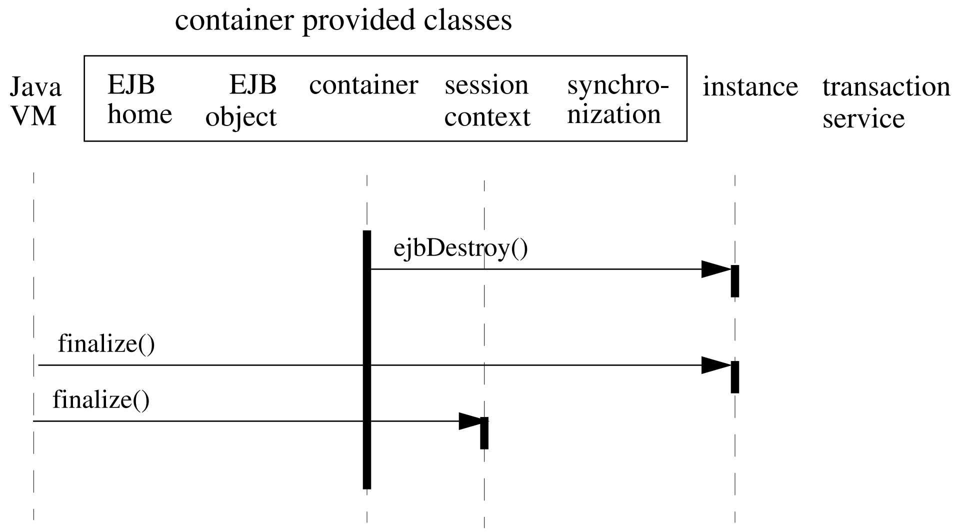


6.10.4 Adding instance to the pool

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



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



6.11 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.

6.11.1 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.

6.11.2 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 must not implement the enterprise bean's remote interface.

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

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

6.11.3 *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 *void*.

The methods arguments must be legal types for Java RMI.

The throws clause may define arbitrary application specific exceptions. The throws clause may also include *javax.ejb.EJBException,* but it must not include the *java.rmi.RemoteException*.

6.11.4 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. The throws clause may also include the *javax.ejb.EJBException*, and it must not include the *java.rmi.RemoteException*.

6.11.5 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. The only exception to this rule is the *javax.ejb.EJBException* that must not be defined in the throws clause of the remote method.

6.11.6 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*.

Each method defined in the home interface must be one of the following:

- A *create* 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 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 only exception to this rule is the *javax.ejb.EJBException* that must not be defined in the throws clause of the *create* method.

6.12 The responsibilities of the container provider

This section describes the responsibilities of the container provider to support a session bean.

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

6.12.2 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 *destroy(...)* methods defined in the *javax.ejb.EJBHome* interface must activate the instance (if the instance is in the passive state) and invoke the *ejbDestroy* method on the instance.

6.12.3 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 *destroy(...)* method (defined in the *javax.ejb.EJBObject* interface) must activate the instance (if the instance is in the passive state) and invoke the *ejbDestroy* 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.

6.12.4 Single-threaded rule

The container must ensure that only one thread can be executing an instance at any time.The exception to this rule is that a container must allow callbacks to an instance within the same transaction. For example, if a client invokes enterprise bean A, A calls another enterprise bean B, and B invokes a callback method on A, the container must allow the callback from B to A, even though A is still executing the client's call.

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. The EJB specification leaves the container behavior implementation specific if concurrent access occurs.

6.12.5 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.

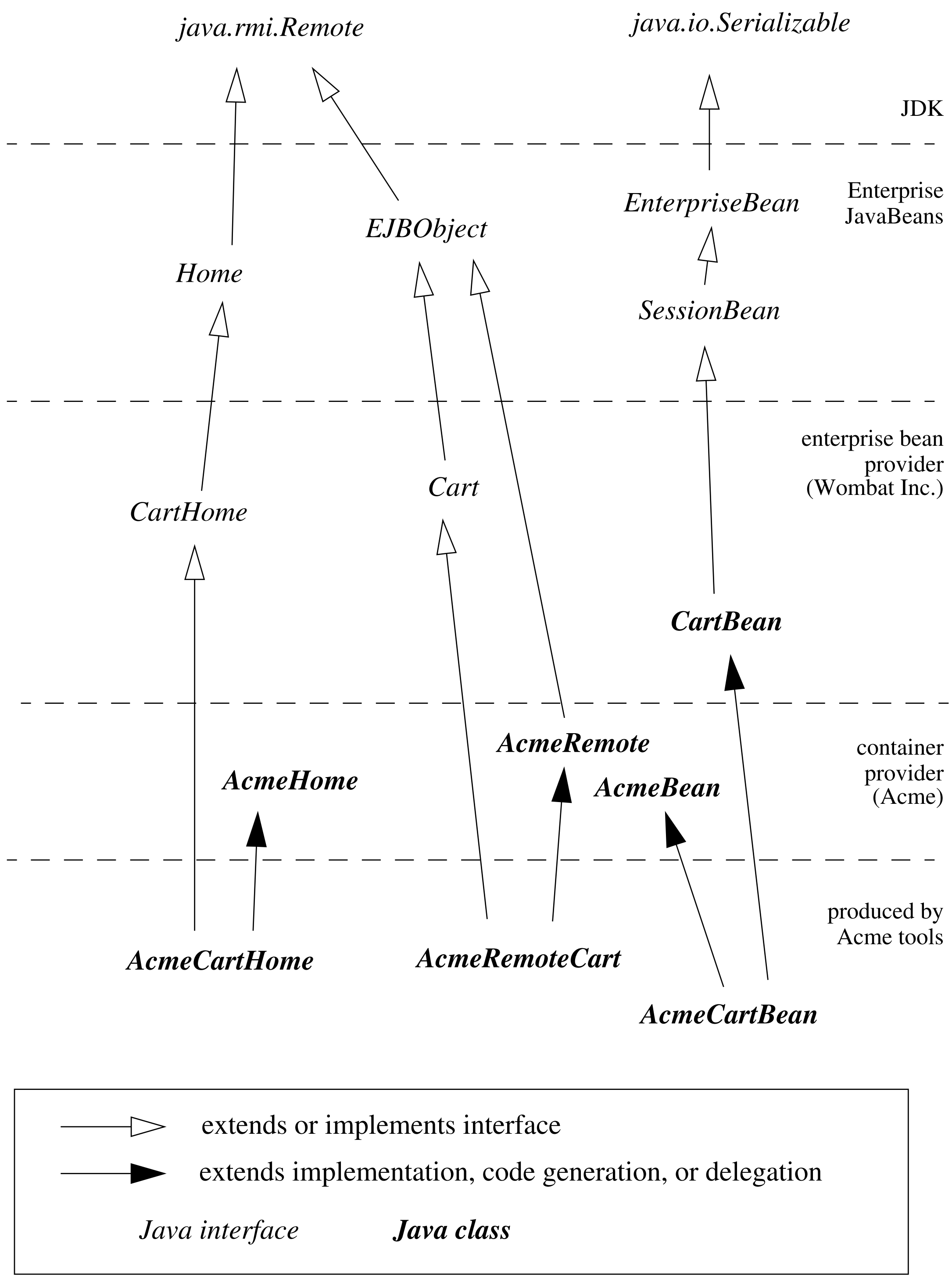
The classes generated by the container provider's tools in this scenario should be considered illustrative rather than prescriptive. Container providers are free to implement the contract between a session enterprise bean and its container in a different way that achieves an equivalent effect (from the 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 must not 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.

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.

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 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 destroy 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 does not break if an enterprise bean is moved to a different container, or its implementation is changed.

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 destroy EJB objects of the corresponding EJB class. The container makes the enterprise beans’ home interfaces available in the JNDI name space to allow clients to look up the home interfaces through JNDI.

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 now way for a client to tell in which container an enterprise bean is installed.

8.2.1 Locating enterprise bean’s home interface

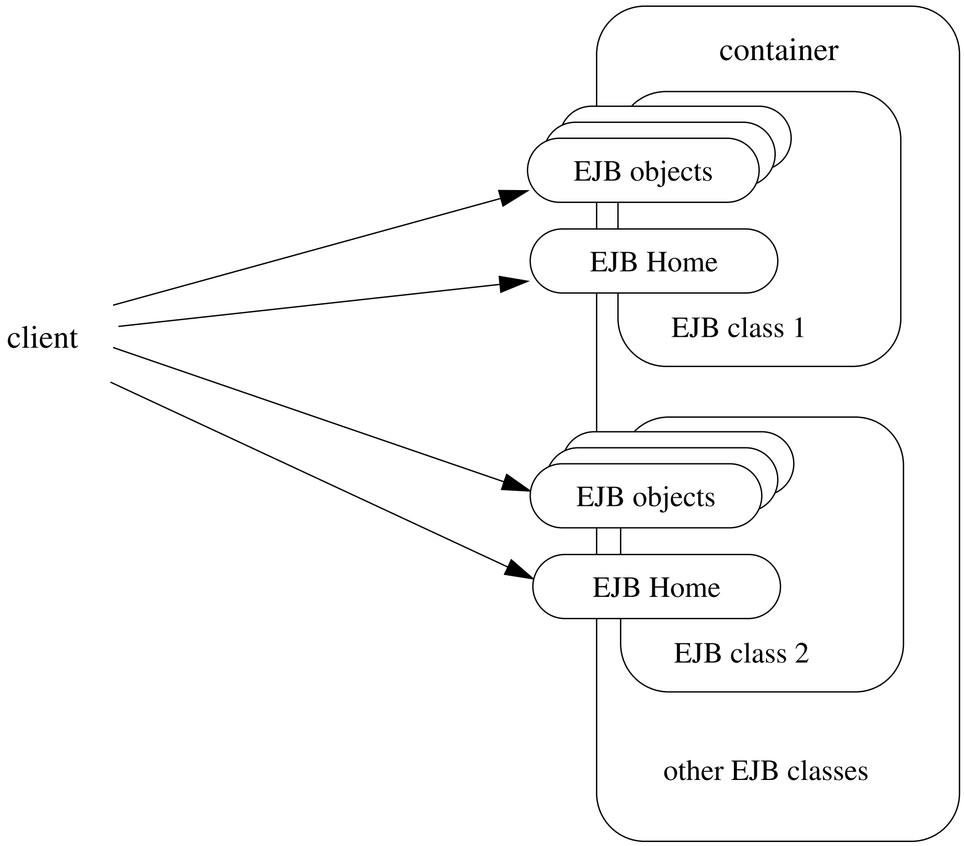
A client locates an enterprise bean’s home interface using JNDI. For example, the home interface for the Account enterprise bean can be located using the following code segment:

```
Context initialContext = new InitialContext();
AccountHome accountHome = (AccountHome)
        initialContext.lookup(“applications/bank/accounts”);
```

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.
- Destroy 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 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*. The throws clause typically includes also the *javax.ejb.CreateException*.

The following is an example of the *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;
         ...
}
```

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 a 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 single argument that is of the enterprise bean's primary key type, and its return type is the enterprise bean's remote interface.

The following is an example of the *findByPrimaryKey* method:

```
public interface AccountHome extends javax.ejb.EJBHome {
         ...
```

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

```
        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 *destroy* methods

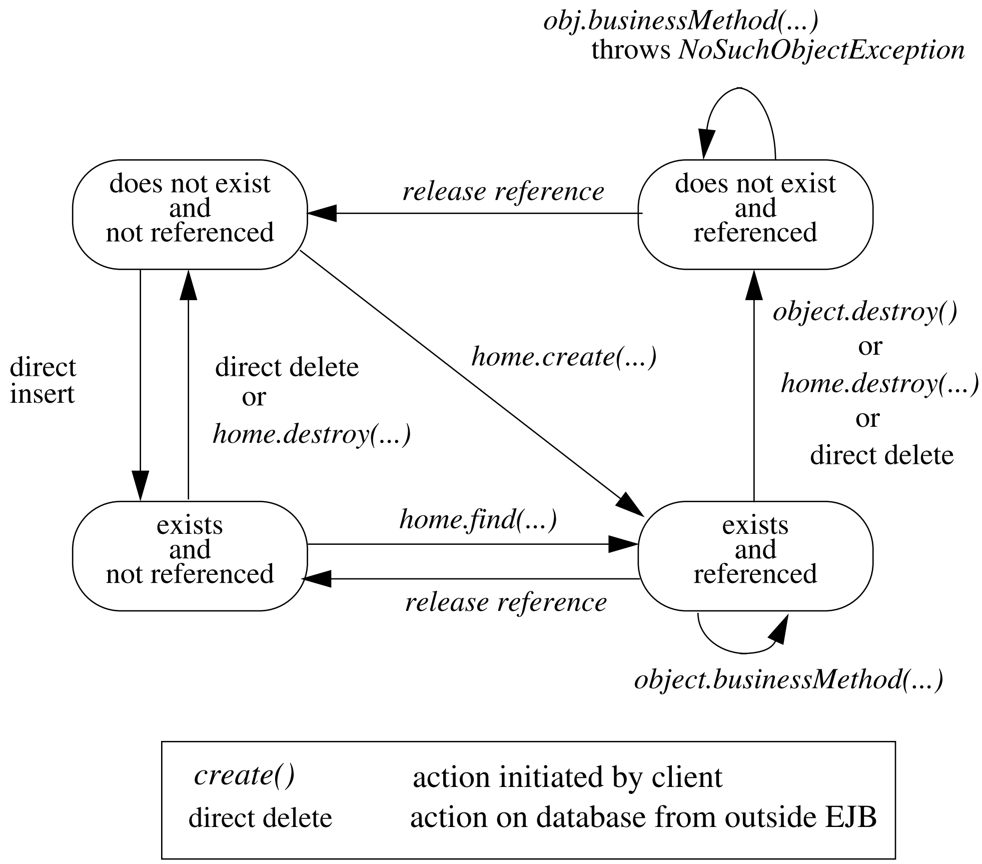
The *javax.ejb.EJBHome* interface defines several methods that allow the client to destroy EJB objects.

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

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 of 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 *destroy()* operation (e.g. by deletion of a database record). The "direct insert" and "direct delete" transitions in the diagram represent such direct database manipulation.

For an existing EJB object, a client can get a reference to an 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.
- Destroy 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 EJB object's 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 difference 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 equality operator. This method works only 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 *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.
- Destroy 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* and *javax.ejb.EntityControl* interfaces to the client. These interfaces are not intended for the client—they are for 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 implements 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 resurrect
// an object reference to the account EJB object from the handle.
//
ObjectInputStream stream = ...;
Handle handle = (Handle) stream.readObject(handle);
Account account = (Account) handle.getEJBObject();
account.debit(100.00);
```

9 Entity bean component contract

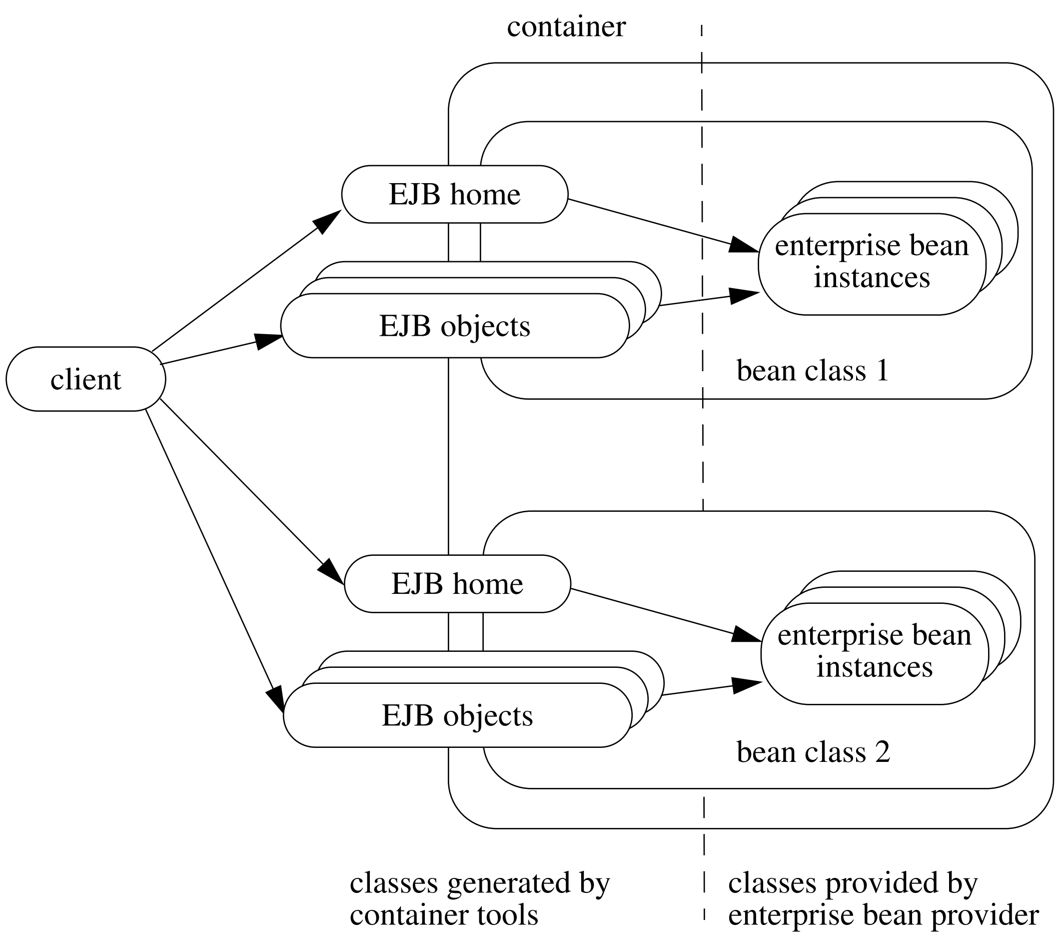
Note: Container support for entity enterprise beans is an optional feature for EJB 1.0 compliance. Container support for entity enterprise beans will become mandatory in EJB 2.0.

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, destroy, 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.

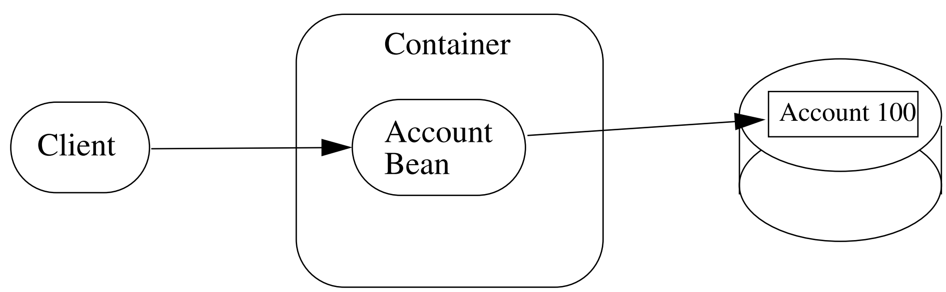
9.2 Entity persistence

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 (e.g. CICS or SAP R/3). The protocol for transferring the state of the entity between the instance variables of an enterprise bean instance and the underlying database is referred as object *persistence*.

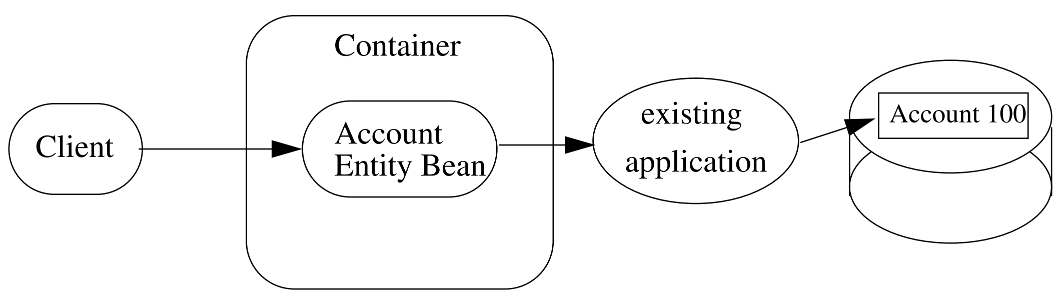
The entity component protocol allows the enterprise bean provider either to implement the enterprise bean's persistence directly in the enterprise bean class (we call this bean-managed persistence), or delegate the enterprise bean's persistence to the container (we call this container-managed persistence).

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(...)*, *ejbDestroy()*, *ejbFindMETHOD*(), *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 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 fields and the data source. The container-managed fields must be of Java primitive types.

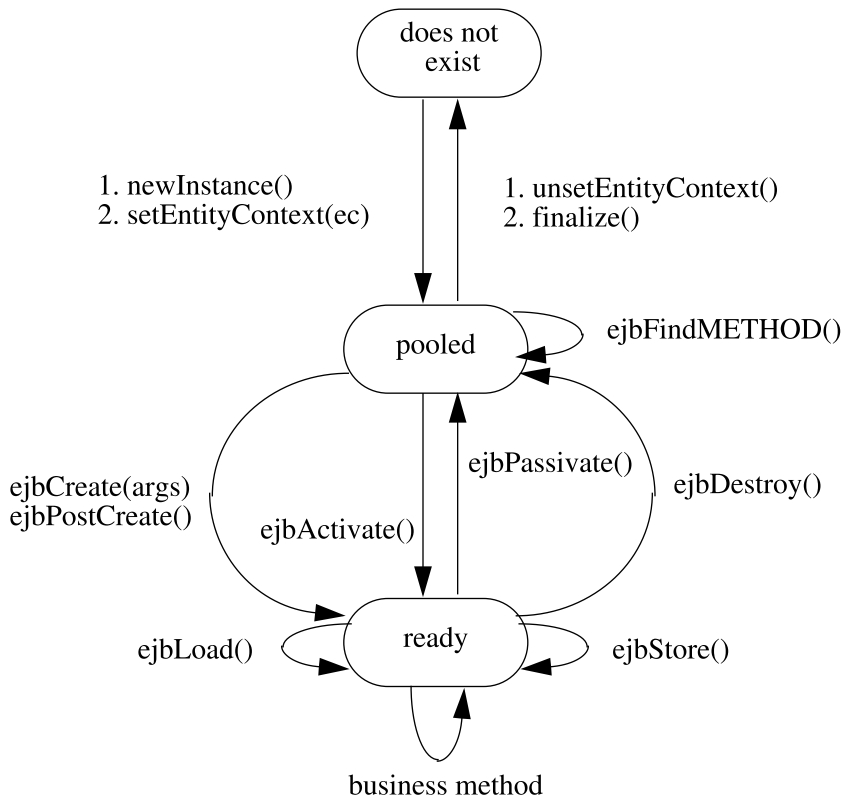
> *This restriction to only primitive types can, in the future, be relaxed to allow persisting bean handles, references to other enterprise beans, and any java.io.Serializable objects.*

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, such as SAP R/3 or CICS 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 pool 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 *ejbFindMETHOD(...)* 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 *ejbDestroy()* method. The container invokes the *ejbPassivate()* method when the container wants to disassociate the instance from the EJB object without destroying the EJB object. The container invokes the *ejbDestroy()* method when the container is destroying the EJB object (i.e. when the client invoked the *destroy()* method on the EJB object, or one of the *destroy()* 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 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 pool 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 9.9.

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 resource 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.

---

1. An entity enterprise bean has no *ejbCreate(...)* methods if it does not define any create methods in its home interface. Such an entity enterprise bean does not allow the clients to create new EJB objects. The enterprise bean restricts the clients to accessing entities that were created through direct database inserts.

The 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();*

  The *ejbPostCreate()* method gives the enterprise bean instance a chance to perform additional initialization after an *ejbCreate(...)* method has completed. 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 *ejbDestroy()*.

  Note that 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 resource typically had been allocated during the *ejbActivate()* method).

  This method executes in an unspecified transaction context. The instance can still obtain the identity of the EJB object via the *getPrimaryKey() or getEJBObject()* method on the entity context.

  Note that instance should not use the *ejbPassivate()* method to write its state to the database; the instance should store its state only in the *ejbStore()* method.

- *public void ejbDestroy();*

The container invokes this method on an instance as a result of a client's invoking a destroy method. The instance is in the ready state when *ejbDestroy()* 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 *destroy* 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 destroy 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 void ejbFindMETHOD(...);*

  The container invokes this method on the instance when the container selects the instance to execute a matching client-invoked *findMETHOD(...)* 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 *ejbFindMETHOD* method on it, and is returned to the pooled state when the execution of the *ejbFindMethod* method completes.

  The *ejbFindMETHOD* method executes in the proper transaction context.

  The implementation of an *ejbFindMETHOD* 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 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 invokes the *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 ejbDestroy()*;

  The container invokes this method before it ends the life of an EJB object as a result of a client's invoking a destroy operation.

  The container invokes this method in the transaction context of the client's *destroy* 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 void ejbFindMETHOD(...);*

  The container invokes the *ejbFindMETHOD(...)* method on an instance when a client invokes a matching *findMETHOD(...)* 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 *ejbFindMETHOD(...)* method.

  The container must invoke the *ejbFindMETHOD(...)* method in the proper transaction context.

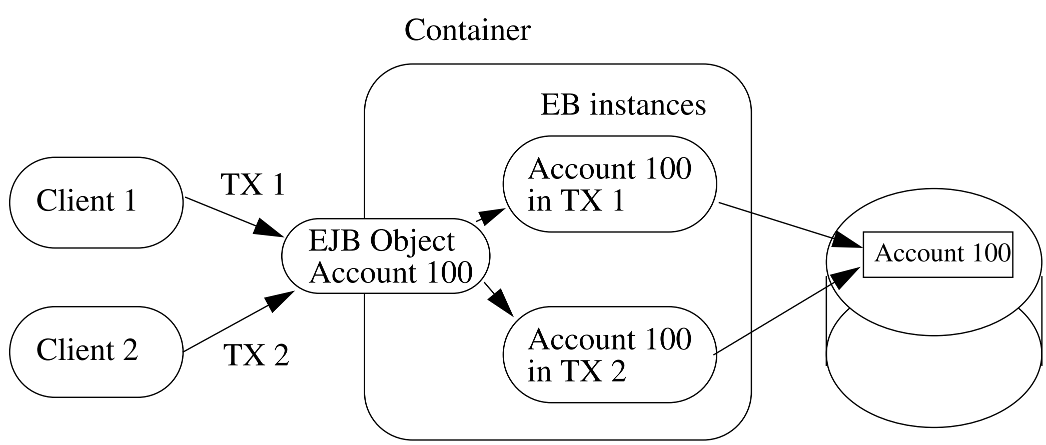
If the *ejbFindMETHOD* method is declared to return a single primary key, the container creates and EJB object reference for the primary key and returns it to the client. If the *ejbFindMETHOD* method is declared to return a collection of primary keys, the container creates a collection of EJB objects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

9.5 Concurrent access from multiple transactions

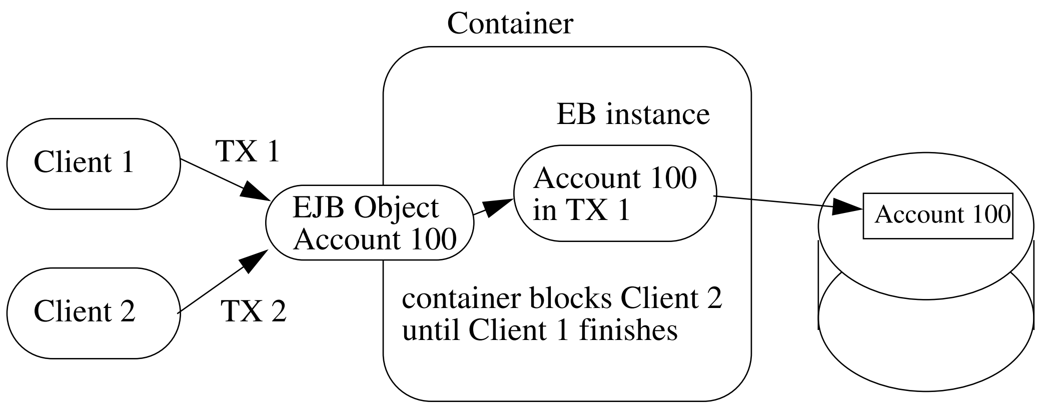
The enterprise bean developer does not have to worry about concurrent access from multiple transactions when writing the business methods. The enterprise bean developer writes the methods assuming that the container will ensure appropriate synchronization for entity beans that are accessed concurrently 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 *ejbDestroy* methods. The database system provides all the necessary transaction synchronization (provided that the developer has set the proper JDBC isolation level), the container does not have to perform any synchronization logic. The commit-time options B and C in Subsection 9.10.4 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 9.10.4 is applicable to this type of container.



9.6 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.

9.6.1 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.

9.6.2 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 must not implement the enterprise bean's remote interface.

The class must implement the business methods, and the *ejbCreate, ejbPostCreate,* and *ejbFindMETHOD* methods as described later in this section.

9.6.3 *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 also include *javax.ejb.EJBException* and *javax.ejb.CreateException*, but it must not include the *java.rmi.RemoteException*.

The return type of an *ejbCreate* method must be either the primary key type, or a collection (See 9.8.1).

9.6.4 *ejbFind* methods

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

The enterprise bean class may also define additional *ejbFindMETHOD(...)* 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 9.8.1).

A finder method's throws clause may define arbitrary application specific exceptions. The throws clause may also include the *javax.ejb.EJBException* and *javax.ejb.FinderException*, but it must not include the *java.rmi.RemoteException*.

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

9.6.5 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 *javax.ejb.EJBException*, and it must not include the *java.rmi.RemoteException*.

9.6.6 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. The only exception to this rule is the *javax.ejb.EJBException* that must not be defined in the throws clause of the remote method.

9.6.7 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 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 only exception to this rule is the *javax.ejb.EJBException* that must not be defined in the throws clause of the *create* method.

9.7 The responsibilities of the container provider

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

9.7.1 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.
- 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 is used to customize the business logic to 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.

9.7.2 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, and then the *ejbPostCreate()* method.

The implementation of the *destroy(...)* 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 *ejbDestroy* method on the instance.

The implementation of each *findMETHOD(...)* methods invokes a matching *ejbFindMETHOD(...)* method. The implementation of the *findMETHOD(...)* method must create an EJB object for the primary key returned from the *ejbFindMETHOD*, and return the EJB object reference to the client. If the *ejbFindMETHOD* method returns a collection of primary keys, the implementation of the *findMETHOD(...)* method must create a collection of EJB objects for the primary keys, and return the collection to the client.

9.7.3 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 *destroy(...)* 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 *ejbDestroy* method on the instance.

The implementation of each business method must activate and instance (if an instance is not already in the ready state) and invoke the matching business method on the instance.

9.7.4 Single-threaded rule

The container must ensure that only one thread can be executing an instance at any time. Consequently, the container must not invoke any of the life cycle methods while a business method invocation is in-progress, or invoke multiple business methods on an instance concurrently. Therefore, an instance is never passivated while it is executing a business method.

The exception to this rule is that a container must allow callbacks to an instance within the same transaction. For example, if a client invokes enterprise bean A, A calls another enterprise bean B, and B invokes a callback method on A, the container must allow the callback from B to A, even though A is still executing the client's call.

9.7.5 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.

9.8 Miscellaneous

9.8.1 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 *ejbFindMETHOD(...)* and *findMETHOD(...)* 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 additions 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, EJBException
        {
                ...
        }
        ...
}
```

9.9 Container-managed entity beans

Sections 9.3 through 9.7 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 from the contract for entities with bean-managed persistence.

9.9.1 *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*, *ejbDestroy*, *ejbLoad*, and *ejbStore*, as described in the below subsections. The container is also responsible for the implementation of the finder methods.

9.9.2 ejbCreate

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 *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.

9.9.3 ejbDestroy

The container invokes the *ejbDestroy()* method on an entity bean instance with container-managed persistence in response to a client-invoked *destroy()* operation on an EJB object reference or on the EJB home interface.

The enterprise bean provider can use the *ejbDestroy* method to implement any actions that must be done before the entity representation is removed from the database.

After *ejbDestroy* returns, the container removes the entity representation from the database.

The container must perform *ejbDestroy* and the database delete operation in the proper transaction context.

9.9.4 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).

9.9.5 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.

9.9.6 finder methods

The enterprise bean provider does not write the finder (*ejbFindMETHOD(...)*) 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 *ejbFindMethod()* methods, or the tools can generate the implementation of the finder methods directly in the class that implements the enterprise bean's home interface.

9.9.7 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).

9.9.8 Isolation levels

The enterprise bean provider must specify the requested transaction isolation level for an entity bean with container-managed persistence in the bean's deployment descriptor.

The possible isolation levels are:

- NONE (illegal value in the deployment descriptor)
- READ_UNCOMMITTED
- READ_COMMITTED
- REPEATABLE_READ
- SERIALIZABLE

These isolation levels correspond to the JDBC isolation levels, and are defined in the *javax.ejb.deployment.IsolationLevel* class.

The isolation level is specified at the level of the whole bean, and may be overridden at the level of an individual method.

The container is allowed to use a higher isolation level than that specified in the bean's deployment descriptor.

9.10 Sequence diagrams

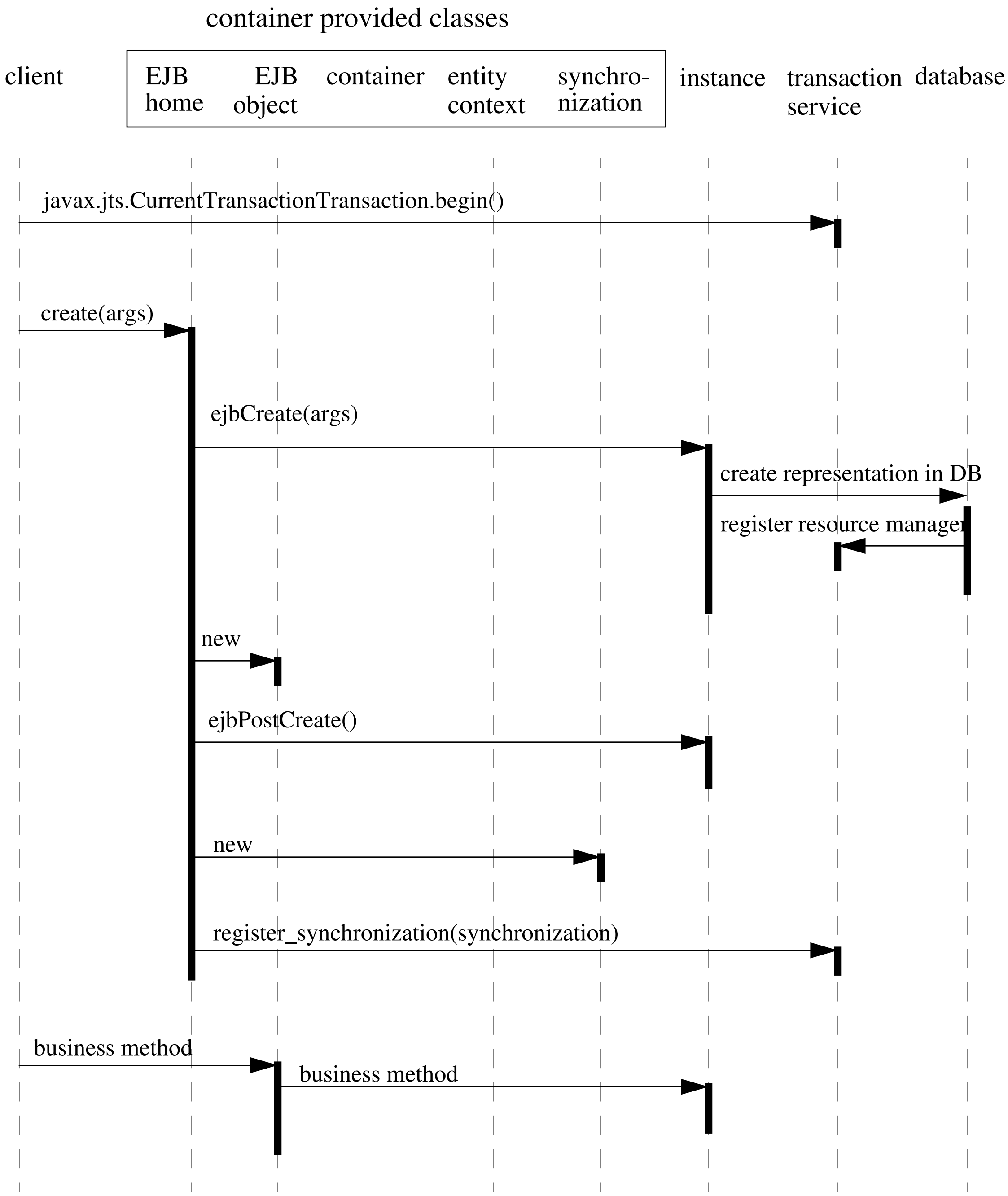
This section uses sequence diagrams to illustrate the interactions between an entity bean instance and its container.

9.10.1 Notes

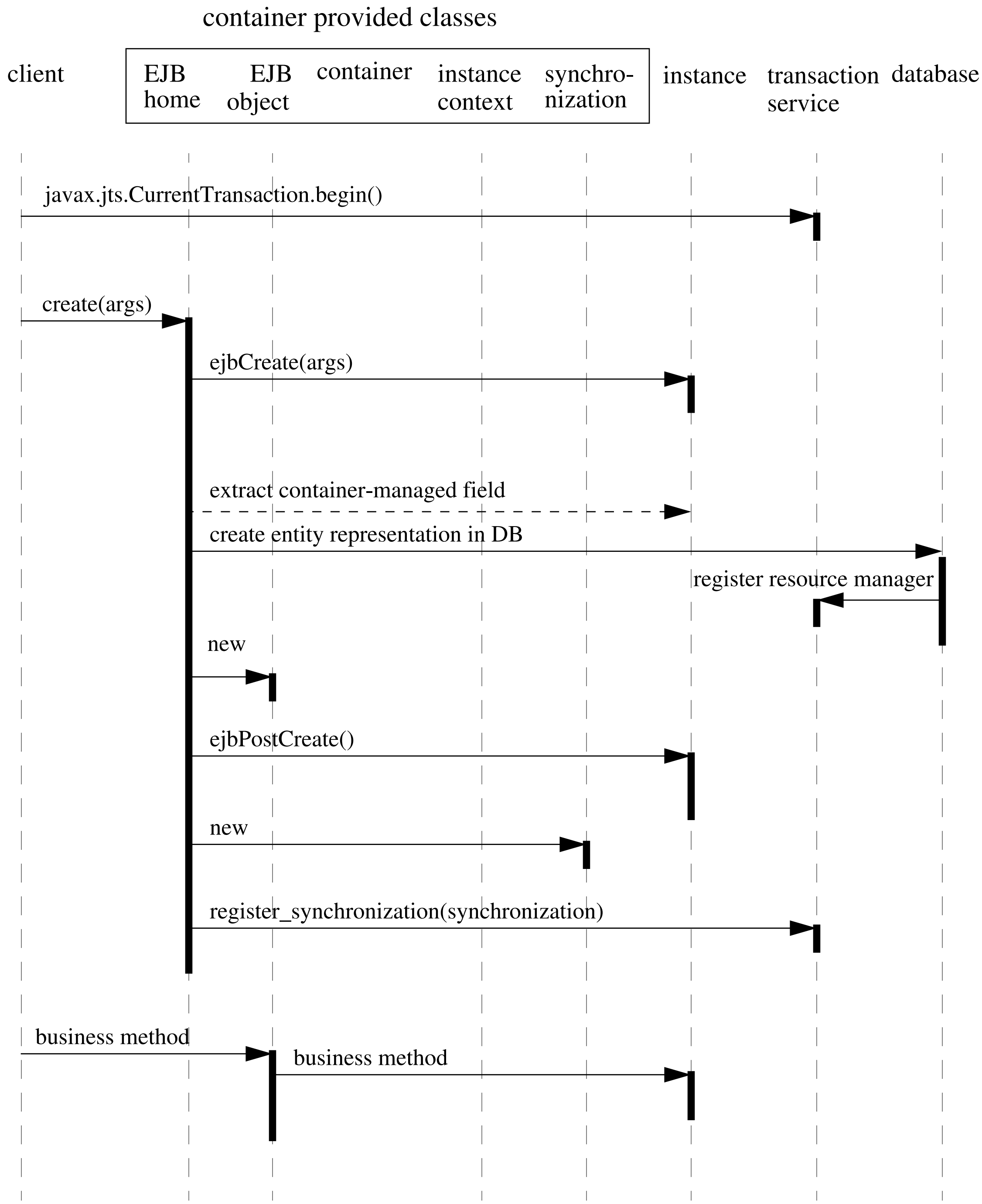
The sequence diagrams illustrate a box labeled "container provided classes". These are either classes that are part of the container, or classes that were generated by the container tools. These classes communicate with each other through protocols that are container implementation specific. Therefore,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

9.10.2 Creating an entity object

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

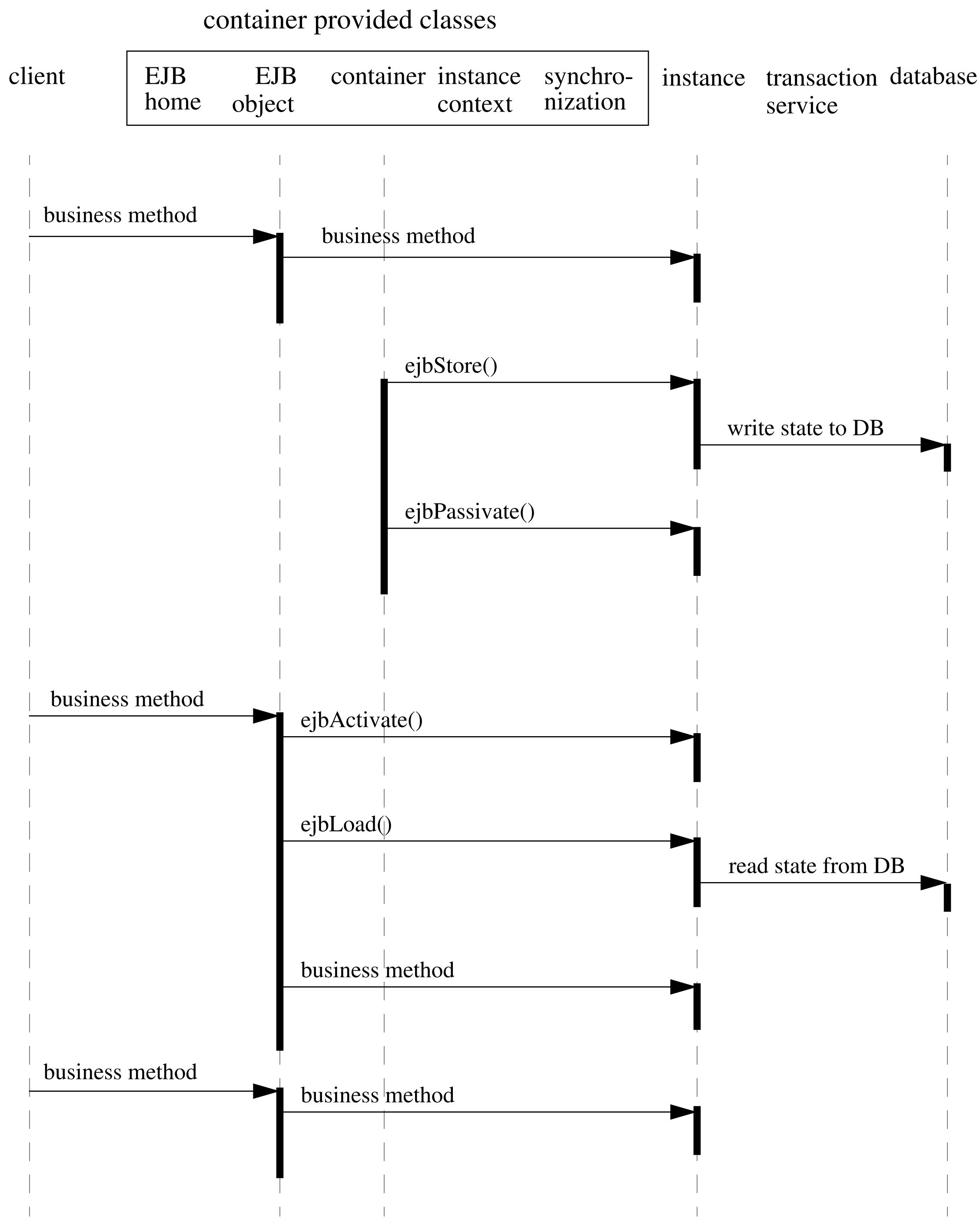


The following diagram illustrates the creation of an enterprise bean with container-managed persistence:

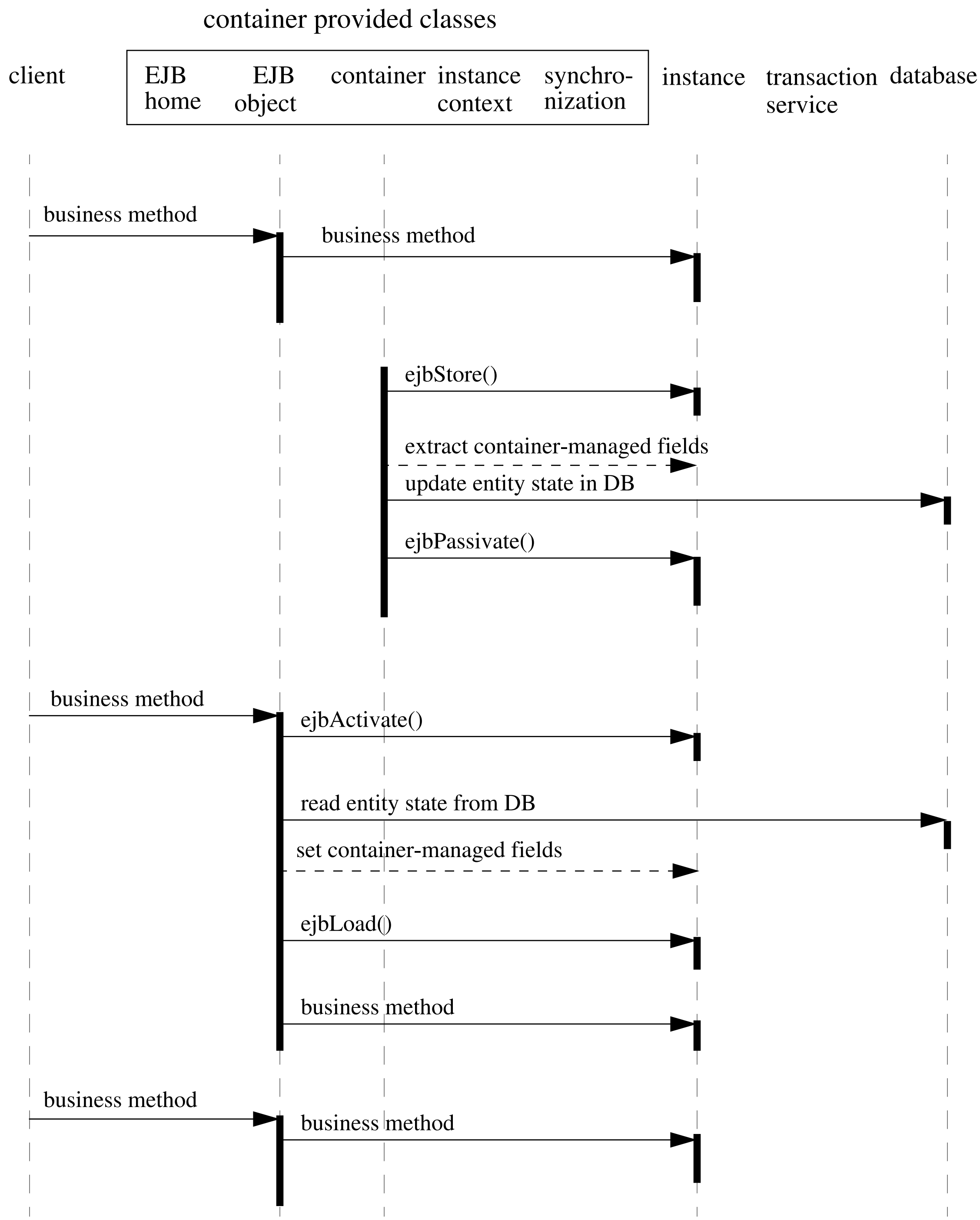


9.10.3 Passivating and activating an instance in a transaction

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.



9.10.4 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:

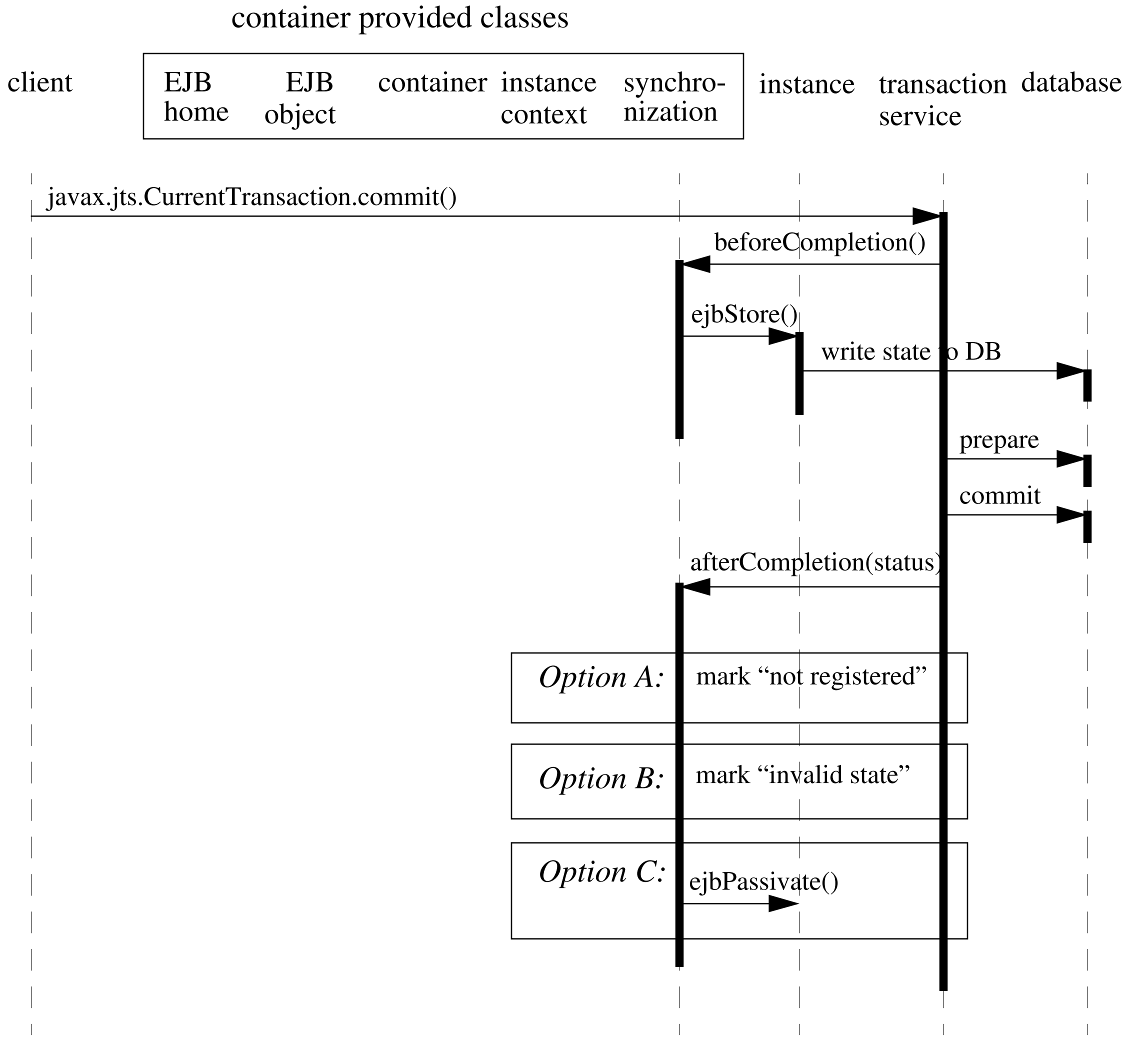
- Option A: The container caches a "ready" instance between transactions. The instance has exclusive access to the state of the object in the persistent storage, and therefore the container does not have to synchronize the instance's state from the persistent storage at the beginning of the next transaction.
- 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.

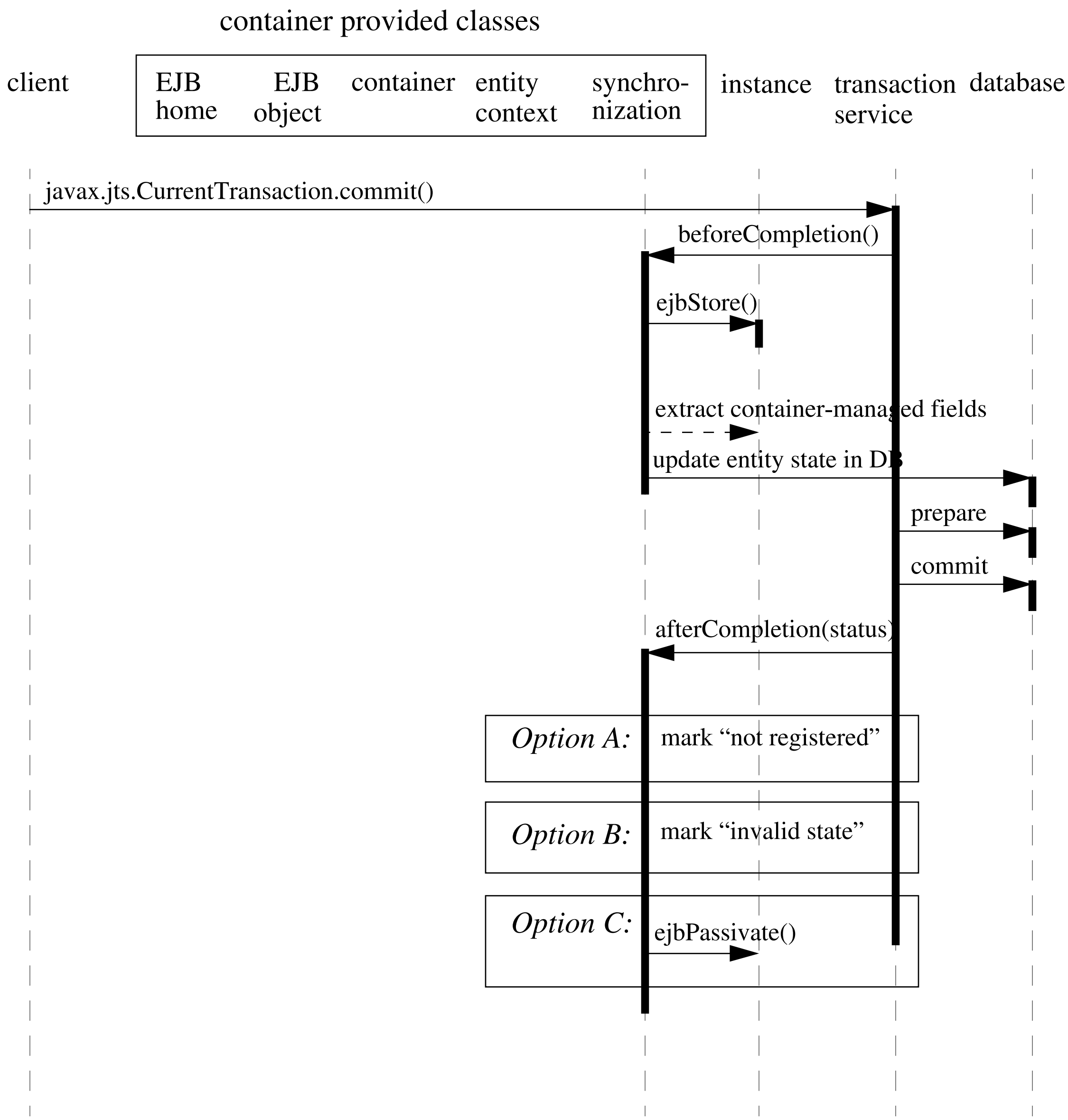
Table 1: Summary of commit-time options

	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.

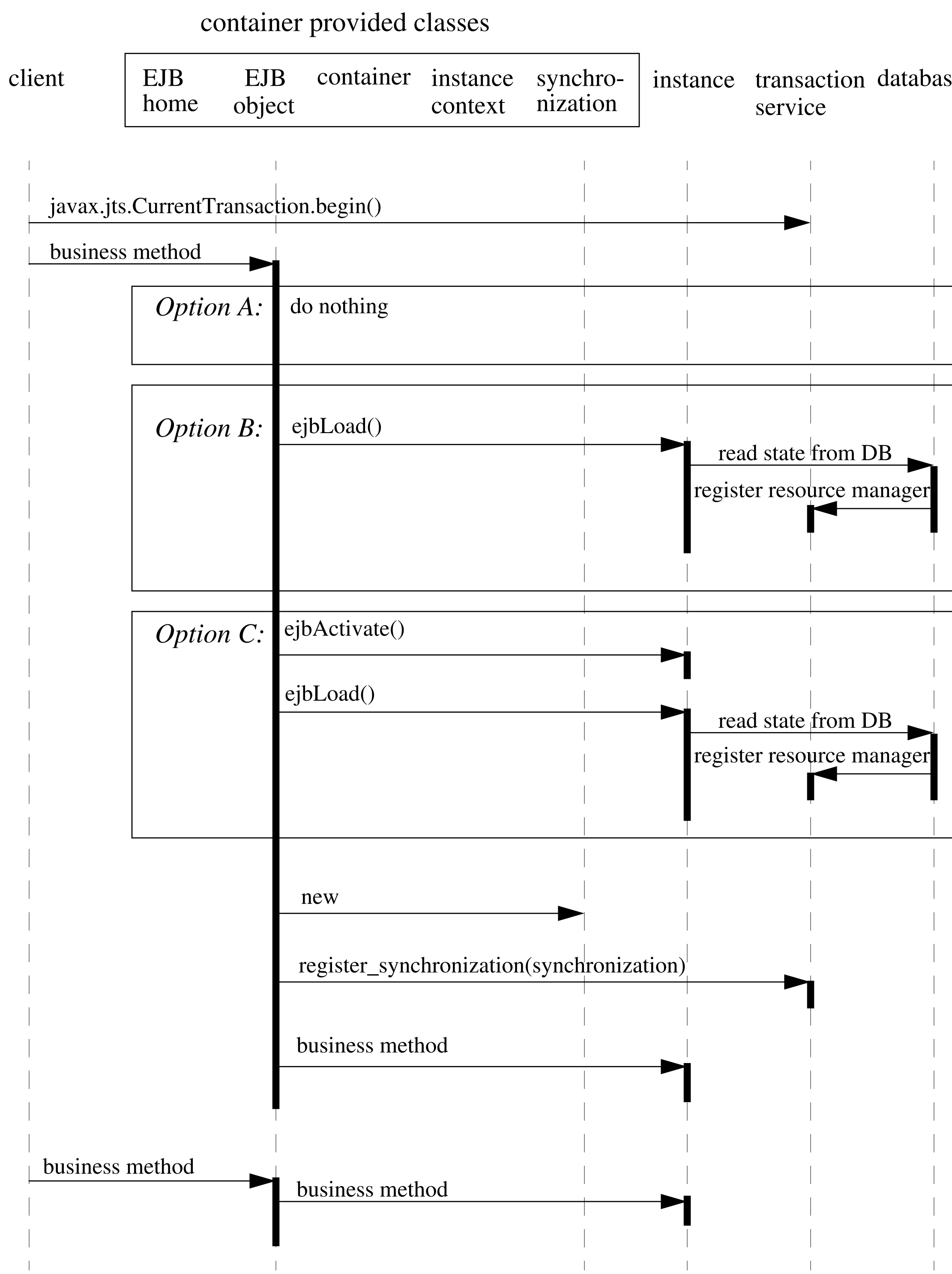


The following diagram illustrates the transaction commit protocol for an enterprise bean instance with container-managed persistence.

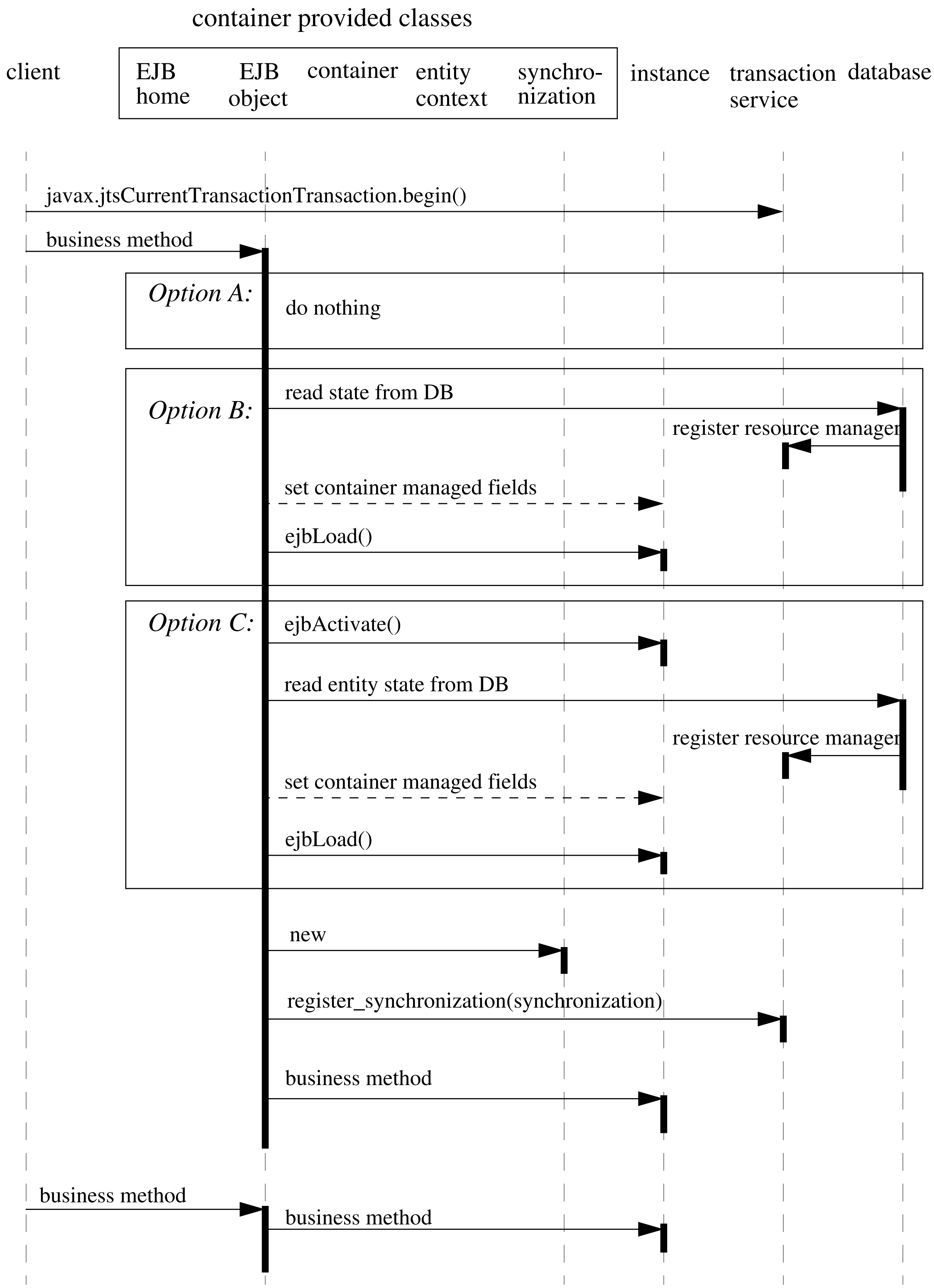


9.10.5 Starting the next transaction

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.

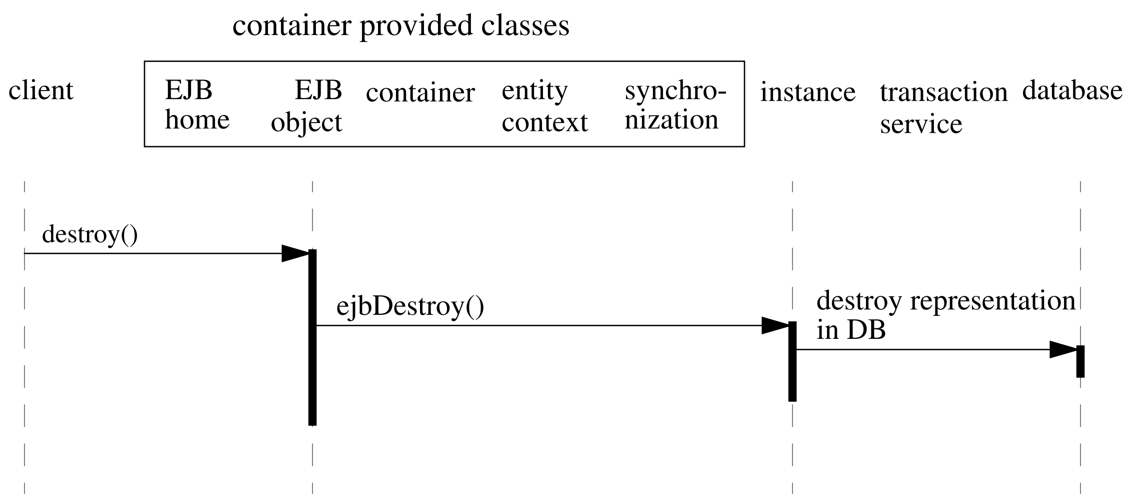


The following diagram illustrates the protocol performed for a bean with container-managed persistence at the beginning of a new transaction.

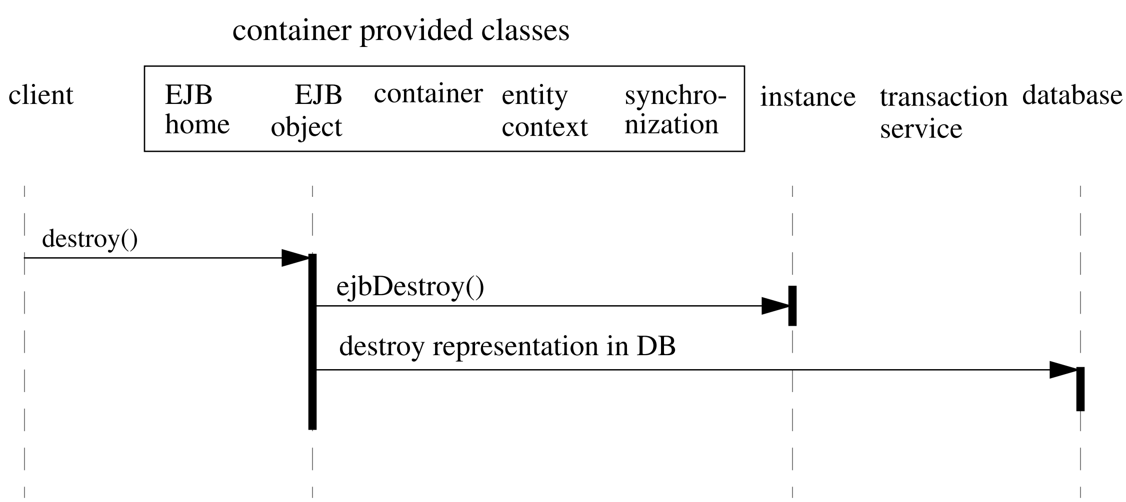


9.10.6 Destroying an entity object

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

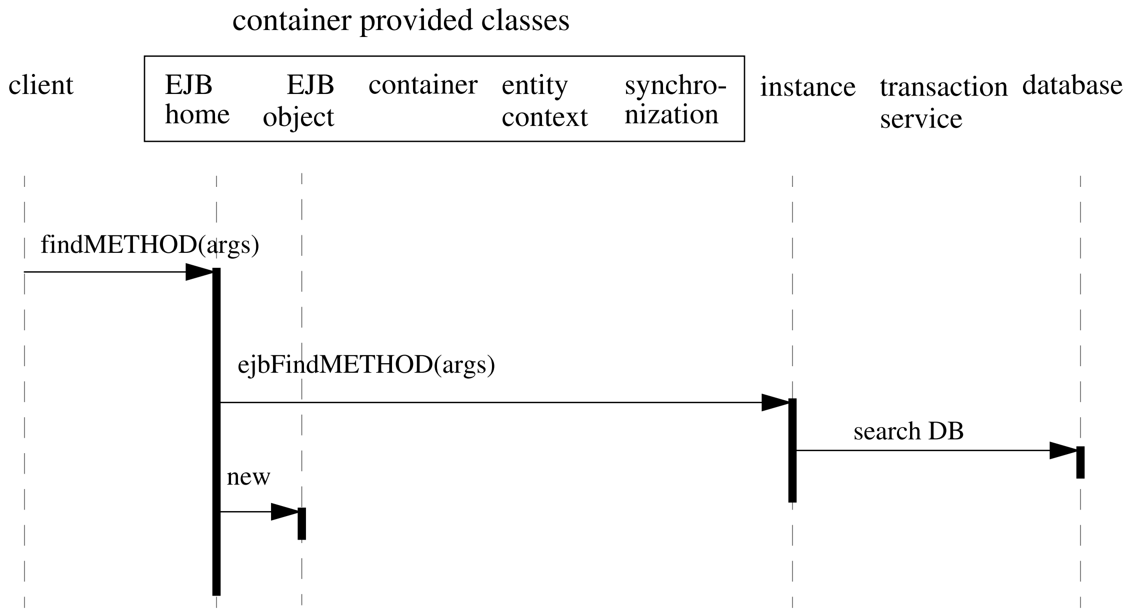


The following diagram illustrates the destruction of an entity enterprise bean with container-managed persistence.

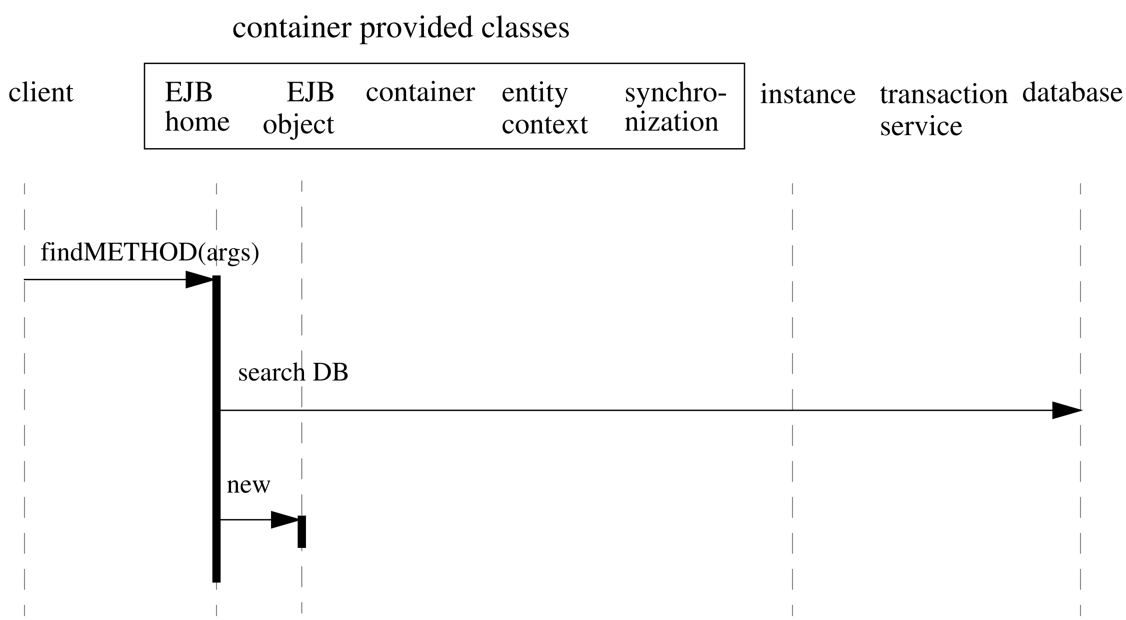


9.10.7 Finding an object

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



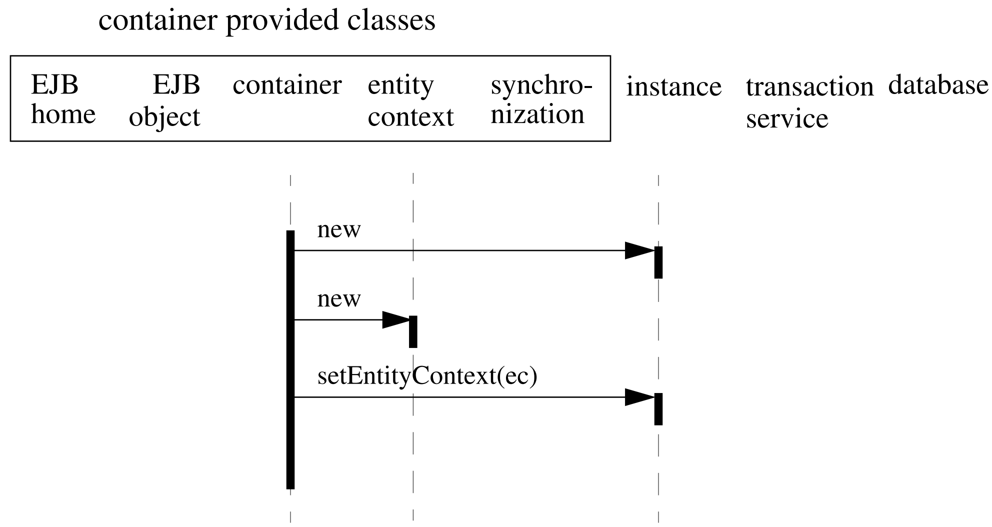
The following diagram illustrates the execution of a finder method on an entity enterprise bean with container-managed persistence.



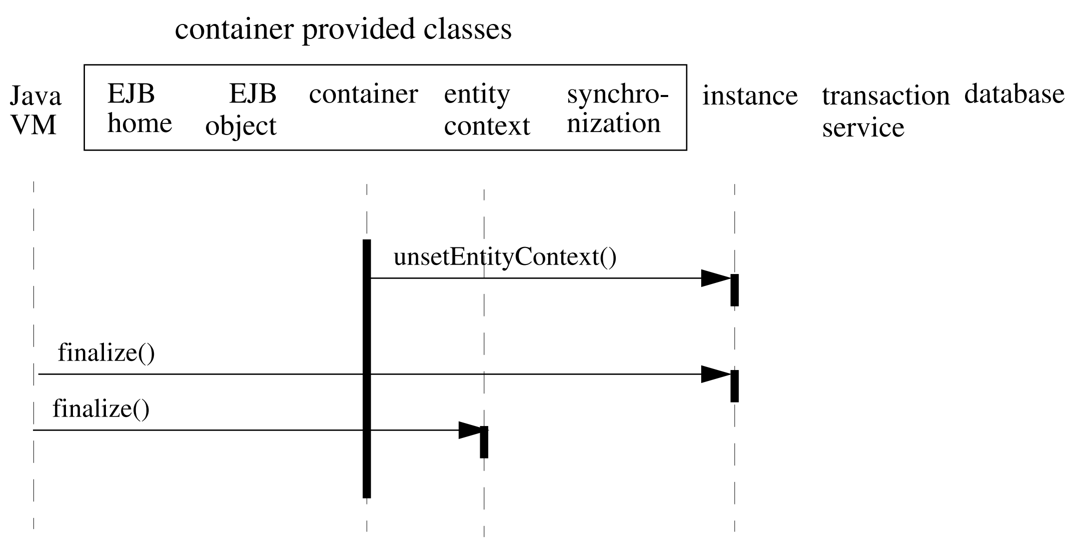
9.10.8 Adding and removing instance from the pool

The diagrams in Subsections 9.10.2 through 9.10.7 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 container's adding an instance to the pool.



The following diagram illustrates the sequence for container's 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.

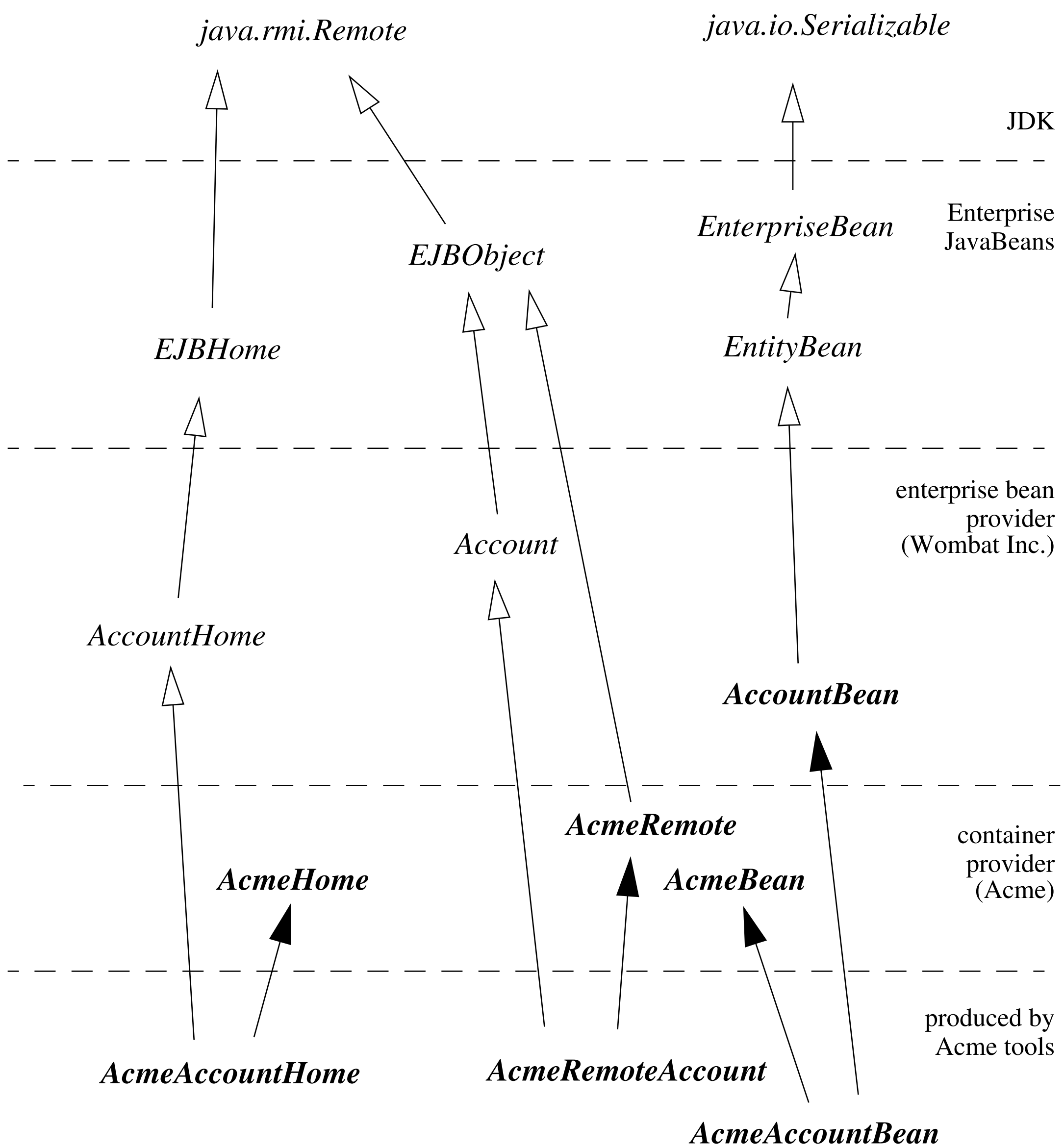
The classes generated by the container provider's tools in this scenario should be considered illustrative rather than prescriptive. Container providers are free to implement the contract between an enterprise bean and its container in a different way that achieves an equivalent effect (from the perspectives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

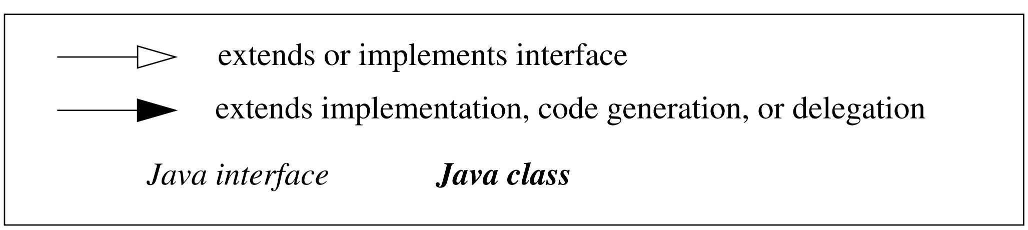
10.1 Overview

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 must not 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.

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.

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 does not have to deal with 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 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 high-level component framework that attempts to hide system complexity from the application developer. Therefore, most enterprise beans do not directly access transaction management.

JTS is a lower-level API, which may be implemented by the EJB server and used by the container. However, EJB server and container implementors are free to use other transaction manager API for their integration.

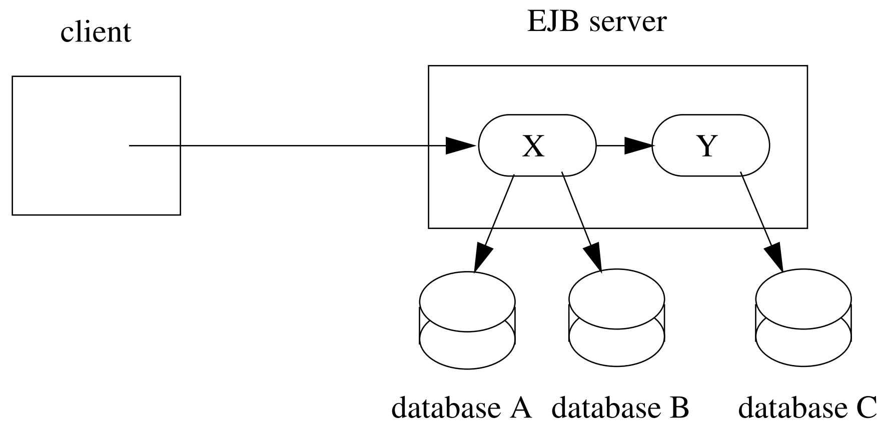
11.3 Scenarios

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.



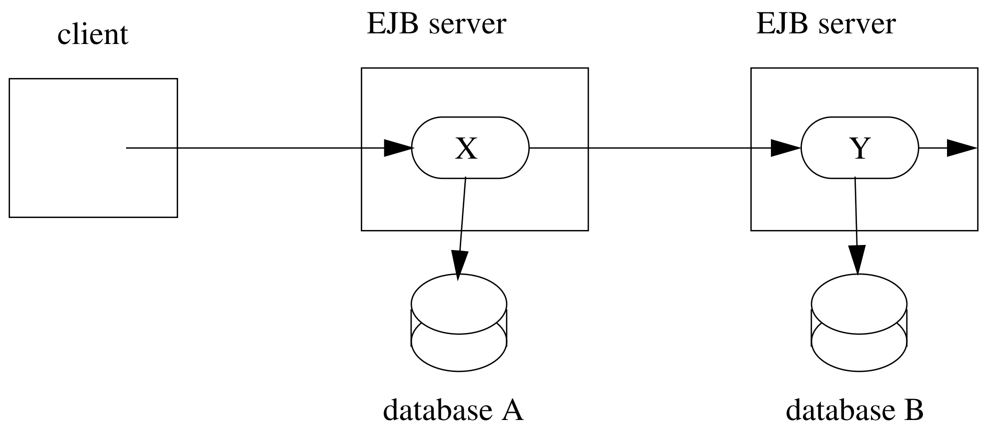
The application programmer does not have to do anything to handle transaction semantics. The enterprise beans X and Y perform the database updates using the standard JDBC API. Behind the scenes, the EJB server enlists the database connections as part of the transaction. When the transaction commits, 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.



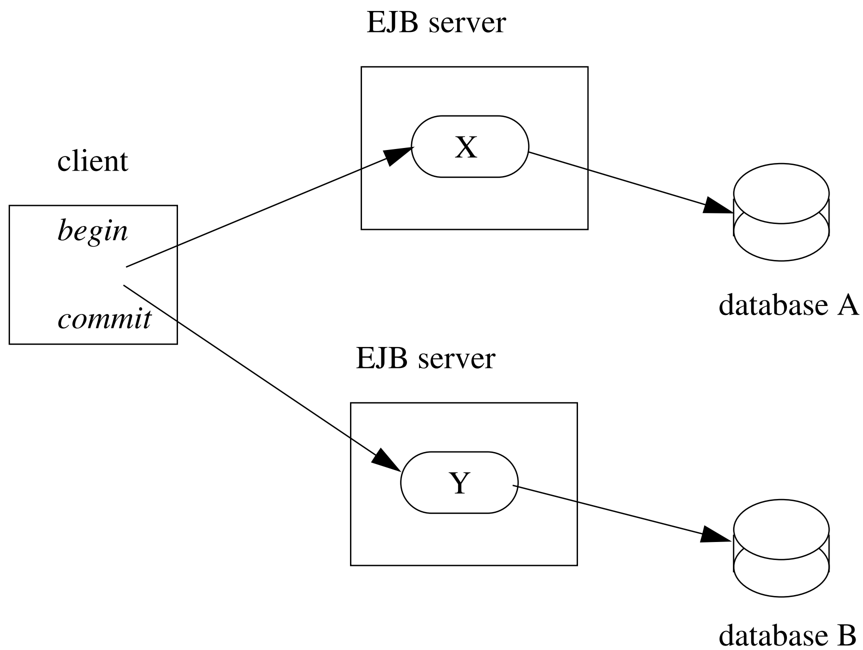
When X invokes Y, the two EJB servers cooperate to propagate the transaction context from X to Y. This transaction context propagation is transparent to the application-level code.

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.CurrentTransaction* 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.



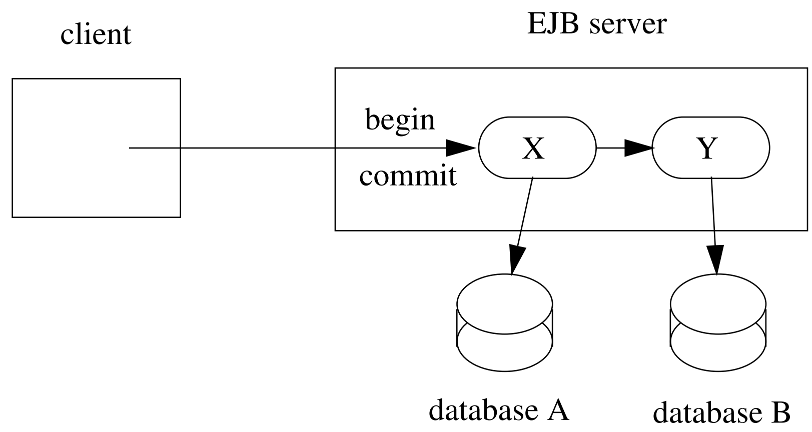
The application programmer does not have to do anything to make the updates to databases A and B performed by enterprise beans X and Y atomic other than demarcate the transaction by the *begin* and *commit* calls. A proxy of a transaction service on the client automatically propagates the transaction 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 *REQUIRES* 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 automatically 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

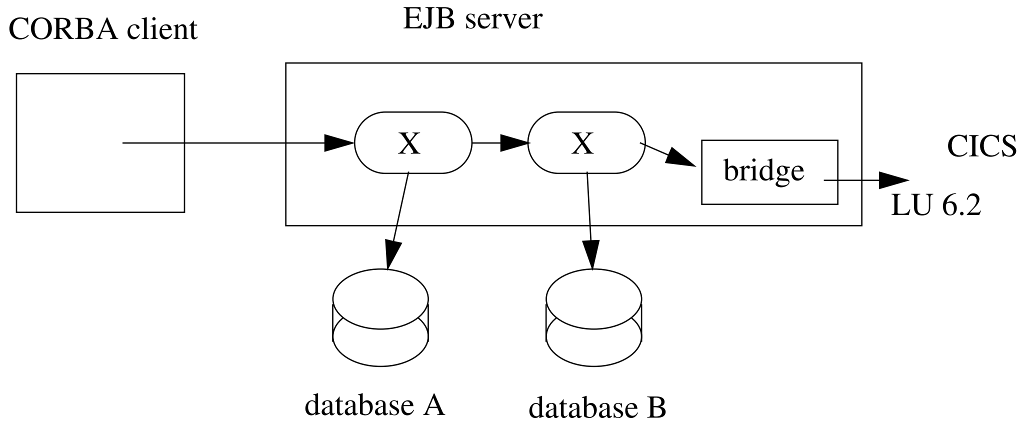
An enterprise bean with the *BEAN_MANAGED* transaction attribute can use the *javax.jts.CurrentTransaction* interface to demarcate transactions.

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 can also interoperate 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 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:

- *NOT_SUPPORTED*
- *BEAN_MANAGED*
- *REQUIRES*
- *SUPPORTS*
- *REQUIRES_NEW*
- *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.

11.4.1 NOT_SUPPORTED

A container must always invoke an enterprise bean that has the *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 BEAN_MANAGED

An enterprise bean with the *BEAN_MANAGED* attribute can use the *javax.jts.CurrentTransaction* interface to demarcate transaction boundaries.

> *TODO: need to described the rules that the enterprise bean must follow when doing transaction demarcation and the rules for the container to deal with suspending and resuming caller's transaction.*

11.4.3 REQUIRES

If a client invokes an enterprise bean object that has the *REQUIRES* 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 SUPPORTS

An enterprise bean object that has the *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 REQUIRES_NEW

An enterprise bean that has the *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 MANDATORY

An enterprise bean object that has the *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
NOT_SUPPORTED	-	-
	T1	-
BEAN_MANAGED	-	-
	T1	T1
REQUIRES	-	T2
	T1	T1
SUPPORTS	-	-
	T1	T1
REQUIRES_NEW	-	T2
	T1	T2
MANDATORY	-	error
	T1	T1

A dash means "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 Bean-managed demarcation

An enterprise bean with the *BEAN_MANAGED* attribute is allowed to use the *javax.jts.CurrentTransaction* interface to demarcate transaction boundaries.

The container makes the *javax.jts.CurrentTransaction* interface available to the enterprise bean though the *InstanceContext.getCurrentTransaction()* method, as illustrated in the following example.

```
import javax.jts.CurrentTransaction;
...
InstanceContext ic = ...;
...
CurrentTransaction tx = ic.getCurrentTransaction();
tx.begin();
...
tx.commit();
```

Enterprise beans deployed with a transaction attribute other than *BEAN_MANAGED* are not allowed to access directly the underlying transaction manager. This means that the container makes the JTS API unavailable to the enterprise bean.

11.6 Transaction management and exceptions

The EJB server and EJB container may throw the Transaction*RollbackException*, *TransactionRequiredException*, and *InvalidTransactionException* exceptions in the situations defined in the JTS specification. See Appendix E 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 clients access an enterprise bean through the enterprise bean's remote and home interfaces. Both these interface are Java RMI interfaces. Therefore, the throws clause of every method of these interfaces includes the mandatory *java.rmi.RemoteException* that 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, or by the container. 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.

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.

12.1.1 Exceptions and transactions

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

The client can assume that the transaction has been marked for rollback if the exception is the *java.rmi.TransactionRollbackException* exception. 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 *java.rmi.TransactionRollbackException* (note that *java.rmi.TransactionRollbackException* 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.

12.2 Rules for the enterprise bean developer

The business methods, the *ejbCreate*, *ejbDestroy*, and *ejbFindMETHOD* methods, defined and implemented by the enterprise bean class may define application-level exceptions and the *javax.ejb.EJBException* in their throws clauses.

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). The enterprise bean developer may assume that the container passes these exceptions unchanged to the client and does not rollback the client's transaction (if the enterprise bean method was executed as part of the client's transaction). Therefore, the enterprise bean developer must be careful to leave its state consistent when throwing an application-level exception because the client may recover the exception and try to commit the transaction.

The enterprise bean developer should throw the *javax.ejb.EJBException* from any of its method (business method, *ejbCreate*, *ejbDestroy*, and any of the other container callback methods) to indicate an unexpected system-level failure (e.g. failure to open database connection).

12.3 Rules for the container provider

The container must deal with exceptions thrown by the enterprise bean's methods as follows.

The container must pass all the application-level exceptions (i.e. all exceptions defined in the method's throws clause other than the *javax.ejb.EJBException*) thrown by the enterprise bean's business methods, *ejbCreate*, *ejbPostCreate*, *ejbDestroy*, and *ejbFindMETHOD* methods to the client. The container must not convert an application-level exception into, for example, the *java.rmi.RemoteException*.

The container must catch the *javax.ejb.EJBException* thrown by the enterprise bean's methods. The container should log the exception to alert the system administrator of the problem. If the *javax.ejb.EJBException* prevents successful completion of a client's request, the container must throw the *java.rmi.RemoteException* to the client.

If an enterprise bean's instance throws an *javax.ejb.EJBException*, the container must assume that the instance is in an undefined state. The container must no longer use the instance for processing.

The container must catch all unchecked exceptions thrown by the enterprise bean's methods, and treat them the same way as an *javax.ejb.EJBException*.

12.3.1 Exceptions and transactions

In general, an exception thrown by an enterprise bean method should not cause the container to automatically rollback a transaction or mark the transaction for rollback. This is to allow the client a chance to recover from application-level exception.

Only the following cases require that the container roll back a transaction, or mark it for rollback, as a result of an exception thrown by the enterprise bean:

- 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). In this case, the container must rollback the transaction when it catches any exception from the enterprise bean's method (including an application-level exceptions) before it throws an exception to the client. The exception thrown to the client is determined using the rules in the previous subsection.

- The enterprise bean's method executed in the transaction context imported from the client, and the method threw the *javax.ejb.EJBException* or an unchecked exception. The container must mark the transaction for rollback, and throw the *javax.jts.TransactionRollbackException* to the client. The *javax.jts.TransactionRollbackException* 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.

12.4 Exceptions from *create* and *finder* methods

12.4.1 Exceptions from the *create* methods

The throws clause of every *create(...)* method defined in the enterprise bean's remote interface must include the *java.rmi.RemoteException* to allow reporting of system-level errors.

The throws clause may include zero or more application-level exceptions that are specific to the enterprise bean.

For entity beans, it t is recommended that the application-level exceptions include the *javax.ejb.CreateException* to allow standard reporting of application-level errors that may occur during the attempt to create the entity.

Currently, the EJB specification defines the *javax.ejb.DuplicateKeyException* as the only standard create exception. The *javax.ejb.DuplicateKeyException* exception is a subclass of *javax.ejb.CreateException*.

> *Note: Future release of EJB may define additional standard "create" exceptions as subclasses of javax.ejb.CreateException.*

12.4.2 Exceptions from the *find* methods

[Relevant only to entity beans since sessions have no finders.]

The throws clause of every *find(...)* method defined in the enterprise bean's remote interface must include the *java.rmi.RemoteException* to allow reporting of system-level errors.

The throws clause may include zero or more application-level exceptions that are specific to the enterprise bean.

It t is recommended that the application-level exceptions include the *javax.ejb.FinderException* to allow standard error reporting from the finder methods.

Currently, the EJB specification defines the *javax.ejb.ObjectNotFoundException* as the standard way for a finder method to report that the specified object does not exist. The finder methods that are defined to return a single object should use this exception to report that an EJB object does not exist. The finder methods that are defined to return a collection of EJB objects should return an empty collection—they should not throw the *javax.ejb.ObjectNotFoundException*.

13 Support for distribution

13.1 Overview

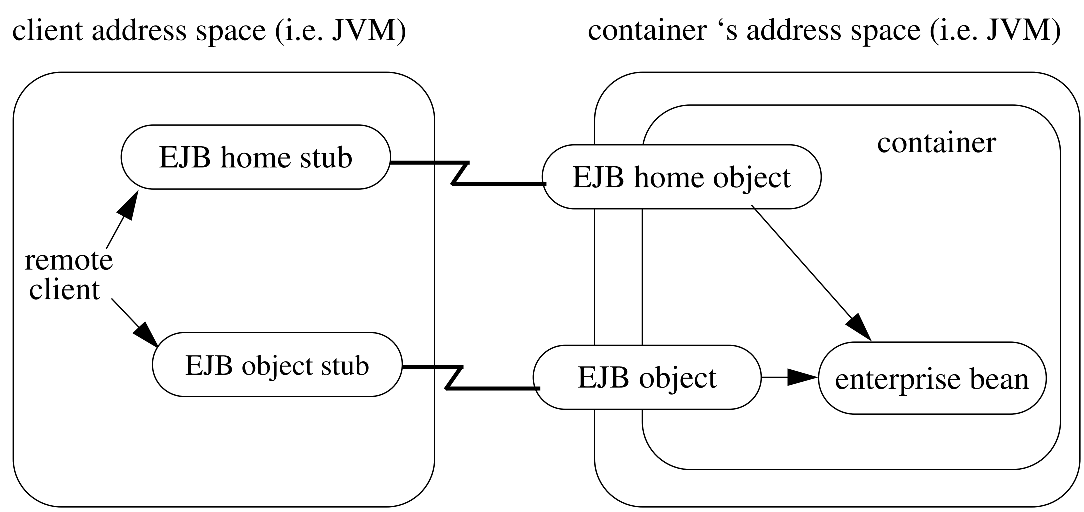
Support for remote client access to an enterprise bean object is through the standard Java API for remote method invocation (Java RMI) [3]. This API allows a client to invoke an enterprise bean object using any distributed object protocol, including the industry standard IIOP protocol, as defined in 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 remote objects in the sense of Java RMI. 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

Support for security in Enterprise JavaBeans includes the following components:

- Use of the existing Java security APIs defined in the core package *java.security.*
- Security-related methods in *InstanceContext.*
- Security-related attributes in the deployment descriptor.

The following sections describe support for security in more detail.

TODO: this chapter needs more examples

14.1 Package *java.security*

The package *java.security* provides the generic Java security-related interfaces. Enterprise JavaBeans uses the applicable existing Java security APIs. This section describes the parts of the java.security API that are relevant to Enterprise JavaBeans.

14.1.1 class *java.security.Identity*

The *java.security.Identity* class encapsulates the concept of "user identity" for security purposes. Please refer to the reference page of *java.security.Identity* for the description.

14.2 Security-related methods in *InstanceContext*

The InstanceContext interface contains the following security-related method:

- *getCallerIdentity*
- *hasRole*

Please refer to reference page of *javax.ejb.InstanceContext* for the description of this method.

14.3 Security-related deployment descriptor properties

An enterprise bean's deployment descriptor allows a container to perform security management outside of the enterprise bean code. The security management of an enterprise bean is determined by the bean's security descriptor. Please see the reference page for *javax.ejb.deployment.SecurityDescriptor*.

14.4 Examples

The following example illustrates programmatic access to the security information.

14.4.1 Obtain client's Identity

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

/* getName returns a printable representation of identity. */
String clientAccount = caller.getName();
```

14.4.2 Check client's role

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

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

15 Ejb-jar file

Enterprise JavaBeans defines the format for 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 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 6.10, 9.6, and 9.9.

Furthermore, the Java 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 file 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 16.3.

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 JTS interfaces directly. The only exception are enterprise beans with the *BEAN_MANAGED* transaction attribute which are allowed to use the *javax.jts.CurrentTransaction* 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:

- Deserialized 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 include 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 6.12 and 9.7.
- 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.

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.
- If the container supports Entities, it must implement the Entity protocol described in Chapters 8 and 9.
- Implement the support for transaction described in Chapter 11.
- Implement the support for security described in Chapter 14.
- Handle exceptions as described in Chapter 12.
- If the container uses IIOP as the distributed object protocol, it should implement the EJB to CORBA mapping 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.

18 Enterprise JavaBeans API Reference

The following interfaces and classes comprise the Enterprise JavaBeans API:

package *javax.ejb*:

Interfaces:

```
public interface EJBHome
public interface EJBMetaData
public interface EJBObject
public interface EnterpriseBean
public interface EntityBean
public interface EntityContext
public interface Handle
public interface InstanceContext
public interface SessionBean
public interface SessionContext
public interface SessionSynchronization
```

Classes:

```
public class CreateException
public class DestroyException
public class DuplicateKeyException
public class EJBException
public class FinderException
public class ObjectNotFoundException
```

package *javax.ejb.deployment*:

Classes:

```
public class DeploymentDescriptor
public class EntityDescriptor
public class IsolationLevel
public class MethodDescriptor
public class SecurityDescriptor
public class SessionDescriptor
public class TransactionAttribute
```

Interface EJBHome

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

The EJBHome interface is an interface that 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 destroy 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

- **destroy**

```
public abstract void destroy(Handle handle)
     throws RemoteException, DestroyException
```

Destroy an EJB object identified by its handle.

Throws: DestroyException
Thrown if the enterprise bean or the container does not allow the client to destroy the object.

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

Throws: DestroyException
The enterprise bean or the container does not allow destruction of the object.

- **destroy**

```
public abstract void destroy(Object primaryKey)
     throws RemoteException, DestroyException
```

Destroy an EJB object identified by its primary key.

Throws: DestroyException
Thrown if the enterprise bean or the container does not allow the client to destroy the object.

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

Throws: DestroyException
The enterprise bean or the container does not allow destruction of the object.

- **`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.

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. The meta-data is intended to for development tools used for building applications that use deployed enterprise beans.

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 EJB's home interface.

- **getPrimaryKeyClass**

  `public abstract Class getPrimaryKeyClass()`

  Obtain the Class object for the EJB's primary key class.

- **getRemoteInterfaceClass**

  `public abstract Class getRemoteInterfaceClass()`

  Obtain the Class object for the EJB'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 void destroy();
    public abstract EJBHome getEJBHome();
    public abstract Handle getHandle();
    public abstract Object getPrimaryKey();
    public abstract boolean
        isIdentical(EJBObject obj);
}
```

The EJBObject interface is an interface that 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 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

- **destroy**

```
public abstract void destroy()
     throws RemoteException, DestroyException
```

Destroy the EJB object.

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

Throws: DestroyException
The enterprise bean or the container does not allow destruction of the object.

- **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 destroy 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.

Interface EnterpriseBean

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

The EnterpriseBean interface is an interface that every enterprise bean class must implement. It is a common superinterface for the SessionBean and EntityBean interfaces.

Interface EntityBean

```
public interface javax.ejb.EntityBean
    extends javax.ejb.EnterpriseBean
{
    public abstract void ejbActivate();
    public abstract void ejbDestroy();
    public abstract void ejbLoad();
    public abstract void ejbPassivate();
    public abstract void ejbPostCreate();
    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 EJBException
```

A container invokes this method on the instance 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: EJBException
Thrown if the instance could not perform the function requested by the container.

- **ejbDestroy**

```
public abstract void ejbDestroy()
     throws EJBException, DestroyException
```

A container invokes this method before it destroys the EJB object that is currrently associated with the instance. This method is invoked when a client invokes a destroy 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 destroy operation.

Throws: EJBException
Thrown if the instance could not perform the function requested by the container.

Throws: DestroyException
The enterprise bean does not allow destruction of the object.

- **ejbLoad**

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

A container invokes this method on the instance 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: EJBException
Thrown if the instance could not perform the function requested by the container.

- **ejbPassivate**

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

A container invokes this method on 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: EJBException
Thrown if the instance could not perform the function requested by the container.

- **ejbPostCreate**

```
public abstract void ejbPostCreate()
     throws EJBException
```

A container invokes this function as part of the protocol for the creation of an EJB object. The container calls ejbPostCreate on the instance after it calls the appropriate ejbCreate(...) method on the instance.

This method is called in the proper transaction context.

The EJB object identity has been established before the container calls ejbPostCreate() and therefore the instance can, for example, pass a reference to the EJB object to other enterprise beans in a method argument.

Throws: EJBException
Thrown if the instance could not perform the function requested by the container.

- **ejbStore**

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

A container invokes this method on the instance 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: EJBException
Thrown if the instance could not perform the function requested by the container.

- **setEntityContext**

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

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: EJBException

Thrown if the instance could not perform the function requested by the container.

- **`unsetEntityContext`**

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

Unset the associated entity context. The container calls this method before destroying 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: EJBException

Thrown if the instance could not perform the function requested by the container.

Interface EntityContext

```
public interface javax.ejb.EntityContext
    extends javax.ejb.InstanceContext
{
    public abstract EJBObject getEJBObject();
    public abstract Object getPrimaryKey();
    public abstract void
        setPrimaryKey(Object primaryKey);
}
```

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 in the "ready" state, i.e. after and within the ejbActivate or ejbCreate method, and before the ejbPassivate or ejbDestroy method. If the instance calls this method from the ejbCreate method, it may do it only after it called setPrimaryKey.

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 entity enterprise bean instance can call this method only when the instance is in the ready state; from the ejbActivate(), ejbPassivate(), and ejbDestroy() methods; and from the ejbCreate(...) methods after the instance has called the set primary key method on the InstanceContext interface.

Note: The result of this method is that same as the 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.

- **setPrimaryKey**

```
public abstract void
setPrimaryKey(Object primaryKey)
     throws IllegalStateException, DuplicateKeyException
```

Associate the primary key with the EJB object that is being currently created. This method is callable only from the ejbCreate(...) methods of an entity enterprise bean class.

Throws: DuplicateKeyException

Thrown if creating of the entity would result in creation of a record with duplicate primary key. This exception is thrown only if the entity bean uses container managed persistence.

Throws: IllegalStateException

Thrown if the instance invokes this method from a method other than ejbCreate(...).

Interface Handle

```
public interface javax.ejb.Handle
    extends java.io.Serializable
{
    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.

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 InstanceContext

```
public interface javax.ejb.InstanceContext
{
    public abstract Identity
        getCallerIdentity();
    public abstract CurrentTransaction
        getCurrentTransaction();
    public abstract EJBHome getEJBHome();
    public abstract Properties
        getEnvironment();
    public abstract boolean
        isCallerInRole(Identity role);
    public abstract void setRollbackOnly();
}
```

The InstanceContext 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.

- **getCurrentTransaction**

  ```
  public abstract CurrentTransaction
  getCurrentTransaction()
       throws IllegalStateException
  ```

  Obtain the transaction demaraction interface.

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

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

- **getEJBHome**

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

  Obtain the enterprise bean's home interface.

  Returns:

The enterprise bean's home interfac.

- **`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.

- **`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 SessionBean

```
public interface javax.ejb.SessionBean
    extends javax.ejb.EnterpriseBean
{
    public abstract void ejbActivate();
    public abstract void ejbDestroy();
    public abstract void ejbPassivate();
    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 EJBException
```

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 in no transaction context.

Throws: EJBException
Thrown if the instance could not perform the function requested by the container.

- **ejbDestroy**

```
public abstract void ejbDestroy()
     throws EJBException
```

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 destroy operation, or when a container decides to terminate the session object after a timeout.

This method is called in no transaction context.

Throws: EJBException
Thrown if the instance could not perform the function requested by the container.

- **ejbPassivate**

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

The passivate method is called before the instance enters the "passive" state. The instance should release any resources that 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 in no transaction context.

Throws: EJBException

Thrown if the instance could not perform the function requested by the container.

- **`setSessionContext`**

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

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 in no transaction context.

Parameters:

`ctx`

An SessionContext interface for the instance.

Throws: EJBException

Thrown if the instance could not perform the function requested by the container.

Interface SessionContext

```
public interface javax.ejb.SessionContext
    extends javax.ejb.InstanceContext
{
    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 ejbDestroy() methods, including from within the ejbCreate() and ejbDestroy() 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 EJBException
```

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: EJBException
Thrown if the instance could not perform the function requested by the container.

- **afterCompletion**

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

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 committted, false if is has been rolled back.

Throws: EJBException
Thrown if the instance could not perform the function requested by the container.

- **beforeCompletion**

```
public abstract void beforeCompletion()
     throws EJBException
```

The beforeCompletions 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: EJBException
Thrown if the instance could not perform the function requested by the container.

Class CreateException

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

The CreateException exception should be included in the throws clause of the create(...) methods of entity enterprise beans. 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 DestroyException

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

The DestroyException exception is thrown at an attempt to destroy an EJB object, and the enterprise bean or the container does not allow the object to be destroyed.

Constructors

- **DestroyException**

  `public DestroyException()`

  Constructs an DestroyException with no detail message.

- **DestroyException**

  `public DestroyException(String message)`

  Constructs an DestroyException 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 EJBException

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

The EJBException exception is thrown by an enterprise bean instance to its container to report that the invoked business method or callback method could not be completed because of an unexpected error (e.g. the instance failed to open a database connection).

The enterprise bean provider should provide a detail descriptive message as part of the exception string. The container should log all reported EJBExceptions and make them available to the system administrator.

If a client-invoked method cannot not be completed because of an instance's throwing the EJBException, the container must throw the java.rmi.RemoteException, or an exception that is a subclass of java.rmi.RemoteException, to the client.

If the EJBException was thrown by an instance while the instance performed work in the scope of a transaction that was initiated by the container, the container must rollback the transaction must rollback the transaction before throwing the java.rmi.RemoteException to the client.

If the EJBException was thrown by an instance while the instance performed work in the scope of a transaction that was imported from the client, the container must mark the transaction for rollback and throw the javax.jts.TransactionRollbackException to the client. The javax.jts.TransactionRollbackException is a subclass of java.rmi.RemoteException, and it indicates to the client that it would be fruitless to continue work in the scope of the transaction.

Constructors

- **EJBException**

  `public EJBException()`

  Constructs an EJBException with no detail message.

- **EJBException**

  `public EJBException(Exception ex)`

  Constructs an EJBException that embeds the exception that caused the instance to thrown the EJBException.

- **EJBException**

  `public EJBException(String message)`

  Constructs an EJBException with the specified detail message.

Methods

- **`getCausedByException`**

  ```
  public Exception getCausedByException()
  ```

  Obtain the exception that caused the EJBException being thrown.

Class FinderException

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

/** The FinderException exception should be included in the throws clause of the findMETHOD(...) methods of entity enterprise beans. The exception is used as a standard application-level exception to report a failure to find the request 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 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 method 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 DeploymentDescriptor

```
public class javax.ejb.deployment.DeploymentDescriptor
    extends java.lang.Object
    implements java.io.Serializable
{
    public DeploymentDescriptor();
    public Class getEnterpriseBeanClass();
    public Properties
        getEnvironmentProperties();
    public Class getHomeInterface();
    public Name getJNDIName();
    public MethodDescriptor
        getMethodDescriptor(int index);
    public MethodDescriptor[]
        getMethodDescriptors();
    public Class getRemoteInterface();
    public SecurityDescriptor
        getSecurityDescriptor();
    public int getTransactionAttribute();
    public void
        setEnterpriseBeanClass(Class value);
    public void
        setEnvironmentProperties(Properties value);
    public void
        setHomeInterface(Class value);
    public void setJNDIName(Name value);
    public void
        setMethodDescriptor(int index,
                            MethodDescriptor value);
    public void
        setMethodDescriptors(MethodDescriptor value[]);
    public void
        setRemoteInterface(Class value);
    public void
        setSecurityDescriptor(SecurityDescriptor value);
    public void
        setTransactionAttribute(int value);
}
```

The DeploymentDescriptor class is the common baseclass for the SessionDescriptor and EntityDescriptor deployment descriptor classes. The class is designed as a JavaBean component.

A serialized instance of a deployment descriptor class is used as the standard format for passing the enterprise bean's declarative deployment attributes in the ejb-jar file. The deployment descriptor setter functions are used by the enterprise bean provider's tools to create the deployment descriptor before it is put into the ejb-jar file, and the getter functions are used by the container provider's tools to read the deployment descriptor from the ejb-jar file when the enterprise bean is installed into the container.

After an enterprise bean has been installed into a container, the container's tools can be then used to view and change the setting of the deployment attributes.

Constructors

- **DeploymentDescriptor**

  `public DeploymentDescriptor()`

  Create an instance of the deployment descriptor.

Methods

- **getEnterpriseBeanClass**

  `public Class getEnterpriseBeanClass()`

  Get the enterprise bean's class.

  Returns:
  The Class object for the enterprise bean's class.

- **getEnvironmentProperties**

  `public Properties getEnvironmentProperties()`

  Get enterprise bean's environment properties.

  Returns:
  Enterprise bean's environment properties.

- **getHomeInterface**

  `public Class getHomeInterface()`

  Get the enterprise bean's home interface.

  Returns:
  The Class object for the enterprise bean's home.

- **getJNDIName**

  `public Name getJNDIName()`

  Get the name to associate with the enterprise bean in the JNDI name space. The cointainer will make the enterprise bean's home interface available at a JNDI name that ends with this relative path.

  For example, if getJNDIName() returns "bank/Account", the container must make the bean's home interface available under the JNDI name "/prefix/bank/Account", where "prefix" can be an arbitrary composite JNDI name, e.g. "/x/y/z/bank/Account". The "prefix" part can also be empty.

  Returns:
  A JNDI name for this enterprise bean.

- **getMethodDescriptor**

  ```
  public MethodDescriptor
  getMethodDescriptor(int index)
  ```

  Get the method descriptor by an index.

  Parameters:
  `index`

The index of the method descriptor.

Returns:
The method descriptor at the specified index.

- **`getMethodDescriptors`**

  `public MethodDescriptor[] getMethodDescriptors()`

  Get the array of the enterprise bean's method descriptors.

  Returns:
  An array of enterprise bean's method descriptors, or null of the enterprise bean does not provide method-level descriptors.

- **`getRemoteInterface`**

  `public Class getRemoteInterface()`

  Get the enterprise bean's remote interface.

  Returns:
  The Class object for the enterprise bean's remote interface.

- **`getSecurityDescriptor`**

  `public SecurityDescriptor getSecurityDescriptor()`

  Get the enterprise bean's security descriptor.

  Returns:
  The enterprise bean's security descriptor.

- **`getTransactionAttribute`**

  `public int getTransactionAttribute()`

  Get the enterprise bean's transaction attribute. The values of the transaction attribute are defined in the class TransactionAttribute.

  Returns:
  The enterprise bean's transaction attribute.

- **`setEnterpriseBeanClass`**

  `public void setEnterpriseBeanClass(Class value)`

  Set the enterprise bean's class.

  Parameters:
  `value`
  The Class object for the enterprise bean's class.

- **`setEnvironmentProperties`**

  `public void`
  `setEnvironmentProperties(Properties value)`

  Set enterprise bean's environment properties.

  Parameters:
  `value`
  Enterprise bean's environment properties.

- **setHomeInterface**

```
public void setHomeInterface(Class value)
```

Set the enterprise bean's home interface.

Parameters:

`value`
The Class object for the enterprise beans home.

- **setJNDIName**

```
public void setJNDIName(Name value)
```

Set the name to associate with the enterprise bean in the JNDI name space. The cointainer will make the enterprise bean's home interface available at a JNDI name that ends with this relative path.

Parameters:

`value`
A JNDI name for this enterprise bean.

- **setMethodDescriptor**

```
public void
setMethodDescriptor(int index,
                    MethodDescriptor value)
```

Set the method descriptor by an index.

Parameters:

`index`
The index of the method descriptor.

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

- **setMethodDescriptors**

```
public void
setMethodDescriptors(MethodDescriptor value[])
```

Set the array of the enterprise bean's method descriptors.

A deployment descriptor may define zero or several MethodDescriptors. If a MethodDescriptor is defined for a method, the MethodDescriptor overrides the values of the transactionAttribute and securityDescriptor set at the bean level.

A MethodDescriptor can be provided for the methods defined in the enterprise bean's remote interface, home interface,

Parameters:

`value`
An array of the enterprise bean's method descriptors.

- **setRemoteInterface**

```
public void setRemoteInterface(Class value)
```

Set the enterprise bean's remote interface.

Parameters:

`value`
The Class object for the enterprise bean's remote interface.

- **`setSecurityDescriptor`**

  ```
  public void
  setSecurityDescriptor(SecurityDescriptor value)
  ```

  Set the enterprise bean's security descriptor.

  The SecurityDescriptor applies to all methods of the enterprise bean, unless it is overridden by a MethodDescriptor.

  Parameters:

  `value`
  The enterprise bean's security descriptor.

- **`setTransactionAttribute`**

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

  Set the enterprise bean's transaction attribute. The values of the transaction attribute are defined in the class TransactionAttribute.

  The transaction attribute applies to all methods of the enterprise bean, unless it is overridden by a MethodDescriptor.

  Parameters:

  `value`
  The enterprise bean's transaction attribute.

Class EntityDescriptor

```
public class javax.ejb.deployment.EntityDescriptor
    extends javax.ejb.deployment.DeploymentDescriptor
{
    public EntityDescriptor();
    public Field
        getContainerManagedField(int index);
    public Field[]
        getContainerManagedFields();
    public Class getPrimaryKeyClass();
    public void
        setContainerManagedField(int index,
                                 Field value);
    public void
        setContainerManagedFields(Field values[]);
    public void
        setPrimaryKeyClass(Class value);
}
```

The SessionDescriptor class defines the deployment descriptor for an entity enterprise bean.

A serialized instance of a deployment descriptor class is used as the standard format for passing an enterprise bean's declarative attributes in the ejb-jar file. The deployment descriptor setter functions are used by the enterprise bean provider's tools to create the deployment descriptor before it is put into the ejb-jar file, and the getter functions are used by the container provider's tools to read the deployment descriptor from the ejb-jar file.

Constructors

- **EntityDescriptor**

  `public EntityDescriptor()`

  Create an instance of the deployment descriptor.

Methods

- **getContainerManagedField**

  `public Field getContainerManagedField(int index)`

  Get the container-managed field at the given index.

  Parameters:

  `index`
  The index of the field.

  Returns:
  The Field of the specified container-managed field.

- **getContainerManagedFields**

  `public Field[] getContainerManagedFields()`

Get the array of container-managed enterprise bean's fields.

Returns:
The array of container-managed fields.

- **`getPrimaryKeyClass`**

  `public Class getPrimaryKeyClass()`

  Get the enterprise bean's primary key class.

  Returns:
  The Class object for the primary key.

- **`setContainerManagedField`**

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

  Set the container-managed field at the given index.

  Parameters:
  `index`
  The index of the field.
  `value`
  The Field of the specified container-managed field.

- **`setContainerManagedFields`**

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

  Set the array of container-managed enterprise bean's fields.

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

- **`setPrimaryKeyClass`**

  `public void setPrimaryKeyClass(Class value)`

  Get the enterprise bean's primary key class.

  Parameters:
  `value`
  The Class object for the primary key.

Class IsolationLevel

```
public class javax.ejb.deployment.IsolationLevel
    extends java.lang.Object
{
    public final static int NONE;
    public final static int READ_COMMITTED;
    public final static int READ_UNCOMMITTED;
    public final static int REPEATABLE_READ;
    public final static int SERIALIZABLE;
}
```

The IsolationLevel class defines the value of the enterprise bean's transaction isolation levels.

The deployment descriptor of an entity bean with container-managed persistence uses the values defined in the IsolationLevel class to tell the container the requested isolation level for the database access operations that will be generated by the container provider tools.

Variables

- **NONE**

  `public final static int NONE`

  Transactions are not supported. This is an illegal value for a an entity with container-managed persistence.

- **READ_COMMITTED**

  `public final static int READ_COMMITTED`

  The container tools must generate database access calls with an isolation degree that is equivalent to the JDBC TRANSACTION_READ_COMMITTED level.

- **READ_UNCOMMITTED**

  `public final static int READ_UNCOMMITTED`

  The container tools must generate database access calls with an isolation degree that is equivalent to the JDBC TRANSACTION_READ_UNCOMMITTED level.

- **REPEATABLE_READ**

  `public final static int REPEATABLE_READ`

  The container tools must generate database access calls with an isolation degree that is equivalent to the JDBC TRANSACTION_REPEATABLE_READ level.

- **SERIALIZABLE**

  `public final static int SERIALIZABLE`

  The container tools must generate database access calls with an isolation degree that is equivalent to the JDBC TRANSACTION_SERIALIZABLE level.

Class MethodDescriptor

```
public class javax.ejb.deployment.MethodDescriptor
    extends java.lang.Object
{
    public MethodDescriptor(Method method);
    public Method getMethod();
    public SecurityDescriptor
        getSecurityDescriptor();
    public int getTransactionAttribute();
    public void
        setSecurityDescriptor(SecurityDescriptor value);
    public void
        setTransactionAttribute(int value);
}
```

The MethodDescriptor is used to provide deployment information specific to a single method. The method-level information overrides any information set at the enterprise bean-level.

A MethodDescriptor can be provided for the following methods of a session enterprise bean class: ejbCreate(...), ejbDestroy(), and any business method.

A MethodDescriptor can be provided for the following methods of an entity enterprise bean class: ejbCreate(...), ejbDestroy(), ejbFindMETHOD(...), and any business method.

Constructors

- **MethodDescriptor**

  `public MethodDescriptor(Method method)`

  Construct a MethodDescriptor for a given Method.

Methods

- **getMethod**

  `public Method getMethod()`

  Get the Method to which this MethodDescriptor applies.

  Returns:
  The Method associated with this MethodDescriptor.

- **getSecurityDescriptor**

  `public SecurityDescriptor getSecurityDescriptor()`

  Get the SecurityDescriptor for the method.

  Returns:
  The SecurityDescriptor for the method.

- **getTransactionAttribute**

  `public int getTransactionAttribute()`

Get the transaction attribute for the method.

Returns:
The transaction attribute for the method.

- **`setSecurityDescriptor`**

```
public void
setSecurityDescriptor(SecurityDescriptor value)
```

Set the SecurityDescriptor for the method.

Parameters:
`value`
The SecurityDescriptore for the method.

- **`setTransactionAttribute`**

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

Set the transaction attribute for the method.

Parameters:
`value`
The transaction attribute for the method.

Class SecurityDescriptor

```
public class javax.ejb.deployment.SecurityDescriptor
    extends java.lang.Object
{
    public final static int Client;
    public final static int SpecifiedUser;
    public final static int System;
    public SecurityDescriptor();
    public Identity
        getAllowedUser(int index);
    public Identity[] getAllowedUsers();
    public int getRunAsAttribute();
    public Identity getRunAsIdentity();
    public void setAllowedUser(int index,
                               Identity value);
    public void
        setAllowedUsers(Identity values[]);
    public void setRunAsAttribute(int value);
    public void
        setRunAsIdentity(Identity value);
}
```

The SecurityDescriptor descriptor defines security-related attributes for enterprise bean. A SecurityDescriptor can be used both at the level of the entire bean, or the level of individual methods.

Variables

- **Client**

  `public final static int Client`

  Run the enterprise bean method with the client's Identity.

- **SpecifiedUser**

  `public final static int SpecifiedUser`

  Run the enterprise bean method with the Identity of a specified user account.

- **System**

  `public final static int System`

  Run the enterprise bean method with the Identity of a "privileged account". The container maps the abstract notion of a "privileged account" to some privileged account on the target system, such as the database administrator, or the operating system administrator account.

Constructors

- **SecurityDescriptor**

  `public SecurityDescriptor()`

  Constructor.

Methods

- **getAllowedUser**

  `public Identity getAllowedUser(int index)`

  Get the Identity at the specified index in the array of Identities that are granted access to execute the enterprise bean method.

  Parameters:

  `index`
  Index of the Identity to be obtained.

  Returns:
  The Identity at the specified index.

- **getAllowedUsers**

  `public Identity[] getAllowedUsers()`

  Get the array of Identities that are granted access to execute the enterprise bean method.

  Returns:
  An array of Identities.

- **getRunAsAttribute**

  `public int getRunAsAttribute()`

  Get the "runAsAttribute" security attribute.

  Returns:
  The runAs attribute. The value must be one of Client, SpecifiedUser, and System.

- **getRunAsIdentity**

  `public Identity getRunAsIdentity()`

  Get the Identity of the specified user account with whose credentials to run the enterprise bean method. This attribute is consulted only if the value of the runAsAttribute is SpecifiedUser.

  Returns:
  The Identity to associate with the execution of the enterprise bean method.

- **setAllowedUser**

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

  Set the Identity at the specified index in the array of Identities that are granted access to execute the enterprise bean method.

  Parameters:

  `index`
  Index of the Identity to be set.

  `value`
  The Identity to be set.

- **setAllowedUsers**

  `public void setAllowedUsers(Identity values[])`

Set the array of Identities that are granted access to execute the enterprise bean method. param value An array of Identities.

- **`setRunAsAttribute`**

  `public void setRunAsAttribute(int value)`

  Set the "runAsAttribute" security attribute.

  Parameters:

  `value`
  : The runAs attribute. The value must be one of Client, SpecifiedUser, and System.

- **`setRunAsIdentity`**

  `public void setRunAsIdentity(Identity value)`

  Set the Identity of the specified user account with whose credentials to run the enterprise bean method. This attribute is consulted only if the value of the runAsAttribute is set to SpecifiedUser.

  Parameters:

  `value`
  : The Identity to associate with the execution of the enterprise bean method.

Class SessionDescriptor

```
public class javax.ejb.deployment.SessionDescriptor
    extends javax.ejb.deployment.DeploymentDescriptor
{
    public final static int STATEFUL;
    public final static int STATELESS;
    public SessionDescriptor();
    public int getSessionTimeout();
    public int getStateManagement();
    public void setSessionTimeout(int value);
    public void
        setStateManagement(int value);
}
```

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

A serialized instance of a deployment descriptor class is used as the standard format for passing an enterprise bean's declarative attributes in the ejb-jar file. The deployment descriptor setter functions are used by the enterprise bean provider's tools to create the deployment descriptor before it is put into the ejb-jar file, and the getter functions are used by the container provider's tools to read the deployment descriptor from the ejb-jar file.

Variables

- **STATEFUL**

  `public final static int STATEFUL`

  The session bean is stateful and can be passivated between transactions.

- **STATELESS**

  `public final static int STATELESS`

  The session bean is stateless. A stateless bean can be reused for multiple session objects.

Constructors

- **SessionDescriptor**

  `public SessionDescriptor()`

  Create an instance of the deployment descriptor.

Methods

- **getSessionTimeout**

  `public int getSessionTimeout()`

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

  Returns:

  The timeout value in seconds.

- **getStateManagement**

```
public int getStateManagement()
     throws IllegalStateException
```

Get the session bean's state management attribute.

Returns:
The session bean's state management attribute. Its value must be either STATELESS or STATEFUL.

Throws: IllegalStateException
Thrown if the attribute's value has not yet been set.

- **setSessionTimeout**

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

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

Parameters:
`value`
The timeout value in seconds.

- **setStateManagement**

```
public void setStateManagement(int value)
```

Set the session bean's state management attribute.

Returns:
The session bean's state management attribute. Its value must be either STATELESS or STATEFUL.

Class TransactionAttribute

```
public class javax.ejb.deployment.TransactionAttribute
    extends java.lang.Object
{
    public final static int BEAN_MANAGED;
    public final static int MANDATORY;
    public final static int NOT_SUPPORTED;
    public final static int REQUIRED;
    public final static int REQUIRES_NEW;
    public final static int SUPPORTS;
}
```

The TransactionAttribute class defines the value of the enterprise bean's transaction attribute.

Variables

- **BEAN_MANAGED**

  `public final static int BEAN_MANAGED`

  The enterprise bean manages transaction boundaries itself using the java.jts.CurrentTransaction interface. The container does not perform any transaction management on the bean's behalf.

- **MANDATORY**

  `public final static int MANDATORY`

  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 java.jts.TransactionRequiredException to the caller.

- **NOT_SUPPORTED**

  `public final static int NOT_SUPPORTED`

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

- **REQUIRED**

  `public final static int REQUIRED`

  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 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.

- **REQUIRES_NEW**

  `public final static int REQUIRES_NEW`

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.

- **SUPPORTS**

```
public final static int SUPPORTS
```

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.

19 Related documents

[1] JavaBeans. *http://www.javasoft.com/beans*.

[2] Java Naming and Directory Interface (JNDI). *http://www.javasoft.com/products/jndi*.

[3] Java Remote Method Invocation (RMI). *http://www.javasoft.com/products/rmi.*

[4] Java Security. *http://www.javasoft.com/security.*

[5] Java to IDL Mapping. Joint Initial Submission. OMG TC Document TC orbos/97-08-06.

[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: Glossary of terms

Appendix B: Example application

TODO

Appendix C: 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.

C.1 Programmatic access to security

We would like to allow expert-level enterprise beans to temporarily change their effective security Identity.

C.2 Enterprise beans with extended transactional semantics

Some enterprise beans may need to themselves be resource managers. One example is an enterprise bean that tries to provide transactional semantics across resource managers that do not participate in two-phase commit. The access to legacy data or business logic may be transaction monitor requests or a database stored procedures, each of which is itself a transaction with a corresponding compensating transaction. The enterprise bean would keep track of these transactions and call the compensating transaction if rollback occurred. This allows the same simple transactional programming model to be used externally on the enterprise bean (i.e., transaction begin, commit, rollback), while not holding locks on the real resources.

These enterprise beans need a much richer contract with the transaction service that allows full participation in two-phase commit.

Appendix D: Issues and dependencies

D.1 Pending issues

We track the important pending issues here on which we would like to receive input from the reviewers.

D.1.1 Argument passing semantics

Enterprise JavaBeans should define the rules for the semantics of parameter passing in calls between two enterprise beans. We believe that the arguments that implement the *java.rmi.Remote* interface should be passed by-reference, and that all other arguments should be passed by-value even if the target enterprise bean is in the same container and JVM.

We believe that this needs to be made part of the architecture to prevent unpredictable behaviour of applications built from enterprise beans when deployed in different containers.

These proposed semantics are consistent with those used by RMI and the Java to IDL Mapping [5] and therefore would be familiar to Java programmers.

D.1.2 Runtime significance of ejb-jar file

In this revision of the specification, an ejb-jar file has no significance at runtime. This means that a container treats a method invocation between two enterprise bean objects in the same way, independent from whether the two enterprise beans are part of the same or different ejb-jar files.

Some reviewers suggested that it would make sense to treat the interactions between the enterprise beans that are part of the same ejb-jar *special*, as follows.

- The *runAsIdentity* applies to the whole ejb-jar file, not to the individual enterprise beans.
- All arguments are passed by-reference (i.e. using the Java semantics, not the Java RMI semantics).

This would imply that an ejb-jar should treated as a single application with multiple entry points (each enterprise bean is an entry point) rather than a collection of independent components (independent at least as far as the transaction, state, and security management is concerned). For a given transaction, a container would have to execute all the enterprise beans that are part of an ejb-jar in the same JVM process in order to preserve the by-reference semantics of argument passing.

This proposal would effectively lead to a two-level component architecture (ejb-jar files and individual enterprise beans) with more complex rules than in the current single-level component architecture. We have currently no plans to introduce such a two-level component architecture.

D.1.3 Declarative transaction attribute at method-level

Many reviewers gave us input that enterprise beans are likely to be coarse-grained objects potentially with a large number of client callable methods. A single declarative transaction attribute at the object-level may often become a problem because different methods may require different transaction attributes.

The suggested solution is to allow a per-method transaction attribute to override the one at object-level. This would be analogous to the security permissions defined on per-method basis.

D.1.4 *NOT_SUPPORTED* transaction attribute

It was pointed out that the name of this transaction attribute might be confusing if the enterprise bean in fact uses JTS to perform explicit transaction demarcation. A suggestion was made that we add a new attribute (e.g. *UsesCurrentTransaction*) that distinguishes an enterprise bean that uses the *CurrentTransaction* interface from one that does not support transactions. The container would not make the *CurrentTransaction* interface available to an enterprise bean that does not support transaction.

We have added the BEAN_MANAGED transaction attribute to the 0.796 specification.

D.1.5 Deployment descriptor format

Many reviewers suggested that using a serialized bean as a deployment descriptor would be a better alternative to using a *java.util.Properties* file.

We have changed the format of the deployment descriptor to a serialized Java object in the 0.796 release.

Appendix E: 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 CurrentTransaction
interface TransactionControl

class Status

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

Interface CurrentTransaction

```
public interface javax.jts.CurrentTransaction
{
    public abstract void begin();
    public abstract void commit();
    public abstract TransactionControl
        getControl();
    public abstract int getStatus();
    public abstract void
        resume(TransactionControl suspended);
    public abstract void rollback();
    public abstract void rollbackOnly();
    public abstract void
        setTransactionTimeout(int seconds);
    public abstract TransactionControl
        suspend();
}
```

The CurrentTransaction interface defines the methods that allow an application to to explicitly manage transaction boundaries and control the association of transactions and threads.

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.

- **getControl**

```
public abstract TransactionControl getControl()
```

Obtain the TransactionControl for the transaction that is associated associated with the current thread.

The returned TransactionControl can be used later by the thread as an argument to the resume method.

Returns:
The TransactionControl control for the transaction that is associated with the thread, or null if no transaction is associated with the thread.

- **getStatus**

```
public abstract int getStatus()
```

Obtain the status of the transaction associated with the current thread.

Returns:
The transaction status. The values of the transaction status are defined in the javax.jts.Status class. If no transaction is associated with the current thread, this method returns the Status.NoTransaction value.

- **resume**

```
public abstract void
resume(TransactionControl suspended)
     throws IllegalArgumentException
```

Resume association of a transaction with the current thread.

Parameters:
`suspended`
A TransactionControl obtained previously by the current thread via the suspend or getControl method.

Throws: IllegalArgumentException
Thrown if the TransactionControl is invalid for the current thread (i.e. it was not obtained via the suspend or getControl method.

- **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.

- **rollbackOnly**

```
public abstract void rollbackOnly()
     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.

- **`suspend`**

```
public abstract TransactionControl suspend()
```

Suspend the association of the current thread with a transaction. When this method completes, the thread becomes associated with no transaction.

The returned TransactionControl can be used later by the thread as an argument to the resume method.

Returns:
The TransactionControl control for the transaction that was associated with the thread, or null if no transaction was associated with the thread.

Interface TransactionControl

```
public interface javax.jts.TransactionControl
{
}
```

The TransactionControl interface represents a transactions. It does not define any methods that would allow the application to directly manipulate the transaction.

Class Status

```
public class javax.jts.Status
    extends java.lang.Object
{
    public final static int Active;
    public final static int Committed;
    public final static int Committing;
    public final static int MarkedRollback;
    public final static int NoTransaction;
    public final static int Prepared;
    public final static int Preparing;
    public final static int RolledBack;
    public final static int RollingBack;
    public final static int Unknown;
}
```

The class Status defines the values of a transaction status.

Variables

- **Active**

  `public final static int 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.

- **Committed**

  `public final static int 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.

- **Committing**

  `public final static int 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.

- **MarkedRollback**

  `public final static int MarkedRollback`

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

- **NoTransaction**

  `public final static int NoTransaction`

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

- **Prepared**

  `public final static int Prepared`

  A transaction is associated with the target object and 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.

- **Preparing**

  `public final static int 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.

- **RolledBack**

  `public final static int 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.

- **RollingBack**

  `public final static int RollingBack`

  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.

- **Unknown**

  `public final static int Unknown`

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

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 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 occured when trying to register a resource.

Constructors

- **InvalidTransactionException**

  `public InvalidTransactionException()`

- **InvalidTransactionException**

  `public InvalidTransactionException(String msg)`

Appendix F: Revision history

F.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 *CurrentTransaction.begin()*.

Added the specification of isolation levels for container-managed entity beans.