SCBCD 5.0 Study Guide

By Mikalai Zaikin

Table of Contents

Identify correct and incorrect statements or examples about implementing a session bean as a web service endpoint, including rules for writing a web service endpoint interface and use of the @WebService and @WebMethod annotations.

Identify correct and incorrect statements or examples about the client view of a session bean, including the client view of a session object's life cycle, obtaining and using a session object, and session object identity.

4. EJB 3.0 Message-Driven Bean Component Contract

Develop code that implements a Message-Driven Bean Class.

Identify correct and incorrect statements or examples about the interface(s) and methods a JMS Message-Driven bean must implement, and the required metadata.

Describe the use and behavior of a JMS message driven bean, including concurrency of message processing, message redelivery, and message acknowledgement.

Identify correct and incorrect statements or examples about the client view of a message driven bean.

5. Java Persistence API Entities

Identify correct and incorrect statements or examples about the characteristics of Java Persistence entities.

Develop code to create valid entity classes, including the use of fields and properties, admissible types, and embeddable classes.

Identify correct and incorrect statements or examples about primary keys and entity identity, including the use of compound primary keys.

Implement association relationships using persistence entities, including the following associations: bidirectional for @OneToOne, @ManyToOne, @OneToMany, and @ManyToMany; unidirectional for @OneToOne, @ManyToOne, @OneToMany, and @ManyToMany.

Given a set of requirements and entity classes choose and implement an appropriate object-relational mapping for association relationships.

Given a set of requirements and entity classes, choose and implement an appropriate inheritance hierarchy strategy and/or an appropriate mapping strategy.

Describe the use of annotations and XML mapping files, individually and in combination, for object-relational mapping.

6. Java Persistence Entity Operations

Describe how to manage entities, including using the EntityManager API and the cascade option.

Identify correct and incorrect statements or examples about entity instance lifecycle, including the new, managed, detached, and removed states.

Identify correct and incorrect statements or examples about EntityManager operations for managing an instance's state, including eager/lazy fetching, handling detached entities, and merging detached entities.

Identify correct and incorrect statements or examples about Entity Listeners and Callback Methods, including: @PrePersist, @PostPersist, @PreRemove, @PostRemove, @PreUpdate, @PostUpdate, and @PostLoad, and when they are invoked.

Identify correct and incorrect statements about concurrency, including how it is managed through the use of @Version attributes and optimistic locking.

7. Persistence Units and Persistence Contexts

Identify correct and incorrect statements or examples about JTA and resource-local entity managers.

Identify correct and incorrect statements or examples about container-managed persistence contexts.

Identify correct and incorrect statements or examples about application-managed persistence contexts.

Identify correct and incorrect statements or examples about transaction management for persistence contexts, including persistence context propagation, the use of the EntityManager.joinTransaction() method, and the EntityTransaction API.

Identify correct and incorrect statements or examples about persistence units, how persistence units are packaged, and the use of the persistence.xml file.

Identify correct and incorrect statements or examples about the effect of persistence exceptions on transactions and persistence contexts.

8. Java Persistence Query Language

Develop queries that use the SELECT clause to determine query results, including the use of entity types, use of aggregates, and returning multiple values.

Develop queries that use Java Persistence Query Language syntax for defining the domain of a query using JOIN clauses, IN, and prefetching.

Use the WHERE clause to restrict query results using conditional expressions, including the use of literals, path expressions, named and positional parameters, logical operators, the following expressions (and their NOT options): BETWEEN, IN, LIKE, NULL, EMPTY, MEMBER [OF], EXISTS, ALL, ANY, SOME, and functional expressions.

Develop Java Persistence Query Language statements that update a set of entities using UPDATE/SET and DELETE FROM.

Declare and use named queries, dynamic queries, and SQL (native) queries.

Obtain javax.persistence.Query objects and use the javax.persistence.Query API.

9. Transactions

Identify correct and incorrect statements or examples about bean-managed transaction demarcation.

Identify correct and incorrect statements or examples about container-managed transaction demarcation, and given a list of transaction behaviors, match them with the appropriate transaction attribute.

Identify correct and incorrect statements or examples about transaction propagation semantics.

Identify correct and incorrect statements or examples about specifying transaction information via annotations and/or deployment descriptors.

Identify correct and incorrect statements or examples about the use of the EJB API for transaction management, including getRollbackOnly, setRollbackOnly and the SessionSynchronization interfaces.

10. Exceptions

Identify correct and incorrect statements or examples about exception handling in EJB.

Identify correct and incorrect statements or examples about application exceptions and system exceptions in session beans and message-driven beans, and defining a runtime exception as an application exception.

Given a list of responsibilities related to exceptions, identify those which are the bean provider's, and those which are the responsibility of the container provider. Be prepared to recognize responsibilities for which neither the bean nor container provider is responsible.

Identify correct and incorrect statements or examples about the client's view of exceptions received from an enterprise bean invocation.

Given a particular method condition, identify the following: whether an exception will be thrown, the type of exception thrown, the container's action, and the client's view.

Part I. Exam Objectives

Chapter 1. EJB 3.0 Overview

- **Identify the uses, benefits, and characteristics of Enterprise JavaBeans technology, for version 3.0 of the EJB specification.**

- [EJB_3.0_CORE]

- **Identify the APIs that all EJB 3.0 containers must make available to developers.**

- [EJB_3.0_CORE] 21.2; 21.2.2; 21.2.3; 21.2.4; 21.2.5; 21.2.6; 21.2.7

An EJB 3.0 container must make the following APIs available to the enterprise bean instances at runtime:

Java 2 Platform, Standard Edition v5 (J2SE) APIs, which include the following APIs:

- JDBC

The EJB container must include the JDBC 3.0 extension and provide its functionality to the enterprise bean instances, with the exception of the low-level XA and connection pooling interfaces. These low-level interfaces are intended for integration of a JDBC driver with an application server, not for direct use by enterprise beans.

- RMI-IIOP

An enterprise bean's remote business interfaces and/or remote home and remote interfaces are *remote interfaces* for Java RMI. The container must ensure the semantics for passing arguments conforms to Java RMI-IIOP. Non-remote objects must be passed by value.

Specifically, the EJB container is not allowed to pass non-remote objects by reference on inter-EJB invocations when the calling and called enterprise beans are collocated in the same JVM. Doing so could result in the multiple beans sharing the state of a Java object, which would break the enterprise bean's semantics. Any local optimizations of remote interface calls must ensure the semantics for passing arguments conforms to Java RMI-IIOP.

An enterprise bean's local business interfaces and/or local home and local interfaces are *local Java interfaces*. The caller and callee enterprise beans that make use of these local interfaces are typically collocated in the same JVM. The EJB container must ensure the semantics for passing arguments across these interfaces conforms to the standard argument passing semantics of the Java programming language.

- JNDI

At the minimum, the EJB container must provide a JNDI API name space to the

enterprise bean instances. The EJB container must make the name space available to an instance when the instance invokes the javax.naming.InitialContext default (no-arg) constructor.

The EJB container must make available at least the following objects in the name space:

The business interfaces of other enterprise beans.

The resource factories used by the enterprise beans.

The entity managers and entity manager factories used by the enterprise beans.

The web service interfaces used by the enterprise beans.

The home interfaces of other enterprise beans.

ORB objects

UserTransaction objects

EJBContext objects

TimerService objects

JAXP

Java IDL

EJB 3.0 APIs, including the Java Persistence API

The container must implement the semantics of the metadata annotations that are supported by EJB 3.0

The container must implement (or provide through a third-party implementation) the javax.persistence interfaces and metadata annotations.

- JTA 1.1, the UserTransaction interface only

The EJB container must include the JTA 1.1 extension, and it must provide the javax.transaction.UserTransaction interface to enterprise beans with bean-managed transaction demarcation through the javax.ejb.EJBContext interface, and also in JNDI under the name java:comp/UserTransaction, in the cases required by the EJB specification.

- JMS 1.1

The EJB container must include the JMS 1.1 extension and provide its functionality to the enterprise bean instances, with the exception of the low-level interfaces that are intended for integration of a JMS provider with an application server, not for direct use by enterprise beans.

- JavaMail 1.4, sending mail only
- JAF 1.1
- JAXP 1.2

- JAXR 1.0
- JAX-RPC 1.1
- JAX-WS 2.0
- JAXB 2.0
- SAAJ 1.3
- Connector 1.5
- Web Services 1.2
- Web Services Metadata 2.0
- Common Annotations 1.0
- StAX 1.0

- NOTE: The EJB architecture DOES NOT require the EJB container to support the JTS interfaces.

Identify correct and incorrect statements or examples about EJB programming restrictions.

- An enterprise bean must not use read/write static fields. Using read-only static fields is allowed. Therefore, it is recommended that all static fields in the enterprise bean class be declared as final.

  This rule is required to ensure consistent runtime semantics because while some EJB containers may use a single JVM to execute all enterprise bean's instances, others may distribute the instances across multiple JVMs.

- An enterprise bean must not use thread synchronization primitives to synchronize execution of multiple instances.

  This is for the same reason as above. Synchronization would not work if the EJB container distributed enterprise bean's instances across multiple JVMs.

- An enterprise bean must not use the AWT functionality to attempt to output information to a display, or to input information from a keyboard.

  Most servers do not allow direct interaction between an application program and a keyboard/display attached to the server system.

- An enterprise bean must not use the java.io package to attempt to access files and directories in the file system.

  The file system APIs are not well-suited for business components to access data. Business components should use a resource manager API, such as JDBC, to store data.

- An enterprise bean must not attempt to listen on a socket, accept connections on a socket, or use a socket for multicast.

  The EJB architecture allows an enterprise bean instance to be a network socket client, but

it does not allow it to be a network server. Allowing the instance to become a network server would conflict with the basic function of the enterprise bean - to serve the EJB clients.

- The enterprise bean must not attempt to query a class to obtain information about the declared members that are not otherwise accessible to the enterprise bean because of the security rules of the Java language. The enterprise bean must not attempt to use the Reflection API to access information that the security rules of the Java programming language make unavailable.

  Allowing the enterprise bean to access information about other classes and to access the classes in a manner that is normally disallowed by the Java programming language could compromise security.

- The enterprise bean must not attempt to create a class loader; obtain the current class loader; set the context class loader; set security manager; create a new security manager; stop the JVM; or change the input, output, and error streams.

  These functions are reserved for the EJB container. Allowing the enterprise bean to use these functions could compromise security and decrease the container's ability to properly manage the runtime environment.

- The enterprise bean must not attempt to set the socket factory used by ServerSocket, Socket, or the stream handler factory used by URL.

  These networking functions are reserved for the EJB container. Allowing the enterprise bean to use these functions could compromise security and decrease the container's ability to properly manage the runtime environment.

- The enterprise bean must not attempt to manage threads. The enterprise bean must not attempt to start, stop, suspend, or resume a thread, or to change a thread's priority or name. The enterprise bean must not attempt to manage thread groups.

  These functions are reserved for the EJB container. Allowing the enterprise bean to manage threads would decrease the container's ability to properly manage the runtime environment.

- The enterprise bean must not attempt to directly read or write a file descriptor.

  Allowing the enterprise bean to read and write file descriptors directly could compromise security.

- The enterprise bean must not attempt to obtain the security policy information for a particular code source.

  Allowing the enterprise bean to access the security policy information would create a security hole.

- The enterprise bean must not attempt to load a native library.

  This function is reserved for the EJB container. Allowing the enterprise bean to load native code would create a security hole.

- The enterprise bean must not attempt to gain access to packages and classes that the usual rules of the Java programming language make unavailable to the enterprise bean.

This function is reserved for the EJB container. Allowing the enterprise bean to perform this function would create a security hole.

- The enterprise bean must not attempt to define a class in a package.

This function is reserved for the EJB container. Allowing the enterprise bean to perform this function would create a security hole.

- The enterprise bean must not attempt to access or modify the security configuration objects (Policy, Security, Provider, Signer, and Identity).

These functions are reserved for the EJB container. Allowing the enterprise bean to use these functions could compromise security.

- The enterprise bean must not attempt to use the subclass and object substitution features of the Java Serialization Protocol.

Allowing the enterprise bean to use these functions could compromise security.

- The enterprise bean must not attempt to pass this as an argument or method result. The enterprise bean must pass the result of SessionContext.getBusinessObject, SessionContext.getEJBObject, SessionContext.getEJBLocalObject, EntityContext.getEJBObject, or EntityContext.getEJBLocalObject instead.

Match the seven EJB roles with the corresponding description of the role's responsibilities.

1. **Enterprise Bean Provider**

The **Enterprise Bean Provider** (**Bean Provider** for short) is the producer of enterprise beans. His or her output is an ejb-jar file that contains one or more enterprise beans. The Bean Provider is responsible for the Java classes that implement the enterprise beans' business methods; the definition of the beans' client view interfaces; and declarative specification of the beans' metadata. The beans' metadata may take the form of metadata annotations applied to the bean classes and/or an external XML deployment descriptor. The beans' metadata - whether expressed in metadata annotations or in the deployment descriptor - includes the structural information of the enterprise beans and declares all the enterprise beans' external dependencies (e.g. the names and types of resources that the enterprise beans use).

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

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

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

2. **Application Assembler**

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

   The Application Assembler can also combine enterprise beans with other types of application components when composing an application.

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

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

3. **Deployer**

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

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

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

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

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

      The Deployer first generates the additional classes and interfaces that enable the container to manage the enterprise beans at runtime. These classes are container-

specific.

The Deployer performs the actual installation of the enterprise beans and the additional classes and interfaces into the EJB container.

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

4. **EJB Server Provider**

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

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

5. **EJB Container Provider**

   The **EJB Container Provider** (**Container Provider** for short) provides:

   The deployment tools necessary for the deployment of enterprise beans.

   The runtime support for the deployed enterprise bean instances.

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

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

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

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

   The Container Provider typically provides tools that allow the System Administrator to monitor and manage the container and the beans running in the container at runtime.

6. **Persistence Provider**

   The expertise of the **Persistence Provider** is in object/relational mapping, query processing, and caching. The focus of the Persistence Provider is on the development of a scalable, transaction-enabled runtime environment for the management of persistence.

   The Persistence Provider provides the tools necessary for the object/relational mapping of persistent entities to a relational database, and the runtime support for the management of persistent entities and their mapping to the database.

   The Persistence Provider insulates the persistent entities from the specifics of the underlying persistence substrate, providing a standard API between the persistent entities and the object/relational runtime.

   The Persistence Provider may be the same vendor as the EJB Container vendor or the Persistence Provider may be a third-party vendor that provides a pluggable persistence environment.

7. **System Administrator**

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

Describe the packaging and deployment requirements for enterprise beans.

The ejb-jar file is the standard format for the packaging of enterprise beans. The ejb-jar file format is used to package un-assembled enterprise beans (the Bean Provider's output), and to package assembled applications (the Application Assembler's output).

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

An ejb-jar file produced by the Bean Provider contains one or more enterprise beans that typically do not contain application assembly instructions. The ejb-jar file produced by an Application Assembler (which can be the same person or organization as the Bean Provider) contains one or more enterprise beans, plus application assembly information describing how the enterprise beans are combined into a single application deployment unit.

Deployment Descriptor

The ejb-jar file must contain the deployment descriptor (if any) in the format defined in the Deployment Descriptor XML Schema. The deployment descriptor must be stored with the name META-INF/ejb-jar.xml in the ejb-jar file.

NOTE: The biggest difference with EJB 2.0 is that with the EJB 3.0 release the deployment descriptor becomes optional.

ejb-jar File Requirements

The ejb-jar file must contain, either by inclusion or by reference, the class files of each enterprise bean as follows:

- The enterprise bean class.
- The enterprise bean business interfaces, web service endpoint interfaces, and home and component interfaces.
- Interceptor classes.
- The primary key class if the bean is an entity bean.

We say that a jar file contains a second file "by reference" if the second file is named in the Class-Path attribute in the Manifest file of the referencing jar file or is contained (either by inclusion or by reference) in another jar file that is named in the Class-Path attribute in the Manifest file of the referencing jar file.

The ejb-jar file must also contain, either by inclusion or by reference, the class files for all the classes and interfaces that each enterprise bean class and the home interfaces, component interfaces, and/or web service endpoints depend on, except Java EE and J2SE classes. This includes their superclasses and superinterfaces, dependent classes, and the classes and interfaces used as method parameters, results, and exceptions.

The Application Assembler must not package the stubs of the EJBHome and EJBObject interfaces in the ejb-jar file. This includes the stubs for the enterprise beans whose implementations are provided in the ejb-jar file as well as the referenced enterprise beans. Generating the stubs is the responsibility of the container. The stubs are typically generated by the Container Provider's deployment tools for each class that extends the EJBHome or EJBObject interfaces, or they may be generated by the container at runtime.

●Describe the purposes and uses of annotations and deployment descriptors, including how the two mechanisms interact, how overriding is handled, and how these mechanisms function at the class, method, and field levels.

One of the key enabling technologies introduced by J2SE 5.0 is the program annotation facility defined by JSR-175. This facility allows developers to annotate program elements in Java programming language source files to control the behavior and deployment of an application.

Metadata annotations are a key element in the simplification of the development of EJB 3.0 applications.

Annotations available for:

- Defining Web Services
- Defining Enterprise Beans
- Calling out EJB Lifecycle callbacks or Interceptors

- Dependency Injection
- Almost anything that you used to previously have a Deployment Descriptor entry for
- Transaction Attributes
- Security

Annotations: Pros

- Easier to write than .xml
- Easier to understand than .xml
- Fewer files to maintain

Annotations: Cons

- Only visible in source-code
- Can't express all Java Platform, EE 5 metadata
- Blurs lines between Java EE platform roles (e.g., Component Provider vs. Application Assembler)

Best Uses for Annotations

- Metadata that does not change often
- Metadata tied to component development time

Examples:

- Structural metadata
  - @Stateless
  - @WebService
  - @Entity
- Environment dependencies
  - @EJB
  - @Resource
  - @PersistenceContext
- Callbacks
  - @PostConstruct
  - @Timeout
  - @Remove

Best Uses for Deployment Descriptors

- Overriding annotations and defaults
- Application assembler metadata
  - EJB Security Method Permissions

- Typically not known until assembly/deployment time
- Likely to change
- Independent of business logic

- Dependency linking info
  - e.g. cross-module ejb-link

- Metadata that has no corresponding annotation
  - e.g. EJB Default interceptors, EJB 2.x Entity Beans, Message Destinations

NOTE. Although it is not anticipated as a typical use case, it is possible for the application developer to combine the use of metadata annotations and deployment descriptors in the design of an application. When such a combination is used, the rules for the use of deployment descriptors as an overriding mechanism apply.

Guidelines for .xml Overriding

- Use sparingly

Overuse can make app difficult to understand/maintain

- Good with linking metadata

e.g., ejb-link, persistence-unit-name

- Keep in mind not all annotations are overridable

e.g., Session bean type (Stateful vs. Stateless) can't be overridden

Don't Forget Spec-Defined Defaults

Default values can be easier than annotations and .xml

- EJB based transaction demarcation type

Default: Container managed transaction

- EJB based transaction attribute

Default: TX_REQUIRED

- Environment annotation name()

Default: Derived from class and field/method

Component Dependency Annotations

- For declaring environment dependencies
- For eliminating JNDI lookups

ejb-ref, resource-ref, service-ref, etc.

- Available on container-managed classes

Enterprise beans and interceptors, Servlets, Filters, ServletListeners, JSF Managed Beans, Web service endpoints and Handlers

NOT FOR JSP, JSP beans, or other plain Java classes that are not available at deployment

- Declared at Class, Field, or Method level
- Field/method level dependencies injected at runtime

Annotation example:

```
@Resource(name="Foo") private DataSource ds;
```

Deployment Descriptor example:

```
<resource-ref>
	<res-ref-name>Foo</res-ref-name>
	<res-ref-type>javax.sql.DataSource</res-ref-type>
	<injection-target>
		<injection-target-class>com.acme.FooBean</injection-target-class>
		<injection-target-name>ds</injection-target-name>
	</injection-target>
</resource-ref>
```

Dependency Injection

- Available for Fields as well as Methods
- Available anywhere on the inheritence hierarchy

Follows normal language overriding rules

- @PostConstruct annotation available to provide initialization after injection

Injected Field/Method Access Modifiers

Spec allows public, package, protected, private

Which should you use?

- Private is best
- Injected data is typically internal to the .class

Exception: Overriding of environment dependencies within a class hierarchy

- Use sparingly
- Tightly couples classes
- Harder to understand/maintain

Which Is Best: Field, Method, or Class-level?

1. Field-level: Easiest
    - e.g., @EJB Converter converter
    - Takes fewest characters to declare
    - Supports injection

2 Method-level

- Useful for logic tied to a specific dependency injection
- But Field-level + @PostConstruct would work too

3 Class-level

- Useful for dependency declaration WITHOUT injection
- Declare environment dependency for use by non container-managed classes

Class-Level Dependency Example:

FooBean.java

```
@EJB(name="ejb/bar", beanInterface=Bar.class)

public class FooBean implements Foo {
	...
}
```

Utility.java

```
Bar bar = (Bar) context.lookup("java:comp/env/ejb/bar");
```

Another Class-Level Dependency Example:

Scenario: Stateful Session Bean creation

- EJB 3.0 SFSBs are created as a side-effect of injection/lookup
- Common need : many instances of same SFSB
- Using Field-based dependency + injection:

```
@EJB Cart cart1;
@EJB Cart cart2;
@EJB Cart cart3;
```

is too static.

Alternative: Class-level dependency + lookup

```
@EJB(name="ejb/Cart", beanInterface=Cart.class)

public class CartClient {
	...
	Cart[] carts = new Cart[numCarts];
	for(int i = 0; i < carts.length; i++) {
		carts[i] = (Cart)
		ctx.lookup("java:comp/env/ejb/Cart");
	}
}
```

Concurrency and Injection

Injection does not solve concurrency issues

If an object obtained through lookup() is non-sharable, it's non-sharable when injected

Be careful with Servlet instance injection:

```
public class MyServlet ... {
      private @EJB StatelessEJB stateless; // OK
      private @EJB StatefulEJB stateful;   // dangerous!
}
```

Most common issues: Stateful Session Beans, PersistenceContexts

Recommended alternative: lookup() and store in HttpSession:

```
@PersistenceContext(name="pc", type=EntityManager.class)

public class MyServlet ... {
      EntityManager em = ctx.lookup("java:comp/env/pc");
      httpSession.setAttribute("entityManager", em);
}
```

Performance and Injection

Use of injection is unlikely to cause performance issues

Injection is essentially a ctx.lookup() + one reflective operation

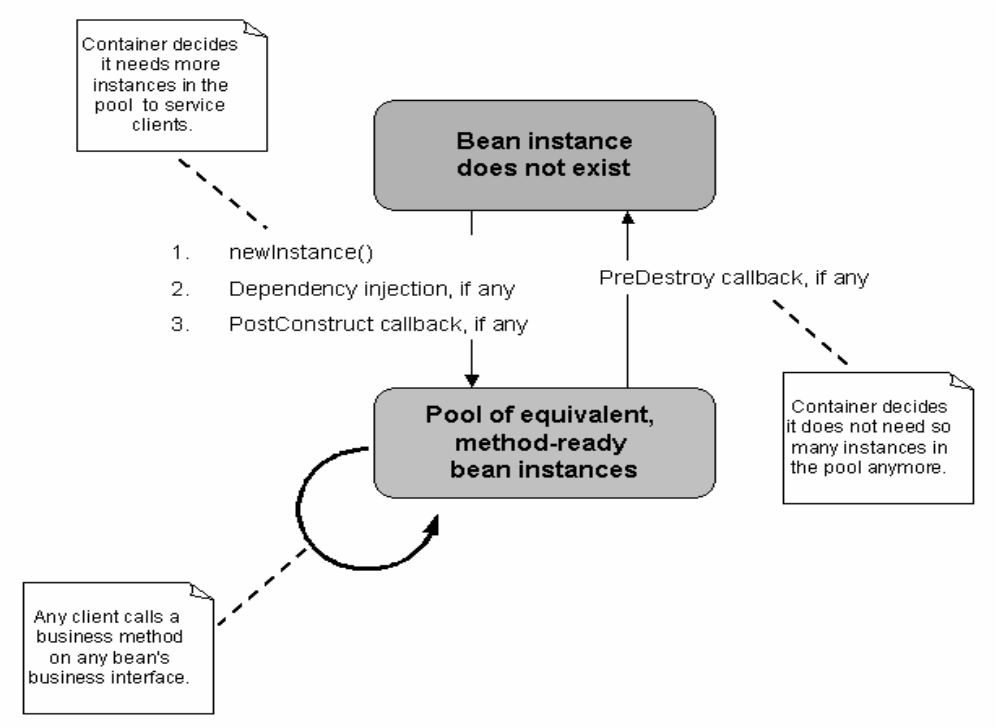
Injection occurs at instance creation-time

- Overhead of injection typically small compared to instance creation itself
- Most lookups() resolved locally within server
- Instances are typically long-lived/reused

Chapter 2. General EJB 3.0 Enterprise Bean Knowledge

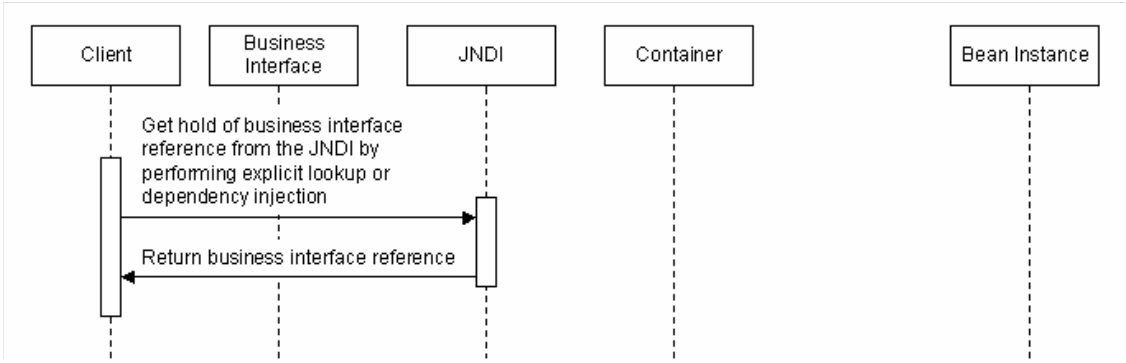
Identify correct and incorrect statements or examples about the lifecycle of all 3.0 Enterprise Bean instances, including the use of the @PostConstruct and @PreDestroy callback methods.

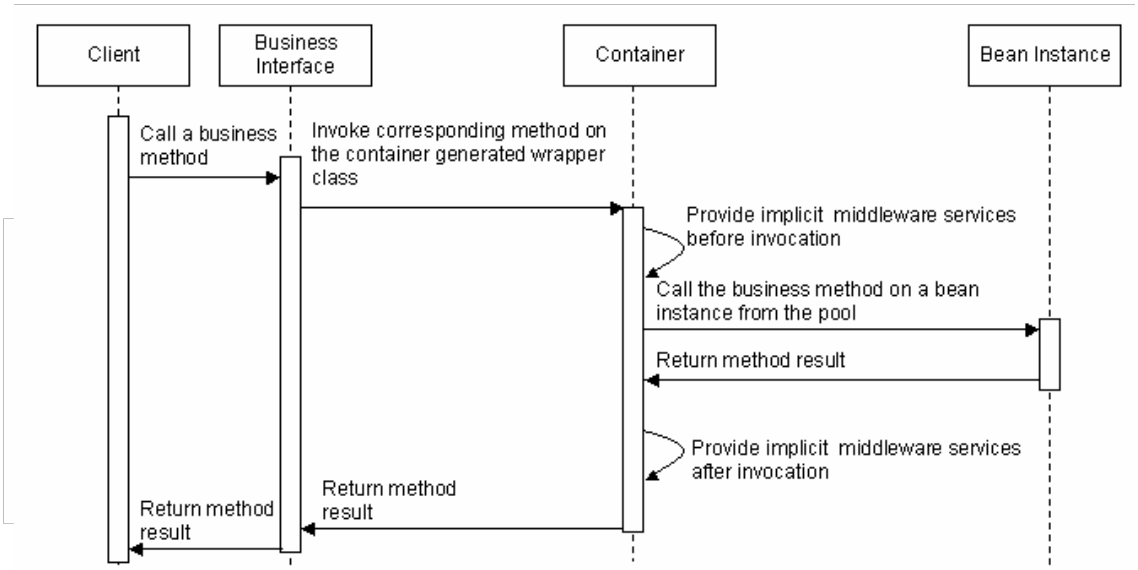
Lifecycle Callbacks for Stateless Session Beans



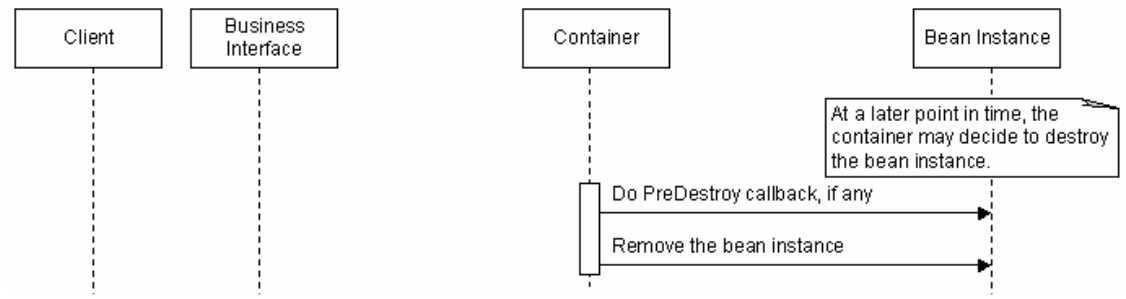
Addition of a new instance to the Stateless Session Bean Pool:

Retrieving reference to Stateless Session Bean business interface from JNDI:





Servicing a business method:



Removing the Stateless Session Bean instance:

The following lifecycle event callbacks are supported for stateless session beans:

- **PostConstruct**

PostConstruct callbacks occur after any dependency injection has been performed by the container and before the first business method invocation on the bean.

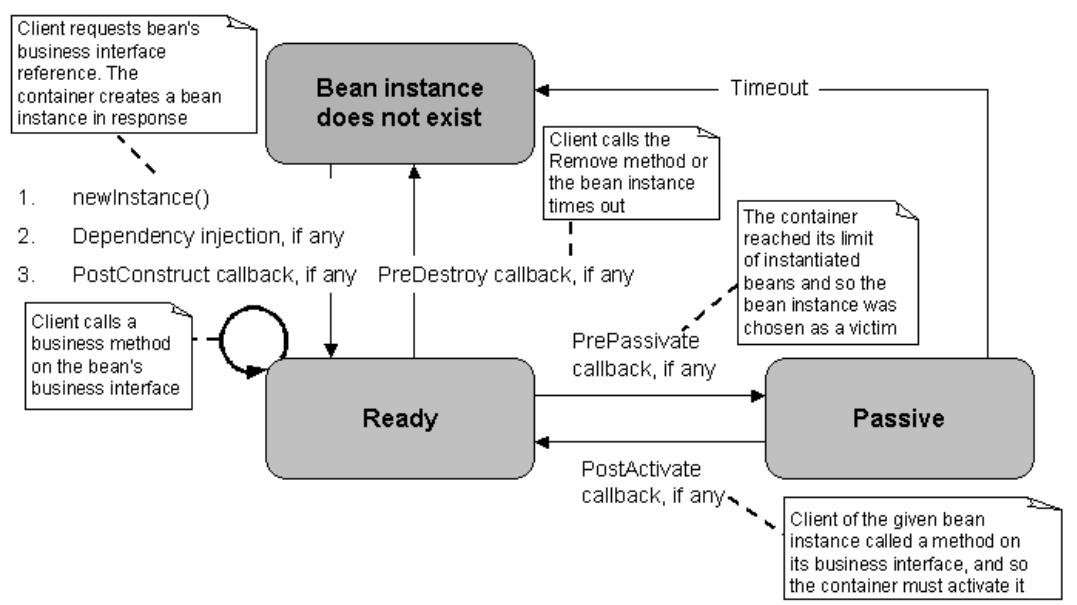
PostConstruct methods are invoked in an unspecified transaction context and security context.

- **PreDestroy**

PreDestroy callbacks occur at the time the bean instance is destroyed.

PreDestroy methods execute in an unspecified transaction and security context.

NOTE. PostActivate and PrePassivate callbacks, if specified, are ignored for stateless session beans.

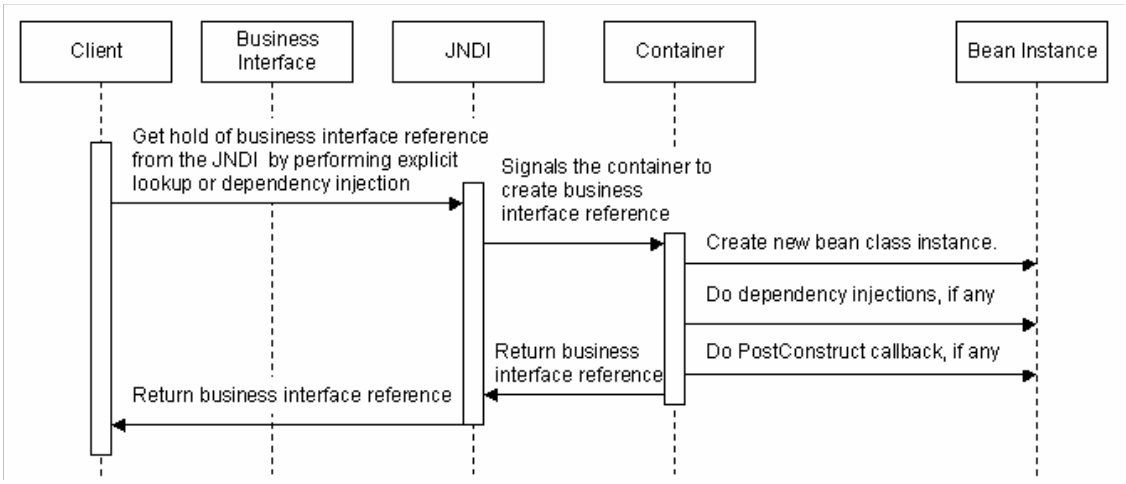


Lifecycle Callbacks for Stateful Session Beans

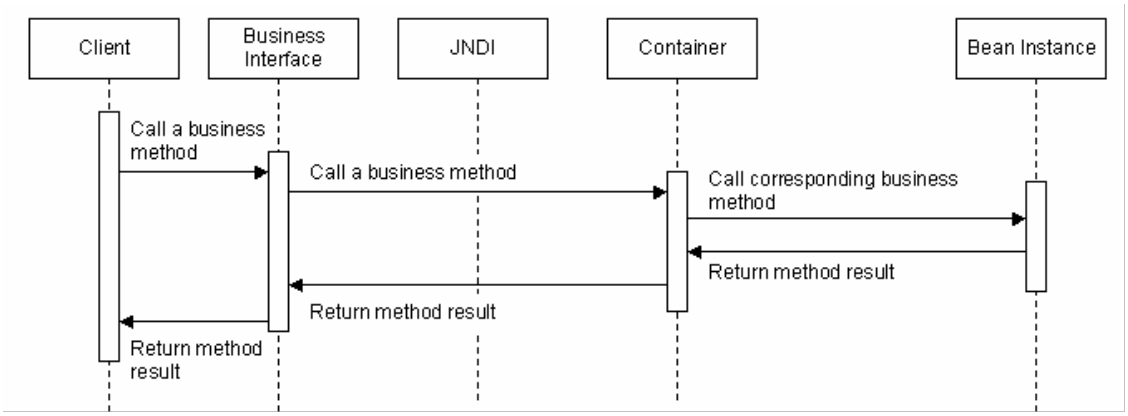
The life cycle of a stateful session bean (does not implement javax.ejb.SessionSynchronization):

Stateful session beans support callbacks for the following lifecycle events: construction, destruction, activation, and passivation.

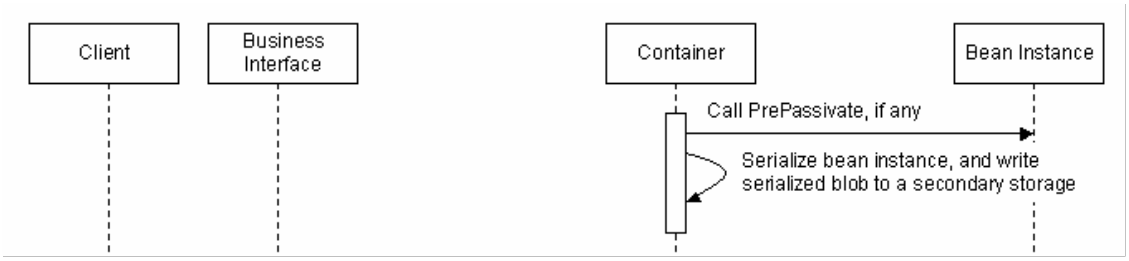
Retrieving reference to Stateful Session Bean business interface from JNDI:

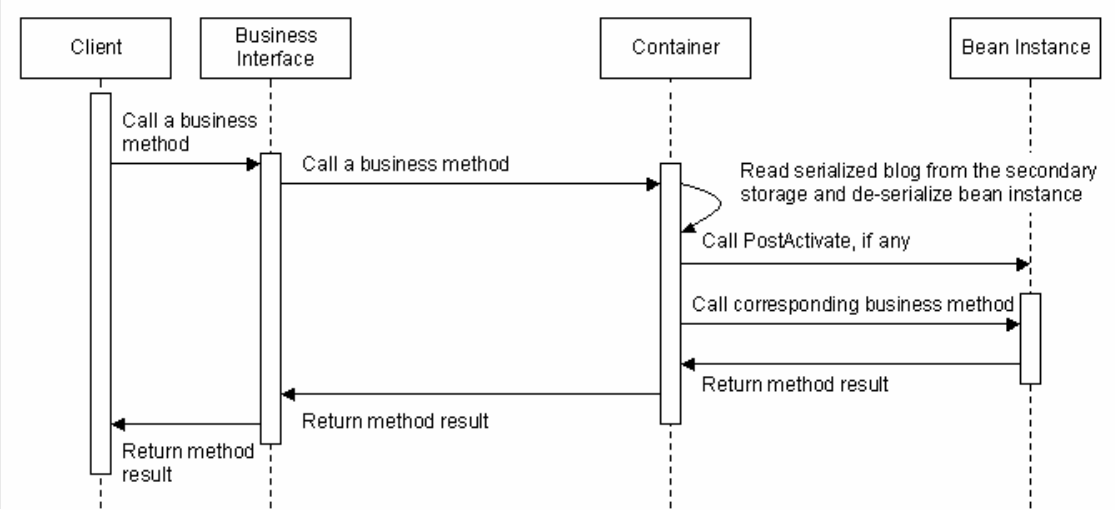


Servicing a business method:



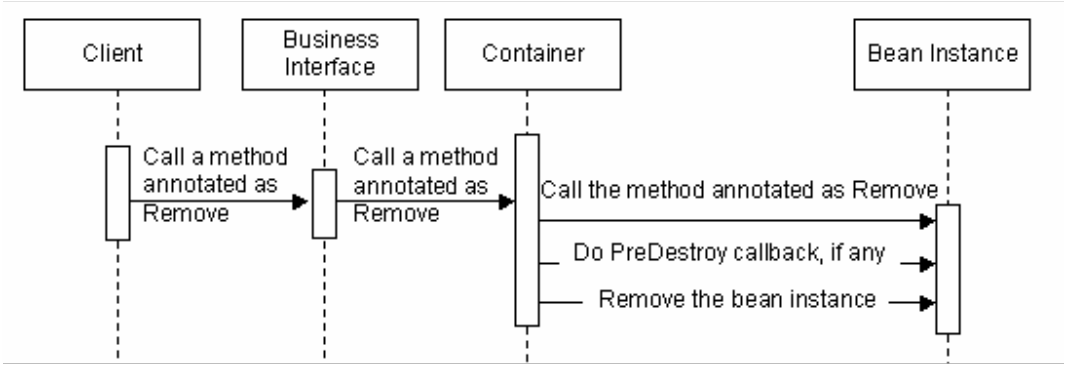
Passivating a Stateful Session Bean:





Activating a Stateful Session Bean:

Servicing a @Remove method:



The lifecycle event callbacks are the following. They may be defined on the bean class or an interceptor class for the bean.

- **PostConstruct**

PostConstruct methods are invoked on the newly constructed instance, after any dependency injection has been performed by the container and before the first business method is invoked on the bean.

PostConstruct methods are invoked in an unspecified transaction and security context.

- **PreDestroy**

PreDestroy methods execute after any method annotated with the Remove annotation has completed.

PreDestroy methods are invoked in an unspecified transaction and security context.

- **PostActivate**

This notification signals the instance it has just been reactivated.

Its purpose is to allow stateful session beans to maintain those resources that need to be reopened during an instance's activation.

- **PrePassivate**

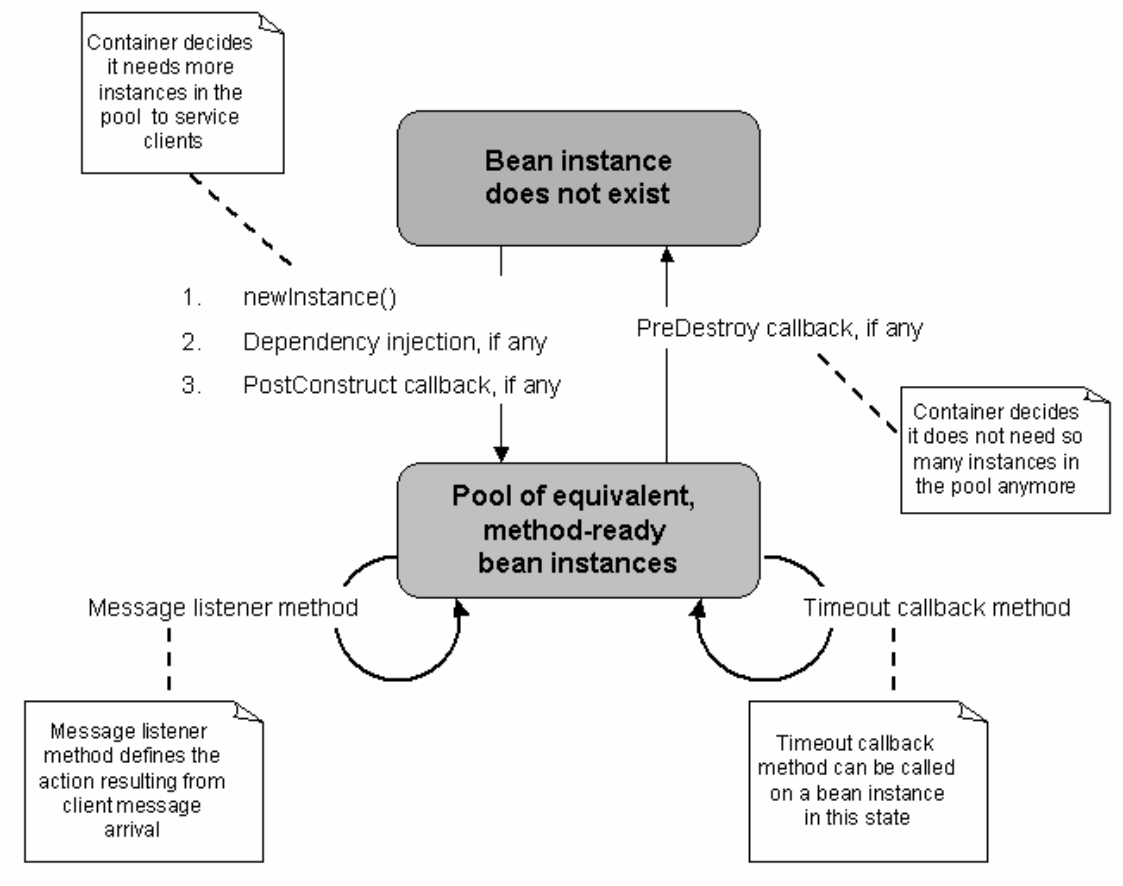
This notification signals the intent of the container to passivate the instance.

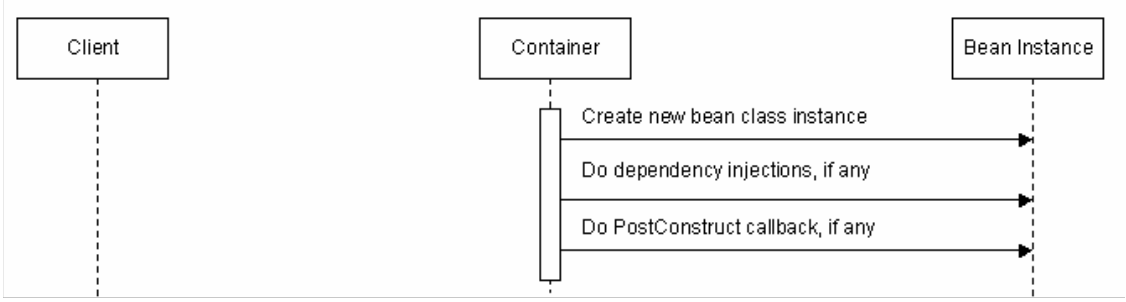
Its purpose is to allow stateful session beans to maintain those open resources that need to be closed prior to an instance's passivation.

NOTE: The callbacks PreConstruct, PostDestroy, PreActivate, and PostPassivate were not introduced because there did not seem to be use cases that justified their introduction.

Lifecycle Callbacks for Message-Driven Beans

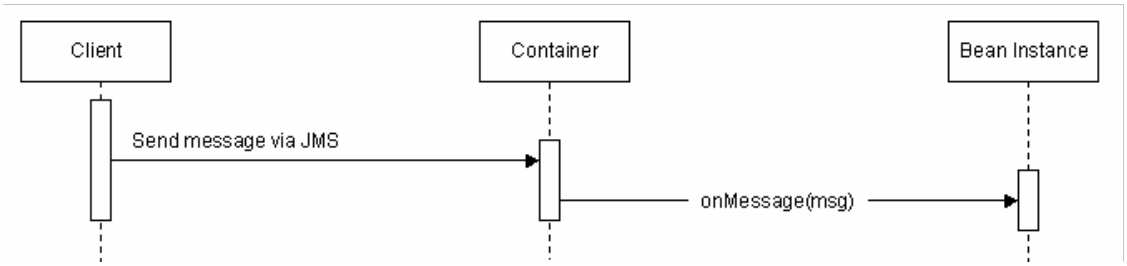
The life cycle of a Message Driven Bean:

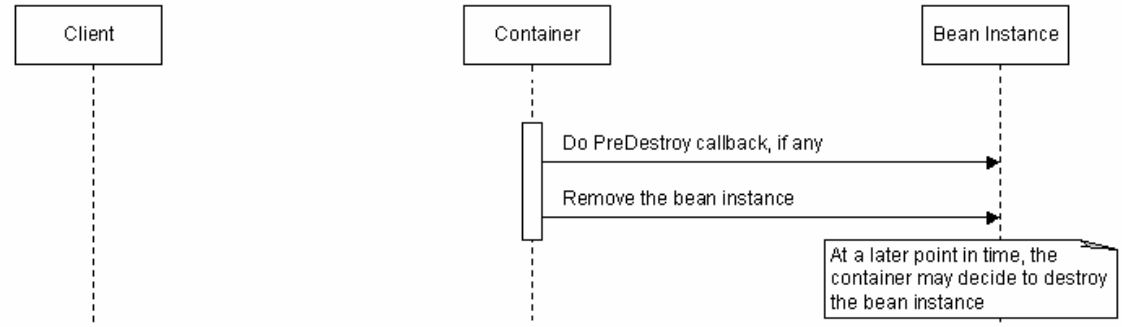




Addition of a new instance to the Message-Driven Bean pool:

Servicing an onMessage(...) method:





Removing the Message-Driven Bean instance:

The following lifecycle event callbacks are supported for message-driven beans:

- **PostConstruct**

PostConstruct callbacks occur before the first message listener method invocation on the bean. This is at a point after which any dependency injection has been performed by the container.

PostConstruct callback methods execute in an unspecified transaction and security context.

- **PreDestroy**

PreDestroy callbacks occur at the time the bean is removed from the pool or destroyed.

PreDestroy callback methods execute in an unspecified transaction and security context.

NOTE: PostActivate and PrePassivate callbacks, if specified, are ignored for message-driven beans.

Identify correct and incorrect statements or examples about interceptors, including implementing an interceptor class, the lifecycle of interceptor instances, @AroundInvoke methods, invocation order, exception handling, lifecycle callback methods, default and method level interceptors, and specifying interceptors in the deployment descriptor.

- [EJB_3.0_CORE] 12.1; 12.2; 12.3; 12.3.1; 12.3.2; 12.4; 12.6; 12.7;

12.8; 12.8.1; 12.8.2; 12.8.2.1

Interceptors are used to interpose on business method invocations and lifecycle events that occur on an enterprise bean instance.

An interceptor method may be defined on an enterprise bean class or on an interceptor class associated with the bean. An interceptor class is a class - distinct from the bean class itself - whose methods are invoked in response to business method invocations and/or lifecycle events on the bean.

Any number of interceptor classes may be defined for a bean class.

It is possible to carry state across multiple interceptor method invocations for a single business method invocation or lifecycle callback event for a bean in the context data of the InvocationContext object.

An interceptor class must have a public no-arg constructor.

The programming restrictions that apply to enterprise bean components apply to interceptors as well. See here: Identify correct and incorrect statements or examples about EJB programming restrictions.

Interceptor Life Cycle (lifecycle of interceptor instances)

The lifecycle of an interceptor instance is the same as that of the bean instance with which it is associated. When the bean instance is created, interceptor instances are created for each interceptor class defined for the bean. These interceptor instances are destroyed when the bean instance is removed. In the case of interceptors associated with stateful session beans, the interceptor instances are passivated upon bean instance passivation, and activated when the bean instance is activated.

Both the interceptor instance and the bean instance are created or activated before any of the respective PostConstruct or PostActivate callbacks are invoked. Any PreDestroy and PrePassivate callbacks are invoked before the respective destruction or passivation of either the bean instance or interceptor instance.

An interceptor instance may hold state. An interceptor instance may be the target of dependency injection. Dependency injection is performed when the interceptor instance is created, using the naming context of the associated enterprise bean. The PostConstruct interceptor callback method is invoked after this dependency injection has taken place on both the interceptor instances and the bean instance.

Interceptors can invoke JNDI, JDBC, JMS, other enterprise beans, and the EntityManager.

The interceptors for a bean share the enterprise naming context of the bean for whose methods and lifecycle events they are invoked. Annotations and/or XML deployment descriptor elements for dependency injection or for direct JNDI lookup refer to this shared naming context.

The EJBContext object may be injected into an interceptor class. The interceptor may use the lookup method of the EJBContext interface to access the bean's JNDI naming context.

The use of an extended persistence context is only supported for interceptors that are associated

with stateful session beans.

Business Method Interceptors (@AroundInvoke methods)

Interceptor methods may be defined for business methods of sessions beans and for the message listener methods of message-driven beans. Business method interceptor methods are denoted by the AroundInvoke annotation or around-invoke deployment descriptor element.

AroundInvoke methods may be defined on superclasses of the bean class or interceptor classes. However, only one AroundInvoke method may be present on a given class. An AroundInvoke method cannot be a business method of the bean.

AroundInvoke methods can have public, private, protected, or package level access. An AroundInvoke method must not be declared as final or static.

AroundInvoke methods have the following signature:

Object <METHOD>(InvocationContext) throws Exception

An AroundInvoke method can invoke any component or resource that a business method can.

Business method interceptor method invocations occur within the same transaction and security context as the business method for which they are invoked.

Business method interceptor methods may be defined to apply to business methods individually, rather than to all the business methods of the bean class.

Multiple Business Method Interceptor Methods (invocation order)

If multiple interceptor methods are defined for a bean, the following rules governing their invocation order apply. The deployment descriptor may be used to override the interceptor invocation order specified in annotations.

1. Default interceptors, if any, are invoked first. Default interceptors can only be specified in the deployment descriptor. Default interceptors are invoked in the order of their specification in the deployment descriptor.
2. If there are any interceptor classes defined on the bean class, the interceptor methods defined by those interceptor classes are invoked before any interceptor methods defined on the bean class itself.
3. The AroundInvoke methods defined on those interceptor classes are invoked in the same order as the specification of the interceptor classes in the Interceptors annotation.
4. If an interceptor class itself has superclasses, the interceptor methods defined by the interceptor class's superclasses are invoked before the interceptor method defined by the interceptor class, most general superclass first.
5. After the interceptor methods defined on interceptor classes have been invoked, then, in order:

   If any method-level interceptors are defined for the business method that is to be

invoked, the AroundInvoke methods defined on those interceptor classes are invoked in the same order as the specification of those interceptor classes in the Interceptors annotation applied to that business method. ZZZZ (See Section 12.7 for the description of method-level interceptors).

If a bean class has superclasses, any AroundInvoke methods defined on those superclasses are invoked, most general superclass first.

The AroundInvoke method, if any, on the bean class itself is invoked.

1 If an AroundInvoke method is overridden by another method (regardless of whether that method is itself an AroundInvoke method), it will not be invoked.

The InvocationContext object provides metadata that enables interceptor methods to control the behavior of the invocation chain, including whether the next method in the chain is invoked and the values of its parameters and result.

Exceptions (exception handling)

Business method interceptor methods may throw runtime exceptions or application exceptions that are allowed in the throws clause of the business method.

AroundInvoke methods are allowed to catch and suppress exceptions and recover by calling proceed(). AroundInvoke methods are allowed to throw runtime exceptions or any checked exceptions that the business method allows within its throws clause.

AroundInvoke methods run in the same Java call stack as the bean business method. InvocationContext.proceed() will throw the same exception as any thrown by the business method unless an interceptor further down the Java call stack has caught it and thrown a different exception. Exceptions and initialization and/or cleanup operations should typically be handled in try/catch/finally blocks around the proceed() method.

AroundInvoke methods can mark the transaction for rollback by throwing a runtime exception or by calling the EJBContext setRollbackOnly() method. AroundInvoke methods may cause this rollback before or after InvocationContext.proceed() is called.

If a system exception escapes the interceptor chain the bean instance and any associated interceptor instances are discarded. The PreDestroy callbacks are not invoked in this case: the interceptor methods in the chain should perform any necessary clean-up operations as the interceptor chain unwinds.

Interceptors for LifeCycle Event Callbacks (lifecycle callback methods)

Lifecycle callback interceptor methods may be defined for session beans and message driven beans.

Interceptor methods for lifecycle event callbacks can be defined on an interceptor class and/or directly on the bean class. The PostConstruct, PreDestroy, PostActivate, and PrePassivate annotations are used to define an interceptor method for a lifecycle callback event. If the deployment descriptor is used to define interceptors, the post-construct, pre-destroy, post-activate, and pre-passivate elements are used.

Lifecycle callback interceptor methods and business method interceptor methods may be defined on the same interceptor class.

Lifecycle callback interceptor methods are invoked in an unspecified transaction and security context.

Lifecycle callback interceptor methods may be defined on superclasses of the bean class or interceptor classes. However, a given class may not have more than one lifecycle callback interceptor method for the same lifecycle event. Any subset or combination of lifecycle callback annotations may be specified on a given class.

A single lifecycle callback interceptor method may be used to interpose on multiple callback events (e.g., PostConstruct and PostActivate).

Lifecycle callback interceptor methods defined on an interceptor class have the following signature:

void <METHOD> (InvocationContext)

Lifecycle callback interceptor methods defined on a bean class have the following signature:

void <METHOD>()

Lifecycle callback interceptor methods can have public, private, protected, or package level access. A lifecycle callback interceptor method must not be declared as final or static.

Examples:

```
@Stateful
public class ShoppingCartBean implements ShoppingCart {
	private float total;
	private Vector productCodes;
	public int someShoppingMethod() {...};
	...
	@PreDestroy
	void endShoppingCart() {...};
}
public class MyInterceptor {

	@PostConstruct
	public void someMethod(InvocationContext ctx) {
		...
		ctx.proceed();
		...
	}

	@PreDestroy
	public void anotherMethod(InvocationContext ctx) {
		...
		ctx.proceed();
		...
	}
}
```

Default Interceptors

Default interceptors may be defined to apply to all the components within the ejb-jar. The

deployment descriptor is used to define default interceptors and their relative ordering.

Default interceptors are automatically applied to all components defined in the ejb-jar. The ExcludeDefaultInterceptors annotation or exclude-default-interceptors deployment descriptor element is used to exclude the invocation of default interceptors for a bean.

The default interceptors are invoked BEFORE ANY OTHER interceptors for a bean. The interceptor-order deployment descriptor element may be used to specify alternative orderings.

Method-level Interceptors

A business method interceptor method may be defined to apply to a specific business method invocation, rather than to all of the business methods of the bean class.

NOTE. Method-level interceptor are used to specify business method interceptor methods. If an interceptor class that is used as a method-level interceptor defines lifecycle callback interceptor methods, those lifecycle callback interceptor methods are not invoked.

Method-specific business method interceptors can be defined by applying the Interceptors annotation to the method for which the interceptors are to be invoked, or by means of the interceptor-binding deployment descriptor element. If more than one method-level interceptor is defined for a business method, the interceptors are invoked in the order specified. Method-level business method interceptors are invoked in addition to any default interceptors and interceptors defined for the bean class (and its superclasses). The deployment descriptor may be used to override this ordering.

The same interceptor may be applied to more than one method of the bean class:

```
@Stateless
public class MyBean ... {
	public void notIntercepted() {}

	  @Interceptors(org.acme.MyInterceptor.class)
	public void someMethod() {
		...
	}

	  @Interceptors(org.acme.MyInterceptor.class)
	public void anotherMethod() {
		...
	}
}
```

The applicability of a method-level interceptor to more than one business method of a bean does not affect the relationship between the interceptor instance and the bean class - only a SINGLE instance of the interceptor class is created PER BEAN instance.

The ExcludeDefaultInterceptors annotation or exclude-default-interceptors deployment descriptor element, when applied to a business method, is used to exclude the invocation of default interceptors for that method. The ExcludeClassInterceptors annotation or exclude-class-interceptors deployment descriptor element is used similarly to exclude the invocation of the class-level interceptors.

In the following example, if there are no default interceptors, only the interceptor MyInterceptor will be invoked when someMethod is called:

```
@Stateless
@Interceptors(org.acme.AnotherInterceptor.class)
public class MyBean ... {
	...
	@Interceptors(org.acme.MyInterceptor.class)
	@ExcludeClassInterceptors
	public void someMethod() {
		...
	}
}
```

If default interceptors have also been defined for the bean class, they can be excluded for the specific method by applying the ExcludeDefaultInterceptors annotation on the method:

```
@Stateless
@Interceptors(org.acme.AnotherInterceptor.class)
public class MyBean ... {
	...
	@ExcludeDefaultInterceptors
	@ExcludeClassInterceptors
	@Interceptors(org.acme.MyInterceptor.class)
	public void someMethod() {
		...
	}
}
```

Specification of Interceptors in the Deployment Descriptor

The deployment descriptor can be used as an alternative to metadata annotations to specify interceptors and their binding to enterprise beans or to override the invocation order of interceptors as specified in annotations.

The interceptor deployment descriptor element is used to specify the interceptor methods of an interceptor class. The interceptor methods are specified by using the around-invoke, pre-construct, post-destroy, pre-passivate, and post-activate elements.

At most ONE method of a given interceptor class can be designated as an around-invoke method, a pre-construct method, a post-destroy method, a pre-passivate method, or a post-activate method, regardless of whether the deployment descriptor is used to define interceptors or whether some combination of annotations and deployment descriptor elements is used.

The interceptor-binding element is used to specify the binding of interceptor classes to enterprise beans and their business methods. The subelements of the interceptor-binding element are as follows:

- The ejb-name element must be the name of one of the enterprise beans contained in the ejb-jar or the wildcard value "*" (which is used to define interceptors that are bound to all beans in the ejb-jar).
- The interceptor-class element specifies the interceptor class. The interceptor-order

element is used as an optional alternative to specify a total ordering over the interceptors defined for the given level and above.

- The exclude-default-interceptors and exclude-class-interceptors elements specify that default interceptors and class interceptors, respectively, are not to be applied to a bean class and/or business method.
- The method-name element specifies the method name for a method-level interceptor; and the optional method-params elements identify a single method among multiple methods with an overloaded method name.

Interceptors bound to all classes using the wildcard syntax "*" are default interceptors for the components in the ejb-jar. In addition, interceptors may be bound at the level of the bean class (class-level interceptors) or business methods of the class (method-level interceptors).

The binding of interceptors to classes is additive. If interceptors are bound at the class-level and/or default-level as well as at the method-level, both class-level and/or default-level as well as method-level interceptors will apply. The deployment descriptor may be used to augment the interceptors and interceptor methods defined by means of annotations. When the deployment descriptor is used to augment the interceptors specified in annotations, the interceptor methods specified in the deployment descriptor will be invoked after those specified in annotations. The interceptor-order deployment descriptor element may be used to override this ordering.

The exclude-default-interceptors element disables default interceptors for the level at which it is specified and lower. That is, exclude-default-interceptors when applied at the class-level disables the application of default-interceptors for all methods of the class. The exclude-class-interceptors element applied to a method, disables the application of class-level interceptors for the given method. Explicitly listing an excluded higher-level interceptor at a lower level causes it to be applied at that level and below.

Examples of the usage of the interceptor-binding syntax are given below.

Style 1: The following interceptors apply to all components in the ejb-jar as default interceptors. They will be invoked in the order specified.

```
<interceptor-binding>
      <ejb-name>*</ejb-name>
      <interceptor-class>org.acme.MyDefaultIC</interceptor-class>
      <interceptor-class>org.acme.MyDefaultIC2</interceptor-class>
</interceptor-binding>
```

Style 2: The following interceptors are the class-level interceptors of the EmployeeService bean. They will be invoked in the order specified after any default interceptors.

```
<interceptor-binding>
      <ejb-name>EmployeeService</ejb-name>
      <interceptor-class>org.acme.MyIC</interceptor-class>
      <interceptor-class>org.acme.MyIC2</interceptor-class>
</interceptor-binding>
```

Style 3: The following interceptors apply to all the myMethod methods of the EmployeeService

bean. They will be invoked in the order specified AFTER any default interceptors and class-level interceptors.

```
<interceptor-binding>
      <ejb-name>EmployeeService</ejb-name>
      <interceptor-class>org.acme.MyIC</interceptor-class>
      <interceptor-class>org.acme.MyIC2</interceptor-class>
      <method-name>myMethod</method-name>
</interceptor-binding>
```

Style 4: The following interceptor element refers to the myMethod(String firstName, String LastName) method of the EmployeeService bean.

```
<interceptor-binding>
      <ejb-name>EmployeeService</ejb-name>
      <interceptor-class>org.acme.MyIC</interceptor-class>
      <method-name>myMethod</method-name>
      <method-params>
            <method-param>java.lang.String</method-param>
            <method-param>java.lang.String</method-param>
      </method-params>
</interceptor-binding>
```

The following example illustrates the total ordering of interceptors using the interceptor-order element:

```
<interceptor-binding>
      <ejb-name>EmployeeService</ejb-name>
      <interceptor-order>
            <interceptor-class>org.acme.MyIC</interceptor-class>
            <interceptor-class>org.acme.MyDefaultIC</interceptor-class>
            <interceptor-class>org.acme.MyDefaultIC2</interceptor-class>
            <interceptor-class>org.acme.MyIC2</interceptor-class>
      </interceptor-order>
</interceptor-binding>
```

Identify correct and incorrect statements or examples about how enterprise beans declare dependencies on external resources using JNDI or dependency injection, including the general rules for using JNDI, annotations and/or deployment descriptors, EJB references, connection factories, resource environment entries, and persistence context and persistence unit references.

Enterprise Bean's Environment as a JNDI Naming Context (declare dependencies on external resources)

The enterprise bean's environment is a mechanism that allows customization of the enterprise bean's business logic during deployment or assembly. The enterprise bean's environment allows the enterprise bean to be customized without the need to access or change the enterprise bean's source code.

Annotations and deployment descriptors are the main vehicles for conveying access information

to the application assembler and deployer about beans' requirements for customization of business logic and access to external information.

The container implements the enterprise bean’s environment, and provides it as a JNDI naming context. The enterprise bean's environment is used as follows:

1. The enterprise bean makes use of entries from the environment. Entries from the environment may be injected by the container into the bean's fields or methods, or the methods of the bean may access the environment using the EJBContext lookup method or the JNDI interfaces. The Bean Provider declares in Java language metadata annotations or in the deployment descriptor all the environment entries that the enterprise bean expects to be provided in its environment at runtime.
2. The container provides an implementation of the JNDI naming context that stores the enterprise bean environment. The container also provides the tools that allow the Deployer to create and manage the environment of each enterprise bean.
3. The Deployer uses the tools provided by the container to create and initialize the environment entries that are declared by means of the enterprise bean's annotations or deployment descriptor. The Deployer can set and modify the values of the environment entries.
4. The container injects entries from the environment into the enterprise bean's fields or methods as specified by the bean's metadata annotations or the deployment descriptor.
5. The container makes the environment naming context available to the enterprise bean instances at runtime. The enterprise bean's instances can use the EJBContext lookup method or the JNDI interfaces to obtain the values of the environment entries.

Sharing of Environment Entries

Each enterprise bean defines its own set of environment entries. All instances of an enterprise bean share the same environment entries; the environment entries are not shared with other enterprise beans. Enterprise bean instances are not allowed to modify the bean's environment at runtime.

In general, lookups of objects in the JNDI java: namespace are required to return a NEW INSTANCE of the requested object every time. Exceptions are allowed for the following:

- The container knows the object is immutable (for example, objects of type java.lang.String), or knows that the application can't change the state of the object.
- The object is defined to be a singleton, such that only one instance of the object may exist in the JVM.
- The name used for the lookup is defined to return an instance of the object that might be shared. The name java:comp/ORB is such a name.

In these cases, a shared instance of the object may be returned. In all other cases, a NEW INSTANCE of the requested object must be returned on each lookup. Note that, in the case of resource adapter connection objects, it is the resource adapter's ManagedConnectionFactory implementation that is responsible for satisfying this requirement.

Each injection of an object corresponds to a JNDI lookup. Whether a new instance of the

requested object is injected, or whether a shared instance is injected, is determined by the rules described above.

Annotations for Environment Entries

A field or method of a bean class may be annotated to request that an entry from the bean's environment be injected. Any of the types of resources or other environment entries described in this chapter may be injected. Injection may also be requested using entries in the deployment descriptor corresponding to each of these resource types. The field or method MAY have any access qualifier (public, private, etc.) but MUST NOT be static.

- A field of the bean class may be the target of injection. The field MUST NOT be final. By default, the name of the field is combined with the name of the class in which the annotation is used and is used directly as the name in the bean's naming context. For example, a field named myDatabase in the class MySessionBean in the package com.acme.example would correspond to the JNDI name java:comp/env/com.acme.example.MySessionBean/myDatabase. The annotation also allows the JNDI name to be specified explicitly.
- Environment entries may also be injected into the bean through bean methods that follow the naming conventions for JavaBeans properties. The annotation is applied to the set method for the property, which is the method that is called to inject the environment entry. The JavaBeans property name (not the method name) is used as the default JNDI name. For example, a method named setMyDatabase in the same MySessionBean class would correspond to the JNDI name java:comp/env/com.example.MySessionBean/myDatabase.
- When a deployment descriptor entry is used to specify injection, the JNDI name and the instance variable name or property name are both specified explicitly. Note that the JNDI name is ALWAYS relative to the java:comp/env naming context.

Each resource may only be injected into a SINGLE field or method of the bean. Requesting injection of the java:comp/env/com.example.MySessionBean/myDatabase resource into both the setMyDatabase method and the myDatabase instance variable is an ERROR. Note, however, that either the field or the method could request injection of a resource of a different (non-default) name. By explicitly specifying the JNDI name of a resource, a single resource may be injected into MULTIPLE fields or methods of multiple classes.

Annotations may also be applied to the bean class itself. These annotations declare an entry in the bean's environment, but do not cause the resource to be injected. Instead, the bean is expected to use the EJBContext lookup method or the methods of the JNDI API to lookup the entry. When the annotation is applied to the bean class, the JNDI name and the environment entry type MUST be explicitly specified.

Annotations may appear on the bean class, or on any superclass. A resource annotation on any class in the inheritance hierarchy defines a resource needed by the bean. However, injection of such resources follows the Java language overriding rules for the visibility of fields and methods. A method definition that overrides a method on a superclass defines the resource, if any, to be injected into that method. An overriding method may request injection of a different resource than is requested by the superclass, or it may request no injection even though the superclass method requests injection.

In addition, fields or methods that are not visible in or are hidden (as opposed to overridden) by a subclass may still request injection. This allows, for example, a private field to be the target of injection and that field to be used in the implementation of the superclass, even though the subclass has no visibility into that field and doesn't know that the implementation of the superclass is using an injected resource. Note that a declaration of a field in a subclass with the same name as a field in a superclass always causes the field in the superclass to be hidden.

Annotations and Deployment Descriptors

Environment entries may be declared by the use of annotations, without need for any deployment descriptor entries. Environment entries may also be declared by deployment descriptor entries, without need for any annotations. The same environment entry may be declared using both an annotation and a deployment descriptor entry. In this case, the information in the deployment descriptor entry may be used to override some of the information provided in the annotation. This approach may be used by an Application Assembler to override information provided by the Bean Provider. Deployment descriptor entries should not be used to request injection of a resource into a field or method that has not been designed for injection.

The following rules apply to how a deployment descriptor entry may override a Resource annotation:

- The relevant deployment descriptor entry is located based on the JNDI name used with the annotation (either defaulted or provided explicitly).
- The type specified in the deployment descriptor must be assignable to the type of the field or property or the type specified in the Resource annotation.
- The description, if specified, overrides the description element of the annotation.
- The injection target, if specified, must name exactly the annotated field or property method.
- The res-sharing-scope element, if specified, overrides the shareable element of the annotation. In general, the Application Assembler or Deployer should NEVER change the value of this element, as doing so is likely to break the application.
- The res-auth element, if specified, overrides the authenticationType element of the annotation. In general, the Application Assembler or Deployer should NEVER change the value of this element, as doing so is likely to break the application.

Restrictions on the overriding of environment entry values depend on the type of environment entry.

EJB References

Injection of EJB References

The Bean Provider uses the EJB annotation to annotate a field or setter property method of the bean class as a target for the injection of an EJB reference. The reference may be to a session bean's business interface or to the local home interface or remote home interface of a session bean or entity bean.

The following example illustrates how an enterprise bean uses the EJB annotation to reference another enterprise bean. The enterprise bean reference will have the name

java:comp/env/com.acme.example.ExampleBean/myCart in the referencing bean's naming context, where ExampleBean is the name of the class of the referencing bean and com.acme.example its package. The target of the reference must be resolved by the Deployer.

```
package com.acme.example;

@Stateless public class ExampleBean implements Example {
    @EJB private ShoppingCart myCart;
    ...
}
```

The following example illustrates use of all portable elements of the EJB annotation. In this case, the enterprise bean reference would have the name java:comp/env/ejb/shopping-cart in the referencing bean's naming context. This reference is linked to a bean named cart1.

```
@EJB(
    name="ejb/shopping-cart",
    beanInterface=ShoppingCart.class,
    beanName="cart1",
    description="The shopping cart for this application"
)
private ShoppingCart myCart;
```

If the ShoppingCart bean were instead written to the **EJB 2.1** client view, the EJB reference would be to the bean's home interface. For example:

```
@EJB(
    name="ejb/shopping-cart",
    beanInterface=ShoppingCartHome.class,
    beanName="cart1",
    description="The shopping cart for this application"
)
private ShoppingCartHome myCartHome;
```

EJB Reference Programming Interfaces

The Bean Provider may use EJB references to locate the business interfaces or home interfaces of other enterprise beans as follows.

- Assign an entry in the enterprise bean's environment to the reference.
- The EJB specification RECOMMENDS, but does not require, that all references to other enterprise beans be organized in the ejb subcontext of the bean's environment (i.e., in the java:comp/env/ejb JNDI context). Note that enterprise bean references declared by means of annotations will not, by default, be in any subcontext.
- Look up the business interface or home interface of the referenced enterprise bean in the enterprise bean's environment using the EJBContext lookup method or the JNDI API.

The following example illustrates how an enterprise bean uses an EJB reference to locate the remote home interface of another enterprise bean using the JNDI APIs:

```
@EJB(name="ejb/EmplRecord", beanInterface=EmployeeRecordHome.class)
@Stateless
public class EmployeeServiceBean implements EmployeeService {
      public void changePhoneNumber(...) {
      // Obtain the default initial JNDI context.
      Context initCtx = new InitialContext();

        // Look up the home interface of the EmployeeRecord
      // enterprise bean in the environment.
      Object result = initCtx.lookup("java:comp/env/ejb/EmplRecord");

        // Convert the result to the proper type.
      EmployeeRecordHome emplRecordHome = (EmployeeRecordHome)
            javax.rmi.PortableRemoteObject.narrow(result, EmployeeRecordHome.class);
      ...
      }
}
```

In the example, the Bean Provider of the EmployeeServiceBean enterprise bean assigned the environment entry ejb/EmplRecord as the EJB reference name to refer to the remote home of another enterprise bean.

Declaration of EJB References in Deployment Descriptor

Although the EJB reference is an entry in the enterprise bean's environment, the Bean Provider must not use a env-entry element to declare it. Instead, the Bean Provider must declare all the EJB references using the ejb-ref and ejb-local-ref elements of the deployment descriptor. This allows the ejb-jar consumer (i.e. Application Assembler or Deployer) to discover all the EJB references used by the enterprise bean. Deployment descriptor entries may also be used to specify injection of an EJB reference into a bean.

Each ejb-ref or ejb-local-ref element describes the interface requirements that the referencing enterprise bean has for the referenced enterprise bean. The ejb-ref element is used for referencing an enterprise bean that is accessed through its remote business interface or remote home and component interfaces. The ejb-local-ref element is used for referencing an enterprise bean that is accessed through its local business interface or local home and component interfaces.

The ejb-ref element contains the description, ejb-ref-name, ejb-ref-type, home, and remote elements.

The ejb-local-ref element contains the description, ejb-ref-name, ejb-ref-type, local-home, and local elements.

The ejb-ref-name element specifies the EJB reference name: its value is the environment entry name used in the enterprise bean code. The ejb-ref-name MUST be specified. The optional ejb-ref-type element specifies the expected type of the enterprise bean: its value must be either Entity or Session. The home and remote or local-home and local elements specify the expected Java types of the referenced enterprise bean's interface(s). If the reference is to an EJB 2.1 remote client view interface, the home element is REQUIRED. Likewise, if the reference is to an EJB 2.1 local client view interface, the local-home element is REQUIRED. The remote element of the ejb-ref element refers to either the business interface type or the component interface,

depending on whether the reference is to a bean's EJB 3.0 or EJB 2.1 remote client view. Likewise, the local element of the ejb-local-ref element refers to either the business interface type or the component interface, depending on whether the reference is to a bean's EJB 3.0 or EJB 2.1 local client view.

An EJB reference is scoped to the enterprise bean whose declaration contains the ejb-ref or ejb-local-ref element. This means that the EJB reference is NOT accessible to other enterprise beans at runtime, and that other enterprise beans may define ejb-ref and/or ejb-local-ref elements with the same ejb-ref-name WITHOUT causing a name conflict.

The following example illustrates the declaration of EJB references in the deployment descriptor:

```
<enterprise-beans>
      <session>
            <ejb-name>EmployeeService</ejb-name>
            <ejb-class>com.wombat.empl.EmployeeServiceBean</ejb-class>
            ...
            <ejb-ref>
                  <description>
                        This is a reference to an EJB 2.1 entity bean that
                        encapsulates access to employee records.
                  </description>
                  <ejb-ref-name>ejb/EmplRecord</ejb-ref-name>
                  <ejb-ref-type>Entity</ejb-ref-type>
                  <home>com.wombat.empl.EmployeeRecordHome</home>
                  <remote>com.wombat.empl.EmployeeRecord</remote>
            </ejb-ref>

              <ejb-local-ref>
                  <description>
                        This is a reference to the local business interface
                        of an EJB 3.0 session bean that provides a payroll
                        service.
                  </description>
                  <ejb-ref-name>ejb/Payroll</ejb-ref-name>
                  <local>com.aardvark.payroll.Payroll</local>
            </ejb-local-ref>

              <ejb-local-ref>
                  <description>
                        This is a reference to the local business interface
                        of an EJB 3.0 session bean that provides a pension
                        plan service.
                  </description>
                  <ejb-ref-name>ejb/PensionPlan</ejb-ref-name>
                  <local>com.wombat.empl.PensionPlan</local>
            </ejb-local-ref>

      </session>
</enterprise-beans>
```

The Application Assembler can use the ejb-link element in the deployment descriptor to link an EJB reference to a target enterprise bean.

The Application Assembler specifies the link between two enterprise beans as follows:

- The Application Assembler uses the optional ejb-link element of the ejb-ref or ejb-local-ref element of the referencing enterprise bean. The value of the ejb-link element is the name of the target enterprise bean. (This is the bean name as defined by metadata annotation (or default) in the bean class or in the ejb-name element of the target enterprise bean.) The target enterprise bean can be in any ejb-jar file in the same Java EE application as the referencing application component.
- Alternatively, to avoid the need to rename enterprise beans to have unique names within an entire Java EE application, the Application Assembler may use the following syntax in the ejb-link element of the referencing application component. The Application Assembler specifies the path name of the ejb-jar file containing the referenced enterprise bean and appends the ejb-name of the target bean separated from the path name by # . The path name is relative to the referencing application component jar file. In this manner, multiple beans with the same ejb-name may be uniquely identified when the Application Assembler cannot change ejb-names.
- The Application Assembler must ensure that the target enterprise bean is type-compatible with the declared EJB reference. This means that the target enterprise bean must be of the type indicated in the ejb-ref-type element, if present, and that the business interface or home and component interfaces of the target enterprise bean must be Java type-compatible with the interfaces declared in the EJB reference.

The following illustrates an ejb-link in the deployment descriptor:

```
<enterprise-beans>
    <session>
        <ejb-name>EmployeeService</ejb-name>
        <ejb-class>com.wombat.empl.EmployeeServiceBean</ejb-class>
        ...
        <ejb-ref>
            <ejb-ref-name>ejb/EmplRecord</ejb-ref-name>
            <ejb-ref-type>Entity</ejb-ref-type>
            <home>com.wombat.empl.EmployeeRecordHome</home>
            <remote>com.wombat.empl.EmployeeRecord</remote>
            <ejb-link>EmployeeRecord</ejb-link>
        </ejb-ref>
        ...
    </session>
    ...
    <entity>
        <ejb-name>EmployeeRecord</ejb-name>
        <home>com.wombat.empl.EmployeeRecordHome</home>
        <remote>com.wombat.empl.EmployeeRecord</remote>
        ...
    </entity>
    ...
</enterprise-beans>
```

The Application Assembler uses the ejb-link element to indicate that the EJB reference EmplRecord declared in the EmployeeService enterprise bean has been linked to the

EmployeeRecord enterprise bean.

The following example illustrates using the ejb-link element to indicate an enterprise bean reference to the ProductEJB enterprise bean that is in the same Java EE application unit but in a DIFFERENT ejb-jar file:

```
<entity>
	...
	<ejb-name>OrderEJB</ejb-name>
	<ejb-class>com.wombat.orders.OrderBean</ejb-class>
	...
	<ejb-ref>
		<ejb-ref-name>ejb/Product</ejb-ref-name>
		<ejb-ref-type>Entity</ejb-ref-type>
		<home>com.acme.orders.ProductHome</home>
		<remote>com.acme.orders.Product</remote>
		<ejb-link>../products/product.jar#ProductEJB</ejb-link>
	</ejb-ref>
	...
</entity>
```

The following example illustrates using the ejb-link element to indicate an enterprise bean reference to the ShoppingCart enterprise bean that is in the same Java EE application unit but in a DIFFERENT ejb-jar file. The reference was originally declared in the bean's code using an annotation. The Application Assembler provides ONLY the link to the bean:

```
...
<ejb-ref>
	<ejb-ref-name>ShoppingService/myCart</ejb-ref-name>
	<ejb-link>../products/product.jar#ShoppingCart</ejb-link>
</ejb-ref>
```

Resource Manager Connection Factory References

A resource manager connection factory is an object that is used to create connections to a resource manager. For example, an object that implements the javax.sql.DataSource interface is a resource manager connection factory for java.sql.Connection objects that implement connections to a database management system.

A field or a method of an enterprise bean may be annotated with the Resource annotation. The authenticationType and shareable elements of the Resource annotation may be used to control the type of authentication desired for the resource and the shareability of connections acquired from the factory.

The following code example illustrates how an enterprise bean uses annotations to declare resource manager connection factory references.

```
//The employee database.
@Resource javax.sql.DataSource employeeAppDB;
...
public void changePhoneNumber(...) {
	...
```

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

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

- Assign an entry in the enterprise bean's environment to the resource manager connection factory reference.
- The EJB specification recommends, but does NOT require, that all resource manager connection factory references be organized in the subcontexts of the bean's environment, using a different subcontext for each resource manager type. For example, all JDBC DataSource references might be declared in the java:comp/env/jdbc subcontext, and all JMS connection factories in the java:comp/env/jms subcontext. Also, all JavaMail connection factories might be declared in the java:comp/env/mail subcontext and all URL connection factories in the java:comp/env/url subcontext. Note that resource manager connection factory references declared via annotations will NOT, by default, appear in any subcontext.
- Lookup the resource manager connection factory object in the enterprise bean's environment using the EJBContext lookup method or using the JNDI API.
- Invoke the appropriate method on the resource manager connection factory to obtain a connection to the resource. The factory method is specific to the resource type. It is possible to obtain multiple connections by calling the factory object multiple times.

The Bean Provider can control the shareability of the connections acquired from the resource manager connection factory. By default, connections to a resource manager are shareable across other enterprise beans in the application that use the same resource in the same transaction context. The Bean Provider can specify that connections obtained from a resource manager connection factory reference are not shareable by specifying the value of the shareable annotation element to false or the res-sharing-scope deployment descriptor element to be Unshareable. The sharing of connections to a resource manager allows the container to optimize the use of connections and enables the container's use of local transaction optimizations.

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

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

The Bean Provider uses the authenticationType annotation element or the res-auth deployment descriptor element to indicate which of the two resource manager authentication approaches is used.

The following code sample illustrates obtaining a JDBC connection when the EJBContext lookup method is used:

```
@Resource(name="jdbc/EmployeeAppDB", type=javax.sql.DataSource)
@Stateless public class EmployeeServiceBean implements EmployeeService {
	@Resource SessionContext ctx;
	public void changePhoneNumber(...) {
		...
		// use context lookup to obtain resource manager
		// connection factory
		javax.sql.DataSource ds = (javax.sql.DataSource) ctx.lookup("jdbc/EmployeeAppDB");

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

The following code sample illustrates obtaining a JDBC connection when the JNDI APIs are used directly:

```
@Resource(name="jdbc/EmployeeAppDB", type=javax.sql.DataSource)
@Stateless public class EmployeeServiceBean implements EmployeeService {
	EJBContext ejbContext;
	public void changePhoneNumber(...) {
		...
		// obtain the initial JNDI context
		Context initCtx = new InitialContext();

		   // perform JNDI lookup to obtain resource manager
		// connection factory
		javax.sql.DataSource ds = (javax.sql.DataSource) initCtx.lookup("java:comp/env/jdbc/EmployeeAppDB");

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

Although a resource manager connection factory reference is an entry in the enterprise bean's environment, the Bean Provider must not use an env-entry element to declare it.

Instead, if metadata annotations are not used, the Bean Provider must declare all the resource manager connection factory references in the deployment descriptor using the resource-ref elements. This allows the ejb-jar consumer (i.e. Application Assembler or Deployer) to discover all the resource manager connection factory references used by an enterprise bean. Deployment descriptor entries may also be used to specify injection of a resource manager connection factor reference into a bean.

Each resource-ref element describes a single resource manager connection factory reference. The resource-ref element consists of the description element; the mandatory res-ref-name

element; and the optional res-type, res-auth and res-sharing-scope elements. The res-ref-name element contains the name of the environment entry used in the enterprise bean's code. The name of the environment entry is RELATIVE to the java:comp/env context (e.g., the name should be jdbc/EmployeeAppDB rather than java:comp/env/jdbc/EmployeeAppDB). The res-type element contains the Java type of the resource manager connection factory that the enterprise bean code expects. The res-type element is optional if an injection target is specified for the resource; in this case, the res-type defaults to the type of the injection target. The res-auth element indicates whether the enterprise bean code performs resource manager sign-on programmatically, or whether the container signs on to the resource manager using the principal mapping information supplied by the Deployer. The Bean Provider indicates the sign-on responsibility by setting the value of the res-auth element to Application or Container. If the res-auth element is not specified, Container sign-on is ASSUMED. The res-sharing-scope element indicates whether connections to the resource manager obtained through the given resource manager connection factory reference are to be shared or whether connections are unshareable. The value of the res-sharing-scope element is Shareable or Unshareable. If the res-sharing-scope element is not specified, connections are assumed to be SHAREABLE.

A resource manager connection factory reference is scoped to the enterprise bean whose declaration contains the resource-ref element. This means that the resource manager connection factory reference is not accessible from other enterprise beans at runtime, and that other enterprise beans may define resource-ref elements with the same res-ref-name without causing a name conflict.

The following example is the declaration of resource manager connection factory references used by the EmployeeService enterprise bean illustrated in the previous subsection:

```
<enterprise-beans>
    <session>
        <ejb-name>EmployeeService</ejb-name>
        <ejb-class>com.wombat.empl.EmployeeServiceBean</ejb-class>
        ...
        <resource-ref>
            <description>
                A data source for the database in which
                the EmployeeService enterprise bean will
                record a log of all transactions.
            </description>
            <res-ref-name>jdbc/EmployeeAppDB</res-ref-name>
            <res-type>javax.sql.DataSource</res-type>
            <res-auth>Container</res-auth>
            <res-sharing-scope>Shareable</res-sharing-scope>
        </resource-ref>
        ...
    </session>
</enterprise-beans>
```

The following example illustrates the declaration of JMS resource manager connection factory references:

```
<enterprise-beans>
    <session>
        ...
        <resource-ref>
            <description>
```

```
                A queue connection factory used by the
                MySession enterprise bean to send
                notifications.
            </description>
            <res-ref-name>jms/qConnFactory</res-ref-name>
            <res-type>javax.jms.QueueConnectionFactory</res-type>
            <res-auth>Container</res-auth>
            <res-sharing-scope>Unshareable</res-sharing-scope>
        </resource-ref>
        ...
    </session>
</enterprise-beans>
```

Resource Environment References

Resource environment references are special entries in the enterprise bean's environment. The Deployer binds the resource environment references to administered objects in the target operational environment.

A field or a method of a bean may be annotated with the Resource annotation to request injection of a resource environment reference. The authenticationType and shareable elements of the Resource annotation MUST NOT be specified; resource environment entries are NOT shareable and DO NOT require authentication. The use of the Resource annotation to declare a resource environment reference differs from the use of the Resource annotation to declare simple environment references only in that the type of a resource environment reference is not one of the Java language types used for simple environment references.

The Bean Provider must use resource environment references to locate administered objects that are associated with resources, as follows.

- Assign an entry in the enterprise bean's environment to the reference.
- The EJB specification recommends, but does not require, that all resource environment references be organized in the appropriate subcontext of the bean's environment for the resource type. Note that the resource environment references declared via annotations will not, by default, appear in any subcontext.
- Look up the administered object in the enterprise bean’s environment using the EJBContext lookup method or the JNDI API.

Although the resource environment reference is an entry in the enterprise bean's environment, the Bean Provider MUST NOT use a env-entry element to declare it. Instead, the Bean Provider must declare all references to administered objects associated with resources using either annotations in the bean's source code or the resource-env-ref elements of the deployment descriptor. This allows the ejb-jar consumer to discover all the resource environment references used by the enterprise bean. Deployment descriptor entries may also be used to specify injection of a resource environment reference into a bean.

Each resource-env-ref element describes the requirements that the referencing enterprise bean has for the referenced administered object. The resource-env-ref element contains optional description and resource-env-ref-type elements, and the MANDATORY resource-env-ref-name element. The resource-env-ref-type element is optional if an injection target is specified for the

resource environment reference; in this case the resource-env-ref-type defaults to the type of the injection target.

The resource-env-ref-name element specifies the resource environment reference name: its value is the environment entry name used in the enterprise bean CODE. The name of the environment entry is relative to the java:comp/env context. The resource-env-ref-type element specifies the expected type of the referenced object.

A resource environment reference is scoped to the enterprise bean whose declaration contains the resource-env-ref element. This means that the resource environment reference is NOT accessible to other enterprise beans at runtime, and that other enterprise beans MAY define resource-env-ref elements with the SAME resource-env-ref-name without causing a name conflict.

Persistence Context References

Persistence context references are special entries in the enterprise bean's environment. The Deployer binds the persistence context references to container-managed entity managers for persistence contexts of the specified type and configured in accordance with their persistence unit.

A field or a method of an enterprise bean may be annotated with the PersistenceContext annotation. The name element specifies the name under which a container-managed entity manager for the referenced persistence unit may be located in the JNDI naming context. The OPTIONAL unitName element specifies the name of the persistence unit as declared in the persistence.xml file that defines the persistence unit. The OPTIONAL type element specifies whether a transaction-scoped or extended persistence context is to be used. If the type is not specified, a transaction-scoped persistence context will be used. References to container-managed entity managers with extended persistence contexts can ONLY be injected into stateful session beans. The OPTIONAL properties element specifies configuration properties to be passed to the persistence provider when the entity manager is created.

The following code example illustrates how an enterprise bean uses annotations to declare persistence context references:

```
@PersistenceContext(type=EXTENDED)
EntityManager em;
```

The Bean Provider must use persistence context references to obtain references to a container-managed entity manager configured for a persistence unit as follows:

- Assign an entry in the enterprise bean's environment to the persistence context reference.
- The EJB specification recommends, but does not require, that all persistence context references be organized in the java:comp/env/persistence subcontexts of the bean's environment.
- Lookup the container-managed entity manager for the persistence unit in the enterprise bean's environment using the EJBContext lookup method or using the JNDI API.

The following code sample illustrates obtaining an entity manager for a persistence context when the EJBContext lookup method is used:

```
@PersistenceContext(name="persistence/InventoryAppMgr")
@Stateless
public class InventoryManagerBean implements InventoryManager {
      @Resource SessionContext ctx;

       public void updateInventory(...) {
            ...
            // use context lookup to obtain container-managed entity
            // manager
            EntityManager em =(EntityManager)
            ctx.lookup("persistence/InventoryAppMgr");
            ...
      }
}
```

The following code sample illustrates obtaining an entity manager when the JNDI APIs are used directly:

```
@PersistenceContext(name="persistence/InventoryAppMgr")
@Stateless
public class InventoryManagerBean implements InventoryManager {
      EJBContext ejbContext;

       public void updateInventory(...) {
            ...
            // obtain the initial JNDI context
            Context initCtx = new InitialContext();

               // perform JNDI lookup to obtain container-managed entity
            // manager
            EntityManager em = (EntityManager)
                  initCtx.lookup("java:comp/env/persistence/InventoryAppMgr");
            ...
      }
}
```

Although a persistence context reference is an entry in the enterprise bean's environment, the Bean Provider must not use an env-entry element to declare it.

Instead, if metadata annotations are not used, the Bean Provider MUST declare all the persistence context references in the deployment descriptor using the persistence-context-ref elements. This allows the ejb-jar consumer (i.e. Application Assembler or Deployer) to discover all the persistence context references used by an enterprise bean. Deployment descriptor entries may also be used to specify injection of a persistence context reference into a bean.

Each persistence-context-ref element describes a single container-managed entity manager reference. The persistence-context-ref element consists of the OPTIONAL description, persistence-unit-name, persistence-context-type and persistence-property elements, and the MANDATORY persistence-context-ref-name element.

The persistence-context-ref-name element contains the name of the environment entry used in the enterprise bean's CODE. The name of the environment entry is relative to the java:comp/env

context (e.g., the name should be persistence/InventoryAppMgr rather than java:comp/env/persistence/InventoryAppMgr). The persistence-unit-name element is the name of the persistence unit, as specified in the persistence.xml file for the persistence unit. The persistence-context-type element specifies whether a transaction-scoped or extended persistence context is to be used. Its value is either Transaction or Extended. If the persistence context type is not specified, a *transaction-scoped* persistence context will be used. The optional persistence-property elements specify configuration properties that are passed to the persistence provider when the entity manager is created.

The following example is the declaration of a persistence context reference used by the InventoryManagerBean enterprise bean illustrated in the previous subsection:

```
<enterprise-beans>
    <session>
        <ejb-name>InventoryManagerBean</ejb-name>
        <ejb-class>com.wombat.empl.InventoryManagerBean</ejb-class>
        ...
        <persistence-context-ref>
            <description>
                Persistence context for the inventory management
                application.
            </description>
            <persistence-context-ref-name>
                persistence/InventoryAppMgr
            </persistence-context-ref-name>
            <persistence-unit-name>
                InventoryManagement
            </persistence-unit-name>
        </persistence-context-ref>
    </session>
</enterprise-beans>
```

The Application Assembler can use the persistence-unit-name element in the deployment descriptor to specify a reference to a persistence unit. In this manner, multiple persistence units with the same persistence unit name may be uniquely identified when persistence unit names cannot be changed.

For example:

```
<enterprise-beans>
    <session>
        <ejb-name>InventoryManagerBean</ejb-name>
        <ejb-class>com.wombat.empl.InventoryManagerBean</ejb-class>
        ...
        <persistence-context-ref>
            <description>
                Persistence context for the inventory management
                application.
            </description>
            <persistence-context-ref-name>
                persistence/InventoryAppMgr
            </persistence-context-ref-name>
            <persistence-unit-name>
```

```
                ../lib/inventory.jar#InventoryManagement
            </persistence-unit-name>
        </persistence-context-ref>
    </session>
</enterprise-beans>
```

The Application Assembler uses the persistence-unit-name element to link the persistence unit name InventoryManagement declared in the InventoryManagerBean to the persistence unit named InventoryManagement defined in inventory.jar.

The following rules apply to how a deployment descriptor entry may override a PersistenceContext annotation:

- The relevant deployment descriptor entry is located based on the JNDI name used with the annotation (either defaulted or provided explicitly).
- The persistence-unit-name OVERRIDES the unitName element of the annotation. The Application Assembler or Deployer should exercise CAUTION in changing this value, if specified, as doing so is likely to break the application.
- The persistence-context-type, if specified, OVERRIDES the type element of the annotation. In general, the Application Assembler or Deployer SHOULD NEVER change the value of this element, as doing so is likely to break the application.
- Any persistence-property elements are added to those specified by the PersistenceContext annotation. If the name of a specified property is the same as one specified by the PersistenceContext annotation, the value specified in the annotation IS OVERRIDEN.
- The injection target, if specified, must name exactly the annotated field or property method.

Persistence Unit References

Persistence unit references are special entries in the enterprise bean's environment. The Deployer binds the persistence unit references to entity manager factories that are configured in accordance with the persistence.xml specification for the persistence unit.

A field or a method of an enterprise bean may be annotated with the PersistenceUnit annotation. The name element specifies the name under which the entity manager factory for the referenced persistence unit may be located in the JNDI naming context. The OPTIONAL unitName element specifies the name of the persistence unit as declared in the persistence.xml file that defines the persistence unit.

The following code example illustrates how an enterprise bean uses annotations to declare persistence unit references:

```
@PersistenceUnit
EntityManagerFactory emf;

@PersistenceUnit(unitName="InventoryManagement")
EntityManagerFactory inventoryEMF;
```

The Bean Provider must use persistence unit references to obtain references to entity manager factories as follows.

- Assign an entry in the enterprise bean's environment to the persistence unit reference.
- The EJB specification RECOMMENDS, but does not require, that all persistence unit references be organized in the java:comp/env/persistence subcontexts of the bean's environment.
- Lookup the entity manager factory for the persistence unit in the enterprise bean's environment using the EJBContext lookup method or using the JNDI API.

The following code sample illustrates obtaining an entity manager factory when the EJBContext lookup method is used:

```
@PersistenceUnit(name="persistence/InventoryAppDB")
@Stateless
public class InventoryManagerBean implements InventoryManager {
      @Resource SessionContext ctx;

        public void updateInventory(...) {
            ...
            // use context lookup to obtain entity manager factory
            EntityManagerFactory emf = (EntityManagerFactory)
            ctx.lookup("persistence/InventoryAppDB");

               // use factory to obtain application-managed entity manager
            EntityManager em = emf.createEntityManager();
            ...
      }
}
```

The following code sample illustrates obtaining an entity manager factory when the JNDI APIs are used directly:

```
@PersistenceUnit(name="persistence/InventoryAppDB")
@Stateless
public class InventoryManagerBean implements InventoryManager {
      EJBContext ejbContext;

      public void updateInventory(...) {
            ...
            // obtain the initial JNDI context
            Context initCtx = new InitialContext();

               // perform JNDI lookup to obtain entity manager factory
            EntityManagerFactory emf = (EntityManagerFactory)
                  initCtx.lookup("java:comp/env/persistence/InventoryAppDB");

            // use factory to obtain application-managed entity manager
            EntityManager em = emf.createEntityManager();
            ...
      }
}
```

Although a persistence unit reference is an entry in the enterprise bean's environment, the Bean Provider MUST NOT use an env-entry element to declare it.

Instead, if metadata annotations are not used, the Bean Provider must declare all the persistence unit references in the deployment descriptor using the persistence-unit-ref elements. This allows the ejb-jar consumer (i.e. Application Assembler or Deployer) to discover all the persistence unit references used by an enterprise bean. Deployment descriptor entries may also be used to specify injection of a persistence unit reference into a bean.

Each persistence-unit-ref element describes a single entity manager factory reference for the persistence unit. The persistence-unit-ref element consists of the optional description and persistence-unit-name elements, and the MANDATORY persistence-unit-ref-name element.

The persistence-unit-ref-name element contains the name of the environment entry used in the enterprise bean's CODE. The name of the environment entry is relative to the java:comp/env context (e.g., the name should be persistence/InventoryAppDB rather than java:comp/env/persistence/InventoryAppDB). The optional persistence-unit-name element is the name of the persistence unit, as specified in the persistence.xml file for the persistence unit.

The following example is the declaration of a persistence unit reference used by the InventoryManagerBean enterprise bean illustrated in the previous code sample:

```
<enterprise-beans>
    <session>
        <ejb-name>InventoryManagerBean</ejb-name>
        <ejb-class>com.wombat.empl.InventoryManagerBean</ejb-class>
        ...
        <persistence-unit-ref>
            <description>
                Persistence unit for the inventory management
                application.
            </description>
            <persistence-unit-ref-name>
                persistence/InventoryAppDB
            </persistence-unit-ref-name>
            <persistence-unit-name>
                InventoryManagement
            </persistence-unit-name>
        </persistence-unit-ref>
        ...
    </session>
</enterprise-beans>
```

The Application Assembler can use the persistence-unit-name element in the deployment descriptor to specify a reference to a persistence unit. The Application Assembler (or Bean Provider) may use the following syntax in the persistence-unit-name element of the referencing application component to avoid the need to rename persistence units to have unique names within a Java EE application. The Application Assembler specifies the path name of the root of the referenced persistence unit and appends the name of the persistence unit separated from the path name by #. The path name is relative to the referencing application component jar file. In this manner, multiple persistence units with the same persistence unit name may be uniquely

identified when persistence unit names cannot be changed.

For example:

```
<enterprise-beans>
    <session>
        ...
        <ejb-name>InventoryManagerBean</ejb-name>
        <ejb-class>com.wombat.empl.InventoryManagerBean</ejb-class>
        ...
        <persistence-unit-ref>
            <description>
                Persistence unit for the inventory management
                application.
            </description>
            <persistence-unit-ref-name>
                persistence/InventoryAppDB
            </persistence-unit-ref-name>
            <persistence-unit-name>
                ../lib/inventory.jar#InventoryManagement
            </persistence-unit-name>
        </persistence-unit-ref>
        ...
    </session>
</enterprise-beans>
```

The Application Assembler uses the persistence-unit-name element to link the persistence unit name InventoryManagement declared in the InventoryManagerBean to the persistence unit named InventoryManagement defined in inventory.jar.

The following rules apply to how a deployment descriptor entry may override a PersistenceUnit annotation:

- The relevant deployment descriptor entry is located based on the JNDI name used with the annotation (either defaulted or provided explicitly).
- The persistence-unit-name overrides the unitName element of the annotation. The Application Assembler or Deployer should exercise CAUTION in changing this value, if specified, as doing so is likely to break the application.
- The injection target, if specified, must name exactly the annotated field or property method.

Identify correct and incorrect statements or examples about Timer Services, including the bean provider's view and responsibilities, the TimerService, Timer and TimerHandle interfaces, and @Timeout callback methods.

- [EJB_3.0_CORE] 18.1; 18.2; 18.2.1; 18.2.2; 18.2.3; 18.2.4; 18.2.5; 18.3.1; 18.3.2; 4.3.8; 5.4.7

The EJB Timer Service is a container-managed service that provides methods to allow callbacks to be scheduled for time-based events. The container provides a reliable and transactional

notification service for timed events. Timer notifications may be scheduled to occur at a specific time, after a specific elapsed duration, or at specific recurring intervals.

The Timer Service is implemented by the EJB container. An enterprise bean accesses this service by means of dependency injection, through the EJBContext interface, or through lookup in the JNDI namespace.

The EJB Timer Service is a coarse-grained timer notification service that is designed for use in the modeling of application-level processes. It is not intended for the modeling of real-time events.

Bean Provider's View of the Timer Service

The EJB Timer Service is a container-provided service that allows enterprise beans to be registered for timer callback methods to occur at a specified time, after a specified elapsed time, or after specified intervals. The timer service provides methods for the creation and cancellation of timers, as well as for locating the timers that are associated with a bean.

A timer is created to schedule timed callbacks. The bean class of an enterprise bean that uses the timer service must provide a **timeout callback method**. This method may be a method that is annotated with the Timeout annotation, or the bean may implement the javax.ejb.TimedObject interface. The javax.ejb.TimedObject interface has a single method, the timer callback method ejbTimeout.

Timers CAN be created for:

- stateless session beans
- message-driven beans
- 2.1 entity beans

Timers CANNOT be created for:

- stateful session beans
- EJB 3.0 entities

A timer that is created for a 2.1 entity bean is associated with the entity bean's identity. The timeout callback method invocation for a timer that is created for a stateless session bean or a message-driven bean may be called on ANY bean instance in the pooled state.

When the time specified at timer creation elapses, the container invokes the timeout callback method of the bean. A timer may be cancelled by a bean before its expiration. If a timer is cancelled, the timeout callback method is not called. A timer is cancelled by calling its cancel method.

Invocations of the methods to create and cancel timers and of the timeout callback method are typically made within a transaction.

The timer service is intended for the modelling of long-lived business processes. Timers SURVIVE container crashes, server shutdown, and the activation/passivation and load/store cycles of the enterprise beans that are registered with them.

The Timer Service Interface

The Timer Service is accessed via dependency injection, through the getTimerService method of the EJBContext interface, or through lookup in the JNDI namespace. The TimerService interface has the following methods:

```
public interface javax.ejb.TimerService {

    public Timer createTimer(long duration, java.io.Serializable info);
    public Timer createTimer(long initialDuration, long intervalDuration, java.io.Serializable info);
    public Timer createTimer(java.util.Date expiration, java.io.Serializable info);
    public Timer createTimer(java.util.Date initialExpiration, long intervalDuration, java.io.Serializable info);

     public Collection getTimers();

}
```

The timer creation methods allow a timer to be created as a single-event timer or as an interval timer. The timer expiration (initial expiration in the case of an interval timer) may be expressed either in terms of a duration or as an absolute time.

The bean may pass some client-specific information at timer creation to help it recognize the significance of the timer's expiration. This information is stored by the timer service and available through the timer. The information object must be serializable.

The timer duration is expressed in terms of MILLISECONDS. The timer service begins counting down the timer duration upon timer creation.

The createTimer methods return a Timer object that allows the bean to cancel the timer or to obtain information about the timer prior to its cancellation and/or expiration (if it is a single-event timer).

The getTimers method returns the active timers associated with the bean. For an EJB 2.1 entity bean, the result of getTimers is a collection of those timers that are associated with the bean's identity.

Timeout Callbacks

The enterprise bean class of a bean that is to be registered with the timer service for timer callbacks must provide a timeout callback method.

This method may be a method annotated with the Timeout annotation (or a method specified as a timeout-method in the deployment descriptor) or the bean may implement the javax.ejb.TimedObject interface. This interface has a single method, ejbTimeout. If the bean implements the TimedObject interface, the Timeout annotation or timeout-method deployment descriptor element can only be used to specify the ejbTimeout method. A bean can have AT MOST ONE timeout method.

```
public interface javax.ejb.TimedObject {

    public void ejbTimeout(Timer timer);

}
```

Any method annotated as a Timeout method (or designated in the deployment descriptor as such) MUST have the signature below, where <METHOD> designates the method name. A timeout method can have public, private, protected, or package level access. A timeout method MUST NOT be declared as final or static.

```
void <METHOD>(Timer timer)
```

Timeout callback methods MUST NOT throw application exceptions.

When the timer expires (i.e., after the number of milliseconds specified at its creation expires or after the absolute time specified has passed), the container calls the timeout method of the bean that was registered for the timer. The timeout method contains the business logic that the Bean Provider supplies to handle the timeout event. The container calls the timeout method with the timer that has expired. The Bean Provider can use the getInfo method to retrieve the information that was supplied when the timer was created. This information may be useful in enabling the timed object to recognize the significance of the timer expiration.

The container interleaves calls to the timeout callback method with the calls to the business methods and the life cycle callback methods of the bean. The time at which the timeout callback method is called may therefore not correspond exactly to the time specified at timer creation. If multiple timers have been created for a bean and will expire at approximately the same times, the Bean Provider must be prepared to handle timeout callbacks that are OUT OF SEQUENCE. The Bean Provider must be prepared to handle EXTRANEOUS calls to the timeout callback method in the event that a timer expiration is outstanding when a call to the cancellation method has been made.

In general, the timeout callback method can perform the same operations as business methods from the component interface or methods from the message listener interface.

Since the timeout callback method is an internal method of the bean class, it has NO CLIENT SECURITY CONTEXT. When getCallerPrincipal is called from within the timeout callback method, it returns the container's representation of the unauthenticated identity.

If the timed object needs to make use of the identity of the timer to recognize the significance of the timer expiration, it may use the equals method to compare it with any other timer references it might have outstanding.

If the timer is a single-action timer, the container removes the timer after the timeout callback method has been successfully invoked (e.g., when the transaction that has been started for the invocation of the timeout callback method commits). If the bean invokes a method on the timer after the termination of the timeout callback method, the NoSuchObjectLocalException is thrown.

The Timer Interface

The javax.ejb.Timer interface allows the Bean Provider to cancel a timer and to obtain information about the timer.

```
public interface javax.ejb.Timer {

      public void cancel();
    public long getTimeRemaining();
    public java.util.Date getNextTimeout();
    public javax.ejb.TimerHandle getHandle();
    public java.io.Serializable getInfo();

}
```

The TimerHandle Interface

The javax.ejb.TimerHandle interface allows the Bean Provider to obtain a serializable timer handle that may be persisted. Since timers are LOCAL objects, a TimerHandle MUST NOT be passed through a bean's remote business interface, remote interface or web service interface.

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

    public javax.ejb.Timer getTimer();

}
```

Timer Identity

The Bean Provider cannot rely on the == operator to compare timers for "object equality". The Bean Provider MUST use the Timer.equals(Object obj) method.

Transactions

An enterprise bean typically creates a timer within the scope of a transaction. If the transaction is then rolled back, the timer creation is rolled back.

An enterprise bean typically cancels a timer within a transaction. If the transaction is rolled back, the container rescinds the timer cancellation.

The timeout callback method is typically has transaction attribute REQUIRED or REQUIRES_NEW (Required or RequiresNew if the deployment descriptor is used to specify the transaction attribute). If the transaction is rolled back, the container retries the timeout.

Enterprise Bean Class

An enterprise bean that is to be registered with the Timer Service must have a timeout callback method. The enterprise bean class may have superclasses and/or superinterfaces. If the bean class has superclasses, the timeout method may be defined in the bean class, or in any of its superclasses.

TimerHandle

Since the TimerHandle interface extends java.io.Serializable, a client may serialize the handle. The serialized handle may be used later to obtain a reference to the timer identified by the handle. A TimerHandle is intended to be storable in persistent storage.

A TimerHandle MUST NOT be passed as an argument or result of an enterprise bean's remote business interface, remote interface, or web service method.

Timeout Callbacks for Stateless Session Beans

A stateless session bean can be registered with the EJB Timer Service for time-based event notifications if it provides a timeout callback method. The container invokes the bean instance's timeout callback method when a timer for the bean has expired. Stateful session beans CANNOT be registered with the EJB Timer Service, and therefore should not implement timeout callback methods.

Timeout Callbacks for Message-Driven Beans

A message driven bean can be registered with the EJB timer service for time-based event notifications if it provides a timeout callback method. The container invokes the bean instance's

timeout callback method when a timer for the bean has expired.

Identify correct and incorrect statements or examples about the EJB context objects that the container provides to 3.0 Session beans and 3.0 Message-Driven beans, including the security, transaction, timer, and lookup services the context can provide.

- [EJB_3.0_CORE] 4.3.3; 5.4.4

[EJB_3.0_SIMPLIFIED] 8; 8.1; 8.1.1; 8.1.2; 8.1.3; 8.1.4

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

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

```
package javax.ejb;

public interface EJBContext {

    // Lookup a resource within the component's private naming context
    // "name" - Name of the entry (relative to java:comp/env)
    public Object lookup(String name);

    // Get access to the EJB Timer Service
    // IllegalStateException is thrown for Stateful Session Beans
    public TimerService getTimerService() throws IllegalStateException;

    // Security methods
    public java.security.Principal getCallerPrincipal();
    public boolean isCallerInRole(java.lang.String roleName);

    // Transaction methods
    public javax.transaction.UserTransaction getUserTransaction() throws IllegalStateException;
    public boolean getRollbackOnly() throws IllegalStateException;
    public void setRollbackOnly() throws IllegalStateException;

    // Deprecated and Obsolete methods
    public java.security.Identity getCallerIdentity();
    public boolean isCallerInRole(java.security.Identity role);
    public java.util.Properties getEnvironment();

    // Obtain the enterprise bean's Remote Home interface
    public EJBHome getEJBHome() IllegalStateException;

     // Obtain the enterprise bean's Local Home interface.
    public EJBLocalHome getEJBLocalHome() IllegalStateException;
```

The SessionContext Interface

If the bean specifies a dependency on the SessionContext interface (or if the bean class

implements the SessionBean interface), the container MUST provide the session bean instance with a SessionContext. This gives the session bean instance access to the instance's context maintained by the container. The SessionContext interface has the following methods:

- The getCallerPrincipal method returns the java.security.Principal that identifies the invoker.
- The isCallerInRole method tests if the session bean instance's caller has a particular role.
- The setRollbackOnly method allows the instance to mark the current transaction such that the only outcome of the transaction is a ROLLBACK. Only instances of a session bean with container-managed transaction (CMT) demarcation can use this method.
- The getRollbackOnly method allows the instance to test if the current transaction has been marked for ROLLBACK. Only instances of a session bean with container-managed transaction (CMT) demarcation can use this method.
- The getUserTransaction method returns the javax.transaction.UserTransaction interface. The instance can use this interface to demarcate transactions and to obtain transaction status. Only instances of a session bean with bean-managed transaction (BMT) demarcation can use this method.
- The getTimerService method returns the javax.ejb.TimerService interface. Only STATELESS session beans can use this method. Stateful session beans CANNOT be timed objects.
- The getMessageContext method returns the javax.xml.rpc.handler.MessageContext interface of a stateless session bean that implements a JAX-RPC web service endpoint. Only STATELESS session beans with web service endpoint interfaces can use this method.
- The getBusinessObject(Class businessInterface) method returns the session bean's business interface. Only session beans with an EJB 3.0 business interface can call this method.
- The getInvokedBusinessInterface method returns the session bean business interface through which the bean was invoked.
- The getEJBObject method returns the session bean's remote interface. Only session beans with a remote EJBObject interface can call this method.
- The getEJBHome method returns the session bean's remote home interface. Only session beans with a remote home interface can call this method.
- The getEJBLocalObject method returns the session bean's local interface. Only session beans with a local EJBLocalObject interface can call this method.
- The getEJBLocalHome method returns the session bean's local home interface. Only session beans with a local home interface can call this method.
- The lookup method enables the session bean to look up its environment entries in the JNDI naming context.

Explicit Dependency Lookup Through the EJBContext API

A new method, "lookup", has been added to the javax.ejb.EJBContext interface. This method can be used to lookup the resources or references bound in a bean's JNDI environment naming context. The method's signature is:

```
Object lookup(String name)
```

The lookup method internally wraps a call to InitialContext.lookup() so that a developer doesn't need to additionally learn the JNDI API. The SessionContext is injected by the EJB Container immediately after the bean is created, and the SessionContext object is then used to lookup resources.

Here's a comparison of explicitly looking up a bean reference in EJB 2.1 and explicitly looking up a bean reference in EJB 3.0 (without annotation).

EJB 2.1:

```
InitialContext ic = new InitialContext();

Object homeObj = ic.lookup("java:comp/env/ejb/FooEJB");

FooHome fooHome = (FooHome) PortableRemoteObject.narrow(homeObj, FooHome.class);

Foo foo = fooHome.create(...);
```

EJB 3.0:

```
@Resource SessionContext sc;

public void setup(){

    Foo foo = (Foo) sc.lookup("ejb/FooEJB");

}
```

The MessageDrivenContext Interface

If the bean specifies a dependency on the MessageDrivenContext interface (or if the bean class implements the MessageDrivenBean interface), the container must provide the message-driven bean instance with a MessageDrivenContext. This gives the message-driven bean instance access to the instance's context maintained by the container. The MessageDrivenContext interface has the following methods:

- The setRollbackOnly method allows the instance to mark the current transaction such that the only outcome of the transaction is a ROLLBACK. Only instances of a message-driven bean with container-managed transaction (CMT) demarcation can use this method.
- The getRollbackOnly method allows the instance to test if the current transaction has been marked for ROLLBACK. Only instances of a message-driven bean with container-managed transaction (CMT) demarcation can use this method.
- The getUserTransaction method returns the javax.transaction.UserTransaction interface that the instance can use to demarcate transactions, and to obtain transaction status. Only instances of a message-driven bean with bean-managed transaction (BMT) demarcation can use this method.

- The getTimerService method returns the javax.ejb.TimerService interface.
- The getCallerPrincipal method returns the java.security.Principal that is associated with the invocation.
- The isCallerInRole method is inherited from the EJBContext interface. Message-driven bean instances MUST NOT call this method.
- The getEJBHome and getEJBLocalHome methods are inherited from the EJBContext interface. Message-driven bean instances MUST NOT call these methods.
- The lookup method enables the message-driven bean to look up its environment entries in the JNDI naming context.

Enterprise Bean Context and Environment

The enterprise bean's context comprises its container context and its resource and environment context.

The bean may gain access to references to resources and other environment entries in its context by having the container supply it with those references. In this case, bean instance variables or setter methods are annotated as target for dependency injection.

Alternatively, the lookup method added to the javax.ejb.EJBContext interface or the JNDI APIs may be used to look up resources in the bean's environment.

The same set of metadata annotations are used to express context dependencies for both these approaches.

Annotation of Context Dependencies

A bean declares a dependency upon a resource or other entry in its environment context through a dependency annotation.

A dependency annotation specifies the type of object or resource on which the bean is dependent, its characteristics, and the name through which it is to be accessed.

The following are examples of dependency annotations:

```
@EJB(name="mySessionBean", beanInterface=MySessionIF.class)

@Resource(name="myDB", type=javax.sql.DataSource.class)
```

Dependency annotations may be attached to the bean class or to its instance variables or methods.

The amount of information that needs to be specified for a dependency annotation depends upon its usage context and how much information can be inferred from that context.

Annotation of Instance Variables

The developer may annotate instance variables of the enterprise bean class to indicate dependencies upon resources or other objects in the bean's environment. The container automatically initializes these annotated variables with the external references to the specified environment objects. This initialization occurs BEFORE any business methods are invoked on the bean instance and AFTER the time the the bean's EJBContext is set.

Example:

```
@Stateless public class MySessionBean implements MySession {

	@Resource(name="myDB") //type is inferred from variable
	public DataSource customerDB;

	@EJB //reference name and type inferred from variable
	public AddressHome addressHome;

	public void myMethod1(String myString) {
		try {
			Connection conn = customerDB.getConnection();
			...
		} catch (Exception ex) {
			...
		}
	}

	public void myMethod2(String myString) {
		Address a = addressHome.create(myString);
	}
}
```

When the resource type can be determined by the variable type, the annotation need not contain the type of the object to be accessed. If the name for the resource reference in the bean's environment is the same as the variable name, it does not need to be explicitly specified:

```
@EJB public ShoppingCart myShoppingCart;

@Resource public DataSource myDB;

@Resource public UserTransaction utx;

@Resource SessionContext ctx;
```

Setter Injection

Setter injection provides an alternative to the container's initialization of variables described above.

When setter injection is to be used, the dependency annotations are applied to setter methods of the bean class defined for that purpose:

```
@Resource(name="customerDB")
public void setDataSource(DataSource myDB) {
	this.ds = myDB;
}

@Resource // reference name is inferred from the property name
public void setCustomerDB(DataSource myDB) {
	this.customerDB = myDB;
}

@Resource
```

```
public void setSessionContext(SessionContext ctx) {
      this.ctx = ctx;
}
```

When the resource type can be determined by the parameter type, the annotation need not specify the type of the object to be accessed. If the name of the resource is the same as the property name corresponding to the setter method, it does not need to be explicitly specified.

A setter method that is annotated with the Resource or other dependency annotation will be used by the container for dependency injection. Such setter injection methods will be called by the container BEFORE any business methods are invoked on the bean instance and AFTER the bean's EJBContext is set.

Injection and Lookup

Resources, references to components, and other objects that may be looked up in the JNDI name space may be injected by means of the injection mechanisms listed above.

References to injected objects are looked up name. These lookups are performed in the referencing bean's java:comp/env namespace.

EJBContext

The method Object lookup(String name) is added to the javax.ejb.EJBContext interface. This method can be used to lookup resources and other environment entries bound in the bean's JNDI environment naming context.

Injection of the bean's EJBContext object may be obtained as described above.

Identify correct and incorrect statements or examples about EJB 3.0 / EJB 2.x interoperability, including how to adapt an EJB 3.0 bean for use with clients written to the EJB 2.x API and how to access beans written to the EJB 2.x API from beans written to the EJB 3.0 API.

- [EJB_3.0_CORE]

[EJB_3.0_SIMPLIFIED] 9.2; 9.2.1; 9.2.2; 9.2.3; 9.3; 9.3.1; 9.3.2; 9.4

Interoperability of EJB 3.0 and Earlier Components

This release of Enterprise JavaBeans supports migration and interoperability among client and server components written to different versions of the EJB APIs.

Clients written to the EJB 2.x APIs

An enterprise bean that is written to the EJB 2.1 or earlier API release may be a client of components written to EJB 3.0 API using the earlier EJB APIs when deployed in an EJB 3.0 container.

Such an EJB 2.1 or earlier client component does not need to be rewritten or recompiled to be a client of a component written to the EJB 3.0 API.

Such clients may access components written to the EJB 3.0 APIs and components written to the

earlier EJB APIs within the same transaction.

Clients written to the new EJB 3.0 API

A client written to the EJB 3.0 API may be a client of a component written to the EJB 2.1 or earlier API.

Such clients may access components written to the EJB 3.0 APIs and components written to the earlier EJB APIs within the same transaction.

Such clients access components written to the earlier EJB APIs using the EJB 2.1 client view home and component interfaces. The EJB annotation (or the ejb-ref and ejb-local-ref deployment descriptor elements) may be used to specify the injection of home interfaces into components that are clients of beans written to the earlier EJB client view.

Combined use of EJB 2.x and EJB 3.0 persistence APIs

EJB clients may access EJB 3.0 entities and/or the EntityManager together with EJB 2.x entity beans together within the same transaction as well as within separate transactions.

NOTE: In general, the same database data should not be accessed by both EJB 3.0 and EJB 2.x entities within the same application: behavior is unspecified if data aliasing occurs.

Adapting EJB 3.0 Session Beans to Earlier Client Views

Clients written to the EJB 2.1 and earlier client view depend upon the existence of a home and component interface.

A session bean written to the EJB 3.0 API may be adapted to such earlier preexisting client view interfaces.

The session bean designates the interfaces to be adapted by using the RemoteHome and/or LocalHome metadata annotations (or equivalent deployment descriptor elements).

When the client is deployed, the container classes that implement the EJB 2.1 home and remote interfaces (or local home and local interfaces) referenced by the client MUST provide the implementation of the javax.ejb.EJBHome and javax.ejb.EJBObject interfaces (or the javax.ejb.EJBLocalHome and javax.ejb.EJBLocalObject interfaces) respectively.

In addition, the container implementation classes must implement the methods of the home and component interfaces to apply to the EJB 3.0 component being referenced.

Adapting EJB 3.0 Stateless Session Beans

The invocation of the home create() method must return the corresponding local or remote component interface of the bean. This may or may not cause the creation of the bean instance, depending on the container's implementation strategy. For example, the container may preallocate bean instances (e.g., in a pooling strategy) or may defer the creation of the bean instance until the first invocation of a business method on the bean class. When the bean instance is created, the container invokes the setSessionContext method (if any), performs any other dependency injection, and invokes the PostConstruct lifecycle callback method(s) (if any).

It is likewise implementation-dependent as to whether the invocation of the EJBHome remove(Handle) or EJBObject or EJBLocalObject remove() method causes the immediate removal of the bean instance, depending on the container's implementation strategy. When the

bean instance is removed, the PreDestroy callback method (if any) is invoked.

The invocations of the business methods of the component interface are delegated to the bean class.

Adapting EJB 3.0 Stateful Session Beans

The invocation of a create<METHOD>() method causes construction of the bean instance, invocation of the PostConstruct callback (if any), and invocation of the matching Init method, and returns the corresponding local or remote component interface of the bean. The invocation of these methods occurs in the same transaction and security context as the client's call to the create method.

The invocation of the EJBHome remove(Handle) or EJBObject or EJBLocalObject remove() method causes the invocation of the the PreDestroy callback method (if any) and removal of bean instance.

The invocations of the business methods of the component interface are delegated to the bean class.

The Init annotation is used to specify the correspondence of a method on the bean class with a create method of the adapted EJBHome and/or EJBLocalHome interface. The result type of such an Init method is required to be void, and its parameter types must be exactly the same as those of the referenced create<METHOD>() method.

Combined Use of EJB 3.0 and EJB 2.1 APIs in a Bean Class

This document describes the typical usage of annotations to specify the enterprise bean type and callback methods. It is permitted to combine the use of such annotations with the bean's implementation of one of the javax.ejb.EnterpriseBean interfaces as such combination may be useful in facilitating migration to the EJB 3.0 simplified programming model.

In addition to the business interface, a session bean may define EJBHome, EJBLocalHome, EJBObject, and/or EJBLocalObject interfaces in accordance with the rules of the EJB 2.1 specification. A deployment descriptor or metadata annotations may be used to associate the bean class with these interfaces.

EJB 2.1 Client / EJB 3.0 Component

Enterprise bean components, which need to work with EJB 2.1 and earlier specifications, don't need to be written using old specifications any more. EJB 3.0 beans can utilize metadata annotations to make them work with older clients that expect to access them using the home and the remote interface. The methods required as per older specifications are mapped to corresponding methods in the enterprise bean written using the EJB 3.0 specification. As an example, the create() method as desired in the old specification could now map to a method that initializes the bean.

Older beans written to EJB 2.1 client view can talk to new components

- Allows server components to be updated or replaced without affecting existing clients
- New beans can support EJB 3.0 clients as well as earlier clients

- Home and component interfaces are mapped to EJB 3.0 bean class
- New EJB 3.0 components can support both EJB 2.1 clients and EJB 3.0 clients

Example:

```
// EJB 2.1 client view of 3.0 bean
...
Context initialContext = new InitialContext();
ShoppingCartHome myCartHome = (ShoppingCartHome)
      initialContext.lookup("java:comp/env/ejb/cart");
ShoppingCart cart= myCartHome.create();
cart.addItem(...);
...
cart.remove();
...
```

Using EntityManager API from EJB 2.x Stateless Session Bean

The javax.persistence.EntityManager API is used for creating, finding, updating entity bean instances. The MySessionEJB EJB 2.1 stateless session bean uses EntityManager API to create Employee EJB 3.0 bean instances. The stateless EJB makes a JNDI lookup to obtain an instance of EntityManager and uses persist method on EntityManager instance to create EJB 3.0 entity bean objects:

```
public class MySessionEJBBean implements SessionBean {

      public void addEmployee(int empNo, String name, double sal) {

            Employee emp = null;

            try {
                  Context ctx = new InitialContext();
                  EntityManager em =
                        (EntityManager) ctx.lookup("java:comp/env/persistence/EntMngr");

                  emp = new Employee();
                  emp.setEmpNo(empNo);
                  emp.setName(name);
                  emp.setSal(sal);
                  em.persist(emp);

            } catch (Exception ex) {
                  ...
            }
      }

      public void ejbCreate() {
      }
      ...
}
```

Please note this release makes EntityManager available in ENC by using the persistence-context-ref. For packaging EJB 2.x beans with EJB 3.0 beans or entities you need to specify the version="3.0" in the ejb-jar tag in the deployment descriptor as follows:

```
<ejb-jar version="3.0">
```

```
    <enterprise-beans>
        <session>
            <ejb-name>MySessionEJB</ejb-name>
            ..
            <persistence-context-ref>
                <persistence-context-ref-name>
                    persistence/EntMngr
                </persistence-context-ref-name>
                <persistence-unit-name>
                    HRDept
                </persistence-unit-name>
            </persistence-context-ref>
        </session>
    </enterprise-beans>
</ejb-jar>
```

EJB 3.0 Client / EJB 2.1 Component

EJB 3.0 beans can access EJB 2.1 and earlier beans. Dependency injection is utilized to inject the EJB 2.1 component references. JNDI lookup calls are avoided. Once a handle is gained to the injected EJB home interface, the create() method is called to instantiate and subsequently utilize the bean.

Beans written to new APIs can be clients of older beans

- Reuse existing components in new applications
- Allows piecemeal migration of applications
- Injection of Homes simplifies client view

Example:

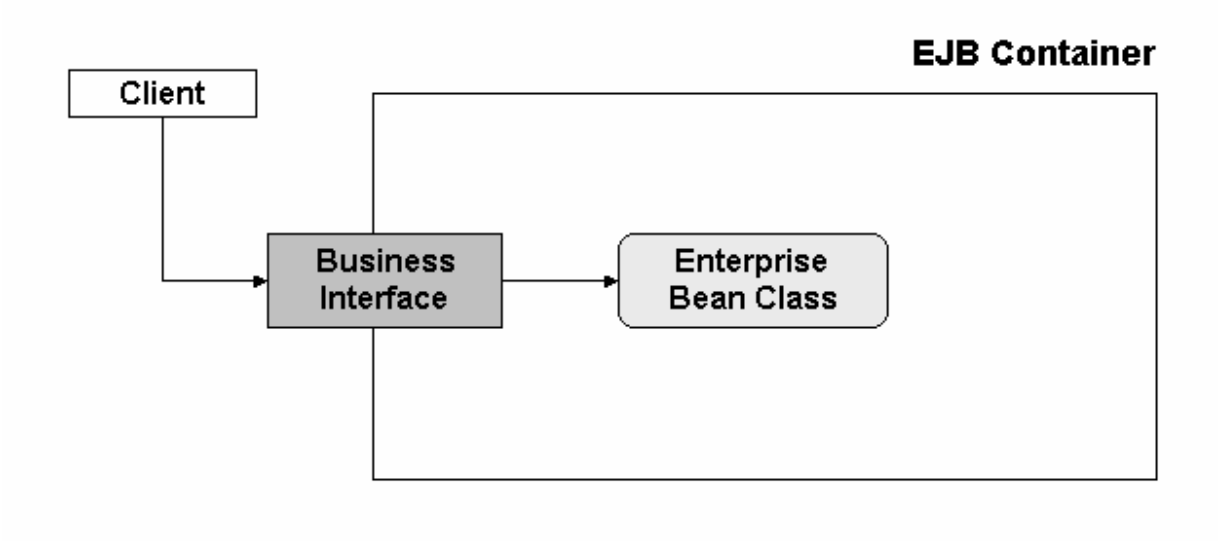
```
// EJB 3.0 client view of 2.1 bean
...
@EJB ShoppingCartHome cartHome;
Cart cart = cartHome.create();
cart.addItem(...);
...
cart.remove();
...
```

Chapter 3. EJB 3.0 Session Bean Component Contract & Lifecycle

Identify correct and incorrect statements or examples that compare the purpose and use of Stateful and Stateless Session Beans.

- [EJB_3.0_CORE] 3.1; 4.1; 4.2; 4.2.1

Client View of a Session Bean Overview



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

A client NEVER directly accesses instances of the session bean's class. A client accesses a session object through the session bean's client view interface(s).

The client of a session bean may be a local client, a remote client, or a web service client, depending on the interface provided by the bean and used by the client.

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

The interface used by a remote client of a session bean is implemented by the container as a remote business interface (or a remote EJBObject interface), and the remote client view of a session bean is location-independent. A client running in the same JVM as the session object uses the same API as a client running in a different JVM on the same or different machine.

Use of a session bean's local business interface(s) or local interface entails the collocation of the local client and the session. The local client of an enterprise bean must be collocated in the same container as the bean. The local client view IS NOT location-independent.

The client of a stateless session bean may be a web service client. Only a STATELESS session bean may provide a web service client view. A web service client makes use of the enterprise bean's web service client view, as described by a WSDL document. The bean's client view web service endpoint is in terms of a JAX-WS endpoint or JAX-RPC endpoint interface.

While it is possible to provide more than one client view for a session bean, typically only one will be provided.

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

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

Multiple enterprise beans can be installed in a container. The container allows the clients of session beans that provide local or remote client views to obtain the business interfaces and/or home interfaces of the installed enterprise beans through dependency injection or to look them up via JNDI.

The client view of a session object is independent of the implementation of the session bean and the container.

Session Bean Component Contract Overview

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

A session bean instance is an extension of the client that creates it:

- In the case of a stateful session bean, its fields contain a conversational state on behalf of the session object's client. This state describes the conversation represented by a specific client/ session object pair.
- It typically reads and updates data in a database on behalf of the client.
- In the case of a STATEFUL session bean, its lifetime is controlled by the CLIENT.

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

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

stateful session bean's persistent conversational state.

A session bean that does not make use of the Java Persistence API must explicitly manage cached database data. A session bean instance must write any cached database updates prior to a transaction completion, and it must refresh its copy of any potentially stale database data at the beginning of the next transaction. A session bean must also refresh any java.sql Statement objects before they are used in a new transaction context. Use of the Java Persistence API provides a session bean with automatic management of database data, including the automatic flushing of cached database updates upon transaction commit.

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

A session bean may be either:

- **stateless** - the session bean instances contain NO conversational STATE between methods; ANY instance can be used for any client.
- **stateful** - the session bean instances CONTAIN conversational STATE which must be retained across methods and transactions.

Conversational State of a Stateful Session Bean

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

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

In advanced cases, a session object's conversational state may contain open resources, such as open sockets and open database cursors. A container cannot retain such open resources when a session bean instance is passivated. A developer of a stateful session bean must close and open the resources in the PrePassivate and PostActivate lifecycle callback interceptor methods.

A container may only passivate a stateful session bean instance when the instance is NOT in a transaction.

A container must not passivate a stateful session bean with an extended persistence context unless the following conditions are met:

- All the entities in the persistence context are serializable.
- The EntityManager is serializable.

A STATELESS session bean is NEVER passivated.

Instance Passivation and Conversational State

The Bean Provider is required to ensure that the PrePassivate method leaves the instance fields and the fields of its associated interceptors ready to be serialized by the container. The objects

that are assigned to the instance's non-transient fields and the non-transient fields of its interceptors after the PrePassivate method completes must be one of the following.

- A serializable object.
- A null.
- A reference to an enterprise bean's business interface.
- A reference to an enterprise bean's remote interface, even if the stub class is not serializable.
- A reference to an enterprise bean's remote home interface, even if the stub class is not serializable.
- A reference to an entity bean's local interface, even if it is not serializable.
- A reference to an entity bean's local home interface, even if it is not serializable.
- A reference to the SessionContext object, even if it is not serializable.
- A reference to the environment naming context (that is, the java:comp/env JNDI context) or any of its subcontexts.
- A reference to the UserTransaction interface.
- A reference to a resource manager connection factory.
- A reference to a container-managed EntityManager object, even if it is not serializable.
- A reference to an EntityManagerFactory object obtained via injection or JNDI lookup, even if it is not serializable.
- A reference to a javax.ejb.Timer object.
- An object that is not directly serializable, but becomes serializable by replacing the references to an enterprise bean's business interface, an enterprise bean's home and component interfaces, the references to the SessionContext object, the references to the java:comp/env JNDI context and its subcontexts, the references to the UserTransaction interface, and the references to the EntityManager and/or EntityManagerFactory by serializable objects during the object's serialization.

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

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

Identify correct and incorrect statements or examples about remote and local business interfaces for Session Beans.

- [EJB_3.0_CORE] 4.6.6

Classes and Interfaces

The session Bean Provider is responsible for providing the following class files:

- Session bean class.
- Session bean's business interface(s), if the session bean provides an EJB 3.0 local or remote client view.
- Session bean's remote interface and remote home interface, if the session bean provides an EJB 2.1 remote client view.
- Session bean's local interface and local home interface, if the session bean provides an EJB 2.1 local client view.
- Session bean's web service endpoint interface, if any.
- Interceptor classes, if any.

The Bean Provider for a stateless session bean that provides a web service client view may also define JAX-WS or JAX-RPC message handlers for the bean.

Session Bean's Business Interface

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

- The interface **MUST NOT** extend the javax.ejb.EJBObject or javax.ejb.EJBLocalObject interface.
- If the business interface is a remote business interface, the argument and return values must be of valid types for RMI/IIOP. The remote business interface is NOT required or expected to be a java.rmi.Remote interface. The throws clause should NOT include the java.rmi.RemoteException. The methods of the business interface may only throw the java.rmi.RemoteException if the interface extends java.rmi.Remote.
- The interface is allowed to have superinterfaces.
- If the interface is a remote business interface, its methods MUST NOT expose local interface types, timers or timer handles, or the managed collection classes that are used for EJB 2.1 entity beans with container-managed persistence as arguments or results.
- The bean class MUST implement the interface or the interface MUST be designated as a local or remote business interface of the bean by means of the Local or Remote annotation or in the deployment descriptor. The following rules apply:

- If bean class implements a single interface, that interface is assumed to be the business interface of the bean. This business interface will be a local interface unless the interface is designated as a remote business interface by use of the Remote annotation on the bean class or interface or by means of the deployment descriptor.
- A bean class is permitted to have MORE THAN ONE interface. If a bean class has more than one interface тАФ excluding the interfaces listed below тАФ any business interface of the bean class MUST be explicitly designated as a business interface of the bean by means of the Local or Remote annotation on the bean class or interface or in the deployment descriptor.
- The following interfaces are excluded when determining whether the bean class has more than one interface: java.io.Serializable; java.io.Externalizable; any of the interfaces defined by the javax.ejb package.

- The same business interface cannot be both a local and a remote business interface of the bean.

It is also an error if the Local and/or Remote annotations are specified both on the bean class and on the referenced interface and the values differ.

This is **WRONG**:

```
import javax.ejb.Local;

@Local
public interface ProcessPayment {
	...
}

@Stateless
@Remote(ProcessPayment.class)
public class ProcessPaymentBean {
	...
}
```

- While it is expected that the bean class will typically implement its business interface(s), if the bean class uses annotations or the deployment descriptor to designate its business interface(s), it is NOT REQUIRED that the bean class also be specified as implementing the interface(s).

Write code for the bean classes of Stateful and Stateless Session Beans.

- [EJB_3.0_CORE] 4.6.2

[EJB_3.0_SIMPLIFIED] 4.1.3; 5.1.3

Session Bean Class

The following are the requirements for the session bean class:

- The class MUST be defined as public, MUST NOT be final, and MUST NOT be abstract. The class MUST be a top level class.
- The class MUST HAVE a public constructor that takes NO parameters. The container uses this constructor to create instances of the session bean class.
- The class MUST NOT define the finalize() method.
- The class must implement the bean's business interface(s) OR the methods of the bean's business interface(s), if any.
- The class must implement the business methods of the bean's EJB 2.1 client view interfaces, if any.

Optionally:

- The class may implement, directly or indirectly, the javax.ejb.SessionBean interface.
- If the class is a STATEFUL session bean, it may implement the javax.ejb.SessionSynchronization interface.
- The class may implement the session bean's web service endpoint or component interface.
- If the class is a STATELESS session bean, it may implement the javax.ejb.TimedObject interface.
- The class may implement the ejbCreate method(s).
- The session bean class may have superclasses and/or superinterfaces. If the session bean has superclasses, the business methods, lifecycle callback interceptor methods, the timeout callback method, the methods of the optional SessionSynchronization interface, the Init or ejbCreate<METHOD> methods, the Remove methods, and the methods of the SessionBean interface, may be defined in the session bean class, or in any of its superclasses. A session bean class MUST NOT have a superclass that is itself a session bean class.
- The session bean class is allowed to implement other methods (for example helper methods invoked internally by the business methods) in addition to the methods required by the EJB specification.

A STATELESS session bean MUST be annotated with the Stateless annotation or denoted in the deployment descriptor as a stateless session bean. The bean class NEED NOT implement the javax.ejb.SessionBean interface.

A STATEFUL session bean MUST be annotated with the Stateful annotation or denoted in the deployment descriptor as a stateful session bean. The bean class NEED NOT implement the javax.ejb.SessionBean interface or the java.io.Serializable interface (the container must be able

to handle the passivation of the bean instance even if the bean class does not implement the Serializable interface).

A STATEFUL session bean MAY implement the SessionSynchronization interface.

Identify correct and incorrect statements or examples about the lifecycle of a Stateful Session Bean including the @PrePassivate and @PostActivate lifecycle callback methods and @Remove methods.

- [EJB_3.0_CORE] 4.6.3; 4.3.4; 4.3.5; 3.4.3

[EJB_3.0_SIMPLIFIED] 10.1.2.2

Lifecycle Callback Interceptor Methods

PostConstruct, PreDestroy, PrePassivate, and PostActivate lifecycle callback interceptor methods may be defined for session beans. If PrePassivate or PostActivate lifecycle callbacks are defined for STATELESS session beans, they are IGNORED.

The PrePassivate and PostActivate lifecycle callback interceptor methods are only called on STATEFUL session bean instances.

The PrePassivate callback notification signals the intent of the container to passivate the instance. The PostActivate notification signals the instance it has just been reactivated. Because containers automatically maintain the conversational state of a stateful session bean instance when it is passivated, these notifications are not needed for most session beans. Their purpose is to allow stateful session beans to maintain those open resources that need to be closed prior to an instance's passivation and then reopened during an instance's activation.

For example, the Bean Provider must close all JDBC connections in the PrePassivate method and assign the instance's fields storing the connections to null.

```
@Stateful
public class SessionCalculator implements Calculator  {

      @PrePassivate
      public void serialize () {
            // close JDBC connection here
      }

        @PostActivate
      private void activate () { // 'private' Ok too
            // re-open JDBC connection here
      }
      ...
}
```

The PrePassivate and PostActivate lifecycle callback interceptor methods execute in an UNSPECIFIED transaction and security context.

If the session bean implements the SessionBean interface, the PreDestroy annotation can only

be applied to the ejbRemove method; the PostActivate annotation can only be applied to the ejbActivate method; the PrePassivate annotation can only be applied to the ejbPassivate method. Similar requirements apply to use of deployment descriptor metadata as an alternative to the use of annotations.

@Remove method

A client may remove a STATEFUL session bean by invoking a method of its business interface designated as a Remove method.

```
@Stateful
public class SessionCalculator implements Calculator {

    @Remove
    public void stopSession() {
        // Call to this method signals the container
        // to remove this bean instance and terminates
        // the session
        ...
    }
    ...
}
```

The lifecycle of a STATELESS session bean does not require that it be removed by the client. Removal of a stateless session bean instance is performed by the CONTAINER, transparently to the client.

Remove Annotation for Stateful Session Beans

The Remove annotation is used to denote a remove method of a STATEFUL session bean. Completion of this method causes the container to destroy the stateful session bean, first invoking the bean's PreDestroy method, if any. The retainIfException element allows the removal to be prevented if the method terminates abnormally with an APPLICATION exception.

```
@Target(METHOD) @Retention(RUNTIME)
public @interface Remove{
    boolean retainIfException() default false;
}
```

Given a list of methods of a Stateful or Stateless Session Bean class, define which of the following operations can be performed from each of those methods: SessionContext interface methods, UserTransaction methods, access to the java:comp/env environment naming context, resource manager access, and other enterprise bean access.

- [EJB_3.0_CORE] 4.4.1; 4.5.2

Identify correct and incorrect statements or examples about implementing a session bean as a web service endpoint, including rules for writing a web service endpoint interface and use of the @WebService and @WebMethod annotations.

- [EJB_3.0_CORE] JSR-181: Web Services Metadata for the Java

Platform. http//jcp.org/en/jsr/detail?id=181.

●Identify correct and incorrect statements or examples about the client view of a session bean, including the client view of a session object's life cycle, obtaining and using a session object, and session object identity.

- [EJB_3.0_CORE] 3.4.3; 3.4.4; 3.4.5; 3.4.5.1; 3.4.5.2

●Chapter 4. EJB 3.0 Message-Driven Bean Component Contract

●Develop code that implements a Message-Driven Bean Class.

Message-Driven Bean Class

The following are the requirements for the message-driven bean class:

- The class MUST implement, directly or indirectly, the message listener interface required by the messaging type that it supports or the methods of the message listener interface. In the case of JMS, this is the javax.jms.MessageListener interface.
- The class MUST be defined as public, MUST NOT be final, and MUST NOT be abstract. The class must be a top level class.
- The class MUST have a public constructor that takes NO arguments. The container uses this constructor to create instances of the message-driven bean class.
- The class MUST NOT define the finalize method.

Optionally:

- The class may implement, directly or indirectly, the javax.ejb.MessageDrivenBean interface.
- The class may implement, directly or indirectly, the javax.ejb.TimedObject interface.
- The class may implement the ejbCreate method.

The message-driven bean class may have superclasses and/or superinterfaces. If the message-driven bean has superclasses, the methods of the message listener interface, lifecycle callback interceptor methods, the timeout method, the ejbCreate method, and the methods of the MessageDrivenBean interface may be defined in the message-driven bean class or in any of its superclasses. A message-driven bean class MUST NOT have a superclass that is itself a message-driven bean class.

The message-driven bean class is allowed to implement other methods (for example, helper methods invoked internally by the message listener method) in addition to the methods required by the EJB specification.

●Identify correct and incorrect statements or examples about the interface(s) and methods a JMS Message-Driven bean must implement, and the required metadata.

- [EJB_3.0_CORE] 5.4.1; 5.4.2

Required MessageDrivenBean Metadata

A message-driven bean must be annotated with the MessageDriven annotation or denoted in the deployment descriptor as a message-driven bean.

```
@MessageDriven(activationConfig = {
```

```
        @ActivationConfigProperty(
                propertyName = "destinationType",
                propertyValue = "javax.jms.Topic"
        )
})
public class CalculatorMessagingBean implements MessageListener {

        public void onMessage(Message msg) {
                ...
    }
}
```

or

```
public class CalculatorMessagingBean implements MessageListener {
    public void onMessage(Message msg) {
        ...
    }
}
<ejb-jar>
        <enterprise-beans>
                <message-driven>
                        <ejb-name>CalculatorMessagingBean</ejb-name>
                        <ejb-class>by.iba.ejb.CalculatorMessagingBean</ejb-class>
                        <messaging-type>javax.jms.MessageListener</messaging-type>
                        <transaction-type>Container</transaction-type>
                        <message-destination-type>javax.jms.Topic</message-destination-type>
                        <activation-config>
                                <activation-config-property>
                                        <activation-config-property-name>destinationType</activation-config-property-name>
                                        <activation-config-property-value>javax.jms.Topic</activation-config-property-value>
                                </activation-config-property>
                        </activation-config>
                </message-driven>
        </enterprise-beans>
</ejb-jar>
```

The Required Message Listener Interface

The message-driven bean class MUST implement the appropriate MESSAGE LISTENER INTERFACE for the messaging type that the message-driven bean supports or specify the message listener interface using the MessageDriven metadata annotation or the messaging-type deployment descriptor element. The specific message listener interface that is implemented by a message-driven bean class distinguishes the messaging type that the message-driven bean supports.

The message-driven bean class's implementation of the javax.jms.MessageListener interface distinguishes the message-driven bean as a JMS message-driven bean.

The bean's message listener method (e.g., onMessage in the case of javax.jms.MessageListener) is called by the container when a message has arrived for the bean to service. The message listener method contains the business logic that handles the processing

of the message.

A bean's message listener interface may define more than one message listener method. If the message listener interface contains more than one method, it is the resource adapter that determines which method is invoked.

If the message-driven bean class implements more than one interface other than java.io.Serializable, java.io.Externalizable, or any of the interfaces defined by the javax.ejb package, the message listener interface must be specified by the messageListenerInterface element of the MessageDriven annotation or the messaging-type element of the message-driven deployment descriptor element.

●Describe the use and behavior of a JMS message driven bean, including concurrency of message processing, message redelivery, and message acknowledgement.

- [EJB_3.0_CORE] 5.4.10; 5.4.11; 5.4.14; 13.6.3; 13.6.3.1; 13.6.3.2

Serializing Message-Driven Bean Methods

The container serializes calls to each message-driven bean instance. Most containers will support many instances of a message-driven bean executing concurrently; however, each instance sees only a serialized sequence of method calls. Therefore, a message-driven bean DOES NOT have to be coded as reentrant.

The container MUST serialize all the container-invoked callbacks (e.g., lifecycle callback interceptor methods and timeout callback methods), and it must serialize these callbacks with the message listener method calls.

Concurrency of Message Processing

A container allows MANY instances of a message-driven bean class to be executing concurrently, thus allowing for the concurrent processing of a stream of messages. No guarantees are made as to the exact order in which messages are delivered to the instances of the message-driven bean class, although the container should attempt to deliver messages in order when it does not impair the concurrency of message processing. Message-driven beans should therefore be prepared to handle messages that are out of sequence: for example, the message to cancel a reservation may be delivered before the message to make the reservation.

Message Acknowledgment for JMS Message-Driven Beans

JMS message-driven beans should NOT attempt to use the JMS API for message acknowledgment. Message acknowledgment is automatically handled by the CONTAINER. If the message-driven bean uses container-managed transaction (CMT) demarcation, message acknowledgment is handled automatically as a part of the transaction commit. If bean-managed transaction (BMT) demarcation is used, the message receipt cannot be part of the bean-managed transaction, and, in this case, the receipt is acknowledged by the container. If **BEAN-MANAGED TRANSACTION** demarcation is used, the Bean Provider CAN indicate whether JMS AUTO_ACKNOWLEDGE semantics or DUPS_OK_ACKNOWLEDGE semantics should apply by using the activationConfig element of the MessageDriven annotation or by using the activation-

config-property deployment descriptor element. The property name used to specify the acknowledgment mode is acknowledgeMode. If the acknowledgeMode property is NOT specified, JMS AUTO_ACKNOWLEDGE semantics are ASSUMED. The value of the acknowledgeMode property MUST be either Auto-acknowledge or Dups-ok-acknowledge for a JMS message-driven bean.

●Identify correct and incorrect statements or examples about the client view of a message driven bean.

- [EJB_3.0_CORE] 5.3

●Chapter 5. Java Persistence API Entities

●Identify correct and incorrect statements or examples about the characteristics of Java Persistence entities.

- [EJB_3.0_PERSISTENCE] 2.1
[OREILLY_EJB_3.0] 6.1

Requirements on the Entity Class

The entity class MUST be annotated with the Entity annotation or denoted in the XML descriptor as an entity.

```
@Entity
public class Operation {
    ...
}
```

The entity class MUST have a no-arg constructor. The entity class may have other constructors as well. The no-arg constructor MUST be public OR protected.

The entity class MUST be a top-level class. An enum or interface SHOULD NOT be designated as an entity. The entity class MUST NOT be final. No methods or persistent instance variables of the entity class may be final.

If an entity instance is to be passed by value as a DETACHED object (e.g., through a remote interface), the entity class MUST implement the Serializable interface.

```
@Entity
public class Operation implements Serializable {
    ...
}
```

Entities support inheritance, polymorphic associations, and polymorphic queries.

Both abstract and concrete classes can be entities. Entities may extend non-entity classes as well as entity classes, and non-entity classes may extend entity classes.

The persistent state of an entity is represented by instance variables, which may correspond to JavaBeans properties. An instance variable may be directly accessed only from within the methods of the entity by the entity instance itself. Instance variables must not be accessed by clients of the entity. The state of the entity is available to clients only through the entity's ACCESSOR METHODS (getter/setter methods) or other business methods. Instance variables MUST be private, protected, or package visibility.

```
@Entity
public class Operation implements Serializable {

    private int id;

    private Timestamp timestamp;

    private double operandA;

    private double operandB;

    private String operation;

    private double result;

    ...
}
```

Develop code to create valid entity classes, including the use of fields and properties, admissible types, and embeddable classes.

- [EJB_3.0_PERSISTENCE] 2.1.1; 2.1.5; 9.1.34

Persistent Fields and Properties

The persistent state of an entity is accessed by the persistence provider runtime either via JavaBeans style property accessors or via instance variables. A single access type (field or property access) applies to an entity hierarchy. When annotations are used, the placement of the mapping annotations on either the persistent fields or persistent properties of the entity class specifies the access type as being either **field-** or **property-**based access respectively.

If the entity has **field-based access**, the persistence provider runtime accesses instance variables DIRECTLY. All non-transient instance variables that are not annotated with the Transient annotation are persistent. When field-based access is used, the object/relational mapping annotations for the entity class annotate the **instance variables**.

```
@Entity
public class Operation implements Serializable {

    @Id
    private long id;

    public long getId( ) {
        return id;
    }
```

```
    public void setId(long id) {
        this.id = id;
    }

    ...
}
```

If the entity has **property-based access**, the persistence provider runtime accesses persistent state via the property ACCESSOR METHODS. All properties not annotated with the Transient annotation are persistent. The property accessor methods MUST be public or protected. When property-based access is used, the object/relational mapping annotations for the entity class annotate the **getter property accessors**. These annotations MUST NOT be applied to the setter methods.

```
@Entity
public class Operation implements Serializable {

    private long id;

    @Id
    public long getId( ) {
        return id;
    }

    public void setId(long id) {
        this.id = id;
    }

    ...
}
```

Mapping annotations CANNOT be applied to fields or properties that are transient or Transient.

- The behavior is unspecified if mapping annotations are applied to both persistent fields and properties or if the XML descriptor specifies use of different access types within a class hierarchy.

It is required that the entity class follow the method signature conventions for JavaBeans read/write properties (as defined by the JavaBeans Introspector class) for persistent properties when persistent properties are used.

In this case, for every persistent property property of type Type of the entity, there is a getter method, getProperty, and setter method setProperty. For boolean properties, isProperty is an alternative name for the getter method.

For SINGLE-valued persistent properties, these method signatures are:

- Type getProperty()
- void setProperty(Type t)

COLLECTION-valued persistent fields and properties MUST be defined in terms of one of the following collection-valued interfaces regardless of whether the entity class otherwise adheres to the JavaBeans method conventions noted above and whether field or property-based access is used:

- java.util.Collection
- java.util.Set
- java.util.List
- java.util.Map

For COLLECTION-valued persistent properties, type Type MUST be one of these collection interface types in the method signatures above. Generic variants of these collection types may also be used (for example, Set<Order>).

In addition to returning and setting the persistent state of the instance, the property accessor methods may contain other business logic as well, for example, to perform validation. The persistence provider runtime executes this logic when **property-based access is used**.

Caution should be exercised in adding business logic to the accessor methods when property-based access is used. The order in which the persistence provider runtime calls these methods when loading or storing persistent state is NOT DEFINED. Logic contained in such methods therefore cannot rely upon a specific invocation order.

If property-based access is used and lazy fetching is specified, portable applications should not directly access the entity state underlying the property methods of managed instances until after it has been fetched by the persistence provider.

Runtime exceptions thrown by property accessor methods cause the current transaction to be rolled back. Exceptions thrown by such methods when used by the persistence runtime to load or store persistent state cause the persistence runtime to rollback the current transaction and to throw a PersistenceException that wraps the application exception.

Entity subclasses may override the property accessor methods. However, portable applications MUST NOT override the object/relational mapping metadata that applies to the persistent fields or properties of entity superclasses.

The persistent fields or properties of an entity MAY BE of the following types:

- Java primitive types
- java.lang.String
- other Java serializable types (including wrappers of the primitive types, java.math.BigInteger, java.math.BigDecimal, java.util.Date, java.util.Calendar, java.sql.Date, java.sql.Time, java.sql.Timestamp, user-defined serializable types, byte[], Byte[], char[], and Character[])
- enums
- entity types and/or collections of entity types
- embeddable classes

Example:

```
@Entity
public class Customer implements Serializable {
```

```
private Long id;

private String name;

private Address address;

private Collection<Order> orders = new HashSet();

private Set<PhoneNumber> phones = new HashSet();

// No-arg constructor
public Customer() {}

@Id // property access is used
public Long getId() {
	return id;
}

public void setId(Long id) {
	this.id = id;
}

public String getName() {
	return name;
}

public void setName(String name) {
	this.name = name;
}

public Address getAddress() {
	return address;
}

public void setAddress(Address address) {
	this.address = address;
}

@OneToMany
public Collection<Order> getOrders() {
	return orders;
}

public void setOrders(Collection<Order> orders) {
	this.orders = orders;
}

@ManyToMany
public Set<PhoneNumber> getPhones() {
	return phones;
}

public void setPhones(Set<PhoneNumber> phones) {
	this.phones = phones;
}
```

```
    // Business method to add a phone number to the customer
    public void addPhone(PhoneNumber phone) {
        this.getPhones().add(phone);

        // Update the phone entity instance to refer to this customer
        phone.addCustomer(this);
    }
}
```

Embeddable Classes

An entity may use other fine-grained classes to represent entity state. Instances of these classes, unlike entity instances themselves, do not have persistent identity. Instead, they exist only as embedded objects of the entity to which they belong. Such embedded objects belong strictly to their owning entity, and are not sharable across persistent entities. Attempting to share an embedded object across entities has undefined semantics. Because these objects have no persistent identity, they are typically mapped together with the entity instance to which they belong.

Embeddable classes must adhere to the requirements specified in the Requirements on the Entity Class section for entities with the exception that embeddable classes are NOT annotated as Entity. Embeddable classes MUST be annotated as Embeddable or denoted in the XML descriptor as such. The access type for an embedded object is determined by the access type of the entity in which it is embedded. Support for only one level of embedding is required by EJB 3.0 specification.

Embeddable Annotation

The Embeddable annotation is used to specify a class whose instances are stored as an intrinsic part of an owning entity and share the identity of the entity. Each of the persistent properties or fields of the embedded object is mapped to the database table for the entity. Only Basic, Column, Lob, Temporal, and Enumerated mapping annotations may portably be used to map the persistent fields or properties of classes annotated as Embeddable.

```
@Target({TYPE}) @Retention(RUNTIME)
public @interface Embeddable {
}
```

Example:

```
@Embeddable
public class EmploymentPeriod {

    java.util.Date startDate;

    java.util.Date endDate;

    ...
}
```

Embedded Annotation

The Embedded annotation is used to specify a persistent field or property of an entity whose value is an instance of an embeddable class.

The AttributeOverride and/or AttributeOverrides annotations may be used to override the column mappings declared within the embeddable class, which are mapped to the entity table.

Implementations are NOT required to support embedded objects that are mapped across more than one table (e.g., split across primary and secondary tables or multiple secondary tables).

```
@Target({METHOD, FIELD}) @Retention(RUNTIME)
public @interface Embedded {}
```

Example below shows how to use this annotation to specify that @Embeddable class EmploymentPeriod may be embedded in the entity class:

```
@Entity
public class Employee implements Serializable {

   ...

      @Embedded
      @AttributeOverrides({
            @AttributeOverride(name="startDate", column=@Column("EMP_START")),
            @AttributeOverride(name="endDate", column=@Column("EMP_END"))
      })
      public EmploymentPeriod getEmploymentPeriod() {
            ...
      }

      ...
}
```

●Identify correct and incorrect statements or examples about primary keys and entity identity, including the use of compound primary keys.

- [EJB_3.0_PERSISTENCE] 2.1.4; 9.1.8; 9.1.14; 9.1.15
  [OREILLY_EJB_3.0] 6.3

Primary Keys and Entity Identity

Every entity MUST HAVE a **primary key**.

The primary key MUST be defined on the entity that is the root of the entity hierarchy or on a mapped superclass of the entity hierarchy. The primary key MUST be defined EXACTLY ONCE in an entity hierarchy.

A SIMPLE (i.e., non-composite) primary key must correspond to a **single persistent field or property** of the entity class. The Id annotation is used to denote a simple primary key.

Id Annotation

The Id annotation specifies the primary key property or field of an entity. The Id annotation may be applied in an entity or mapped superclass.

By default, the mapped column for the primary key of the entity is assumed to be the primary key of the primary table. If no Column annotation is specified, the primary key column name is assumed to be the name of the primary key property or field.

```
@Target({METHOD, FIELD}) @Retention(RUNTIME)
public @interface Id {}
```

Example:

```
@Id
public Long getId() {
    return id;
}
```

A COMPOSITE primary key MUST correspond to either a single persistent field or property OR to a set of such fields or properties as described below. A **primary key class** MUST be defined to represent a composite primary key. Composite primary keys typically arise when mapping from legacy databases when the database key is comprised of several columns. The EmbeddedId and IdClass annotations are used to denote **composite primary keys**.

The primary key (or field or property of a composite primary key) should be one of the following types:

- any Java primitive type
- any primitive wrapper type
- java.lang.String
- java.util.Date
- java.sql.Date

In general, however, approximate numeric types (e.g., floating point types) should NEVER be used in primary keys. Entities whose primary keys use types other than these will NOT be portable. If generated primary keys are used, ONLY INTEGRAL types will be portable. If java.util.Date is used as a primary key field or property, the temporal type should be specified as DATE.

The access type (**field-** or **property-based** access) of a primary key class is determined by the access type of the entity for which it is the primary key.

The following rules apply for **composite primary keys**:

- The primary key class MUST be public and MUST HAVE a public no-arg constructor.
- If property-based access is used, the properties of the primary key class MUST be public or protected.
- The primary key class must be Serializable.
- The primary key class MUST define equals and hashCode methods. The semantics of value equality for these methods must be consistent with the database equality for the database types to which the key is mapped.

- A composite primary key MUST either be represented and mapped as an embeddable class or must be represented and mapped to multiple fields or properties of the entity class.
- If the composite primary key class is mapped to multiple fields or properties of the entity class, the names of primary key fields or properties in the primary key class and those of the entity class MUST correspond and their types MUST be the same.

The application MUST NOT CHANGE the value of the primary key. The behavior is undefined if this occurs.

EmbeddedId Annotation

Use the @EmbeddedId annotation to specify an embeddable composite primary key class (usually made up of two or more primitive or Java object types) owned by the entity. Composite primary keys typically arise when mapping from legacy databases when the database key is comprised of several columns.

The EmbeddedId annotation is applied to a persistent field or property of an entity class or mapped superclass to denote a **composite primary key** that is an embeddable class. The embeddable class must be annotated as Embeddable (note that the Id annotation is NOT used in the embeddable class).

There must be ONLY ONE EmbeddedId annotation and NO Id annotation when the EmbeddedId annotation is used.

```
@Target({METHOD, FIELD}) @Retention(RUNTIME)
public @interface EmbeddedId {}
```

Example:

```
@EmbeddedId
protected EmployeePK empPK;
```

Example below shows a typical composite primary key class, annotated as @Embeddable:

```
@Embeddable
public class EmployeePK implements Serializable {

    private String name;
    private long id;

    public EmployeePK() {}

    public String getName(){
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
```

```
    public long getId() {
        return id;
    }

    public void setId(long id) {
        this.id = id;
    }

    public int hashCode() {
        return (int) name.hashCode() + id;
    }

    public boolean equals(Object obj) {
        if (obj == this) return true;
        if (!(obj instanceof EmployeePK)) return false;
        if (obj == null) return false;
        EmployeePK pk = (EmployeePK) obj;
        return pk.id == id && pk.name.equals(name);
    }
}
```

Example below shows how to configure an entity with this embeddable composite primary key class using the @EmbeddedId annotation:

```
@Entity
public class Employee implements Serializable {

    EmployeePK primaryKey;

    public Employee() {}

    @EmbeddedId
    public EmployeePK getPrimaryKey() {
        return primaryKey;
    }

    public void setPrimaryKey(EmployeePK pk) {
        primaryKey = pk;
    }

    ...
}
```

IdClass Annotation

The IdClass annotation is applied to an entity class or a mapped superclass to specify a composite primary key class that is mapped to multiple fields or properties of the entity.

The names of the fields or properties in the primary key class and the primary key fields or properties of the entity MUST CORRESPOND and their TYPES MUST BE THE SAME.

The Id annotation MUST also be applied to the corresponding fields or properties of the entity.

```
@Target({TYPE}) @Retention(RUNTIME)
public @interface IdClass {
	Class value();
}
```

Example:

```
@Entity
@IdClass(com.acme.EmployeePK.class)
public class Employee {
	@Id String empName;
	@Id Date birthDay;
	...

}
```

Example below shows a non-embedded composite primary key class (fields empName and birthDay MUST correspond in name and type to properties in the entity class):

```
public class EmployeePK implements Serializable {

	private String empName;
	private Date birthDay;

	public EmployeePK() {}

	public String getName() {
		return empName;
	}

	public void setName(String name) {
		empName = name;
	}

	public long getDateOfBirth() {
		return birthDay;
	}

	public void setDateOfBirth(Date date) {
		birthDay = date;
	}

	public int hashCode() {
		return (int) empName.hashCode();
	}

	public boolean equals(Object obj) {
		if (obj == this) return true;
		if (!(obj instanceof EmployeePK)) return false;
		if (obj == null) return false;
		EmployeePK pk = (EmployeePK) obj;
		return pk.birthDay == birthDay && pk.empName.equals(empName);
	}
}
```

Example below shows how to configure a JPA entity with this non-embedded composite primary key class using the @IdClass annotation. NOTE: Because entity class fields empName and

birthDay are used in the primary key, you MUST also annotate them using the @Id annotation:

```
@IdClass(EmployeePK.class)
@Entity
public class Employee {

    @Id String empName;

    @Id Date birthDay;

    ...
}
```

●Implement association relationships using persistence entities, including the following associations: bidirectional for @OneToOne, @ManyToOne, @OneToMany, and @ManyToMany; unidirectional for @OneToOne, @ManyToOne, @OneToMany, and @ManyToMany.

- [EJB_3.0_PERSISTENCE] 2.1.8.1; 2.1.8.2; 2.1.8.4; 2.1.8.3; 2.1.8.3.1; 2.1.8.3.2; 2.1.8.5; 2.1.8.5.1; 2.1.8.5.2

Bidirectional OneToOne Relationships

Assuming that:

- Entity A references a single instance of Entity B.
- Entity B references a single instance of Entity A.
- Entity A is specified as the owner of the relationship.

The following mapping defaults apply:

- Entity A is mapped to a table named A.
- Entity B is mapped to a table named B.
- Table A contains a foreign key to table B. The foreign key column name is formed as the concatenation of the following: **the name of the relationship property or field** of entity A; "_"; the name of the primary key column in table B. The foreign key column has the same type as the primary key of table B and there is a unique key constraint on it.

Example:

```
@Entity
public class Employee {

    private Cubicle assignedCubicle;

    @OneToOne
    public Cubicle getAssignedCubicle() {
        return assignedCubicle;
    }
```

```
        public void setAssignedCubicle(Cubicle cubicle) {
            this.assignedCubicle = cubicle;
        }

        ...
}

@Entity
public class Cubicle {

        private Employee residentEmployee;

        @OneToOne(mappedBy="assignedCubicle")
        public Employee getResidentEmployee() {
            return residentEmployee;
        }

        public void setResidentEmployee(Employee employee) {
            this.residentEmployee = employee;
        }

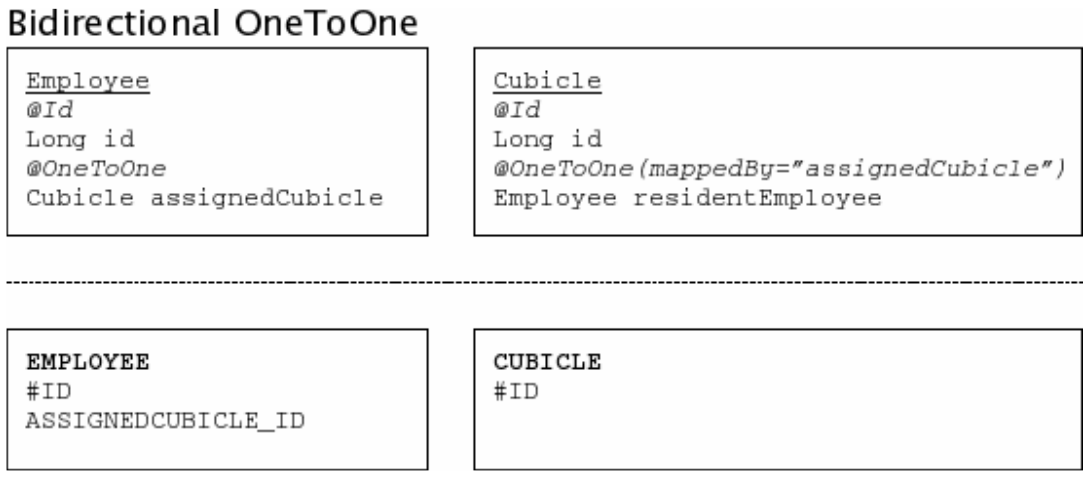
        ...
}
```

In this example:

- Entity Employee references a single instance of Entity Cubicle.
- Entity Cubicle references a single instance of Entity Employee.
- Entity Employee is the **owner of the relationship**.

The following mapping defaults apply:

- Entity Employee is mapped to a table named EMPLOYEE.
- Entity Cubicle is mapped to a table named CUBICLE.
- Table EMPLOYEE contains a foreign key to table CUBICLE. The foreign key column is named ASSIGNEDCUBICLE_<PK of CUBICLE>, where <PK of CUBICLE> denotes the name of the primary key column of table CUBICLE. The foreign key column has the same type as the primary key of CUBICLE, and there is a unique key constraint on it.



Bidirectional ManyToOne / OneToMany Relationships

Assuming that:

- Entity A references a single instance of Entity B.
- Entity B references a collection of Entity A.

Entity A must be the owner of the relationship.

The following mapping defaults apply:

- Entity A is mapped to a table named A.
- Entity B is mapped to a table named B.
- Table A contains a foreign key to table B. The foreign key column name is formed as the concatenation of the following: **the name of the relationship property or field** of entity A; "_"; the name of the primary key column in table B. The foreign key column has the same type as the primary key of table B.

Example:

```
@Entity
public class Employee {

	private Department department;

	@ManyToOne
	public Department getDepartment() {
		return department;
	}

	public void setDepartment(Department department) {
		this.department = department;
	}

	...
}

@Entity
public class Department {

	private Collection<Employee> employees = new HashSet();

	@OneToMany(mappedBy="department")
	public Collection<Employee> getEmployees() {
		return employees;
	}

	public void setEmployees(Collection<Employee> employees) {
		this.employees = employees;
	}

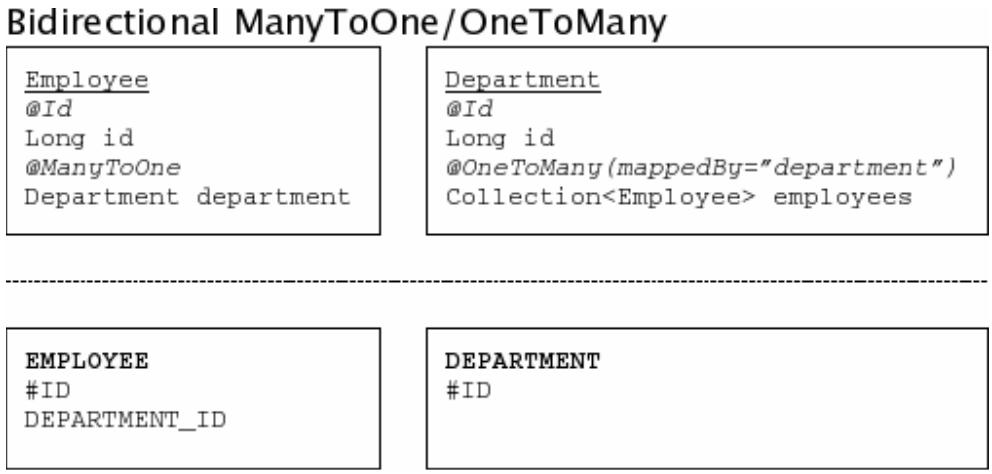
	...
}
```

In this example:

- Entity Employee references a single instance of Entity Department.
- Entity Department references a collection of Entity Employee.
- Entity Employee is the **owner** of the relationship.

The following mapping defaults apply:

- Entity Employee is mapped to a table named EMPLOYEE.
- Entity Department is mapped to a table named DEPARTMENT.
- Table EMPLOYEE contains a foreign key to table DEPARTMENT. The foreign key column is named DEPARTMENT_<PK of DEPARTMENT>, where <PK of DEPARTMENT> denotes the name of the primary key column of table DEPARTMENT. The foreign key column has the same type as the primary key of DEPARTMENT.



Bidirectional ManyToMany Relationships

Assuming that:

- Entity A references a collection of Entity B.
- Entity B references a collection of Entity A.
- Entity A is the **owner** of the relationship.

The following mapping defaults apply:

- Entity A is mapped to a table named A.
- Entity B is mapped to a table named B.
- There is a **join table** that is named A_B (owner name first). This join table has two foreign key columns. One foreign key column refers to table A and has the same type as the primary key of table A. The name of this foreign key column is formed as the concatenation of the following: **the name of the relationship property or field** of entity B; "_"; the name of the primary key column in table A. The other foreign key column refers to

table B and has the same type as the primary key of table B. The name of this foreign key column is formed as the concatenation of the following: the name of the relationship property or field of entity A; "_"; the name of the primary key column in table B.

Example:

```
@Entity
public class Project {

	private Collection<Employee> employees;

	@ManyToMany
	public Collection<Employee> getEmployees() {
		return employees;
	}

	public void setEmployees(Collection<Employee> employees) {
		this.employees = employees;
	}

	...
}
@Entity
public class Employee {

	private Collection<Project> projects;

	@ManyToMany(mappedBy="employees")
	public Collection<Project> getProjects() {
		return projects;
	}

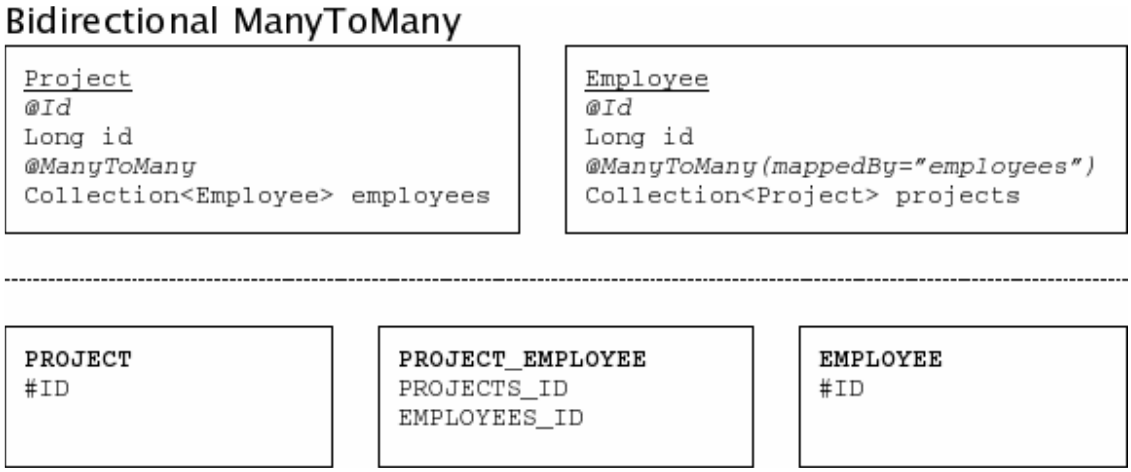
	public void setProjects(Collection<Project> projects) {
		this.projects = projects;
	}

	...
}
```

In this example:

- Entity Project references a collection of Entity Employee.
- Entity Employee references a collection of Entity Project.

- Entity Project is the **owner** of the relationship.



The following mapping defaults apply:

- Entity Project is mapped to a table named PROJECT.
- Entity Employee is mapped to a table named EMPLOYEE.
- There is a **join table** that is named PROJECT_EMPLOYEE (owner name first). This join table has two foreign key columns. One foreign key column refers to table PROJECT and has the same type as the primary key of PROJECT. The name of this foreign key column is PROJECTS_<PK of PROJECT>, where <PK of PROJECT> denotes the name of the primary key column of table PROJECT. The other foreign key column refers to table EMPLOYEE and has the same type as the primary key of EMPLOYEE. The name of this foreign key column is EMPLOYEES_<PK of EMPLOYEE>, where <PK of EMPLOYEE> denotes the name of the primary key column of table EMPLOYEE.

Unidirectional OneToOne Relationships

Assuming that:

- Entity A references a single instance of Entity B.
- Entity B does not reference Entity A.

A unidirectional relationship **has only an owning side**, which in this case must be Entity A.

The following mapping defaults apply:

- Entity A is mapped to a table named A.
- Entity B is mapped to a table named B.
- Table A contains a foreign key to table B. The foreign key column name is formed as the concatenation of the following: **the name of the relationship property or field** of entity A; "_"; the name of the primary key column in table B. The foreign key column has the same type as the primary key of table B and there is a unique key constraint on it.

Example:

@Entity

```
public class Employee {

	private TravelProfile profile;

	@OneToOne
	public TravelProfile getProfile() {
		return profile;
	}

	public void setProfile(TravelProfile profile) {
		this.profile = profile;
	}

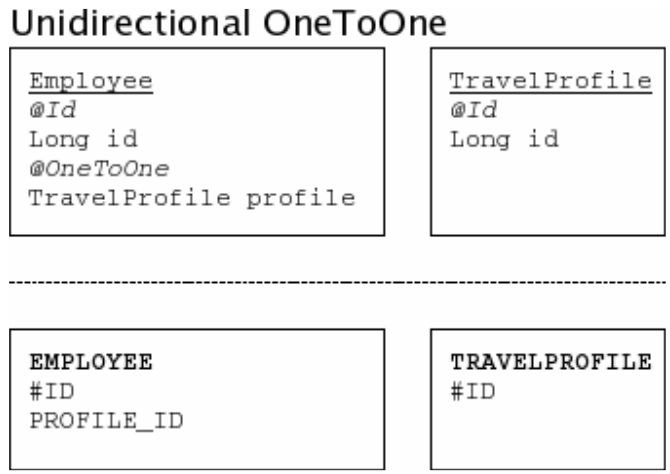
	...
}
@Entity
public class TravelProfile {
	...
}
```

In this example:

- Entity Employee references a single instance of Entity TravelProfile.
- Entity TravelProfile does not reference Entity Employee.
- Entity Employee is the **owner of the relationship**.

The following mapping defaults apply:

- Entity Employee is mapped to a table named EMPLOYEE.
- Entity TravelProfile is mapped to a table named TRAVELPROFILE.
- Table EMPLOYEE contains a foreign key to table TRAVELPROFILE. The foreign key column is named PROFILE_<PK of TRAVELPROFILE>, where <PK of TRAVELPROFILE> denotes the name of the primary key column of table TRAVELPROFILE. The foreign key column has the same type as the primary key of TRAVELPROFILE, and there is a unique key constraint on it.



Unidirectional ManyToOne Relationships

Assuming that:

- Entity A references a single instance of Entity B.
- Entity B does not reference Entity A.

A unidirectional relationship **has only an owning side**, which in this case must be Entity A.

The following mapping defaults apply:

- Entity A is mapped to a table named A.
- Entity B is mapped to a table named B.
- Table A contains a foreign key to table B. The foreign key column name is formed as the concatenation of the following: the **name of the relationship property or field** of entity A; "_"; the name of the primary key column in table B. The foreign key column has the same type as the primary key of table B.

Example:

```
@Entity
public class Employee {

	private Address address;

	@ManyToOne
	public Address getAddress() {
		return address;
	}

	public void setAddress(Address address) {
		this.address = address;
	}

	...
}


@Entity
public class Address {
	...
}
```

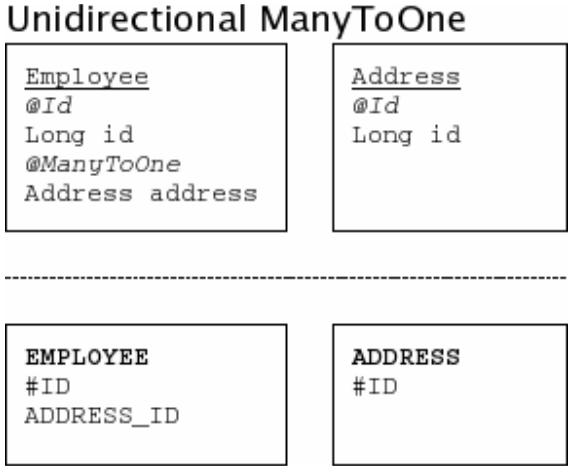
In this example:

- Entity Employee references a single instance of Entity Address.
- Entity Address does NOT reference Entity Employee.
- Entity Employee is the **owner** of the relationship.

The following mapping defaults apply:

- Entity Employee is mapped to a table named EMPLOYEE.
- Entity Address is mapped to a table named ADDRESS.

- Table EMPLOYEE contains a foreign key to table ADDRESS. The foreign key column is named ADDRESS_<PK of ADDRESS>, where <PK of ADDRESS> denotes the name of the primary key column of table ADDRESS. The foreign key column has the same type as the primary key of ADDRESS.



Unidirectional OneToMany Relationships

Assuming that:

- Entity A references a collection of Entity B.
- Entity B does not reference Entity A.

A UNIDIRECTIONAL relationship **has only an owning side**, which in this case must be Entity A.

The following mapping defaults apply:

- Entity A is mapped to a table named A.
- Entity B is mapped to a table named B.
- There is a **join table** that is named A_B (owner name first). This join table has two foreign key columns. One foreign key column refers to table A and has the same type as the primary key of table A. The name of this foreign key column is formed as the concatenation of the following: the name of entity A; "_"; the name of the primary key column in table A. The other foreign key column refers to table B and has the same type as the primary key of table B and there is a unique key constraint on it. The name of this foreign key column is formed as the concatenation of the following: the **name of the relationship property or field** of entity A; "_"; the name of the primary key column in table B.

Example:

```
@Entity
public class Employee {

    private Collection<AnnualReview> annualReviews;
```

```
	@OneToMany
	public Collection<AnnualReview> getAnnualReviews() {
		return annualReviews;
	}

	public void setAnnualReviews(Collection<AnnualReview> annualReviews) {
		this.annualReviews = annualReviews;
	}

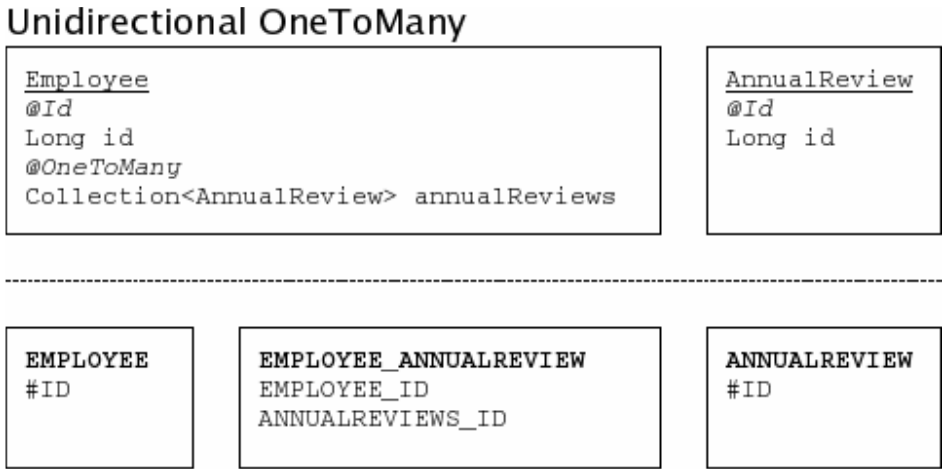
	...
}
@Entity
public class AnnualReview {
	...
}
```

In this example:

- Entity Employee references a collection of Entity AnnualReview.
- Entity AnnualReview does NOT reference Entity Employee.
- Entity Employee is the **owner** of the relationship.

The following mapping defaults apply:

- Entity Employee is mapped to a table named EMPLOYEE.
- Entity AnnualReview is mapped to a table named ANNUALREVIEW.
- There is a **join table** that is named EMPLOYEE_ANNUALREVIEW (owner name first). This join table has **two foreign key columns**. One foreign key column refers to table EMPLOYEE and has the same type as the primary key of EMPLOYEE. This foreign key column is named EMPLOYEE_<PK of EMPLOYEE>, where <PK of EMPLOYEE> denotes the name of the primary key column of table EMPLOYEE. The other foreign key column refers to table ANNUALREVIEW and has the same type as the primary key of ANNUALREVIEW. This foreign key column is named ANNUALREVIEWS_<PK of ANNUALREVIEW>, where <PK of ANNUALREVIEW> denotes the name of the primary key column of table ANNUALREVIEW. There is a **unique key constraint** on the foreign key that refers to table ANNUALREVIEW.



Unidirectional ManyToMany Relationships

Assuming that:

- Entity A references a collection of Entity B.
- Entity B does not reference Entity A.

A UNIDIRECTIONAL relationship **has only an owning side**, which in this case must be Entity A.

The following mapping defaults apply:

- Entity A is mapped to a table named A.
- Entity B is mapped to a table named B.
- There is a **join table** that is named A_B (owner name first). This join table has two foreign key columns. One foreign key column refers to table A and has the same type as the primary key of table A. The name of this foreign key column is formed as the concatenation of the following: the name of entity A; "_"; the name of the primary key column in table A. The other foreign key column refers to table B and has the same type as the primary key of table B. The name of this foreign key column is formed as the concatenation of the following: the name of the relationship property or field of entity A; "_"; the name of the primary key column in table B.

Example:

```
@Entity
public class Employee {

    private Collection<Patent> patents;

    @ManyToMany
    public Collection<Patent> getPatents() {
        return patents;
    }

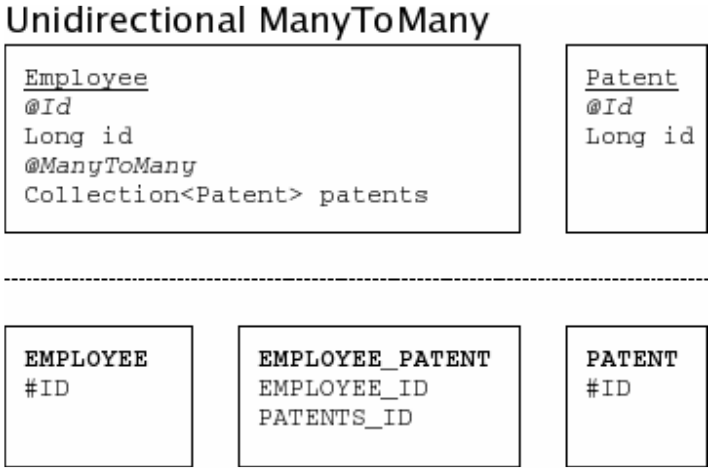
    public void setPatents(Collection<Patent> patents) {
        this.patents = patents;
```

```
    }

    ...
}
@Entity
public class Patent {
    ...
}
```

In this example:

- Entity Employee references a collection of Entity Patent.
- Entity Patent does NOT reference Entity Employee.
- Entity Employee is the **owner** of the relationship.



The following mapping defaults apply:

- Entity Employee is mapped to a table named EMPLOYEE.
- Entity Patent is mapped to a table named PATENT.
- There is a **join table** that is named EMPLOYEE_PATENT (owner name first). This join table has two foreign key columns. One foreign key column refers to table EMPLOYEE and has the same type as the primary key of EMPLOYEE. This foreign key column is named EMPLOYEE_<PK of EMPLOYEE>, where <PK of EMPLOYEE> denotes the name of the primary key column of table EMPLOYEE. The other foreign key column refers to table PATENT and has the same type as the primary key of PATENT. This foreign key column is named PATENTS_<PK of PATENT>, where <PK of PATENT> denotes the name of the primary key column of table PATENT.

- **Given a set of requirements and entity classes choose and implement an appropriate object-relational mapping for association relationships.**
  - [EJB_3.0_PERSISTENCE] 2.1.7

Relationships among entities may be one-to-one, one-to-many, many-to-one, or many-to-many.

Relationships are polymorphic.
If there is an association between two entities, one of the following relationship modeling annotations MUST be applied to the corresponding **persistent property** or **instance variable** of the referencing entity: OneToOne, OneToMany, ManyToOne, ManyToMany. For associations that do not specify the target type (e.g., where Java generic types are not used for collections), it is necessary to specify the entity that is the target of the relationship.

Relationships may be **bidirectional** or **unidirectional**. A **bidirectional** relationship has BOTH an **owning side** and an **inverse side**. A **unidirectional** relationship has ONLY an **owning side**. The **owning side** of a relationship DETERMINES the updates to the relationship in the database.

The following rules apply to **bidirectional** relationships:

- The **inverse** side of a bidirectional relationship MUST refer to its owning side by use of the mappedBy element of the OneToOne, OneToMany, or ManyToMany annotation. The mappedBy element DESIGNATES the property or field in the entity that is the **owner** of the relationship.
- The **many** side of one-to-many / many-to-one **bidirectional** relationships MUST be the owning side, hence the mappedBy element CANNOT be specified on the ManyToOne annotation.
- For one-to-one **bidirectional** relationships, the owning side corresponds to the side that contains the corresponding foreign key.
- For many-to-many **bidirectional** relationships either side may be the owning side.

The relationship modeling annotation constrains the use of the cascade=REMOVE specification. The cascade=REMOVE specification should ONLY be applied to associations that are specified as OneToOne or OneToMany. Applications that apply cascade=REMOVE to other associations are not portable.

Note that it is the application that bears responsibility for maintaining the consistency of runtime relationships - for example, for insuring that the "one" and the "many" sides of a bidirectional relationship are consistent with one another when the application updates the relationship at runtime.

If there are no associated entities for a multi-valued relationship of an entity fetched from the database, the persistence provider is responsible for returning an EMPTY COLLECTION as the value of the relationship.

Given a set of requirements and entity classes, choose and implement an appropriate inheritance hierarchy strategy and/or an appropriate mapping strategy.

- [EJB_3.0_PERSISTENCE] 2.1.9; 2.1.9.1; 2.1.9.2; 2.1.9.3; 2.1.10; 2.1.10.1; 2.1.10.2; 2.1.10.3

An entity may inherit from another entity class. Entities support inheritance, polymorphic associations, and polymorphic queries.
Both **abstract** and **concrete** classes can be entities. Both **abstract** and **concrete** classes can be annotated with the Entity annotation, mapped as entities, and queried for as entities.

Entities can extend non-entity classes and non-entity classes can extend entity classes.

Abstract Entity Classes

An abstract class CAN be specified as an entity. An abstract entity differs from a concrete entity only in that it CANNOT be directly instantiated. An abstract entity is mapped as an entity and can be the target of queries (which will operate over and/or retrieve instances of its concrete subclasses).

An abstract entity class is annotated with the Entity annotation or denoted in the XML descriptor as an entity.

The following example shows the use of an abstract entity class in the entity inheritance hierarchy:

```
@Entity
@Table(name="EMP")
@Inheritance(strategy=JOINED)
public abstract class Employee {

    @Id protected Integer empId;

    @Version protected Integer version;

    @ManyToOne protected Address address;

    ...
}
@Entity
@Table(name="FT_EMP")
@DiscriminatorValue("FT")
@PrimaryKeyJoinColumn(name="FT_EMPID")
public class FullTimeEmployee extends Employee {

    // Inherit empId, but mapped in this class to FT_EMP.FT_EMPID

    // Inherit version mapped to EMP.VERSION

    // Inherit address mapped to EMP.ADDRESS fk

    // Defaults to FT_EMP.SALARY

    protected Integer salary;

    ...
}
@Entity
@Table(name="PT_EMP")
@DiscriminatorValue("PT")
// PK field is PT_EMP.EMPID due to PrimaryKeyJoinColumn default
public class PartTimeEmployee extends Employee {

    protected Float hourlyWage;

    ...
}
```

Mapped Superclasses

An entity may inherit from a superclass that provides persistent entity state and mapping information, but which is not itself an entity. Typically, the purpose of such a mapped superclass is to define state and mapping information that is common to multiple entity classes.

A mapped superclass, unlike an entity, is NOT queryable and CANNOT be passed as an argument to EntityManager or Query operations. A mapped superclass CANNOT be the target of a persistent relationship.

Both abstract and concrete classes may be specified as mapped superclasses. The MappedSuperclass annotation (or mapped-superclass XML descriptor element) is used to designate a mapped superclass.

A class designated as MappedSuperclass has NO separate table defined for it. Its mapping information is applied to the entities that **inherit from it**.

A class designated as MappedSuperclass can be mapped in the same way as an entity except that the mappings will apply only to its subclasses since no table exists for the mapped superclass itself. When applied to the subclasses, the inherited mappings will apply in the context of the subclass tables. Mapping information can be overridden in such subclasses by using the AttributeOverride and AssociationOverride annotations or corresponding XML elements.

All other entity mapping defaults apply equally to a class designated as MappedSuperclass.

The following example illustrates the definition of a concrete class as a mapped superclass:

```
@MappedSuperclass
public class Employee {

    @Id protected Integer empId;

    @Version protected Integer version;

    @ManyToOne @JoinColumn(name="ADDR")
    protected Address address;

    public Integer getEmpId() { ... }

    public void setEmpId(Integer id) { ... }

    public Address getAddress() { ... }

    public void setAddress(Address addr) { ... }
}


// Default table is FTEMPLOYEE table
@Entity
public class FTEmployee extends Employee {

    // Inherited empId field mapped to FTEMPLOYEE.EMPID

    // Inherited version field mapped to FTEMPLOYEE.VERSION
```

```
    // Inherited address field mapped to FTEMPLOYEE.ADDR fk

    // Defaults to FTEMPLOYEE.SALARY

    protected Integer salary;

    public FTEmployee() {}

    public Integer getSalary() { ... }

    public void setSalary(Integer salary) { ... }
}


@Entity @Table(name="PT_EMP")
@AssociationOverride(name="address", joincolumns=@JoinColumn(name="ADDR_ID"))
public class PartTimeEmployee extends Employee {

    // Inherited empId field mapped to PT_EMP.EMPID

    // Inherited version field mapped to PT_EMP.VERSION

    // address field mapping overridden to PT_EMP.ADDR_ID fk

    @Column(name="WAGE")
    protected Float hourlyWage;

    public PartTimeEmployee() {}

    public Float getHourlyWage() { ... }

    public void setHourlyWage(Float wage) { ... }
}
```

Non-Entity Classes in the Entity Inheritance Hierarchy

An entity can have a non-entity superclass, which may be either a concrete or abstract class (the superclass may NOT be an embeddable class or id class).

The non-entity superclass serves for inheritance of behavior only. The state of a non-entity superclass is NOT PERSISTENT. Any state inherited from non-entity superclasses is non-persistent in an inheriting entity class. This non-persistent state is NOT managed by the EntityManager. Any annotations on such superclasses are IGNORED.

Non-entity classes CANNOT be passed as arguments to methods of the EntityManager or Query interfaces and CANNOT bear mapping information.

The following example illustrates the use of a non-entity class as a superclass of an entity:

```
public class Cart {

    // This state is transient
    Integer operationCount;
```

```
    public Cart() { operationCount = 0; }

    public Integer getOperationCount() { return operationCount; }

    public void incrementOperationCount() { operationCount++; }
}


@Entity
public class ShoppingCart extends Cart {

    Collection<Item> items = new Vector<Item>();

    public ShoppingCart() { super(); }

    ...

    @OneToMany
    public Collection<Item> getItems() { return items; }

    public void addItem(Item item) {
        items.add(item);
        incrementOperationCount();
    }
}
```

Inheritance Mapping Strategies

The mapping of class hierarchies is specified through metadata.

There are three basic strategies that are used when mapping a class or class hierarchy to a relational database:

1 **Single Table per Class Hierarchy Strategy**

In this strategy, all the classes in a hierarchy are mapped to a **single table**. The table has a column that serves as a "discriminator column", that is, a column whose value identifies the specific subclass to which the instance that is represented by the row belongs.

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

It has the drawback, however, that it requires that the columns that correspond to state specific to the subclasses be NULLABLE.

Example of Single Table Mapping

```
@Entity
@Table(name="SUB", schema="CNTRCT")
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
public class Subscription {
    ...
}
```

```
@Entity(name="Lifetime")
public class LifetimeSubscription extends Subscription {
	...
}
```

The same metadata expressed in XML form:

```
<entity class="Subscription">
	<table name="SUB" schema="CNTRCT"/>
	<inheritance strategy="SINGLE_TABLE"/>
	...
</entity>

<entity class="LifetimeSubscription">
	...
</entity>
```

Single table inheritance is the **default strategy**. Thus, we could omit the @Inheritance annotation in the example above and get the same result.

Advantages

Single table inheritance mapping is the fastest of all inheritance models, since it never requires a join to retrieve a persistent instance from the database. Similarly, persisting or updating a persistent instance requires only a single INSERT or UPDATE statement. Finally, relations to any class within a single table inheritance hierarchy are just as efficient as relations to a base class.

Disadvantages

The larger the inheritance model gets, the "wider" the mapped table gets, in that for every field in the entire inheritance hierarchy, a column must exist in the mapped table. This may have undesirable consequence on the database size, since a wide or deep inheritance hierarchy will result in tables with many mostly-empty columns.

2 Table per Concrete Class Strategy

In this mapping strategy, each class is mapped to a separate table. All properties of the class, including inherited properties, are mapped to columns of the table for the class.

This strategy has the following drawbacks:

- It provides POOR support for polymorphic relationships.
- It typically requires that SQL UNION queries (or a separate SQL query per subclass) be issued for queries that are intended to range over the class hierarchy.

NOTE: Support for the table per concrete class inheritance mapping strategy is optional in EJB 3.0.

Example of the Table Per Class Mapping:

```
@Entity
```

```
@Table(name="MAG")
@Inheritance(strategy=InheritanceType.TABLE_PER_CLASS)
public class Magazine {
	...
}


@Entity
@Table(name="TABLOID")
public class Tabloid extends Magazine {
	...
}
```

The same metadata expressed in XML form:

```
<entity class="Magazine">
	<table name="MAG"/>
	<inheritance strategy="TABLE_PER_CLASS"/>
	...
</entity>

<entity class="Tabloid">
	<table name="TABLOID"/>
	...
</entity>
```

Advantages

The table-per-class strategy is very efficient when operating on instances of a known class. Under these conditions, the strategy never requires joining to superclass or subclass tables. Reads, joins, inserts, updates, and deletes are all efficient in the absence of polymorphic behavior. Also, as in the joined strategy, adding additional classes to the hierarchy does not require modifying existing class tables.

Disadvantages

Polymorphic relations to non-leaf classes in a table-per-class hierarchy have many limitations. When the concrete subclass is not known, the related object could be in any of the subclass tables, making joins through the relation impossible. This ambiguity also affects identity lookups and queries; these operations require multiple SQL SELECTs (one for each possible subclass), or a complex UNION.

3 Joined Subclass Strategy

In the joined subclass strategy, the root of the class hierarchy is represented by a single table. Each subclass is represented by a separate table that contains those fields that are specific to the subclass (not inherited from its superclass), as well as the column(s) that represent its primary key. The primary key column(s) of the subclass table serves as a foreign key to the primary key of the superclass table.

This strategy provides support for polymorphic relationships between entities.

It has the drawback that it requires that one or more join operations be performed to instantiate instances of a subclass. In deep class hierarchies, this may lead to unacceptable performance. Queries that range over the class hierarchy likewise require joins.

Example of Joined Subclass Tables:

```
@Entity
@Table(schema="CNTRCT")
@Inheritance(strategy=InheritanceType.JOINED)
public class Contract extends Document {
	...
}


@Entity
@Table(name="LINE_ITEM", schema="CNTRCT")
@PrimaryKeyJoinColumn(name="ID", referencedColumnName="PK")
public class LineItem extends Contract {
	...
}
```

The same metadata expressed in XML form:

```
<entity class="Contract">
	<table schema="CNTRCT"/>
	<inheritance strategy="JOINED"/>
	...
</entity>

<entity class="LineItem">
	<table name="LINE_ITEM" schema="CNTRCT"/>
	<primary-key-join-column name="ID" referenced-column-name="PK"/>
	...
</entity>
```

Advantages

Using joined subclass tables results in the most *normalized* database schema, meaning the schema with the least spurious or redundant data.

As more subclasses are added to the data model over time, the only schema modification that needs to be made is the addition of corresponding subclass tables in the database (rather than having to change the structure of existing tables).

Relations to a base class using this strategy can be loaded through standard joins and can use standard foreign keys, as opposed to the machinations required to load polymorphic relations to table-per-class base types, described below.

Disadvantages

Aside from certain uses of the table-per-class strategy, the joined strategy is often the slowest of the inheritance models. Retrieving any subclass requires one or more database

joins, and storing subclasses requires multiple INSERT or UPDATE statements. This is only the case when persistence operations are performed on subclasses; if most operations are performed on the least-derived persistent superclass, then this mapping is very fast.

●Describe the use of annotations and XML mapping files, individually and in combination, for object-relational mapping.

- [EJB_3.0_PERSISTENCE] 9.1.1; 9.1.4; 9.1.8; 9.1.15; 9.1.9; 9.1.17; 9.1.18; 9.1.19; 9.1.20; 9.1.21; 9.1.16

Persistence metadata is specified using either the Java 5 annotations defined in the javax.persistence package, XML mapping files, or a mixture of both. In the latter case, XML declarations override conflicting annotations. If you choose to use XML metadata, the XML files must be available at development and runtime, and must be discoverable via either of two strategies:

1. In a resource named orm.xml placed in a META-INF directory within a directory in your classpath or within a jar archive containing your persistent classes.
2. Declared in your persistence.xml configuration file. In this case, each XML metadata file must be listed in a mapping-file element whose content is either a path to the given file or a resource location available to the class' class loader.

Table Annotation

The Table annotation specifies the primary table for the annotated entity. Additional tables may be specified using SecondaryTable or SecondaryTables annotation.

```
@Target({TYPE}) @Retention(RUNTIME)
public @interface Table {
    String name() default "";
    String catalog() default "";
    String schema() default "";
    UniqueConstraint[] uniqueConstraints() default {};
}
```

If you omit the Table annotation, base entity classes default to a table with their **Entity name**. The default table of an entity subclass depends on the inheritance strategy.

Example of Mapping Classes:

```
@Entity
@Table(name="SUB", schema="CNTRCT")
public class Subscription {
    ...
}

@Entity(name="Lifetime")
public class LifetimeSubscription extends Subscription {
    ...
}
```

```
@MappedSuperclass
public abstract class Document {
	...
}

@Embeddable
public class Address {
	...
}
```

The same mapping information expressed in XML:

```
<entity-mappings xmlns="http://java.sun.com/xml/ns/persistence/orm"
	xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
	xsi:schemaLocation="http://java.sun.com/xml/ns/persistence/orm orm_1_0.xsd"
	version="1.0">

	<entity class="org.mag.subcribe.Subscription">
		<table name="SUB" schema="CNTRCT"/>
		...
	</entity>

	<entity class="LifetimeSubscription" name="Lifetime">
		...
	</entity>

	<mapped-superclass class="Document">
		...
	</mapped-superclass>

	<embeddable class="Address">
		...
	</embeddable>
</entity-mappings>
```

UniqueConstraint Annotation

The UniqueConstraint annotation is used to specify that a unique constraint is to be included in the generated DDL for a primary or secondary table.

```
@Target({}) @Retention(RUNTIME)
public @interface UniqueConstraint {
	String[] columnNames();
}
```

Unique constraints ensure that the data in a column or combination of columns is unique for each row. A table's primary key, for example, functions as an implicit unique constraint. In JPA, you represent other unique constraints with an array of UniqueConstraint annotations within the table annotation. The unique constraints you define are used during table creation to generate the proper database constraints, and may also be used at runtime to order INSERT, UPDATE, and DELETE statements. For example, suppose there is a unique constraint on the columns of field F. In the same transaction, you remove an object A and persist a new object B, both with the same F value. The JPA runtime must ensure that the SQL deleting A is sent to the database before the SQL inserting B to avoid a unique constraint violation.

UniqueConstraint has a single property: String[] columnNames - the names of the columns the constraint spans. Note that the columnNames array refers to the LOGICAL column names.

In XML, unique constraints are represented by nesting unique-constraint elements within the table element. Each unique-constraint element in turn nests column-name text elements to enumerate the contraint's columns.

The following defines a unique constraint on the TITLE column of the ART table:

```
@Entity
@Table(name="ART", uniqueConstraints=@UniqueConstraint(columnNames={"TITLE"}))
public class Article {
	...
}
```

The same metadata expressed in XML form:

```
<entity class="Article">
	<table name="ART">
		<unique-constraint>
			<column-name>TITLE</column-name>
		</unique-constraint>
	</table>
	...
</entity>
```

Column Annotation

The Column annotation is used to specify a mapped column for a persistent property or field.

```
@Target({METHOD, FIELD}) @Retention(RUNTIME)
public @interface Column {
	String name() default "";
	boolean unique() default false;
	boolean nullable() default true;
	boolean insertable() default true;
	boolean updatable() default true;
	String columnDefinition() default "";
	String table() default "";
	int length() default 255;
	int precision() default 0; // decimal precision
	int scale() default 0; // decimal scale
}
```

The column(s) used for a property mapping can be defined using the @Column annotation. Use it to override default values (for example: the name of the column is the same as the property or field name). You can use this annotation for properties or fields that are:

- not annotated at all
- annotated with @Basic
- annotated with @Version
- annotated with @Lob
- annotated with @Temporal
- annotated with @Id

Example below shows the name property mapped to the flight_name column, which is not

nullable, has a length of 50 and is not updatable (making the property immutable):

```
@Entity
public class Flight implements Serializable {

	@Column(updatable = false, name = "flight_name", nullable = false, length=50)
	public String getName() {
		...
	}
}
```

Example:

```
@Column(name="DESC", nullable=false, length=512)
public String getDescription() {
	return description;
}
```

Example:

```
@Column(name="DESC", columnDefinition="CLOB NOT NULL", table="EMP_DETAIL")
@Lob
public String getDescription() {
	return description;
}
```

Example:

```
@Column(name="ORDER_COST", updatable=false, precision=12, scale=2)
public BigDecimal getCost() {
	return cost;
}
```

The equivalent XML element is column. This element has attributes that are exactly equivalent to the Column annotation's properties.

Id Annotation

The Id annotation specifies the primary key property or field of an entity. The Id annotation may be applied in an entity or mapped superclass.

By default, the mapped column for the primary key of the entity is assumed to be the primary key of the primary table. If no Column annotation is specified, the primary key column name is assumed to be the name of the primary key property or field.

```
@Target({METHOD, FIELD}) @Retention(RUNTIME)
public @interface Id {}
```

The equivalent XML element is id. It has one required attribute:

name: The name of the identity field or property.

Example:

```
@Entity
@Table(name="COMP")
public class Company {

	@Column(name="CID")
	@Id private long id;
```

```
    ...
}
```

The same metadata for Company expressed in XML form:

```
<entity class="org.mag.pub.Company">
    <table name="COMP"/>
    <attributes>
        <id name="id">
            <column name="CID"/>
        </id>
        ...
    </attributes>
</entity>
```

IdClass Annotation

The IdClass annotation is applied to an entity class or a mapped superclass to specify a **composite primary key class** that is MAPPED to **multiple fields or properties of the entity**.

The names of the fields or properties in the primary key class and the primary key fields or properties of the entity MUST correspond and their types MUST be the SAME.

The Id annotation must also be applied to the corresponding fields or properties of the entity.

```
@Target({TYPE}) @Retention(RUNTIME)
public @interface IdClass {
    Class value();
}
```

Example - Magazine.isbn field is mapped to a VARCHAR(9) column instead of a VARCHAR(255) column, which is the default for String fields.

```
@Entity
@IdClass(MagazineId.class)
@Table(name="MAG")
public class Magazine {

    @Column(length=9)
    @Id private String isbn;
    @Id private String title;

    ...
}
public class MagazineId {

    // Each identity field in the Magazine class must have a
    // corresponding field in the identity class

    public String isbn;
    public String title;

    public boolean equals(Object other) {
        ...
    }

    public int hashCode() {
        ...
    }
}
```

The same metadata for Magazine expressed in XML form:

```
<entity class="Magazine">
	<id-class class="MagazineId"/>
	<table name="MAG"/>
	<attributes>
		<id name="isbn">
			<column length="9"/>
		</id>
		<id name="title"/>
		...
	</attributes>
</entity>
```

GeneratedValue Annotation

The GeneratedValue annotation provides for the specification of generation strategies for the values of primary keys. The GeneratedValue annotation may be applied to a primary key property or field of an entity or mapped superclass in conjunction with the Id annotation.

The types of primary key generation are defined by the GenerationType enum:

```
public enum GenerationType {
	TABLE,
	SEQUENCE,
	IDENTITY,
	AUTO
};
```

The TABLE generator type value indicates that the persistence provider must assign primary keys for the entity using an **underlying database table** to ensure uniqueness.

The SEQUENCE and IDENTITY values specify the use of a **database sequence or identity column**, respectively.

The AUTO value indicates that the persistence provider should pick an appropriate strategy for the particular database. The AUTO generation strategy may expect a database resource to exist, or it may attempt to create one. A vendor may provide documentation on how to create such resources in the event that it does not support schema generation or cannot create the schema resource at runtime.

EJB 3.0 specification does not define the exact behavior of these strategies.

```
@Target({METHOD, FIELD}) @Retention(RUNTIME)
public @interface GeneratedValue {
	GenerationType strategy() default AUTO;
	String generator() default "";
}
```

Example 1:

```
@Entity
@Table(name="ART", uniqueConstraints=@UniqueConstraint(columnNames={"TITLE"}))
@SequenceGenerator(name="ArticleSeq", sequenceName="ART_SEQ")
public class Article {

	@Id
	@GeneratedValue(strategy=GenerationType.SEQUENCE, generator="ArticleSeq")
	private long id;
```

```
    ...
}
```

The same metadata for Article expressed in XML form:

```
<entity class="Article">
    <table name="ART">
        <unique-constraint>
            <column-name>TITLE</column-name>
        </unique-constraint>
    </table>
    <sequence-generator name="ArticleSeq" sequence-name="ART_SEQ"/>
    <attributes>
        <id name="id">
            <generated-value strategy="SEQUENCE" generator="ArticleSeq"/>
        </id>
        ...
    </attributes>
</entity>
```

Example 2:

```
@Entity
@Table(name="AUTH")
public class Author {

    @Id
    @GeneratedValue(strategy=GenerationType.TABLE, generator="AuthorGen")
    @TableGenerator(name="AuthorGen", table="AUTH_GEN", pkColumnName="PK",
        valueColumnName="AID")
    @Column(name="AID", columnDefinition="INTEGER64")
    private long id;

    ...
}
```

The same metadata for Author expressed in XML form:

```
<entity class="Author">
    <table name="AUTH"/>
    <attributes>
        <id name="id">
            <column name="AID" column-definition="INTEGER64"/>
            <generated-value strategy="TABLE" generator="AuthorGen"/>
            <table-generator name="AuthorGen" table="AUTH_GEN"
                pk-column-name="PK" value-column-name="AID"/>
        </id>
        ...
    </attributes>
</entity>
```

Version Annotation

The @Version annotation specifies the version field or property of an entity class that serves as its **optimistic lock value**. The version is used to ensure integrity when performing the merge operation and for **optimistic** concurrency control.

Only a SINGLE @Version property or field should be used per class; applications that use more than one @Version property or field will not be portable.

The Version property should be mapped to the primary table for the entity class; applications that map the Version property to a table other than the primary table will not be portable.

In general, fields or properties that are specified with the @Version annotation SHOULD NOT be updated by the application.

The following types are supported for version properties: int, Integer, short, Short, long, Long, Timestamp.

```
@Target({METHOD, FIELD}) @Retention(RUNTIME)
public @interface Version {}
```

Example:

```
@Version
@Column(name="OPTLOCK")
protected int getVersionNum() {
    return versionNum;
}
```

Example:

```
@Entity
@IdClass(MagazineId.class)
public class Magazine {

    @Id private String isbn;
    @Id private String title;
    @Version private int version;

    ...
}
@Entity
@Table(name="SUB", schema="CNTRCT")
public class Subscription {

    ...
    @Column(name="VERS")
    @Version private int version;
    ...
}
```

The same metadata declarations in XML:

```
<entity-mappings>

    <entity class="Magazine">
    <id-class="MagazineId"/>
    <attributes>
       <id name="isbn"/>
       <id name="title"/>
       <version name="version"/>
       ...
    </attributes>
    </entity>

    <entity class="Subscription">
          <table name="SUB" schema="CNTRCT"/>
```

```
        <attributes>
            ...
            <version name="version">
                <column name="VERS"/>
            </version>
            ...
        </attributes>
    </entity>

</entity-mappings>
```

Basic Annotation

The @Basic annotation is the simplest type of mapping to a database column. The @Basic annotation can be applied to a persistent property or instance variable of any of the following types: Java primitive types, wrappers of the primitive types, java.lang.String, java.math.BigInteger, java.math.BigDecimal, java.util.Date, java.util.Calendar, java.sql.Date, java.sql.Time, java.sql.Timestamp, byte[], Byte[], char[], Character[], enums, and any other type that implements Serializable. The use of the @Basic annotation is OPTIONAL for persistent fields and properties of these types.

```
@Target({METHOD, FIELD}) @Retention(RUNTIME)
public @interface Basic {
    FetchType fetch() default EAGER;
    boolean optional() default true; // can be 'null' ?
}
```

The FetchType enum defines strategies for fetching data from the database:

```
public enum FetchType {
    LAZY,
    EAGER
};
```

The EAGER strategy is a requirement on the persistence provider runtime that data must be eagerly fetched. The LAZY strategy is a hint to the persistence provider runtime that data should be fetched lazily when it is first accessed. The implementation is permitted to eagerly fetch data for which the LAZY strategy hint has been specified. In particular, lazy fetching might ONLY be available for Basic mappings for which **property-based** access is used.

The optional element is a hint as to whether the value of the field or property may be null. It is disregarded for primitive types, which are considered non-optional.

Examples:

```
@Basic
protected String name;
@Basic(fetch=LAZY)
protected String getName() {
    return name;
}
```

Example:

```
@Entity
public class Subscription {

    @Id private long id;
    @Version private int version;
```

```
    private Date startDate; // defaults to @Basic
    private double payment; // defaults to @Basic

    ...
}

@Entity(name="Lifetime")
public class LifetimeSubscription extends Subscription {

    @Basic(fetch=FetchType.LAZY)
    private boolean getEliteClub() { ... }
    public void setEliteClub(boolean elite) { ... }

    ...
}
```

The same metadata declarations in XML:

```
<entity-mappings>
    <entity class="Subscription">
        <attributes>
            <id name="id"/>
            <basic name="payment"/>
            <basic name="startDate"/>
            <version name="version"/>
            ...
        </attributes>
    </entity>

    <entity class="LifetimeSubscription" name="Lifetime" access="PROPERTY">
        <attributes>
            <basic name="eliteClub" fetch="LAZY"/>
            ...
        </attributes>
    </entity>

</entity-mappings>
```

Lob Annotation

A @Lob annotation specifies that a persistent property or field should be persisted as a large object to a database-supported large object type. Portable applications should use the @Lob annotation when mapping to a database LOB type. The @Lob annotation may be used in conjunction with the @Basic annotation. A @Lob may be either a binary or character type. The LOB type is inferred from the type of the persistent field or property, and except for string and character-based types defaults to BLOB.

```
@Target({METHOD, FIELD}) @Retention(RUNTIME)
public @interface Lob {
}
```

Adding the @Lob marker annotation to a basic field or property signals that the data is to be stored as a LOB (Large OBject). If the field holds string or character data, it will map to a CLOB

(Character Large OBject) database column. If the field holds any other data type, it will be stored as binary data in a BLOB (Binary Large OBject) column. The implementation will serialize the Java value if needed.

The equivalent XML element is lob, which has no children or attributes.

Example 1:

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

Example 2:

```
@Lob @Basic(fetch=LAZY)
@Column(name="EMP_PIC", columnDefinition="BLOB NOT NULL")
public byte[] getPic() {
    return pic;
}
```

Example 3:

```
@Entity
public class Contract {

    @Lob
    private String terms;

    ...
}
```

The same metadata expressed in XML:

```
<entity-mappings>

    <entity class="Contract">
        <attributes>
            <basic name="terms">
                <lob/>
            </basic>
            ...
        </attributes>
    </entity>

</entity-mappings>
```

Temporal Annotation

The @Temporal annotation MUST be specified for persistent fields or properties of type java.util.Date and java.util.Calendar. It may ONLY be specified for fields or properties of these types.

The @Temporal annotation may be used in conjunction with the @Basic annotation.

The TemporalType enum defines the mapping for these temporal types.

```
public enum TemporalType {
    DATE, // equivalent to java.sql.Date
    TIME, // equivalent to java.sql.Time
```

```
    TIMESTAMP // equivalent to java.sql.Timestamp
}

@Target({METHOD, FIELD}) @Retention(RUNTIME)
public @interface Temporal {
    TemporalType value();
}
```

Example:

```
@Temporal(DATE)
protected java.util.Date endDate;
```

The @Temporal annotation determines how the implementation handles your basic java.util.Date and java.util.Calendar fields at the JDBC level. The @Temporal annotation's value is a constant from the TemporalType enum. Available values are:

- TemporalType.TIMESTAMP: The default. Use JDBC's timestamp APIs to manipulate the column data.
- TemporalType.DATE: Use JDBC's SQL date APIs to manipulate the column data.
- TemporalType.TIME: Use JDBC's time APIs to manipulate the column data.

If the @Temporal annotation is omitted, the implementation will treat the data as a TIMESTAMP.

The corresponding XML element is temporal, whose text value must be one of: TIME, DATE, or TIMESTAMP.

Example:

```
@Entity
public class Employee {

    ...
    @Temporal(DATE)
    protected java.util.Date startDate;
    ...

}
```

The same metadata expressed in XML:

```
<entity-mappings>
    <entity class="Employee">
        <attributes>
            ...
            <basic name="startDate"/>
                <temporal>DATE</temporal>
            </basic>
            ...
        </attributes>
    </entity>
</entity-mappings>
```

Enumerated Annotation

An @Enumerated annotation specifies that a persistent property or field should be persisted as a enumerated type. The @Enumerated annotation may be used in conjunction with the @Basic

annotation.

An enum can be mapped as either a string or an integer. The EnumType enum defines the mapping for enumerated types.

```
public enum EnumType {
	ORDINAL,
	STRING
}
```

The corresponding XML element is enumerated. Its embedded text must be one of STRING or ORIDINAL.

If the enumerated type is not specified or the @Enumerated annotation is not used, the enumerated type is assumed to be ORDINAL:

```
@Target({METHOD, FIELD}) @Retention(RUNTIME)
public @interface Enumerated {
	EnumType value() default ORDINAL;
}
```

Example:

```
public enum EmployeeStatus {FULL_TIME, PART_TIME, CONTRACT}

public enum SalaryRate {JUNIOR, SENIOR, MANAGER, EXECUTIVE}
@Entity public class Employee {
	...
	public EmployeeStatus getStatus() {...}

	@Enumerated(STRING)
	public SalaryRate getPayScale() {...}
	...
}
```

If the status property is mapped to a column of integer type, and the payscale property to a column of varchar type, an instance that has a status of PART_TIME and a pay rate of JUNIOR will be stored with STATUS set to 1 and PAYSCALE set to "JUNIOR".

```
@Entity
public class Employee {

	...
	@Enumerated (EnumType.STRING)
	public EmployeeType getEmployeeType() {
		return employeeType;
	}
	...
}
```

The same metadata expressed in XML:

```
<entity-mappings>
	<entity class="Employee" access="PROPERTY">
		<attributes>
			...
			<basic name="employeeType">
				<enumerated>STRING</enumerated>
			</basic>
```

```
            ...
        </attributes>
    </entity>
</entity-mappings>
```

Transient Annotation

The @Transient annotation is used to annotate a property or field of an entity class, mapped superclass, or embeddable class. It specifies that the property or field is NOT persistent.

By default, a JPA persistence provider assumes that ALL the fields of an entity are persistent.

Use the @Transient annotation to specify a field or property of an entity that is not persistent, for example, a field or property that is used at run time but that is not part of the entity's state.

A JPA persistence provider will not persist (or create database schema) for a property or field annotated as @Transient.

This annotation can be used with @Entity, @MappedSuperclass, and @Embeddable.

This annotation has no attributes:

```
@Target({METHOD, FIELD}) @Retention(RUNTIME)
public @interface Transient {}
```

The equivalent XML element is transient. It has a single attribute:

- name: The transient field or property name. This attribute is required.

Example:

```
@Entity
public class Magazine {

    ...
    @Transient
    private byte[] data;
    ...
}
```

```
<entity class="Magazine">
    <attributes>
        ...
        <transient name="data"/>
        ...
    </attributes>
</entity>
```

●Chapter 6. Java Persistence Entity Operations

●Describe how to manage entities, including using the EntityManager API and the cascade option.

- [EJB_3.0_PERSISTENCE] 3.1; 3.2

An EntityManager instance is associated with a **persistence context**. A **persistence context** is a set of entity instances in which for any persistent entity identity there is a unique entity instance. Within the persistence context, the entity instances and their lifecycle are managed. The EntityManager interface defines the methods that are used to interact with the persistence context. The EntityManager API is used to create and remove persistent entity instances, to find entities by their primary key, and to query over entities. The set of entities that can be managed by a given EntityManager instance is defined by a **persistence unit**. A **persistence unit** defines the set of all classes that are related or grouped by the application, and which must be colocated in their mapping to a SINGLE database.

EntityManager Interface

The EntityManager is the primary interface used by application developers to interact with the JPA runtime. The methods of the EntityManager can be divided into the following functional categories:

1 **Transaction Association**

```
public EntityTransaction getTransaction();
```

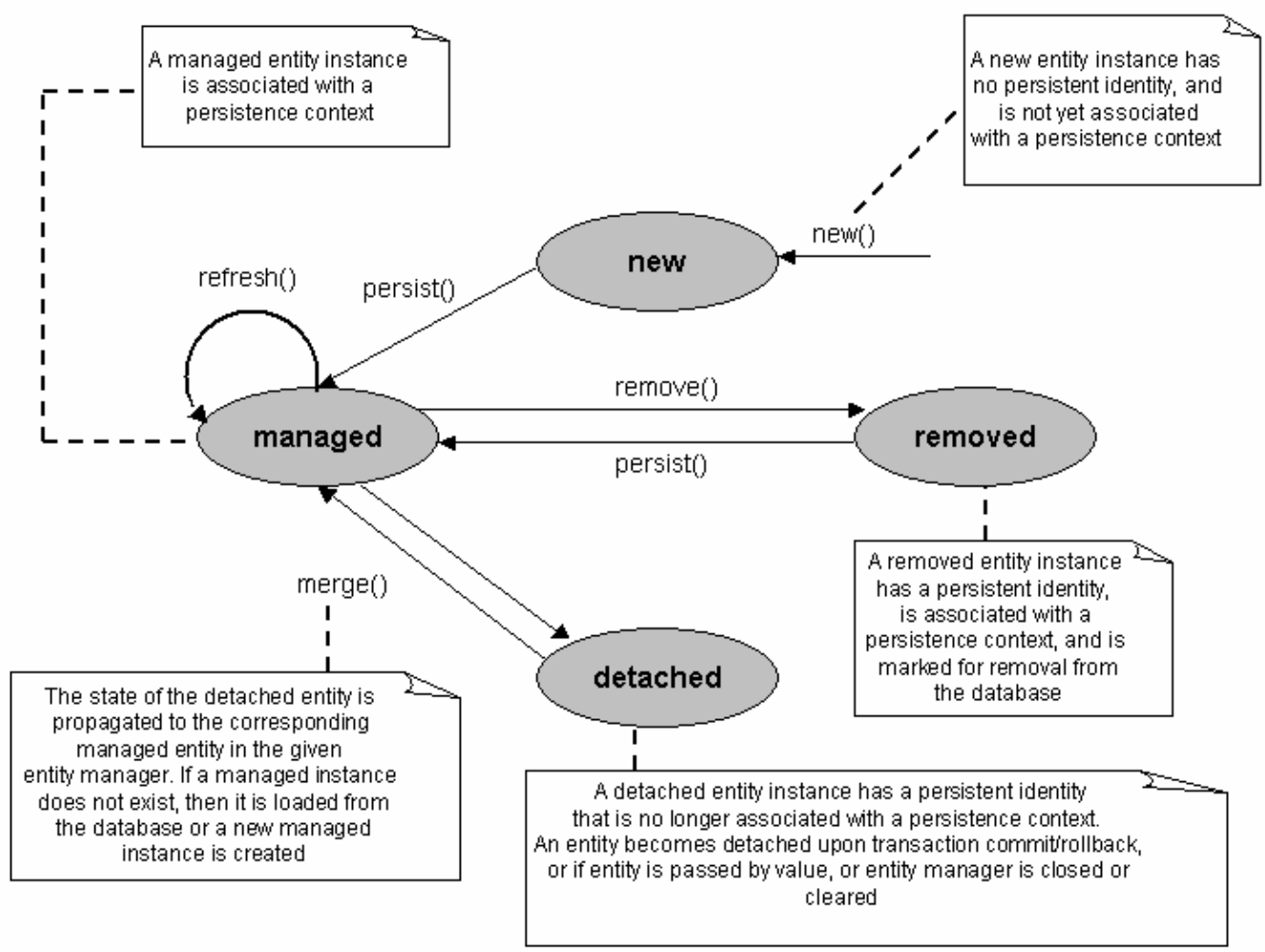
Every EntityManager has a one-to-one relation with an EntityTransaction instance. In fact, many vendors use a single class to implement both the EntityManager and EntityTransaction interfaces. If your application requires multiple concurrent transactions, you will use multiple EntityManagers.

You can retrieve the EntityTransaction associated with an EntityManager through the getTransaction method. Note that most most JPA implementations can integrate with an application server's managed transactions. If you take advantage of this feature, you will control transactions by declarative demarcation or through the Java Transaction API (JTA) rather than through the EntityTransaction.

2 Entity Lifecycle Management

EntityManagers perform several actions that affect the lifecycle state of entity instances.

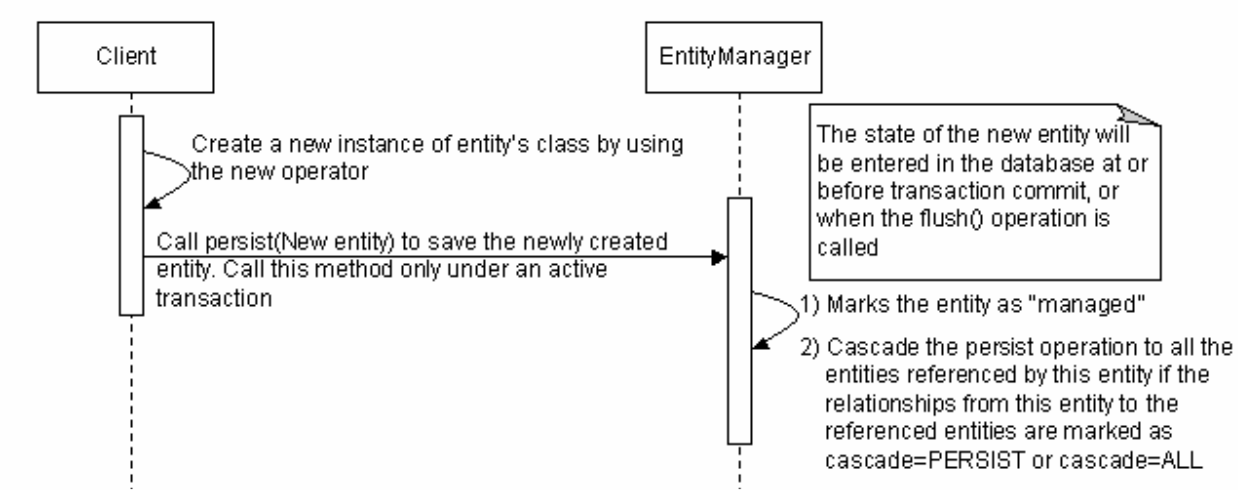
The following diagram illustrates the lifecycle of an entity with respect to the APIs presented in this section.



public void persist(Object entity);
Transitions NEW instances to MANAGED. On the next flush or commit, the newly persisted instances will be INSERTED into the datastore.

For a given entity A, the persist method behaves as follows:

- If A is a NEW entity, it becomes MANAGED.
- If A is an existing MANAGED entity, it is IGNORED. However, the persist operation cascades as defined below.
- If A is a REMOVED entity, it becomes MANAGED.
- If A is a DETACHED entity, an IllegalArgumentException is thrown.
- The persist operation recurses on all relation fields of A whose cascades include CascadeType.PERSIST.



This action can only be used in the context of an ACTIVE transaction.

Persisting Objects Example:

```
// create some objects
Magazine mag = new Magazine("1B78-YU9L", "JavaWorld");

Company pub = new Company("Weston House");
pub.setRevenue(1750000D);
mag.setPublisher(pub);
pub.addMagazine(mag);

Article art = new Article("JPA Rules!", "Transparent Object Persistence");
art.addAuthor(new Author("Fred", "Hoyle"));
mag.addArticle(art);

// persist
EntityManager em = emf.createEntityManager();
em.getTransaction().begin();
em.persist(mag);
em.persist(pub);
em.persist(art);
em.getTransaction().commit();

// or we could continue using the EntityManager...
em.close();
```

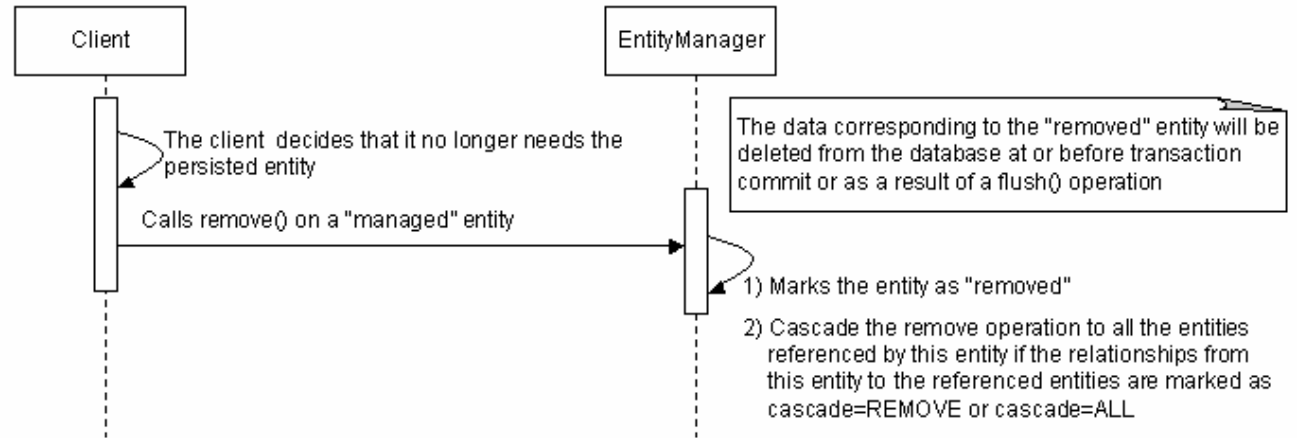
public void remove(Object entity);

Transitions managed instances to REMOVED. The instances will be deleted from the datastore on the next flush or commit. Accessing a removed entity has UNDEFINED results.

For a given entity A, the remove method behaves as follows:

- If A is a NEW entity, it is ignored. However, the remove operation cascades as defined below.

- If A is an existing MANAGED entity, it becomes REMOVED.
- If A is a REMOVED entity, it is IGNORED.
- If A is a DETACHED entity, an IllegalArgumentException is thrown.
- The remove operation recurses on all relation fields of A whose cascades include



CascadeType.REMOVE.

This action can only be used in the context of an ACTIVE transaction.

Removing Objects Example:

```
// assume we have an object id for the company whose subscriptions
// we want to delete
Object oid = ...;

// deletes should always be made within transactions
EntityManager em = emf.createEntityManager();
em.getTransaction().begin();
Company pub = (Company) em.find(Company.class, oid);
for (Subscription sub : pub.getSubscriptions())
      em.remove(sub);
pub.getSubscriptions().clear();
em.getTransaction().commit();

// or we could continue using the EntityManager...
em.close();
```

public void refresh(Object entity);

Use the refresh action to make sure the persistent state of an instance is synchronized with the values in the datastore. refresh is intended for long-running optimistic transactions in which there is a danger of seeing stale data.

For a given entity A, the refresh method behaves as follows:

- If A is a NEW entity, it is IGNORED. However, the refresh operation cascades as defined below.

- If A is an existing MANAGED entity, its state is REFRESHED from the datastore.
- If A is a REMOVED entity, it is IGNORED.
- If A is a DETACHED entity, an IllegalArgumentException is thrown.
- The refresh operation recurses on all relation fields of A whose cascades include CascadeType.REFRESH.

public Object merge(Object entity);

A common use case for an application running in a servlet or application server is to "detach" objects from all server resources, modify them, and then "attach" them again. For example, a servlet might store persistent data in a user session between a modification based on a series of web forms. Between each form request, the web container might decide to serialize the session, requiring that the stored persistent state be disassociated from any other resources. Similarly, a client/server application might transfer persistent objects to a client via serialization, allow the client to modify their state, and then have the client return the modified data in order to be saved. This is sometimes referred to as the data transfer object or value object pattern, and it allows fine-grained manipulation of data objects without incurring the overhead of multiple remote method invocations.

JPA provides support for this pattern by automatically detaching entities when they are serialized or when a persistence context ends. The JPA merge API re-attaches detached entities. This allows you to detach a persistent instance, modify the detached instance offline, and merge the instance back into an EntityManager (either the same one that detached the instance, or a new one). The changes will then be applied to the existing instance from the datastore.

A detached entity maintains its persistent identity, but cannot load additional state from the datastore. Accessing any persistent field or property that was not loaded at the time of detachment has UNDEFINED results. Also, be sure NOT to alter the version or identity fields of detached instances if you plan on merging them later.

The merge method returns a managed copy of the given detached entity. Changes made to the persistent state of the detached entity are applied to this managed instance. Because merging involves changing persistent state, you can only merge WITHIN a transaction.

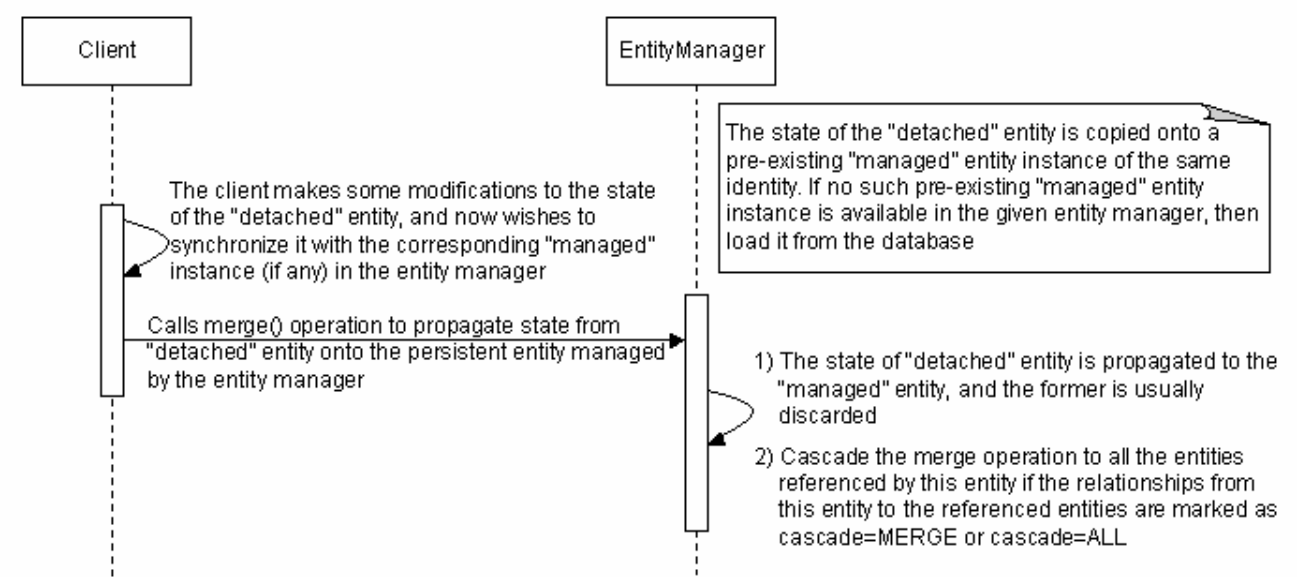
If you attempt to merge an instance whose representation has changed in the datastore since detachment, the merge operation will throw an exception, or the transaction in which you perform the merge will fail on commit, just as if a normal optimistic conflict were detected.

For a given entity A, the merge method behaves as follows:

- If A is a DETACHED entity, its state is COPIED into existing managed instance A' of the same entity identity, or a new managed copy of A is created.
- If A is a NEW entity, a new MANAGED entity A' is created and the state of A is COPIED into A'.
- If A is an existing MANAGED entity, it is IGNORED. However, the merge operation

still cascades as defined below.

- If A is a REMOVED entity, an IllegalArgumentException is thrown.
- The merge operation recurses on all relation fields of A whose cascades include CascadeType.MERGE.



This example demonstrates a common client/server scenario. The client requests objects and makes changes to them, while the server handles the object lookups and transactions:

```
// CLIENT:
// requests an object with a given oid
Record detached = (Record) getFromServer(oid);

// SERVER:
// send object to client; object detaches on EM close
Object oid = processClientRequest();
EntityManager em = emf.createEntityManager();
Record record = em.find(Record.class, oid);
em.close();
sendToClient(record);

// CLIENT:
// makes some modifications and sends back to server
detached.setSomeField("bar");
sendToServer(detached);

// SERVER:
// merges the instance and commit the changes
Record modified = (Record) processClientRequest();
EntityManager em = emf.createEntityManager();
```

```
em.getTransaction().begin();
Record merged = (Record) em.merge(modified);
merged.setLastModified(System.currentTimeMillis());
merged.setModifier(getClientIdentityCode());
em.getTransaction().commit();
em.close();
```

public void lock (Object entity, LockModeType mode);

This method locks the given entity using the named mode. The javax.persistence.LockModeType enum defines two modes:

- READ: Other transactions may concurrently read the object, but cannot concurrently update it.
- WRITE: Other transactions cannot concurrently read or write the object. When a transaction is committed that holds WRITE locks on any entites, those entites will have their version incremented even if the entities themselves did not change in the transaction.

3 Entity Identity Management

Each EntityManager is responsible for managing the persistent identities of the managed objects in the persistence context. The following methods allow you to interact with the management of persistent identities. The behavior of these methods is deeply affected by the persistence context type of the EntityManager.

public <T> T find(Class<T> cls, Object oid);

This method returns the persistent instance of the given type with the given persistent identity. If the instance is already present in the current persistence context, the cached version will be returned. Otherwise, a new instance will be constructed and loaded with state from the datastore. If no entity with the given type and identity exists in the datastore, this method returns null.

Example:

```
MagazineId mi = new MagazineId();
mi.isbn = "1B78-YU9L";
mi.title = "JavaWorld";

// updates should always be made within transactions; note that
// there is no code explicitly linking the magazine or company
// with the transaction; JPA automatically tracks all changes
EntityManager em = emf.createEntityManager();
em.getTransaction().begin();
Magazine mag = em.find(Magazine.class, mi);
mag.setPrice(5.99);
Company pub = mag.getPublisher();
pub.setRevenue(1750000D);
em.getTransaction().commit();

// or we could continue using the EntityManager...
```

em.close();

public <T> T getReference(Class<T> cls, Object oid);

This method is similar to find, but does NOT necessarily go to the database when the entity is not found in cache. The implementation may construct a hollow entity and return it to you instead. Hollow entities do not have any state loaded. The state only gets loaded when you attempt to access a persistent field. At that time, the implementation may throw an EntityNotFoundException if it discovers that the entity does not exist in the datastore. The implementation may also throw an EntityNotFoundException from the getReference method itself. Unlike find, getReference does NOT return null.

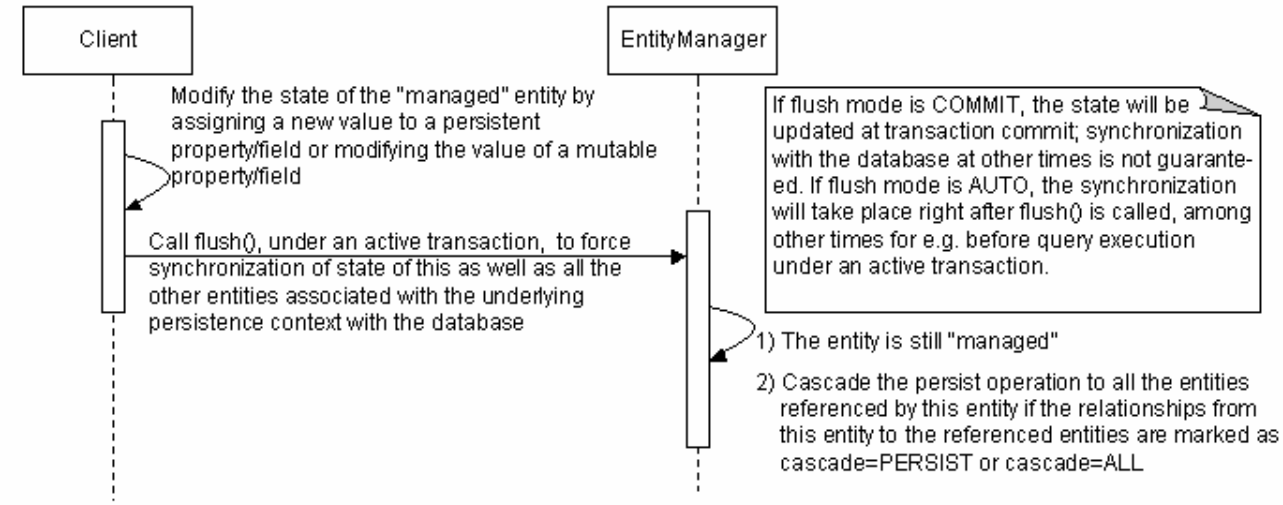
public boolean contains(Object entity);

Returns true if the given entity is part of the current persistence context, and false otherwise. Removed entities are not considered part of the current persistence context.

4 Cache Management

public void flush();

The flush method writes any changes that have been made in the current transaction to the datastore. If the EntityManager does not already have a connection to the datastore, it obtains one for the flush and retains it for the duration of the transaction. Any exceptions



during flush cause the transaction to be marked for rollback.

Flushing requires an ACTIVE transaction. If there isn't a transaction in progress, the flush method throws a TransactionRequiredException.

public FlushModeType getFlushMode();
public void setFlushMode(FlushModeType flushMode);

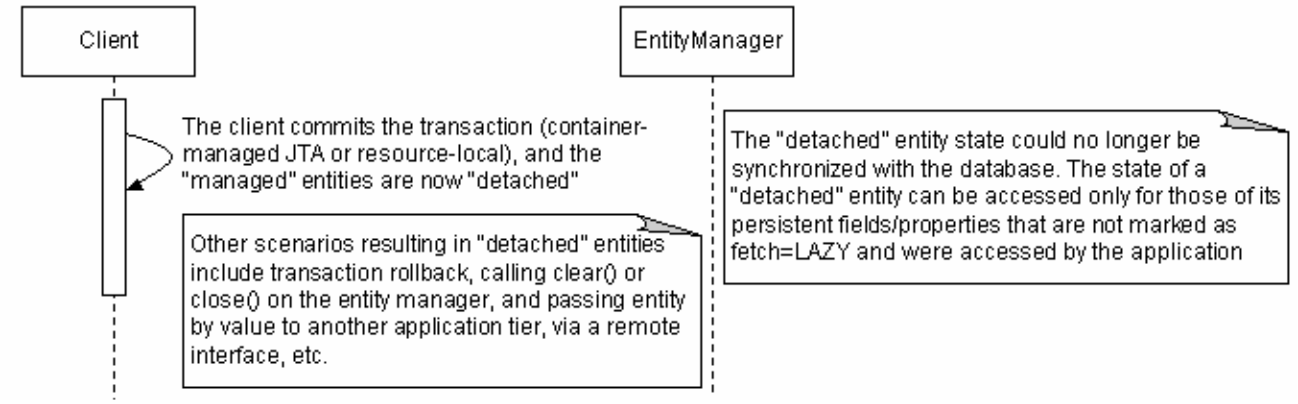
The EntityManager's FlushMode property controls whether to flush transactional changes before executing queries. This allows the query results to take into account changes you have made during the current transaction. Available javax.persistence.FlushModeType constants are:

- COMMIT: Only flush when committing, or when told to do so through the flush method. Query results may not take into account changes made in the current transaction.
- AUTO: The implementation is permitted to flush before queries to ensure that the results reflect the most recent object state.

You can also set the flush mode on individual Query instances.

public void clear();

Clearing the EntityManager effectively ends the persistence context. All entities managed



by the EntityManager become DETACHED.

5 Query Factory

public Query createQuery(String query);

Query objects are used to find entities matching certain criteria. The createQuery method creates a query using the given Java Persistence Query Language (JPQL) string.

public Query createNamedQuery(String name);

This method retrieves a query defined in metadata by name. The returned Query instance is initialized with the information declared in metadata.

public Query createNativeQuery(String sql);
public Query createNativeQuery(String sql, Class resultCls);
public Query createNativeQuery(String sql, String resultMapping);

Native queries are queries in the datastore's native language. For relational databases, this the Structured Query Language (SQL).

6 Closing

public boolean isOpen();
public void close();

When an EntityManager is no longer needed, you should call its close method. Closing an EntityManager releases any resources it is using. The persistence context ends, and the entities managed by the EntityManager become DETACHED. Any Query instances the EntityManager created become INVALID. Calling any method other than isOpen on a closed EntityManager results in an IllegalStateException. You cannot close a EntityManager that is in the middle of a transaction.

If you are in a managed environment using injected entity managers, you SHOULD NOT close them.

NOTE:

The persist, merge, remove, and refresh methods MUST be invoked within a transaction context when an entity manager with a transaction-scoped persistence context is used. If there is no transaction context, the javax.persistence.TransactionRequiredException is thrown.

The find and getReference methods are NOT required to be invoked within a transaction context. If an entity manager with **transaction-scoped persistence context** is in use, the resulting entities will be DETACHED; if an entity manager with an **extended persistence context** is used, they will be MANAGED.

The Query and EntityTransaction objects obtained from an entity manager are valid while that entity manager is OPEN.

If the argument to the createQuery method is not a valid Java Persistence query string, the IllegalArgumentException may be thrown or the query execution will fail. If a native query is not a valid query for the database in use or if the result set specification is incompatible with the result of the query, the query execution will fail and a PersistenceException will be thrown when the query is executed. The PersistenceException should wrap the underlying database exception when possible.

Runtime exceptions thrown by the methods of the EntityManager interface will cause the current transaction to be ROLLED BACK.

The methods close, isOpen, joinTransaction, and getTransaction are used to manage application-managed entity managers and their lifecycle.

●Identify correct and incorrect statements or examples about entity instance lifecycle, including the new, managed, detached, and removed states.

- [EJB_3.0_PERSISTENCE] 3.2; 3.2.1; 3.2.2; 3.2.3

An entity instance may be characterized as being **new**, **managed**, **detached**, or **removed**.

- A **new** entity instance has no persistent identity, and is NOT yet ASSOCIATED with a persistence context.
- A **managed** entity instance is an instance with a persistent identity that is currently ASSOCIATED with a persistence context.
- A **detached** entity instance is an instance with a persistent identity that is NOT (or no longer) ASSOCIATED with a persistence context.
- A **removed** entity instance is an instance with a persistent identity, ASSOCIATED with a persistence context, that is SCHEDULED for removal from the database.

Use of the cascade annotation element may be used to propagate the effect of an operation to associated entities. The cascade functionality is most typically used in parent-child relationships.

Persisting an Entity Instance

A new entity instance becomes both MANAGED and PERSISTENT by invoking the persist method on it or by cascading the persist operation.

The semantics of the persist operation, applied to an entity X are as follows:

- If X is a NEW entity, it becomes MANAGED. The entity X will be entered into the database at or before transaction commit or as a result of the flush operation.
- If X is a preexisting MANAGED entity, it is IGNORED by the persist operation. However, the persist operation is cascaded to entities referenced by X, if the relationships from X to these other entities is annotated with the cascade=PERSIST or cascade=ALL annotation element value or specified with the equivalent XML descriptor element.
- If X is a REMOVED entity, it becomes MANAGED.
- If X is a DETACHED object, the EntityExistsException may be thrown when the persist operation is invoked, or the EntityExistsException or another PersistenceException may be thrown at flush or commit time.
- For all entities Y referenced by a relationship from X, if the relationship to Y has been annotated with the cascade element value cascade=PERSIST or cascade=ALL, the persist operation is applied to Y.

Removal

A MANAGED entity instance becomes REMOVED by invoking the remove method on it or by cascading the remove operation.

The semantics of the remove operation, applied to an entity X are as follows:

- If X is a NEW entity, it is IGNORED by the remove operation. However, the remove operation is cascaded to entities referenced by X, if the relationship from X to these other entities is annotated with the cascade=REMOVE or cascade=ALL annotation element value.
- If X is a MANAGED entity, the remove operation causes it to become REMOVED. The remove operation is cascaded to entities referenced by X, if the relationships from X to these other entities is annotated with the cascade=REMOVE or cascade=ALL annotation element value.

- If X is a DETACHED entity, an IllegalArgumentException will be thrown by the remove operation (or the transaction commit will fail).
- If X is a REMOVED entity, it is IGNORED by the remove operation.
- A removed entity X will be removed from the database at or before transaction commit or as a result of the flush operation.

After an entity has been removed, its state (except for generated state) will be that of the entity at the point at which the remove operation was called.

Synchronization to the Database

The state of persistent entities is synchronized to the database at transaction commit. This synchronization involving writing to the database any updates to persistent entities and their relationships as speci fied above.

An update to the state of an entity includes both the assignment of a new value to a persistent property or field of the entity as well as the modification of a mutable value of a persistent property or field.

Synchronization to the database DOES NOT involve a refresh of any managed entities unless the refresh operation is EXPLICITLY invoked on those entities.

Bidirectional relationships between managed entities will be persisted based on references held by the owning side of the relationship. It is the developer's responsibility to keep the in-memory references held on the owning side and those held on the inverse side consistent with each other when they change. In the case of unidirectional one-to-one and one-to-many relationships, it is the developer's responsibility to insure that the semantics of the relationships are adhered to.

It is particularly important to ensure that changes to the inverse side of a relationship result in appropriate updates on the owning side, so as to ensure the changes are not lost when they are synchronized to the database.

The persistence provider runtime is permitted to perform synchronization to the database at other times as well when a transaction is active. The flush method can be used by the application to FORCE synchronization. It applies to entities associated with the persistence context. The EntityManager and Query setFlushMode methods can be used to control synchronization semantics. If FlushModeType.COMMIT is specified, flushing will occur at transaction commit; the persistence provider is permitted, but not required, to perform to flush at other times. If there is no transaction active, the persistence provider must not flush to the database.

The semantics of the flush operation, applied to an entity X are as follows:

- If X is a MANAGED entity, it is SYNCHRONIZED to the database.
- For all entities Y referenced by a relationship from X, if the relationship to Y has been annotated with the cascade element value cascade=PERSIST or cascade=ALL, the persist operation is applied to Y.
- For any entity Y referenced by a relationship from X, where the relationship to Y has NOT been annotated with the cascade element value cascade=PERSIST or cascade=ALL:
    - If Y is NEW or REMOVED, an IllegalStateException will be thrown by the flush

operation (and the transaction rolled back) or the transaction commit will fail.

- If Y is DETACHED, the semantics depend upon the ownership of the relationship. If X owns the relationship, any changes to the relationship are synchronized with the database; otherwise, if Y owns the relationships, the behavior is undefined.

- If X is a REMOVED entity, it is REMOVED from the database. No cascade options are relevant.

- **Identify correct and incorrect statements or examples about EntityManager operations for managing an instance's state, including eager/lazy fetching, handling detached entities, and merging detached entities.**

  - [EJB_3.0_PERSISTENCE] 3.2.4; 3.2.4.1; 3.2.4.2; 3.2.5

Detached Entities

A **detached** entity may result:

- from transaction commit if a transaction-scoped container-managed entity manager is used;
- from transaction rollback;
- from clearing the persistence context;
- from closing an entity manager;
- and from serializing an entity or otherwise passing an entity by value - e.g., to a separate application tier, through a remote interface, etc.

Detached entity instances continue to live outside of the persistence context in which they were persisted or retrieved, and their state is NO longer guaranteed to be synchronized with the database state.

The application may access the available state of available detached entity instances after the persistence context ends. The available state includes:

- Any persistent field or property NOT MARKED fetch=LAZY.
- Any persistent field or property that was accessed by the application.

If the persistent field or property is an association, the available state of an associated instance MAY ONLY be safely accessed if the associated instance is available. The available instances include:

- Any entity instance retrieved using find().
- Any entity instances retrieved using a query or explicitly requested in a FETCH JOIN clause.
- Any entity instance for which an instance variable holding non-primary-key persistent state was accessed by the application.
- Any entity instance that may be reached from another available instance by navigating

associations marked fetch=EAGER.

Merging Detached Entity State

The merge operation allows for the propagation of state from detached entities onto persistent entities managed by the EntityManager.

The semantics of the merge operation applied to an entity X are as follows:

- If X is a DETACHED entity, the state of X is COPIED onto a pre-existing managed entity instance X' of the same identity or a NEW MANAGED copy X' of X is created.
- If X is a NEW entity instance, a NEW MANAGED entity instance X' is created and the state of X is copied into the NEW MANAGED entity instance X'.
- If X is a REMOVED entity instance, an IllegalArgumentException will be thrown by the merge operation (or the transaction commit will fail).
- If X is a MANAGED entity, it is ignored by the merge operation, however, the merge operation is cascaded to entities referenced by relationships from X if these relationships have been annotated with the cascade element value cascade=MERGE or cascade=ALL annotation.
- For all entities Y referenced by relationships from X having the cascade element value cascade=MERGE or cascade=ALL, Y is merged recursively as Y'. For all such Y referenced by X, X' is set to reference Y'. (Note that if X is managed then X is the same object as X'.)
- If X is an entity merged to X', with a reference to another entity Y, where cascade=MERGE or cascade=ALL is not specified, then navigation of the same association from X' yields a reference to a managed object Y' with the same persistent identity as Y.

The persistence provider MUST NOT merge fields marked LAZY that have not been fetched: it MUST IGNORE such fields when merging.

Any Version columns used by the entity MUST BE CHECKED by the persistence runtime implementation during the merge operation and/or at flush or commit time. In the absence of Version columns there is no additional version checking done by the persistence provider runtime during the merge operation.

Detached Entities and Lazy Loading

Serializing entities and merging those entities back into a persistence context may not be interoperable across vendors when lazy properties or fields and/or relationships are used.

A vendor is required to support the serialization and subsequent deserialization and merging of detached entity instances (which may contain lazy properties or fields and/or relationships that have not been fetched) back into a separate JVM instance of that vendor's runtime, where both runtime instances have access to the entity classes and any required vendor persistence implementation classes.

When interoperability across vendors is required, the application must not use lazy loading.

Managed Instances

It is the responsibility of the application to insure that an instance is managed in only a single

persistence context. The behavior is undefined if the same Java instance is made managed in more than one persistence context.

The contains() method can be used to determine whether an entity instance is managed in the current persistence context.

The contains method returns **true**:

- If the entity has been retrieved from the database, and has NOT been removed or detached.
- If the entity instance is new, and the persist method has been called on the entity or the persist operation has been cascaded to it.

The contains method returns **false**:

- If the instance is detached.
- If the remove method has been called on the entity, or the remove operation has been cascaded to it.
- If the instance is new, and the persist method HAS NOT been called on the entity or the persist operation has not been cascaded to it.

Note that the effect of the cascading of persist or remove is IMMEDIATELY visible to the contains method, whereas the ACTUAL insertion or deletion of the database representation for the entity may be DEFERRED until the end of the transaction.

●Identify correct and incorrect statements or examples about Entity Listeners and Callback Methods, including: @PrePersist, @PostPersist, @PreRemove, @PostRemove, @PreUpdate, @PostUpdate, and @PostLoad, and when they are invoked.

- [EJB_3.0_PERSISTENCE] 3.5; 3.5.1; 3.5.2; 3.5.4; 3.5.5

A method may be designated as a lifecycle callback method to receive notification of entity lifecycle events.
A lifecycle callback method may be defined on an entity class, a mapped superclass, or an entity listener class associated with an entity or mapped superclass. An entity listener class is a class whose methods are invoked in response to lifecycle events on an entity. ANY NUMBER of entity listener classes may be defined for an entity class or mapped superclass.

Default entity listeners - entity listeners that apply to all entities in the persistence unit - can be specified by means of the XML descriptor.

Lifecycle callback methods and entity listener classes are defined by means of metadata annotations or the XML descriptor. When annotations are used, one or more entity listener classes are denoted using the EntityListeners annotation on the entity class or mapped superclass. If multiple entity listeners are defined, the order in which they are invoked is determined by the order in which they are specified in the EntityListeners annotation. The XML descriptor may be used as an alternative to specify the invocation order of entity listeners or to

override the order specified in metadata annotations.

Any subset or combination of annotations may be specified on an entity class, mapped superclass, or listener class. A single class MAY NOT have more than ONE lifecycle callback method for the same lifecycle event. The same method may be used for multiple callback events.

Multiple entity classes and mapped superclasses in an inheritance hierarchy may define listener classes and/or lifecycle callback methods directly on the class.

The entity listener class must have a public no-arg constructor.

Entity listeners are stateless. The lifecycle of an entity listener is unspecified.

The following rules apply to lifecycle callbacks:

- Lifecycle callback methods may throw unchecked/runtime exceptions. A runtime exception thrown by a callback method that executes within a transaction causes that transaction to be rolled back.
- Lifecycle callbacks can invoke JNDI, JDBC, JMS, and enterprise beans.
- In general, portable applications should not invoke EntityManager or Query operations, access other entity instances, or modify relationships in a lifecycle callback method.

When invoked from within a Java EE environment, the callback listeners for an entity share the enterprise naming context of the invoking component, and the entity callback methods are invoked in the transaction and security contexts of the calling component at the time at which the callback method is invoked.

Lifecycle Callback Methods

Entity lifecycle callback methods can be defined on an entity listener class and/or directly on an entity class or mapped superclass.

Lifecycle callback methods are annotated with annotations designating the callback events for which they are invoked or are mapped to the callback event using the XML descriptor.

The annotations used for callback methods on the entity class or mapped superclass and for callback methods on the entity listener class are the same. The signatures of individual methods, however, differ.

Callback methods defined on an entity class or mapped superclass have the following signature:

```
void <METHOD>()
```

When declaring callback methods on a persistent class, any method may be used which takes no arguments and is not shared with any property access fields. Multiple events can be assigned to a single method as well.

Below is an example of how to declare callback methods on persistent classes:

```
/**
 * Example persistent class declaring entity listener.
 */
```

```
@Entity
public class Magazine {

	@Transient
	private byte[][] data;

	@ManyToMany
	private List<Photo> photos;

	@PostLoad
	public void convertPhotos() {
		data = new byte[photos.size()][];
		for (int i = 0; i < photos.size(); i++) {
			data[i] = photos.get(i).toByteArray();
		}
	}

	@PreDelete
	public void logMagazineDeletion() {
		getLog().debug("deleting magazine containing" + photos.size() + " photos.");
	}
}
```

In an XML mapping file, we can define the same methods without annotations:

```
<entity class="Magazine">
	<pre-remove>logMagazineDeletion</pre-remove>
	<post-load>convertPhotos</post-load>
</entity>
```

Callback methods defined on an entity listener class have the following signature:

```
void <METHOD>(Object)
```

The Object argument is the entity instance for which the callback method is invoked. It may be declared as the actual entity type.

Mixing lifecycle event code into your persistent classes is not always ideal. It is often more elegant to handle cross-cutting lifecycle events in a non-persistent listener class. JPA allows for this, requiring only that listener classes have a public no-arg constructor. Like persistent classes, your listener classes can consume any number of callbacks. The callback methods must take in a single java.lang.Object argument which represents the persistent object that triggered the event.

Entities can enumerate listeners using the EntityListeners annotation. This annotation takes an array of listener classes as its value.

Below is an example of how to declare an entity and its corresponding listener classes:

```
/**
 * Example persistent class declaring entity listener.
 */
@Entity
@EntityListeners({ MagazineLogger.class, ... })
public class Magazine {
	...
}


/**
 * Example entity listener.
 */
public class MagazineLogger {

	@PostPersist
	public void logAddition(Object pc) {
		getLog ().debug ("Added new magazine:" + ((Magazine) pc).getTitle ());
	}

	@PreRemove
	public void logDeletion(Object pc) {
		getLog().debug("Removing from circulation:" + ((Magazine) pc).getTitle());
	}
}
```

In XML, we define both the listeners and their callback methods as so:

```
<entity class="Magazine">
	<entity-listeners>
		<entity-listener class="MagazineLogger">
			<post-persist>logAddition</post-persist>
			<pre-remove>logDeletion</pre-remove>
		</entity-listener>
	</entity-listeners>
</entity>
```

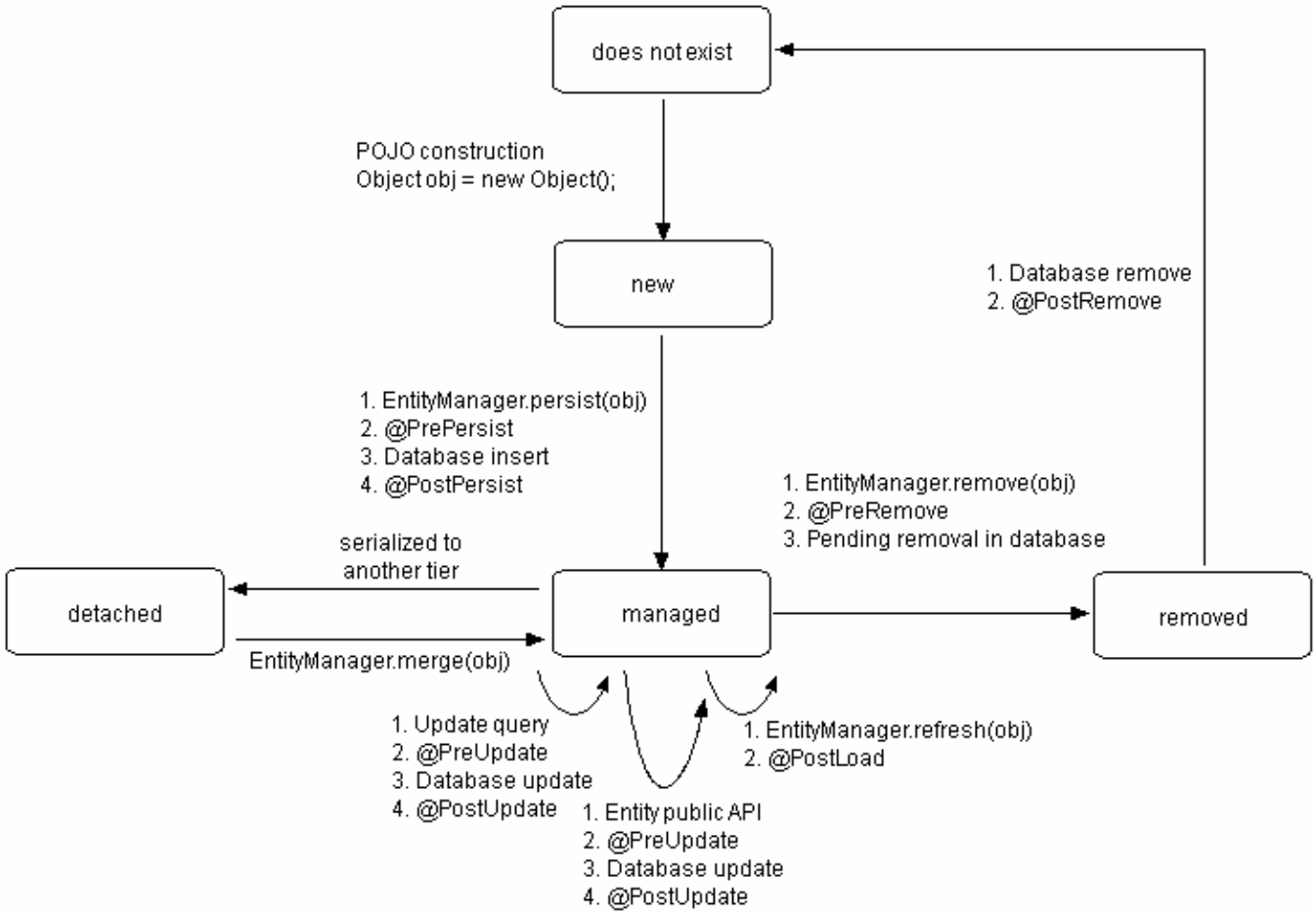
The callback methods can have public, private, protected, or package level access, but MUST NOT be static or final.

The following annotations are defined to designate lifecycle event callback methods of the corresponding types:

- @PrePersist: Methods marked with this annotation will be invoked BEFORE an object is persisted. This could be used for assigning primary key values to persistent objects. This is equivalent to the XML element tag pre-persist.
- @PostPersist: Methods marked with this annotation will be invoked AFTER an object has transitioned to the persistent state. You might want to use such methods to update a screen after a new row is added. This is equivalent to the XML element tag post-persist.

- @PreRemove: Methods marked with this annotation will be invoked BEFORE an object transactions to the DELETED state. Access to persistent fields is valid within this method. You might use this method to cascade the deletion to related objects based on complex criteria, or to perform other cleanup. This is equivalent to the XML element tag pre-remove.
- @PostRemove: Methods marked with this annotation will be invoked AFTER an object has been marked as to be deleted. This is equivalent to the XML element tag post-remove.
- @PreUpdate: Methods marked with this annotation will be invoked just the persistent values in your objects are flushed to the datastore. This is equivalent to the XML element tag pre-update.
- @PreUpdate is the complement to @PostLoad. While methods marked with @PostLoad are most often used to initialize non-persistent values from persistent data, methods annotated with @PreUpdate is normally used to set persistent fields with information cached in non-persistent data.
- @PostUpdate: Methods marked with this annotation will be invoked AFTER changes to a given instance have been stored to the datastore. This is useful for clearing stale data cached at the application layer. This is equivalent to the XML element tag post-update.
- @PostLoad: Methods marked with this annotation will be invoked after all eagerly fetched fields of your class have been loaded from the datastore. No other persistent fields can be accessed in this method. This is equivalent to the XML element tag post-load.

  @PostLoad is often used to initialize non-persistent fields whose values depend on the values of persistent fields, such as a complex datastructure.



Semantics of the Life Cycle Callback Methods for Entities

The @PrePersist and @PreRemove callback methods are invoked for a given entity BEFORE the respective EntityManager persist and remove operations for that entity are executed. For entities to which the merge operation has been applied and causes the creation of newly managed instances, the @PrePersist callback methods will be invoked for the managed instance AFTER the entity state has been copied to it. These @PrePersist and @PreRemove callbacks will also be invoked on all entities to which these operations are cascaded. The @PrePersist and @PreRemove methods will always be invoked as part of the synchronous persist, merge, and remove operations.

The @PostPersist and @PostRemove callback methods are invoked for an entity AFTER the entity has been made persistent or removed. These callbacks will also be invoked on all entities to which these operations are cascaded. The @PostPersist and @PostRemove methods will be invoked AFTER the database INSERT and DELETE operations respectively. These database operations may occur directly after the persist, merge, or remove operations have been invoked or they may occur directly after a flush operation has occurred (which may be at the end of the transaction). Generated primary key values are available in the @PostPersist method.

The @PreUpdate and @PostUpdate callbacks occur BEFORE and AFTER the database

UPDATE operations to entity data respectively. These database operations may occur at the time the entity state is updated or they may occur at the time state is flushed to the database (which may be at the end of the transaction).

The @PostLoad method for an entity is invoked AFTER the entity has been loaded into the current persistence context from the database or after the refresh operation has been applied to it. The @PostLoad method is invoked before a query result is returned or accessed or before an association is traversed.

Multiple Lifecycle Callback Methods for an Entity Lifecycle Event

If multiple callback methods are defined for an entity lifecycle event, the ordering of the invocation of these methods is as follows.

Default listeners, if any, are invoked first, in the order specified in the XML descriptor. Default listeners apply to all entities in the persistence unit, unless explicitly excluded by means of the ExcludeDefaultListeners annotation or exclude-default-listeners XML element.

The lifecycle callback methods defined on the entity listener classes for an entity class or mapped superclass are invoked in the same order as the specification of the entity listener classes in the EntityListeners annotation.

If MULTIPLE classes in an inheritance hierarchy - entity classes and/or mapped superclasses - define entity listeners, the listeners defined for a SUPERCLASS are invoked BEFORE the listeners defined for its subclasses in this order. The ExcludeSuperclassListeners annotation or exclude-superclass-listeners XML element may be applied to an entity class or mapped superclass to exclude the invocation of the listeners defined by the entity listener classes for the superclasses of the entity or mapped superclass. The excluded listeners are excluded from the class to which the ExcludeSuperclassListeners annotation or element has been specified and its subclasses. The ExcludeSuperclassListeners annotation (or exclude-superclass-listeners XML element) DOES NOT cause default entity listeners to be excluded from invocation.

If a lifecycle callback method for the same lifecycle event is also specified on the entity class and/or one or more of its entity or mapped superclasses, the callback methods on the entity class and/or superclasses are invoked AFTER the other lifecycle callback methods, most general superclass first. A class is permitted to override an inherited callback method of the same callback type, and in this case, the overridden method is NOT invoked.

Callback methods are invoked by the persistence provider runtime in the order specified. If the callback method execution terminates normally, the persistence provider runtime then invokes the next callback method, if any.

The XML descriptor may be used to OVERRIDE the lifecycle callback method invocation order specified in annotations.

Example:

```
@Entity
public class Animal {
    ....
    @PostPersist
    protected void postPersistAnimal() {
        ....
    }
```

```
}


@Entity
@EntityListeners(PetListener.class)
public class Pet extends Animal {
	...
}


@Entity
@EntityListeners({CatListener.class, CatListener2.class})
public class Cat extends Pet {
	...
}


public class PetListener {
	@PostPersist
	protected void postPersistPetListenerMethod(Object pet) {
		...
	}
}


public class CatListener {
	@PostPersist
	protected void postPersistCatListenerMethod(Object cat) {
		...
	}
}


public class CatListener2 {
	@PostPersist
	protected void postPersistCatListener2Method(Object cat) {
		...
	}
}
```

CASE 1: If a @PostPersist event occurs on an instance of Cat, the following methods are called in order:

1. postPersistPetListenerMethod
2. postPersistCatListenerMethod
3. postPersistCatListener2Method
4. postPersistAnimal

Assume that SiameseCat is defined as a subclass of Cat:

```
@EntityListeners(SiameseCatListener.class)
@Entity
```

```
public class SiameseCat extends Cat {
	...
	@PostPersist
	protected void postPersistSiameseCat() {
		...
	}
}


public class SiameseCatListener {
	@PostPersist
	protected void postPersistSiameseCatListenerMethod(Object cat) {
		...
	}
}
```

CASE 2: If a @PostPersist event occurs on an instance of SiameseCat, the following methods are called in order:

1. postPersistPetListenerMethod
2. postPersistCatListenerMethod
3. postPersistCatListener2Method
4. postPersistSiameseCatListenerMethod
5. postPersistAnimal
6. postPersistSiameseCat

Assume the definition of SiameseCat were instead:

```
@EntityListeners(SiameseCatListener.class)
@Entity
public class SiameseCat extends Cat {
	...
	@PostPersist
	protected void postPersistAnimal() {
		...
	}
}
```

CASE 3: In this case, the following methods would be called in order, where postPersistAnimal is the @PostPersist method defined in the SiameseCat class:

1. postPersistPetListenerMethod
2. postPersistCatListenerMethod
3. postPersistCatListener2Method
4. postPersistSiameseCatListenerMethod
5. postPersistAnimal

Entity listener methods are invoked in a specific order when a given event is fired. So-called default listeners are invoked first: these are listeners which have been defined in a package annotation or in the root element of XML mapping files. Next, entity listeners are invoked in the order of the inheritance hierarchy, with superclass listeners being invoked before subclass listeners. Finally, if an entity has multiple listeners for the same event, the listeners are invoked in declaration order.

You can exclude default listeners and listeners defined in superclasses from the invocation chain through the use of two class-level annotations:

- ExcludeDefaultListeners: This annotation indicates that no default listeners will be invoked for this class, or any of its subclasses. The XML equivalent is the empty exclude-default-listeners element.
- ExcludeSuperclassListeners: This annotation will cause JPA to skip invoking any listeners declared in superclasses. The XML equivalent is empty the exclude-superclass-listeners element.

- **Identify correct and incorrect statements about concurrency, including how it is managed through the use of @Version attributes and optimistic locking.**

  - [EJB_3.0_PERSISTENCE] 3.4.1; 3.4.2; 3.4.4

Optimistic locking is a technique that is used to insure that updates to the database data corresponding to the state of an entity are made only when NO intervening transaction has updated that data for the entity state since the entity state was read. This insures that updates or deletes to that data are consistent with the current state of the database and that intervening updates are NOT LOST. Transactions that would cause this constraint to be violated result in an OptimisticLockException being thrown and transaction ROLLBACK. Portable applications that wish to enable optimistic locking for entities must specify @Version attributes for those entities - i.e., persistent properties or fields annotated with the @Version annotation or specified in the XML descriptor as version attributes. Applications are strongly encouraged to enable optimistic locking for all entities that may be concurrently accessed or merged from a disconnected state. Failure to use optimistic locking may lead to inconsistent entity state, lost updates and other state irregularities. If optimistic locking is not defined as part of the entity state, the application must bear the burden of maintaining data consistency.

Version Attributes

The @Version field or property is used by the persistence provider to perform optimistic locking. It is accessed and/or set by the persistence provider in the course of performing lifecycle operations on the entity instance. An entity is automatically enabled for optimistic locking if it has a property or field mapped with a @Version mapping.

An entity may access the state of its version field or property or export a method for use by the application to access the version, but MUST NOT modify the version value. Only the persistence provider is permitted to set or update the value of the version attribute in the object.

The version attribute is updated by the persistence provider runtime when the object is WRITTEN to the database. All non-relationship fields and properties and all relationships owned

by the entity are included in version checks.

The persistence provider's implementation of the merge operation MUST examine the version attribute when an entity is being merged and throw an OptimisticLockException if it is discovered that the object being merged is a stale copy of the entity - i.e. that the entity has been updated since the entity became detached. Depending on the implementation strategy used, it is possible that this exception may not be thrown until flush is called or commit time, whichever happens first.

The persistence provider runtime is only required to use the version attribute when performing optimistic lock checking. Persistence provider implementations may provide additional mechanisms beside version attributes to enable optimistic lock checking. However, support for such mechanisms is not required of an implementation of EJB 3.0 specification.

If only some entities contain version attributes, the persistence provider runtime is required to check those entities for which version attributes have been specified. The consistency of the object graph is not guaranteed, but the absence of version attributes on some of the entities will not stop operations from completing.

Example:

```
@Entity
public class Article {

   @Id private long id;

   @Column(name="VERS")
   @Version
   private int ver;

      ...
}
```

The same metadata expressed in XML:

```
<entity-mappings>
      <entity class="Article">
            <attributes>
                  <id name="id"/>
                  <version name="ver">
                        <column name="VERS"/>
                  </version>
                  ...
            </attributes>
      </entity>
</entity-mappings>
```

OptimisticLockException

Provider implementations may defer writing to the database until the end of the transaction, when consistent with the flush mode setting in effect. In this case, the optimistic lock check may

not occur until commit time, and the OptimisticLockException may be thrown in the "before completion" phase of the commit. If the OptimisticLockException must be caught or handled by the application, the flush method should be used by the application to force the database writes to occur. This will allow the application to catch and handle optimistic lock exceptions.

The OptimisticLockException provides an API to return the object that caused the exception to be thrown. The object reference is not guaranteed to be present every time the exception is thrown but should be provided whenever the persistence provider can supply it. Applications cannot rely upon this object being available.

In some cases an OptimisticLockException will be thrown and wrapped by another exception, such as a RemoteException, when VM boundaries are crossed. Entities that may be referenced in wrapped exceptions should be Serializable so that marshalling will not fail.

An OptimisticLockException ALWAYS causes the transaction to ROLL BACK.

Refreshing objects or reloading objects in a new transaction context and then retrying the transaction is a potential response to an OptimisticLockException.

●Chapter 7. Persistence Units and Persistence Contexts

●Identify correct and incorrect statements or examples about JTA and resource-local entity managers.

- [EJB_3.0_PERSISTENCE] 5.5; 5.5.1; 5.5.2; 5.5.2.1; 5.5.3

Depending on the transactional type of the entity manager, transactions involving EntityManager operations may be controlled either through JTA or through use of the resource-local EntityTransaction API, which is mapped to a resource transaction over the resource that underlies the entities managed by the entity manager.
An entity manager whose underlying transactions are controlled through JTA is termed a **JTA entity manager**.

An entity manager whose underlying transactions are controlled by the application through the EntityTransaction API is termed a **resource-local entity manager**.

A **container-managed entity manager** MUST be a JTA entity manager. JTA entity managers are only specified for use in Java EE containers.

An **application-managed entity manager** may be either a JTA entity manager OR a resource-local entity manager.

An entity manager is defined to be of a given transactional type - either JTA or resource-local - at the time its underlying entity manager factory is configured and created.

Both JTA entity managers and resource-local entity managers are required to be supported in Java EE web containers and EJB containers. Within an EJB environment, a JTA entity manager is typically used. In general, in Java SE environments only resource-local entity managers are supported.

JTA EntityManagers

An entity manager whose transactions are controlled through JTA is a JTA entity manager. A JTA entity manager participates in the current JTA transaction, which is begun and committed external to the entity manager and propagated to the underlying resource manager.

An example persistence.xml file (Java EE) is shown below:

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd"
      version="1.0">

      <persistence-unit name="ShopPU" transaction-type="JTA">
            ...
      </persistence-unit>

</persistence>
```

NOTE: by default transaction type is JTA.

Resource-local EntityManagers

An entity manager whose transactions are controlled by the application through the EntityTransaction API is a resource-local entity manager. A resource-local entity manager transaction is mapped to a resource transaction over the resource by the persistence provider. Resource-local entity managers may use server or local resources to connect to the database and are unaware of the presence of JTA transactions that may OR may not be active.

An example persistence.xml file (Java SE) is shown below:

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
	xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
	xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd"
	version="1.0">

	<persistence-unit name="ShopPU" transaction-type="RESOURCE_LOCAL">
		...
	</persistence-unit>
</persistence>
```

The EntityTransaction Interface

The EntityTransaction interface is used to control resource transactions on resource-local entity managers. The EntityManager.getTransaction() method returns the EntityTransaction interface.

When a resource-local entity manager is used, and the persistence provider runtime throws an exception defined to cause transaction rollback, it MUST mark the transaction for rollback.

If the EntityTransaction.commit() operation fails, the persistence provider MUST roll back the transaction.

```
public interface EntityTransaction {

	public void begin();

	public void commit();

	public void rollback();

	public void setRollbackOnly();

	public boolean getRollbackOnly();

	public boolean isActive();
}
```

The begin, commit, and rollback methods demarcate transaction boundaries. The methods

should be self-explanatory: begin starts a transaction, commit attempts to commit the transaction's changes to the datastore, and rollback aborts the transaction, in which case the datastore is "rolled back" to its previous state. JPA implementations will automatically roll back transactions if any exception is thrown during the commit process.

Unless you are using an **extended persistence context**, committing or rolling back also ends the persistence context. All managed entites will be detached from the EntityManager.

The isActive() method returns true if the transaction is in progress (begin has been called more recently than commit or rollback), and false otherwise.

The following example illustrates the creation of an entity manager factory in a Java SE environment, and its use in creating and using a resource-local entity manager:

```
public class PasswordChanger {

	public static void main (String[] args) {
		EntityManagerFactory emf = Persistence.createEntityManagerFactory("Order");
		EntityManager em = emf.createEntityManager();

		em.getTransaction().begin();

		User user = (User)em.createQuery("SELECT u FROM User u WHERE u.name=:name AND
u.pass=:pass")
			.setParameter("name", args[0])
			.setParameter("pass", args[1])
			.getSingleResult();

		if (user!=null) {
			user.setPassword(args[2]);
		}

		em.getTransaction().commit();

		em.close();
		emf.close ();
	}
}
```

Example. Grouping Operations with Transactions:

```
public void transferFunds(EntityManager em, User from, User to, double amnt) {
	EntityTransaction trans = em.getTransaction();
	trans.begin();
	try {
		from.decrementAccount(amnt);
		to.incrementAccount(amnt);
		trans.commit();
	} catch (RuntimeException re) {
		if (trans.isActive()) {
			trans.rollback();   // or could attempt to fix error and retry
		}
		throw re;
	}
}
```

●Identify correct and incorrect statements or examples about container-managed persistence contexts.

- [EJB_3.0_PERSISTENCE] 5.6; 5.6.1; 5.6.2; 5.6.4.1; 5.6.4.2

When a container-managed entity manager is used, the lifecycle of the persistence context is always managed AUTOMATICALLY, transparently to the application, and the persistence context is propagated with the JTA transaction.
A container-managed persistence context may be defined to have either a lifetime that is scoped to a SINGLE transaction or an EXTENDED lifetime that spans MULTIPLE transactions, depending on the PersistenceContextType that is specified when its EntityManager is created. This specification refers to such persistence contexts as *transaction-scoped* persistence contexts and *extended* persistence contexts respectively.

The lifetime of the persistence context is declared using the PersistenceContext annotation or the persistence-context-ref deployment descriptor element. By DEFAULT, a **transaction-scoped** persistence context is used.

Container-managed Transaction-scoped Persistence Context

The application may obtain a container-managed entity manager with transaction-scoped persistence context bound to the JTA transaction by injection or direct lookup in the JNDI namespace. The persistence context type for the entity manager is defaulted or defined as PersistenceContextType.TRANSACTION.

A new persistence context begins when the container-managed entity manager is invoked in the scope of an active JTA transaction, and there is no current persistence context already associated with the JTA transaction. The persistence context is created and then associated with the JTA transaction.

The persistence context ends when the associated JTA transaction commits or rolls back, and all entities that were managed by the EntityManager become detached.

If the entity manager is invoked outside the scope of a transaction, any entities loaded from the database will immediately become detached at the end of the method call.

Example:

```
@Stateless
public class ShoppingCartImpl implements ShoppingCart {

    @PersistenceContext
    EntityManager em;

    public Order getOrder(Long id) {
        return em.find(Order.class, id);
    }

    public Product getProduct(String name) {
        return (Product) em.createQuery("select p from Product p where p.name = :name")
            .setParameter("name", name)
            .getSingleResult();
    }
```

```
    public LineItem createLineItem(Order order, Product product, int quantity) {
        LineItem li = new LineItem(order, product, quantity);
        order.getLineItems().add(li);
        em.persist(li);
        return li;
    }
}
```

Under the **transaction persistence context** model, an EntityManager begins a new persistence context with each transaction, and ends the context when the transaction commits or rolls back. Within the transaction, entities you retrieve through the EntityManager or via Queries are managed entities. They can access datastore resources to lazy-load additional persistent state as needed, and only one entity may exist for any persistent identity.

When the transaction completes, all entities lose their association with the EntityManager and become detached. Traversing a persistent field that wasn't already loaded now has undefined results. And using the EntityManager or a Query to retrieve additional objects may now create new instances with the same persistent identities as detached instances.

If you use an EntityManager with a transaction persistence context model outside of an active transaction, each method invocation creates a new persistence context, performs the method action, and ends the persistence context. For example, consider using the EntityManager.find(...) method outside of a transaction. The EntityManager will create a temporary persistence context, perform the find(...) operation, end the persistence context, and return the detached result object to you. A second call with the same id will return a second detached object.

When the next transaction begins, the EntityManager will begin a new persistence context, and will again start returning managed entities. You can also merge the previously-detached entites back into the new persistence context.

The following code illustrates the behavior of entites under an EntityManager using a **transaction persistence context**:

```
EntityManager em; // injected
...

// outside a transaction:

// each operation occurs in a separate persistence context, and returns
// a new detached instance
Magazine mag1 = em.find(Magazine.class, magId);
Magazine mag2 = em.find(Magazine.class, magId);
assertTrue(mag2 != mag1);
...

// transaction begins:

// within a transaction, a subsequent lookup doesn't return any of the
// detached objects.  however, two lookups within the same transaction
// return the same instance, because the persistence context spans the
// transaction
Magazine mag3 = em.find(Magazine.class, magId);
```

```
assertTrue(mag3 != mag1 && mag3 != mag2);
Magazine mag4 = em.find(Magazine.class (magId);
assertTrue(mag4 == mag3);
...

// transaction commits:

// once again, each operation returns a new instance
Magazine mag5 = em.find(Magazine.class, magId);
assertTrue(mag5 != mag3);
```

Container-managed Extended Persistence Context

A container-managed EXTENDED persistence context can ONLY be initiated within the scope of a **stateful session bean**. It exists from the point at which the stateful session bean that declares a dependency on an entity manager of type PersistenceContextType.EXTENDED is created, and is said to be bound to the stateful session bean. The dependency on the extended persistence context is declared by means of the PersistenceContext annotation or persistence-context-ref deployment descriptor element.

The persistence context is closed by the container when the @Remove method of the stateful session bean completes (or the stateful session bean instance is otherwise destroyed).

Example:

```
@Stateful
@Transaction(REQUIRES_NEW)
public class ShoppingCartImpl implements ShoppingCart {

	@PersistenceContext(type=EXTENDED)
	EntityManager em;

	private Order order;

	private Product product;

	public void initOrder(Long id) {
		order = em.find(Order.class, id);
	}

	public void initProduct(String name) {
		product = (Product) em.createQuery("select p from Product p where p.name = :name")
			.setParameter("name", name)
			.getSingleResult();
	}

	public LineItem createLineItem(int quantity) {
		LineItem li = new LineItem(order, product, quantity);
		order.getLineItems().add(li);
		return li;
	}
}
```

An EntityManager using an extended persistence context maintains the same persistence context for its entire lifecycle. Whether inside a transaction or not, all entities returned from the EntityManager are MANAGED, and the EntityManager never creates two entity instances to represent the same persistent identity. Entities only become detached when you finally close the EntityManager (or when they are serialized).

The following code illustrates the behavior of entites under an EntityManager using an **extended persistence context**:

```
EntityManagerFactory emf = ...
EntityManager em = emf.createEntityManager();

// persistence context active for entire life of EM, so only one entity
// for a given persistent identity
Magazine mag1 = em.find(Magazine.class, magId);
Magazine mag2 = em.find(Magazine.class, magId);
assertTrue(mag2 == mag1);

em.getTransaction().begin();

// same persistence context active within the transaction
Magazine mag3 = em.find(Magazine.class, magId);
assertTrue(mag3 == mag1);
Magazine mag4 = em.find(Magazine.class (magId);
assertTrue(mag4 == mag1);

em.getTransaction.commit ();

// when the transaction commits, instance still managed
Magazine mag5 = em.find(Magazine.class, magId);
assertTrue(mag5 == mag1);

// instance finally becomes detached when EM closes
em.close();
```

●Identify correct and incorrect statements or examples about application-managed persistence contexts.

- [EJB_3.0_PERSISTENCE] 5.7; 5.7.1.1; 5.7.1.2; 5.7.1.3; 5.7.1.4

When an application-managed entity manager is used, the application interacts directly with the persistence provider's entity manager factory to manage the entity manager lifecycle and to obtain and destroy persistence contexts.
All such application-managed persistence contexts are EXTENDED in scope, and may span MULTIPLE transactions.

The EntityManager close and isOpen methods are used to manage the lifecycle of an application-managed entity manager and its associated persistence context.

The EntityManager.close() method closes an entity manager to release its persistence context and other resources. After calling close(), the application MUST NOT invoke any further methods on the EntityManager instance except for getTransaction and isOpen, or the

IllegalStateException will be thrown. If the close() method is invoked when a transaction is active, the persistence context remains managed until the transaction completes.

The EntityManager.isOpen() method indicates whether the entity manager is open. The isOpen() method returns true until the entity manager has been closed.

The **extended** persistence context exists from the point at which the entity manager has been created using EntityManagerFactory.createEntityManager until the entity manager is closed by means of EntityManager.close. The extended persistence context obtained from the application-managed entity manager is a stand-alone persistence context - it is not propagated with the transaction.

When a JTA application-managed entity manager is used, if the entity manager is created outside the scope of the current JTA transaction, it is the responsibility of the application to associate the entity manager with the transaction (if desired) by calling EntityManager.joinTransaction.

Example of application-managed Persistence Context used in Stateless Session Bean:

```
/*
 * Container-managed transaction demarcation is used.
 * Session bean creates and closes an entity manager in
 * each business method.
 */
@Stateless
public class ShoppingCartImpl implements ShoppingCart {

	@PersistenceUnit
	private EntityManagerFactory emf;

	public Order getOrder(Long id) {
		EntityManager em = emf.createEntityManager();
		Order order = (Order)em.find(Order.class, id);
		em.close();
		return order;
	}

	public Product getProduct() {
		EntityManager em = emf.createEntityManager();
		Product product = (Product) em.createQuery("select p from Product p where p.name = :name")
			.setParameter("name", name)
			.getSingleResult();
		em.close();
		return product;
	}

	public LineItem createLineItem(Order order, Product product, int quantity) {
		EntityManager em = emf.createEntityManager();
		LineItem li = new LineItem(order, product, quantity);
		order.getLineItems().add(li);
		em.persist(li);
		em.close();
		return li; // remains managed until JTA transaction ends
	}
}
```

Another example of application-managed Persistence Context used in Stateless Session Bean:

```
/*
 * Container-managed transaction demarcation is used.
 * Session bean creates entity manager in PostConstruct
 * method and clears persistence context at the end of each
 * business method.
 */
@Stateless
public class ShoppingCartImpl implements ShoppingCart {

	@PersistenceUnit
	private EntityManagerFactory emf;

	private EntityManager em;

	@PostConstruct
	public void init()
		em = emf.createEntityManager();
	}

	public Order getOrder(Long id) {
		Order order = (Order)em.find(Order.class, id);
		em.clear(); // entities are detached
		return order;
	}

	public Product getProduct() {
		Product product = (Product) em.createQuery("select p from Product p where p.name = :name")
			.setParameter("name", name)
			.getSingleResult();
		em.clear();
		return product;
	}

	public LineItem createLineItem(Order order, Product product, int quantity) {
		em.joinTransaction();
		LineItem li = new LineItem(order, product, quantity);
		order.getLineItems().add(li);
		em.persist(li);
		// persistence context is flushed to database;
		// all updates will be committed to database on tx commit
		em.flush();
		// entities in persistence context are detached
		em.clear();
		return li;
	}

	@PreDestroy
	public void destroy()
		em.close();
	}
}
```

Example of application-managed Persistence Context used in Stateful Session Bean

```
//Container-managed transaction demarcation is used
@Stateful
public class ShoppingCartImpl implements ShoppingCart {

	@PersistenceUnit
	private EntityManagerFactory emf;

	private EntityManager em;

	private Order order;

	private Product product;

	@PostConstruct
	public void init() {
		em = emf.createEntityManager();
	}

	public void initOrder(Long id) {
		order = em.find(Order.class, id);
	}

	public void initProduct(String name) {
		product = (Product) em.createQuery("select p from Product p where p.name = :name")
			.setParameter("name", name)
			.getSingleResult();
	}

	public LineItem createLineItem(int quantity) {
		em.joinTransaction();
		LineItem li = new LineItem(order, product, quantity);
		order.getLineItems().add(li);
		return li;
	}

	@Remove
	public void destroy() {
		em.close();
	}
}
```

Example of application-managed Persistence Context with Resource Transaction

```
// Usage in an ordinary Java class
public class ShoppingCart {

	private EntityManager em;

	private EntityManagerFactory emf;

	public ShoppingCart() {
		emf = Persistence.createEntityManagerFactory("orderMgt");
		em = emf.createEntityManager();
```

```
    }

    private Order order;

    private Product product;

    public void initOrder(Long id) {
        order = em.find(Order.class, id);
    }

    public void initProduct(String name) {
        product = (Product) em.createQuery("select p from Product p where p.name = :name")
                .setParameter("name", name)
                .getSingleResult();
    }

    public LineItem createLineItem(int quantity) {
        em.getTransaction().begin();
        LineItem li = new LineItem(order, product, quantity);
        order.getLineItems().add(li);
        em.getTransaction().commit();
        return li;
    }

    public void destroy() {
        em.close();
        emf.close();
    }
}
```

To summarize comparison of container- and application-managed EntityManagers:

1. **Application**-managed EntityManager
   - The lifecycle of EntityManager - EntityManagerFactory
   - Transaction тАУ JTA or Resource-local
   - Persistence Context - EntityManager.clear()
2. **Container**-managed EntityManager
   - Container manages lifecycle of EntityManager, Transaction
   - JTA Transaction **ONLY**
   - Transaction-scoped persistence context (by default)
   - Extended persistence context (only in Stateful Session bean) - keeps entities managed across multiple transaction.

- **Identify correct and incorrect statements or examples about transaction management for persistence contexts, including persistence context propagation, the use of the EntityManager.joinTransaction() method, and the EntityTransaction API.**

  - [EJB_3.0_PERSISTENCE] 5.6.3

A single persistence context may correspond to one or more JTA entity manager instances (all associated with the same entity manager factory).
The persistence context is propagated across the entity manager instances as the JTA transaction is propagated.

Propagation of persistence contexts only applies within a local environment. Persistence contexts are NOT propagated to REMOTE tiers.

Persistence contexts are propagated by the container across component invocations as follows.

1. If a component is called and there is no JTA transaction or the JTA transaction is not propagated, the persistence context is not propagated.
2. If an entity manager is then invoked from within the component:
   - Invocation of an entity manager defined with PersistenceContextType.TRANSACTION will result in use of a NEW persistence context.
   - Invocation of an entity manager defined with PersistenceContextType.EXTENDED will result in the use of the existing extended persistence context bound to that component.
   - If the entity manager is invoked within a JTA transaction, the persistence context will be bound to the JTA transaction.
3. If a component is called and the JTA transaction is propagated into that component:
   - If the component is a stateful session bean to which an extended persistence context has been bound and there is a different persistence context bound to the JTA transaction, an EJBException is thrown by the container.
   - Otherwise, if there is a persistence context bound to the JTA transaction, that persistence context is propagated and used.

Application-managed entity manager are always EXTENDED. An injected extended persistence context is automatically associated with a JTA transaction by the EJB container. For other extended contexts created manually with the EntityManagerFactory API, you MUST call EntityManager.joinTransaction() to perform the transaction association.

Example of application-managed entity manager:

```
// JTA transaction is used
EntityManager em = emf.createEntityManager();
ut.begin();
em.joinTransaction();
```

```
em.persist(customer);
ut.commit();
em.close();
```

NOTE: It is NOT legal to call EntityManager.joinTransaction() if no JTA transaction is involved.

NOTE: EntityManager.joinTransaction() is required to be invoked only when an EntityManager is created explicitly using an EntityManagerFactory. If you are using EJB container-managed persistence contexts, then you DO NOT NEED to perform this extra step.

●Identify correct and incorrect statements or examples about persistence units, how persistence units are packaged, and the use of the persistence.xml file.

- [EJB_3.0_PERSISTENCE] 6.1; 6.2; 6.2.1; 6.2.1.8; 6.2.2

A **persistence unit** is a logical grouping that includes:

- An entity manager factory and its entity managers, together with their configuration information.
- The set of managed classes included in the persistence unit and managed by the entity managers of the entity manager factory.
- Mapping metadata (in the form of metadata annotations and/or XML metadata) that specifies the mapping of the classes to the database.

Persistence Unit Packaging

Within Java EE environments, an EJB-JAR, WAR, EAR, or application client JAR can define a persistence unit. Any number of persistence units may be defined within these scopes.

A persistence unit may be packaged within one or more jar files contained within a WAR or EAR, as a set of classes within an EJB-JAR file or in the WAR classes directory, or as a combination of these as defined below.

A persistence unit is defined by a persistence.xml file. The jar file or directory whose META-INF directory contains the persistence.xml file is termed the *root* of the persistence unit. In Java EE, the root of a persistence unit may be one of the following:

- an EJB-JAR file
- the WEB-INF/classes directory of a WAR file
- NOTE: the root of the persistence unit is the WEB-INF/classes directory; the persistence.xml file is therefore contained in the WEB-INF/classes/META-INF directory
- a jar file in the WEB-INF/lib directory of a WAR file
- a jar file in the root of the EAR
- a jar file in the EAR library directory
- an application client jar file

It is not required that an EJB-JAR or WAR containing a persistence unit be packaged in an EAR

unless the persistence unit contains persistence classes in addition to those contained in the EJB-JAR or WAR.

A persistence unit MUST have a name. Only one persistence unit of any given name may be defined within a single EJB-JAR file, within a single WAR file, within a single application client jar, or within an EAR (in the EAR root or lib directory).

The persistence.xml file may be used to designate MORE THAN ONE persistence unit within the same scope.

All persistence classes defined at the level of the Java EE EAR must be accessible to all other Java EE components in the application - i.e. loaded by the application classloader - such that if the same entity class is referenced by two different Java EE components (which may be using different persistence units), the referenced class is the same identical class.

In Java SE environments, the metadata mapping files, jar files, and classes can be used. To insure the portability of a Java SE application, it is necessary to explicitly list the managed persistence classes that are included in the persistence unit.

persistence.xml file

A persistence.xml file defines a persistence unit. It may be used to specify managed persistence classes included in the persistence unit, object/relational mapping information for those classes, and other configuration information for the persistence unit and for the entity manager(s) and entity manager factory for the persistence unit. The persistence.xml file is located in the META-INF directory of the root of the persistence unit. This information may be defined by containment or by reference, as described below.

The object/relational mapping information may take the form of annotations on the managed persistence classes included in the persistence unit, one or more XML files contained in the root of the persistence unit, one or more XML files outside the root of the persistence unit on the classpath and referenced from the persistence.xml, or a combination of these.

The managed persistence classes may either be contained within the root of the persistence unit; or they may be specified by reference - i.e., by naming the classes, class archives, or mapping XML files (which in turn reference classes) that are accessible on the application classpath; or they may be specified by some combination of these means.

The persistence element consists of one or more persistence-unit elements.

The persistence-unit element consists of the name and transaction-type attributes and the following sub-elements:

- description
- provider
- jta-data-source
- non-jta-data-source
- mapping-file
- jar-file
- class

- exclude-unlisted-classes
- properties

The name attribute is REQUIRED; the other attributes and elements are OPTIONAL.

Examples:

```
<persistence>
    <persistence-unit name="OrderManagement">
        <description>
            This unit manages orders and customers.
            It does not rely on any vendor-specific features and can
            therefore be deployed to any persistence provider.
        </description>
        <jta-data-source>jdbc/MyOrderDB</jta-data-source>
        <mapping-file>ormap.xml</mapping-file>
        <jar-file>MyOrderApp.jar</jar-file>
        <class>com.widgets.Order</class>
        <class>com.widgets.Customer</class>
    </persistence-unit>
</persistence>

<persistence>
    <persistence-unit name="OrderManagement2">
        <description>
            This unit manages inventory for auto parts.
            It depends on features provided by the
            com.acme.persistence implementation.
        </description>
        <provider>com.acme.AcmePersistence</provider>
        <jta-data-source>jdbc/MyPartDB</jta-data-source>
        <mapping-file>ormap2.xml</mapping-file>
        <jar-file>MyPartsApp.jar</jar-file>
        <properties>
            <property name="com.acme.persistence.sql-logging" value="on"/>
        </properties>
    </persistence-unit>
</persistence>
```

The following are sample contents of a persistence.xml file.

```
<persistence-unit name="OrderManagement"/>
```

A persistence unit named OrderManagement is created. Any annotated managed persistence classes found in the root of the persistence unit are added to the list of managed persistence classes. If a META-INF/orm.xml file exists, any classes referenced by it and mapping information contained in it are used as specified above. Because no provider is specified, the persistence unit is assumed to be portable across providers. Because the transaction type is not specified, JTA is assumed. The container must provide the data source (it may be specified at application deployment, for example); in Java SE environments, the data source may be specified by others means.

```
<persistence-unit name="OrderManagement2">
    <mapping-file>mappings.xml</mapping-file>
</persistence-unit>
```

A persistence unit named OrderManagement2 is created. Any annotated managed persistence classes found in the root of the persistence unit are added to the list of managed persistence classes. The mappings.xml resource exists on the classpath and any classes and mapping information contained in it are used as specified above. If a META-INF/orm.xml file exists, any classes and mapping information contained in it are used as well. The transaction type, data source, and provider are as described above.

```
<persistence-unit name="OrderManagement3">
    <jar-file>order.jar</jar-file>
    <jar-file>order-supplemental.jar</jar-file>
</persistence-unit>
```

A persistence unit named OrderManagement3 is created. Any annotated managed persistence classes found in the root of the persistence unit are added to the list of managed persistence classes. If a META-INF/orm.xml file exists, any classes and mapping information contained in it are used as specified above. The order.jar and order-supplemental.jar files are searched for managed persistence classes and any annotated managed persistence classes found in them and/or any classes specified in the orm.xml files of these jar files are added. The transaction-type, data source and provider are as described above.

```
<persistence-unit name="OrderManagement4" transaction-type=RESOURCE_LOCAL>
    <non-jta-data-source>jdbc/MyDB</jta-data-source>
    <mapping-file>order-mappings.xml</mapping-file>
    <exclude-unlisted-classes/>
    <class>com.acme.Order</class>
    <class>com.acme.Customer</class>
    <class>com.acme.Item</class>
</persistence-unit>
```

A persistence unit named OrderManagement4 is created. The order-mappings.xml is read as a resource and any classes referenced by it and mapping information contained in it are used. The annotated Order, Customer and Item classes are loaded and are added. No (other) classes contained in the root of the persistence unit are added to the list of managed persistence classes. The persistence unit is portable across providers. A entity manager factory supplying resource-local entity managers will be created. The data source jdbc/MyDB must be used.

```
<persistence-unit name="OrderManagement5">
    <provider>com.acme.AcmePersistence</provider>
    <mapping-file>order1.xml</mapping-file>
    <mapping-file>order2.xml</mapping-file>
    <jar-file>order.jar</jar-file>
    <jar-file>order-supplemental.jar</jar-file>
</persistence-unit>
```

A persistence unit named OrderManagement5 is created. Any annotated managed persistence classes found in the root of the persistence unit are added to the list of managed classes. The order1.xml and order2.xml files are read as resources and any classes referenced by them and mapping information contained in them are also used as specified above. The order.jar is a jar file on the classpath containing another persistence unit, while order-supplemental.jar is just a library of classes. Both of these jar files are searched for annotated managed persistence classes and any annotated managed persistence classes

found in them and/or any classes specified in the orm.xml files (if any) of these jar files are added. The provider com.acme.AcmePersistence must be used.

Note that the persistence.xml file contained in order.jar is not used to augment the persistence unit OrderManagement5 with the classes of the persistence unit whose root is order.jar.

Persistence Unit Scope

An EJB-JAR, WAR, application client jar, or EAR can define a persistence unit.

When referencing a persistence unit using the unitName annotation element or persistence-unit-name deployment descriptor element, the visibility scope of the persistence unit is determined by its point of definition. A persistence unit that is defined at the level of an EJB-JAR,WAR, or application client jar is scoped to that EJB-JAR, WAR, or application jar respectively and is visible to the components defined in that jar or war. A persistence unit that is defined at the level of the EAR is generally visible to all components in the application.

However, if a persistence unit of the same name is defined by an EJB-JAR, WAR, or application jar file within the EAR, the persistence unit of that name defined at EAR level will not be visible to the components defined by that EJB-JAR, WAR, or application jar file unless the persistence unit reference uses the persistence unit name # syntax to specify a path name to disambiguate the reference. When the # syntax is used, the path name is relative to the referencing application component jar file. For example, the syntax ../lib/persistenceUnitRoot.jar#myPersistenceUnit refers to a persistence unit whose name, as specified in the name element of the persistence.xml file, is myPersistenceUnit and for which the relative path name of the root of the persistence unit is ../lib/persistenceUnitRoot.jar. The # syntax may be used with both the unitName annotation element or persistence-unit-name deployment descriptor element to reference a persistence unit defined at EAR level.

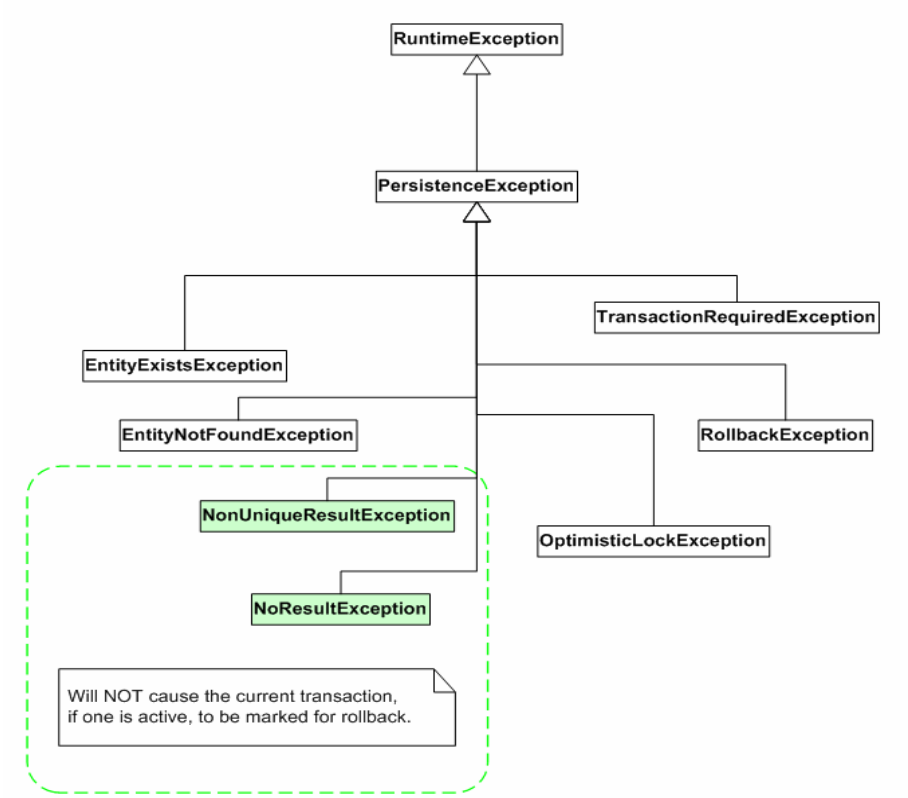
●Identify correct and incorrect statements or examples about the effect of persistence exceptions on transactions and persistence contexts.

- [EJB_3.0_PERSISTENCE] 3.7

PersistenceException

- The PersistenceException is thrown by the persistence provider when a problem occurs. It may be thrown to report that the invoked operation could not complete because of an unexpected error (e.g., failure of the persistence provider to open a database connection). All other exceptions defined by this specification are subclasses of the PersistenceException. All instances of PersistenceException except for instances of NoResultException and NonUniqueResultException will cause the current transaction, if one is active, to be marked for ROLLBACK.
- TransactionRequiredException
- The TransactionRequiredException is thrown by the persistence provider when a transaction is required but is NOT active.
- OptimisticLockException

- The OptimisticLockException is thrown by the persistence provider when an optimistic locking CONFLICT occurs. This exception may be thrown as part of an API call, at flush, or at commit time. The current transaction, if one is active, will be marked for ROLLBACK.
- RollbackException
- The RollbackException is thrown by the persistence provider when EntityTransaction.commit() fails.



- EntityExistsException
- The EntityExistsException may thrown by the persistence provider when the persist operation is invoked and the entity already exists. The EntityExistsException may be thrown when the persist operation is invoked, or the EntityExistsException or another PersistenceException may be thrown at commit time.
- EntityNotFoundException
- The EntityNotFoundException is thrown by the persistence provider when an entity reference obtained by getReference is accessed but the entity does not exist. It is also thrown by the refresh operation when the entity no longer exists in the database. The current transaction, if one is active, will be marked for ROLLBACK.
- NoResultException
- The NoResultException is thrown by the persistence provider when Query.getSingleResult is invoked and there is NO result to return. This exception will

NOT cause the current transaction, if one is active, to be marked for roll back.

- NonUniqueResultException
- The NonUniqueResultException is thrown by the persistence provider when Query.getSingleResult is invoked and there is more than one result from the query. This exception will NOT cause the current transaction, if one is active, to be marked for roll back.

●Chapter 8. Java Persistence Query Language

●Develop queries that use the SELECT clause to determine query results, including the use of entity types, use of aggregates, and returning multiple values.

- [EJB_3.0_PERSISTENCE] 4.2.1; 4.3.1; 4.3.2; 4.8; 4.8.1; 4.8.2; 4.8.4; 4.8.4.1

A select statement is a string which consists of the following clauses:

- a SELECT clause, which determines the type of the objects or values to be selected.
- a FROM clause, which provides declarations that designate the domain to which the expressions specified in the other clauses of the query apply.
- an optional WHERE clause, which may be used to restrict the results that are returned by the query.
- an optional GROUP BY clause, which allows query results to be aggregated in terms of groups.
- an optional HAVING clause, which allows filtering over aggregated groups.
- an optional ORDER BY clause, which may be used to order the results that are returned by the query.

A select statement must always have a SELECT and a FROM clause. The other clauses are optional.

Example. Find all orders:

```
SELECT o
FROM Order o
```

Entities are designated in query strings by their entity names. The entity name is defined by the name element of the @Entity annotation (or the entity-name XML descriptor element), and defaults to the unqualified name of the entity class. Entity names are scoped within the persistence unit and must be unique within the persistence unit.

This example assumes that the application developer provides several entity classes, representing orders, products, line items, shipping addresses, and billing addresses. The abstract schema types for these entities are Order, Product, LineItem, ShippingAddress, and BillingAddress respectively. These entities are logically in the same persistence unit, as shown in the figure below:

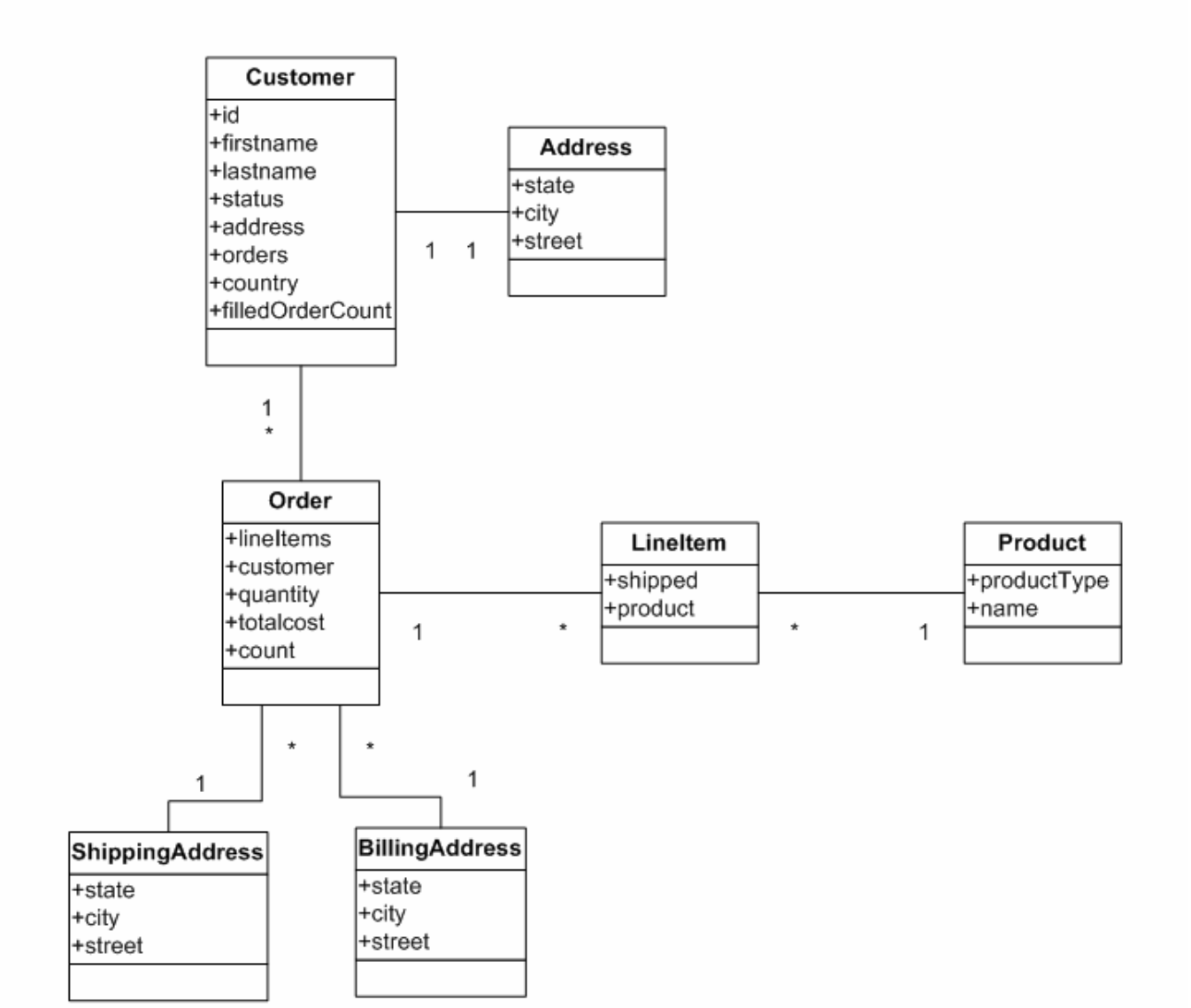
The entities ShippingAddress and BillingAddress each have one-to-many relationships with Order. There is also a one-to-many relationship between Order and Lineitem. The entity LineItem is related to Product in a many-to-one relationship.

Queries to select orders can be defined by navigating over the association-fields and state-fields

defined by Order and LineItem. A query to find all orders with pending line items might be written as follows:

```
SELECT DISTINCT o
FROM Order AS o JOIN o.lineItems AS l
WHERE l.shipped = FALSE
```

This query navigates over the association-field lineItems of the abstract schema type Order to find line items, and uses the state-field shipped of LineItem to select those orders that have at least one line item that has not yet shipped. (Note that this query does not select orders that



have no line items.)

Although predefined reserved identifiers, such as DISTINCT, FROM, AS, JOIN, WHERE, and FALSE appear in upper case in this example, predefined reserved identifiers are case insensitive.

The SELECT clause of this example designates the return type of this query to be of type Order.

Because the same persistence unit defines the abstract persistence schemas of the related entities, the developer can also specify a query over orders that utilizes the abstract schema type for products, and hence the state-fields and association-fields of both the abstract schema types Order and Product. For example, if the abstract schema type Product has a state-field named productType, a query over orders can be specified using this state-field. Such a query might be to find all orders for products with product type office supplies. A query for this might be as follows:

```
SELECT DISTINCT o
FROM Order o JOIN o.lineItems l JOIN l.product p
WHERE p.productType = 'office_supplies'
```

Because Order is related to Product by means of the relationships between Order and LineItem and between LineItem and Product, navigation using the association-fields lineItems and product is used to express the query. This query is specified by using the abstract schema name Order, which designates the abstract schema type over which the query ranges. The basis for the navigation is provided by the association-fields lineItems and product of the abstract schema types Order and LineItem respectively.

SELECT Clause

The SELECT clause denotes the query result. MORE THAN ONE value may be returned from the SELECT clause of a query.

The SELECT clause may contain one or more of the following elements: a single range variable or identification variable that ranges over an entity abstract schema type, a single-valued path expression, an aggregate select expression, a constructor expression.

For example:

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

Note that the SELECT clause must be specified to return ONLY single-valued expressions. The query below is therefore NOT VALID:

```
SELECT o.lineItems FROM Order AS o
```

The DISTINCT keyword is used to specify that duplicate values must be eliminated from the query result. If DISTINCT is not specified, duplicate values are not eliminated.

Result Type of the SELECT Clause

The type of the query result specified by the SELECT clause of a query is an entity abstract schema type, a state-field type, the result of an aggregate function, the result of a construction operation, or some sequence of these.

The result type of the SELECT clause is defined by the the result types of the select_expressions contained in it. When multiple select_expressions are used in the SELECT clause, the result of the query is of type Object[], and the elements in this result correspond in order to the order of their specification in the SELECT clause and in type to the result types of each of the select_expressions.

```
Query q = em.createQuery("SELECT MIN(o.totalcost), MAX(o.totalcost) FROM Order o");
```

```
Object[] stats = (Object[]) q.getSingleResult();
```

Constructor Expressions in the SELECT Clause

A constructor may be used in the SELECT list to return one or more Java instances. The specified class is not required to be an entity or to be mapped to the database. The constructor name must be FULLY qualified:

```
Query q = em.createQuery("SELECT NEW OrderCostData(MIN(o.totalcost), MAX(o.totalcost)) FROM Order o");
OrderCostData stats = (OrderCostData) q.getSingleResult();
```

If an entity class name is specified in the SELECT NEW clause, the resulting entity instances are in the NEW state:

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

Aggregate Functions in the SELECT Clause

The result of a query may be the result of an aggregate function applied to a path expression.

The following aggregate functions can be used in the SELECT clause of a query: AVG, COUNT, MAX, MIN, SUM.

For all aggregate functions except COUNT, the path expression that is the argument to the aggregate function must terminate in a state-field. The path expression argument to COUNT may terminate in either a state-field or a association-field, or the argument to COUNT may be an identification variable.

Arguments to the functions SUM and AVG must be numeric. Arguments to the functions MAX and MIN must correspond to orderable state-field types (i.e., numeric types, string types, character types, or date types).

The Java type that is contained in the result of a query using an aggregate function is as follows:

- COUNT returns Long.
- The following query returns the total number of magazines:
- SELECT COUNT(mag) FROM Magazine mag
- MAX, MIN return the type of the state-field to which they are applied.
- The following query will return the highest price of all the magazines titled "JDJ":
- EntityManager em = ...
- Query q = em.createQuery("SELECT MAX(x.price) FROM Magazine x WHERE x.title = 'JDJ'");
- Number result = (Number) q.getSingleResult();
- AVG returns Double.
- The following query will return the average of all the prices of all the magazines:
- EntityManager em = ...
- Query q = em.createQuery("SELECT AVG(x.price) FROM Magazine x");
- Number result = (Number) q.getSingleResult();
- SUM returns Long when applied to state-fields of integral types (other than BigInteger); Double when applied to state-fields of floating point types; BigInteger when applied to state-fields of type BigInteger; and BigDecimal when applied to state-fields of type

BigDecimal.

- The following query returns the sum total cost of all the prices from all the magazines published by 'Larry':
- SELECT SUM(mag.price) FROM Publisher pub JOIN pub.magazines mag WHERE pub.firstName = 'Larry'

If SUM, AVG, MAX, or MIN is used, and there are no values to which the aggregate function can be applied, the result of the aggregate function is NULL.

If COUNT is used, and there are no values to which COUNT can be applied, the result of the aggregate function is 0.

The argument to an aggregate function may be preceded by the keyword DISTINCT to specify that duplicate values are to be eliminated before the aggregate function is applied.

Null values are eliminated before the aggregate function is applied, regardless of whether the keyword DISTINCT is specified.

The following query returns the average order quantity:

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

The following query returns the total cost of the items that Mikalai Zaikin has ordered:

```
SELECT SUM(l.price)
FROM Order o JOIN o.lineItems l JOIN o.customer c
WHERE c.lastname = 'Zaikin' AND c.firstname = 'Mikalai'
```

The following query returns the total number of orders:

```
SELECT COUNT(o)
FROM Order o
```

The following query counts the number of items in Mikalai Zaikin's order for which prices have been specified:

```
SELECT COUNT(l.price)
FROM Order o JOIN o.lineItems l JOIN o.customer c
WHERE c.lastname = 'Zaikin' AND c.firstname = 'Mikalai'
```

Note that this is equivalent to:

```
SELECT COUNT(l)
FROM Order o JOIN o.lineItems l JOIN o.customer c
WHERE c.lastname = 'Zaikin' AND c.firstname = 'Mikalai'
    AND l.price IS NOT NULL
```

●Develop queries that use Java Persistence Query Language syntax for defining the domain of a query using JOIN clauses, IN, and prefetching.

- [EJB_3.0_PERSISTENCE] 4.4.5; 4.4.5.1; 4.4.5.2; 4.4.5.3; 4.4.6

An inner join may be implicitly specified by the use of a cartesian product in the FROM clause and a join condition in the WHERE clause. In the absence of a join condition, this reduces to the cartesian product.

The main use case for this generalized style of join is when a join condition does not involve a foreign key relationship that is mapped to an **entity relationship**.

Example:

```
SELECT c FROM Customer c, Employee e WHERE c.hatsize = e.shoesize
```

In general, use of this style of inner join (also referred to as theta-join) is less typical than explicitly defined joins over **entity relationships**.

Inner Joins (Relationship Joins)

The syntax for the inner join operation is:

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

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

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

The keyword INNER may optionally be used:

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

This is equivalent to the following query using the earlier IN construct, defined in EJB 2.1. It selects those customers of status 1 for which at least one order exists:

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

Left Outer Joins

LEFT JOIN and LEFT OUTER JOIN are synonymous. They enable the retrieval of a set of entities where matching values in the join condition may be absent.

The syntax for a left outer join is:

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

For example:

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

The keyword OUTER may optionally be used:

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

An important use case for LEFT JOIN is in enabling the prefetching of related data items as a side effect of a query. This is accomplished by specifying the LEFT JOIN as a FETCH JOIN.

Fetch Joins

A FETCH JOIN enables the fetching of an association as a side effect of the execution of a query. A FETCH JOIN is specified over an entity and its related entities.

The syntax for a fetch join is:

fetch_join ::= [LEFT [OUTER] | INNER] JOIN FETCH join_association_path_expression

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

The following query returns a set of departments. As a side effect, the associated employees for those departments are also retrieved, even though they are not part of the explicit query result. The persistent fields or properties of the employees that are eagerly fetched are fully initialized. The initialization of the relationship properties of the employees that are retrieved is determined by the metadata for the Employee entity class.

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

A fetch join has the same join semantics as the corresponding inner or outer join, except that the related objects specified on the right-hand side of the join operation are not returned in the query result or otherwise referenced in the query. Hence, for example, if department 1 has five employees, the above query returns five references to the department 1 entity.

JPQL queries may specify one or more JOIN FETCH declarations, which allow the query to specify which fields in the returned instances will be PRE-FETCHED:

```
SELECT x FROM Magazine x JOIN FETCH x.articles WHERE x.title = 'JDJ'
```

The query above returns Magazine instances and GUARANTEES that the articles field will already be fetched in the returned instances.

Multiple fields may be specified in separate JOIN FETCH declarations:

```
SELECT x FROM Magazine x JOIN FETCH x.articles JOIN FETCH x.authors WHERE x.title = 'JDJ'
```

Collection Member Declarations

An identification variable declared by a *collection_member_declaration* ranges over values of a collection obtained by navigation using a path expression. Such a path expression represents a navigation involving the association-fields of an entity abstract schema type. Because a path expression can be based on another path expression, the navigation can use the association-fields of related entities.

An identification variable of a collection member declaration is declared using a special operator, the reserved identifier IN. The argument to the IN operator is a collection-valued path expression.

The path expression evaluates to a collection type specified as a result of navigation to a collection-valued association-field of an entity abstract schema type.

The syntax for declaring a collection member identification variable is as follows:

collection_member_declaration ::= IN (collection_valued_path_expression) [AS] identification_variable

For example, the query:

```
SELECT DISTINCT o
FROM Order o JOIN o.lineItems l JOIN l.product p
WHERE p.productType = 'office_supplies'
```

may equivalently be expressed as follows, using the IN operator:

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

In this example, lineItems is the name of an association-field whose value is a collection of instances of the abstract schema type LineItem. The identification variable l designates a member of this collection, a single LineItem abstract schema type instance. In this example, o is an identification variable of the abstract schema type Order.

●Use the WHERE clause to restrict query results using conditional expressions, including the use of literals, path expressions, named and positional parameters, logical operators, the following expressions (and their NOT options): BETWEEN, IN, LIKE, NULL, EMPTY, MEMBER [OF], EXISTS, ALL, ANY, SOME, and functional expressions.

- [EJB_3.0_PERSISTENCE] 4.5; 4.6.1; 4.6.3; 4.6.4; 4.6.4.1; 4.6.4.2; 4.6.6; 4.6.7; 4.6.8; 4.6.9; 4.6.10; 4.6.11; 4.6.12; 4.6.13; 4.6.14; 4.6.16; 4.6.16.1; 4.6.16.2; 4.6.16.3

The WHERE clause of a query consists of a conditional expression used to select objects or values that satisfy the expression. The WHERE clause restricts the result of a select statement or the scope of an update or delete operation.
A WHERE clause is defined as follows:

where_clause ::= WHERE conditional_expression

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

Examples:

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

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

Literals

A string literal is enclosed in single quotes—for example: 'literal'. A string literal that includes a single quote is represented by two single quotes—for example: 'literal''s'. String literals in queries, like Java String literals, use unicode character encoding. The use of Java escape notation is NOT supported in query string literals.

Exact numeric literals support the use of Java integer literal syntax as well as SQL exact numeric literal syntax.

Approximate literals support the use Java floating point literal syntax as well as SQL approximate numeric literal syntax.

Enum literals support the use of Java enum literal syntax. The enum class name must be specified.

Appropriate suffixes may be used to indicate the specific type of a numeric literal in accordance with the Java Language Specification. Support for the use of hexadecimal and octal numeric literals is not required by this specification.

The boolean literals are TRUE and FALSE.

Although predefined reserved literals appear in upper case, they are case INSENSITIVE.

Path Expressions

It is illegal to use a *collection_valued_path_expression* within a WHERE or HAVING clause as part of a conditional expression except in an *empty_collection_comparison_expression*, in a *collection_member_expression*, or as an argument to the SIZE operator.

Input Parameters

Either positional or named parameters may be used. Positional and named parameters MAY NOT be mixed in a single query.

Input parameters can ONLY be used in the WHERE clause or HAVING clause of a query.

Note that if an input parameter value is null, comparison operations or arithmetic operations involving the input parameter will return an unknown value.

1 **Positional Parameters**

The following rules apply to positional parameters:

- Input parameters are designated by the question mark (?) prefix followed by an integer. For example: ?1.
- Input parameters are numbered starting from 1.
- Note that the same parameter can be used MORE THAN ONCE in the query string and that the ordering of the use of parameters within the query string need not conform to the order of the positional parameters.
- The following query finds the orders for a product whose name is designated by an input parameter:

```
SELECT DISTINCT o
FROM Order o, IN(o.lineItems) l
WHERE l.product.name = ?1
```

For this query, the input parameter must be of the type of the state-field name, i.e., a string.

The following Query interface method used to define positional parameters:

```
public Query setParameter (int pos, Object value);
```

Specify positional parameters in your JPQL string using an integer prefixed by a question mark. You can then populate the Query object with positional parameter values via calls to the setParameter method above. The method returns the Query instance for optional method chaining.

The following code will substitute JDJ for the ?1 parameter and 5.0 for the ?2 parameter, then execute the query with those values:

```
EntityManager em = ...
Query q = em.createQuery("SELECT x FROM Magazine x WHERE x.title = ?1 and x.price > ?2");
q.setParameter(1, "JDJ").setParameter(2, 5.0);
List<Magazine> results = (List<Magazine>) q.getResultList();
```

2 Named Parameters

A named parameter is an identifier that is prefixed by the ":" symbol. It follows the rules for identifiers (must not be a reserved word, may include underscore (_) character and the dollar sign ($) character, may NOT include question mark (?). Named parameters are case SENSITIVE.

Example:

```
SELECT c
FROM Customer c
WHERE c.status = :stat
```

The following Query interface method used to define named parameters:

```
public Query setParameter(String name, Object value);
```

Named parameters are denoted by prefixing an arbitrary name with a colon in your JPQL string. You can then populate the Query object with parameter values using the method above. Like the positional parameter method, this method returns the Query instance for optional method chaining.

This code substitutes JDJ for the :titleParam parameter and 5.0 for the :priceParam parameter, then executes the query with those values:

```
EntityManager em = ...
Query q = em.createQuery("SELECT x FROM Magazine x WHERE x.title = :titleParam and x.price > :priceParam");
q.setParameter("titleParam", "JDJ").setParameter("priceParam", 5.0);
List<Magazine> results = (List<Magazine>) q.getResultList();
```

3 Logical operators

- NOT
- AND
- OR

4 Between Expressions

The BETWEEN expression:

x BETWEEN y AND z

is semantically equivalent to:

y <= x AND x <= z

The rules for unknown and NULL values in comparison operations apply.

Example 1:

p.age BETWEEN 15 and 19

is equivalent to:

p.age >= 15 AND p.age <= 19

Example 2:

p.age NOT BETWEEN 15 and 19

is equivalent to:

p.age < 15 OR p.age > 19

5 In Expressions

The syntax for the use of the comparison operator [NOT] IN in a conditional expression is as follows:

in_expression ::= state_field_path_expression [NOT]IN (in_item {, in_item}* | subquery)

in_item ::= literal | input_parameter

The *state_field_path_expression* must have a string, numeric, or enum value.

Example:

o.country IN ('UK', 'US', 'Belarus')

is true for Belarus and false for Russia, and is equivalent to the expression:

(o.country = 'UK') OR (o.country = 'US') OR (o.country = 'Belarus')

Example 2:

o.country NOT IN ('UK', 'US', 'Belarus')

is false for UK and true for Germany, and is equivalent to the expression:

NOT ((o.country = 'UK') OR (o.country = 'US') OR (o.country = 'Belarus'))

There must be at least one element in the comma separated list that defines the set of values for the IN expression.

If the value of a *state_field_path_expression* in an IN or NOT IN expression is NULL or unknown, the value of the expression is unknown.

6 Like Expressions

The syntax for the use of the comparison operator [NOT] LIKE in a conditional expression is as follows:

string_expression [NOT] LIKE pattern_value [ESCAPE escape_character]

The string_expression must have a string value. The pattern_value is a string literal or a string-valued input parameter in which an underscore (_) stands for ANY SINGLE character, a percent (%) character stands for any sequence of characters (including the empty sequence), and all other characters stand for themselves. The optional escape_character is a single-character string literal or a character-valued input parameter (i.e., char or Character) and is used to escape the special meaning of the underscore and percent characters in pattern_value.

Examples:

- address.phone LIKE '12%3' is true for '123' and '12993' and false for '1234'
- asentence.word LIKE 'l_se' is true for 'lose' and false for 'loose'
- aword.underscored LIKE '_%' ESCAPE '\' is true for '_foo' and false for 'bar'
- address.phone NOT LIKE '12%3' is false for '123' and '12993' and true for '1234'

7 Null Comparison Expressions

The syntax for the use of the comparison operator IS NULL in a conditional expression is as follows:

{single_valued_path_expression | input_parameter} IS [NOT] NULL

A null comparison expression tests whether or not the single-valued path expression or input parameter is a NULL value.

8 Empty Collection Comparison Expressions

The syntax for the use of the comparison operator IS EMPTY in an *empty_collection_comparison_expression* is as follows:

collection_valued_path_expression IS [NOT] EMPTY

This expression tests whether or not the collection designated by the collection-valued path expression is empty (i.e, has no elements).

Example:

```
SELECT o
FROM Order o
WHERE o.lineItems IS EMPTY
```

9 Collection Member Expressions

The syntax for the use of the comparison operator MEMBER OF (the use of the reserved word OF is optional in this expression) in an *collection_member_expression* is as follows:

```
entity_expression [NOT] MEMBER [OF] collection_valued_path_expression
entity_expression ::= single_valued_association_path_expression | simple_entity_expression
simple_entity_expression ::= identification_variable | input_parameter
```

This expression tests whether the designated value is a member of the collection specified by the collection-valued path expression.

If the collection valued path expression designates an empty collection, the value of the MEMBER OF expression is FALSE and the value of the NOT MEMBER OF expression is TRUE. Otherwise, if the value of the collection-valued path expression or single-valued association-field path expression in the collection member expression is NULL or unknown, the value of the collection member expression is unknown.

10 Exists Expressions

An EXISTS expression is a predicate that is true only if the result of the subquery consists of one or more values and that is false otherwise.

The syntax of an exists expression is:

exists_expression ::= [NOT] EXISTS (subquery)

Example:

```
SELECT DISTINCT emp
FROM Employee emp
WHERE EXISTS (
      SELECT spouseEmp
      FROM Employee spouseEmp
      WHERE spouseEmp = emp.spouse
)
```

The result of this query consists of all employees whose spouses are also employees.

11 All or Any Expressions

An ALL conditional expression is a predicate that is true if the comparison operation is true for all values in the result of the subquery or the result of the subquery is empty. An ALL conditional expression is false if the result of the comparison is false for AT LEAST ONE row, and is unknown if neither true nor false.

An ANY conditional expression is a predicate that is true if the comparison operation is true for some value in the result of the subquery. An ANY conditional expression is false if the result of the subquery is empty or if the comparison operation is false for EVERY VALUE in the result of the subquery, and is unknown if neither true nor false. The keyword SOME is **synonymous** with ANY.

The syntax of an ALL or ANY expression is specified as follows:

```
all_or_any_expression ::= {ALL |ANY |SOME} (subquery)
```

Example:

```
SELECT emp
FROM Employee emp
WHERE emp.salary > ALL (
      SELECT m.salary
      FROM Manager m
      WHERE m.department = emp.department
)
```

12 String Functions

- CONCAT(string1, string2)

  The CONCAT function returns a string that is a concatenation of its arguments.

  ```
  SELECT x FROM Magazine x WHERE CONCAT(x.title, 's') = 'JDJs'
  ```

- SUBSTRING(string, startIndex, length)

  The second and third arguments of the SUBSTRING function denote the starting position and length of the substring to be returned. These arguments are integers. The first position of a string is denoted by 1. The SUBSTRING function returns a string.

```
SELECT x FROM Magazine x WHERE SUBSTRING(x.title, 1, 1) = 'J'
```

- TRIM([LEADING | TRAILING | BOTH] [character FROM] string)

  The TRIM function trims the specified character from a string. If the character to be trimmed is not specified, it is assumed to be space (or blank). The optional trim_character is a single-character string literal or a character-valued input parameter (i.e., char or Character). If a trim specification (LEADING | TRAILING | BOTH) is not provided, BOTH is assumed. The TRIM function returns the trimmed string.

  ```
  SELECT x FROM Magazine x WHERE TRIM(BOTH 'J' FROM x.title) = 'D'
  ```

- LOWER(string), UPPER(string)

  The LOWER and UPPER functions convert a string to lower and upper case, respectively. They return a string.

  ```
  SELECT x FROM Magazine x WHERE LOWER(x.title) = 'jdj'
  ```

  ```
  SELECT x FROM Magazine x WHERE UPPER(x.title) = 'JAVAPRO'
  ```

- LOCATE(searchString, candidateString [, startIndex])

  The LOCATE function returns the position of a given string within a string, starting the search at a specified position. It returns the first position at which the string was found as an integer. The first argument is the string to be located; the second argument is the string to be searched; the optional third argument is an integer that represents the string position at which the search is started (by default, the beginning of the string to be searched). The first position in a string is denoted by 1. If the string is not found, 0 is returned.

  ```
  SELECT x FROM Magazine x WHERE LOCATE('D', x.title) = 2
  ```

- LENGTH(string)

  The LENGTH function returns the length of the string in characters as an integer.

  ```
  SELECT x FROM Magazine x WHERE LENGTH(x.title) = 3
  ```

13 Arithmetic Functions

- ABS(number)

  Returns the absolute value of the argument.

  The ABS function takes a numeric argument and returns a number (integer, float, or double) of the same type as the argument to the function.

  ```
  SELECT x FROM Magazine x WHERE ABS(x.price) >= 5.00
  ```

- SQRT(number)

  Returns the square root of the argument.

  The SQRT function takes a numeric argument and returns a double.

  SELECT x FROM Magazine x WHERE SQRT(x.price) >= 1.00

- MOD(number, divisor)

  Returns the modulo of number and divisor.

  The MOD function takes two integer arguments and returns an integer.

  SELECT x FROM Magazine x WHERE MOD(x.price, 10) = 0

- SIZE

  The SIZE function returns an integer value, the number of elements of the collection. If the collection is empty, the SIZE function evaluates to zero.

14 Datetime Functions

- CURRENT_DATE

  The CURRENT_DATE function returns the value of current date on the database server.

- CURRENT_TIME

  The CURRENT_TIME function returns the value of current time on the database server.

- CURRENT_TIMESTAMP

  The CURRENT_TIMESTAMP function returns the value of current timestamp on the database server.

- **Develop Java Persistence Query Language statements that update a set of entities using UPDATE/SET and DELETE FROM.**
- **Declare and use named queries, dynamic queries, and SQL (native) queries.**

  - [EJB_3.0_PERSISTENCE] 3.6.4; 3.6.1.1;

Named Queries

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

The following is an example of the definition of a named query:

```
@NamedQuery(
    name="findAllCustomersWithName",
    query="SELECT c FROM Customer c WHERE c.name LIKE :custName"
)
```

The following is an example of the use of a named query:

```
@PersistenceContext
public EntityManager em;
	...
	customers = em.createNamedQuery("findAllCustomersWithName")
		.setParameter("custName", "Zaikin")
		.getResultList();
```

Query templates can be statically declared using the @NamedQuery and @NamedQueries annotations. For example:

```
@Entity
@NamedQueries({
	@NamedQuery(
		name="magsOverPrice",
		query="SELECT x FROM Magazine x WHERE x.price > ?1"
	),
	@NamedQuery(
		name="magsByTitle",
	query="SELECT x FROM Magazine x WHERE x.title = :titleParam"
	)
})
public class Magazine {
	...
}
```

These declarations will define two named queries called magsOverPrice and magsByTitle.

You retrieve named queries with the EntityManager.createNamedQuery(String name) method. For example:

```
EntityManager em = ...
Query q = em.createNamedQuery("magsOverPrice");
q.setParameter(1, 5.0f);
List<Magazine> results = (List<Magazine>) q.getResultList();
```

```
EntityManager em = ...
Query q = em.createNamedQuery("magsByTitle");
q.setParameter("titleParam", "JDJ");
List<Magazine> results = (List<Magazine>) q.getResultList();
```

Dynamic Queries

Example:

```
public List findWithName(String name) {
	return em.createQuery("SELECT c FROM Customer c WHERE c.name LIKE :custName")
		.setParameter("custName", name)
		.setMaxResults(10)
```

```
        .getResultList();
}
```

SQL Queries

Queries may be expressed in native SQL. The result of a native SQL query may consist of entities, scalar values, or a combination of the two. The entities returned by a query may be of different entity types.

When multiple entities are returned by a SQL query, the entities must be specified and mapped to the column results of the SQL statement in a @SqlResultSetMapping metadata definition. This result set mapping metadata can then be used by the persistence provider runtime to map the JDBC results into the expected objects.

If the results of the query are limited to entities of a SINGLE entity class, a simpler form may be used and @SqlResultSetMapping metadata is not required.

This is illustrated in the following example in which a native SQL query is created dynamically using the createNativeQuery method and the entity class that specifies the type of the result is passed in as an argument.

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

When executed, this query will return a Collection of all Order entities for items named "widget". The same results could also be obtained using @SqlResultSetMapping:

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

In this case, the metadata for the query result type might be specified as follows:

```
@SqlResultSetMapping(
      name="WidgetOrderResults",
      entities=@EntityResult(entityClass=com.acme.Order.class)
)
```

The following query and @SqlResultSetMapping metadata illustrates the return of MULTIPLE entity types and assumes default metadata and column name defaults:

```
Query q = em.createNativeQuery("SELECT o.id, o.quantity, o.item, " +
      "i.id, i.name, i.description " +
      "FROM Order o, Item i " +
      "WHERE (o.quantity > 25) AND (o.item = i.id)",
      "OrderItemResults"
```

```
);


@SqlResultSetMapping(
	name="OrderItemResults",
	entities={
		@EntityResult(entityClass=com.acme.Order.class),
		@EntityResult(entityClass=com.acme.Item.class)
	}
)
```

When an entity is being returned, the SQL statement should select ALL of the columns that are mapped to the entity object. This should include foreign key columns to related entities. The results obtained when insufficient data is available are undefined. A SQL result set mapping must NOT be used to map results to the non-persistent state of an entity.

The column names that are used in the SQL result set mapping annotations refer to the names of the columns in the SQL SELECT clause. Note that column aliases must be used in the SQL SELECT clause where the SQL result would otherwise contain multiple columns of the same name.

An example of combining multiple entity types and that includes aliases in the SQL statement requires that the column names be explicitly mapped to the entity fields. The @FieldResult annotation is used for this purpose:

```
Query q = em.createNativeQuery("SELECT o.id AS order_id, " +
	"o.quantity AS order_quantity, " +
	"o.item AS order_item, " +
	"i.id, i.name, i.description " +
	"FROM Order o, Item i " +
	"WHERE (order_quantity > 25) AND (order_item = i.id)",
	"OrderItemResults"
);


@SqlResultSetMapping(
	name="OrderItemResults",
	entities={
		@EntityResult(
			entityClass=com.acme.Order.class,
			fields={
				@FieldResult(name="id", column="order_id"),
				@FieldResult(name="quantity", column="order_quantity"),
				@FieldResult(name="item", column="order_item")
			}
		),
		@EntityResult(
			entityClass=com.acme.Item.class
		)
	}
)
```

Scalar result types CAN be included in the query result by specifying the @ColumnResult annotation in the metadata.

```
Query q = em.createNativeQuery("SELECT o.id AS order_id, " +
      "o.quantity AS order_quantity, " +
      "o.item AS order_item, " +
      "i.name AS item_name, " +
      "FROM Order o, Item i " +
      "WHERE (order_quantity > 25) AND (order_item = i.id)",
      "OrderResults"
);


@SqlResultSetMapping(
      name="OrderResults",
      entities={
            @EntityResult(
                  entityClass=com.acme.Order.class,
                  fields={
                        @FieldResult(name="id", column="order_id"),
                        @FieldResult(name="quantity", column="order_quantity"),
                        @FieldResult(name="item", column="order_item")
                  }
            )
      },
      columns={
            @ColumnResult(name="item_name")
      }
)
```

The use of named parameters is not defined for native queries. Only positional parameter binding for SQL queries may be used by portable applications.

Support for joins is currently limited to single-valued relationships.

➢ Obtain javax.persistence.Query objects and use the javax.persistence.Query API.

- [EJB_3.0_PERSISTENCE] 3.1.1; 3.6; 3.6.1; 3.6.3

EntityManager interface provides the following methods for creating Query object:

- **Query createQuery(String qlString)**

  Creates an instance of Query for executing a Java Persistence Query Language statement.

  If the argument to the createQuery method is not a valid Java Persistence Query string, the IllegalArgumentException may be thrown or the query execution will fail.

- **Query createNamedQuery(String name)**

Creates an instance of Query for executing a named query (in the Java Persistence Query Language or in native SQL).

- **Query createNativeQuery(String sqlString)**

  Creates an instance of Query for executing a native SQL statement, e.g., for UPDATE or DELETE.

- **Query createNativeQuery(String sqlString, Class resultClass)**

  Creates an instance of Query for executing a native SQL query. resultClass - the class of the resulting instance(s).

  If a native query is not a valid query for the database in use or if the result set specification is incompatible with the result of the query, the query execution will fail and a PersistenceException will be thrown when the query is executed.

- **Query createNativeQuery(String sqlString, String resultSetMapping)**

  Creates an instance of Query for executing a native SQL query. resultSetMapping - the name of the result set mapping.

The Query API is used for both static queries (i.e., named queries) and dynamic queries. The Query API also supports named parameter binding and pagination control.

Query interface defines the following methods:

- **List getResultList()**

  Executes a SELECT query and returns the query results as a List. Throws IllegalStateException if called for a Java Persistence Query Language UPDATE or DELETE statement

- **Object getSingleResult()**

  Executes a SELECT query that returns a single result. Throws NoResultException if there is no result, NonUniqueResultException if more than one result, IllegalStateException if called for a Java Persistence Query Language UPDATE or DELETE statement.

- **int executeUpdate()**

  Executes an update or delete statement. Returns the number of entities updated or deleted Throws IllegalStateException if called for a Java Persistence Query Language SELECT statement, TransactionRequiredException if there is NO transaction.

- **Query setMaxResults(int maxResult)**

  Sets the maximum number of results to retrieve. Returns the same query instance.

- **Query setFirstResult(int startPosition)**

  Set the position of the first result to retrieve. The parameter is the start position of the first result, numbered from 0. Returns the same query instance.

- **Query setParameter(String name, Object value)**

Binds an argument to a named parameter. name - the parameter name, value - the parameter value. Returns the same query instance. Throws IllegalArgumentException if parameter name does not correspond to parameter in query string or argument is of incorrect type.

A named parameter is an identifier that is prefixed by the ":" symbol in JPQL. Named parameters are case-sensitive.

The parameter names passed to the setParameter methods DO NOT include the ":" prefix.

- **Query setParameter(String name, Date value, TemporalType temporalType)**

  Binds an instance of java.util.Date to a named parameter. Returns the same query instance.

- **Query setParameter(String name, Calendar value, TemporalType temporalType)**

  Binds an instance of java.util.Calendar to a named parameter. Returns the same query instance.

- **Query setParameter(int position, Object value)**

  Binds an argument to a positional parameter. Returns the same query instance.

- **Query setParameter(int position, Date value, TemporalType temporalType)**

  Binds an instance of java.util.Date to a positional parameter. Returns the same query instance.

- **Query setParameter(int position, Calendar value, TemporalType temporalType)**

  Binds an instance of java.util.Calendar to a positional parameter. Returns the same query instance.

- **Query setFlushMode(FlushModeType flushMode)**

  Sets the flush mode type to be used for the query execution. The flush mode type applies to the query regardless of the flush mode type in use for the entity manager.

The elements of the result of a Java Persistence query whose SELECT clause consists of more than one select expression are of type Object[]. If the SELECT clause consists of only one select expression, the elements of the query result are of type Object. When native SQL queries are used, the SQL result set mapping, determines how many items (entities, scalar values, etc.) are returned. If multiple items are returned, the elements of the query result are of type Object[]. If only a single item is returned as a result of the SQL result set mapping or if a result class is specified, the elements of the query result are of type Object.

An IllegalArgumentException is thrown if a parameter name is specified that does not correspond to a named parameter in the query string, if a positional value is specified that does not correspond to a positional parameter in the query string, or if the type of the parameter is not valid for the query. This exception may be thrown when the parameter is bound, or the execution of the query may fail.

The effect of applying setMaxResults or setFirstResult to a query involving fetch joins over collections is undefined.

Query methods other than the executeUpdate method are NOT REQUIRED to be invoked within a transaction context. In particular, the getResultList and getSingleResult methods are not required to be invoked within a transaction context. If an entity manager with transaction-scoped persistence context is in use, the resulting entities will be DETACHED; if an entity manager with an extended persistence context is used, they will be MANAGED.

Runtime exceptions other than the NoResultException and NonUniqueResultException thrown by the methods of the Query interface cause the current transaction to be ROLLED BACK.

➢**Chapter 9. Transactions**

➢Identify correct and incorrect statements or examples about bean-managed transaction demarcation.

- [EJB_3.0_CORE] 13.1.1; 13.2.4; 13.3.3; 13.3.3.1; 13.6.1

Support for transactions is an essential element of the Enterprise JavaBeans architecture. The Enterprise Bean Provider and the client application programmer are not exposed to the complexity of distributed transactions. The Bean Provider can choose between using PROGRAMMATIC transaction demarcation in the enterprise bean code (this style is called *bean-managed transaction demarcation*) or DECLARATIVE transaction demarcation performed automatically by the EJB container (this style is called *container-managed transaction demarcation*).
With **bean-managed transaction** (BMT) demarcation, the enterprise BEAN CODE demarcates transactions using the javax.transaction.UserTransaction interface. All resource manager accesses between the UserTransaction.begin and UserTransaction.commit calls are part of a transaction.

The terms resource and resource manager used in this section refer to the resources declared using the Resource annotation in the enterprise bean class or using the resource-ref element in the enterprise bean's deployment descriptor. This includes not only database resources, but also other resources, such as JMS Connections. These resources are considered to be "managed" by the container.

Regardless of whether an enterprise bean uses bean-managed or container-managed transaction demarcation, the burden of implementing transaction management is on the EJB container and server provider. The EJB container and server implement the necessary low-level transaction protocols, such as the two-phase commit protocol between a transaction manager and a database system or messaging provider, transaction context propagation, and distributed two-phase commit.

Client-Managed Demarcation

A Java client can use the javax.transaction.UserTransaction interface to explicitly demarcate transaction boundaries. The client program obtains the javax.transaction.UserTransaction interface through dependency injection or lookup in the bean's EJBContext or in the JNDI name space.

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

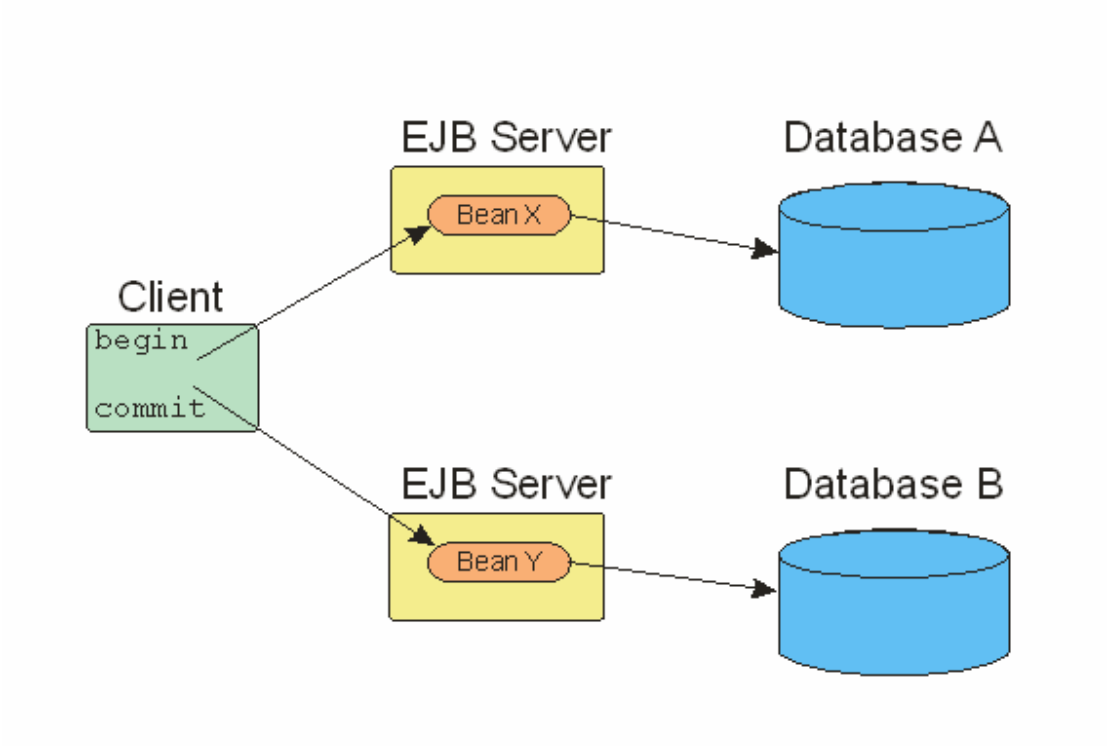
The application programmer demarcates the transaction with beginand commitcalls. If the enterprise beans X and Y are configured to use a client transaction (i.e., their methods have transaction attributes that either require or support an existing transaction context), the EJB server ensures that the updates to databases A and B are made as part of the client's

transaction.

Enterprise Beans Using Bean-Managed Transaction Demarcation

The enterprise bean with bean-managed transaction (BMT) demarcation MUST be a session bean OR a message-driven bean.

An instance that starts a transaction must complete the transaction before it starts a new transaction.



The Bean Provider uses the UserTransaction interface to demarcate transactions.

```
package javax.transaction;

public interface UserTransaction {

    void begin() throws NotSupportedException, SystemException;

    void commit() throws RollbackException, HeuristicMixedException,
        HeuristicRollbackException, SecurityException,
        IllegalStateException, SystemException;

    void rollback() throws IllegalStateException, SecurityException, SystemException;

    void setRollbackOnly() throws IllegalStateException, SystemException;

    int getStatus() throws SystemException;

    void setTransactionTimeout(int seconds) throws SystemException;
}
```

All updates to the resource managers between the UserTransaction.begin() and UserTransaction.commit() methods are performed in a transaction. While an instance is in a transaction, the instance MUST NOT attempt to use the resource-manager specific transaction demarcation API (e.g. it must not invoke the commit or rollback method on the java.sql.Connection interface or on the javax.jms.Session interface). However, use of the Java Persistence API EntityTransaction interface is supported:

```
@PersistenceContext
EntityManager em;

@Resource
UserTransaction tx;

....
tx.begin();
em.persist(new PersistentObject());
tx.commit();
```

or

```
InitialContext ctx = new InitialContext();
UserTransaction tx = (UserTransaction)ctx.lookup("java:comp/UserTransaction"); // Predefined reserved name
EntityManager em = (EntityManager) ctx.lookup("java:comp/env/persistence/EntityManager");

...
tx.begin();
PersistentObject obj = em.merge(pObj); // merge object to the new context
em.remove(obj);
tx.commit();
```

A stateful session bean instance may, but is NOT REQUIRED to, commit a started transaction before a business method returns. If a transaction has not been completed by the end of a business method, the container retains the association between the transaction and the instance across multiple client calls until the instance eventually completes the transaction.

A stateless session bean instance MUST commit a transaction before a business method or timeout callback method returns.

A message-driven bean instance MUST commit a transaction before a message listener method or timeout callback method returns.

The following example illustrates a business method that performs a transaction involving two database connections:

```
@Stateless
@TransactionManagement(BEAN)
public class MySessionBean implements MySession {
    @Resource javax.transaction.UserTransaction ut;
    @Resource javax.sql.DataSource database1;
    @Resource javax.sql.DataSource database2;

    public void someMethod(...) {
        java.sql.Connection con1;
        java.sql.Connection con2;
        java.sql.Statement stmt1;
```

```
        java.sql.Statement stmt2;

        // obtain con1 object and set it up for transactions
        con1 = database1.getConnection();
        stmt1 = con1.createStatement();

        // obtain con2 object and set it up for transactions
        con2 = database2.getConnection();
        stmt2 = con2.createStatement();

        //
        // Now do a transaction that involves con1 and con2.
        //

        // start the transaction
        ut.begin();

        // Do some updates to both con1 and con2. The container
        // automatically enlists con1 and con2 with the transaction.
        stmt1.executeQuery(...);
        stmt1.executeUpdate(...);
        stmt2.executeQuery(...);
        stmt2.executeUpdate(...);
        stmt1.executeUpdate(...);
        stmt2.executeUpdate(...);

        // commit the transaction
        ut.commit();

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

The following example illustrates a business method that performs a transaction involving both a database connection and a JMS connection:

```
@Stateless
@TransactionManagement(BEAN)
public class MySessionBean implements MySession {
    @Resource javax.Transaction.UserTransaction ut;
    @Resource javax.sql.DataSource database1;
    @Resource javax.jms.QueueConnectionFactory qcf1;
    @Resource javax.jms.Queue queue1;

    public void someMethod(...) {
        java.sql.Connection dcon;
        java.sql.Statement stmt;
        javax.jms.QueueConnection qcon;
        javax.jms.QueueSession qsession;
        javax.jms.QueueSender qsender;
        javax.jms.Message message;
```

```
            // obtain db conn object and set it up for transactions
            dcon = database1.getConnection();
            stmt = dcon.createStatement();

            // obtain jms conn object and set up session for transactions
            qcon = qcf1.createQueueConnection();
            qsession = qcon.createQueueSession(true,0);
            qsender = qsession.createSender(queue1);
            message = qsession.createTextMessage();
            message.setText("some message");

            //
            // Now do a transaction that involves the two connections.
            //

            // start the transaction
            ut.begin();

            // Do database updates and send message. The container
            // automatically enlists dcon and qsession with the
            // transaction.
            stmt.executeQuery(...);
            stmt.executeUpdate(...);
            stmt.executeUpdate(...);
            qsender.send(message);

            // commit the transaction
            ut.commit();

            // release connections
            stmt.close();
            qsender.close();
            qsession.close();
            dcon.close();
            qcon.close();
        }
        ...
}
```

The following example illustrates a STATEFUL session bean that retains a transaction across three client calls, invoked in the following order: method1, method2, and method3. Note that the Bean Provider must use the pre-passivation callback method here to close the connections and set the instance variables for the connection to null:

```
@Stateful
@TransactionManagement(BEAN)
public class MySessionBean implements MySession {
    @Resource javax.Transaction.UserTransaction ut;
    @Resource javax.sql.DataSource database1;
    @Resource javax.sql.DataSource database2;
    java.sql.Connection con1;
    java.sql.Connection con2;
```

```
    public void method1(...) {
        java.sql.Statement stmt;

        // start a transaction
        ut.begin();

        // make some updates on con1
        con1 = database1.getConnection();
        stmt = con1.createStatement();
        stmt.executeUpdate(...);
        stmt.executeUpdate(...);

        //
        // The container retains the transaction associated with the
        // instance to the next client call (which is method2(...)).
    }

    public void method2(...) {
        java.sql.Statement stmt;
        con2 = database2.getConnection();
        stmt = con2.createStatement();
        stmt.executeUpdate(...);
        stmt.executeUpdate(...);

        // The container retains the transaction associated with the
        // instance to the next client call (which is method3(...)).
    }

    public void method3(...) {
        java.sql.Statement stmt;

        // make some more updates on con1 and con2
        stmt = con1.createStatement();
        stmt.executeUpdate(...);
        stmt = con2.createStatement();
        stmt.executeUpdate(...);

        // commit the transaction
        ut.commit();

        // release connections
        stmt.close();
        con1.close();
        con2.close();
    }
    ...
}
```

It is possible for an enterprise bean to open and close a database connection in each business method (rather than hold the connection open until the end of transaction). In the following example, if the client executes the sequence of methods (method1, method2, method2, method2, and method3), all the database updates done by the multiple invocations of method2 are performed in the scope of the same transaction, which is the transaction started in method1

and committed in method3:

```
@Stateful
@TransactionManagement(BEAN)
public class MySessionBean implements MySession {

	@Resource javax.Transaction.UserTransaction ut;
	@Resource javax.sql.DataSource database1;

	public void method1(...) {
		// start a transaction
		ut.begin();
	}

	public void method2(...) {
		java.sql.Connection con;
		java.sql.Statement stmt;

		// open connection
		con = database1.getConnection();

		// make some updates on con
		stmt = con.createStatement();
		stmt.executeUpdate(...);
		stmt.executeUpdate(...);

		// close the connection
		stmt.close();
		con.close();
	}

	public void method3(...) {
		// commit the transaction
		ut.commit();
	}
	...
}
```

getRollbackOnly and setRollbackOnly Methods for BMT Beans

An enterprise bean with bean-managed transaction (BMT) demarcation MUST NOT use the getRollbackOnly and setRollbackOnly methods of the EJBContext interface.

An enterprise bean with bean-managed transaction demarcation has no need to use these methods, because of the following reasons:

- An enterprise bean with bean-managed transaction (BMT) demarcation can obtain the status of a transaction by using the getStatus method of the javax.transaction.UserTransaction interface.
- An enterprise bean with bean-managed transaction (BMT) demarcation can rollback a transaction using the rollback method of the javax.transaction.UserTransaction interface.

The container MUST make the javax.transaction.UserTransaction interface available to the enterprise bean's business method, message listener method, interceptor method, or timeout

callback method via dependency injection into the enterprise bean class or interceptor class, and through lookup via the javax.ejb.EJBContext interface, and in the JNDI naming context under java:comp/UserTransaction. When an instance uses the javax.transaction.UserTransaction interface to demarcate a transaction, the container must enlist all the resource managers used by the instance between the begin and commit or rollback methods with the transaction. When the instance attempts to commit the transaction, the container is responsible for the global coordination of the transaction commit.

Via JNDI:

```
Context ctx = new InitialContext( );
UserTransaction ut = (UserTransaction) ctx.lookup("java:comp/UserTransaction");
ut.begin();
...
```

Via EJBContext:

```
@Resource SessionContext ejbContext;

public void myMethod() {
	try {
		UserTransaction ut = ejbContext.getUserTransaction();
		ut.begin();
		...
```

Via dependency injection:

```
@Resource UserTransaction ut;

public void myMethod() {
	try {
		ut.begin();
		...
```

In the case of a **stateful** session bean, it is possible that the business method that started a transaction completes without committing or rolling back the transaction. In such a case, the container must retain the association between the transaction and the instance across multiple client calls until the instance commits or rolls back the transaction. When the client invokes the next business method, the container must invoke the business method (and any applicable interceptor methods for the bean) in this transaction context.

If a **stateless** session bean instance starts a transaction in a business method or interceptor method, it MUST commit the transaction BEFORE the business method (or all its interceptor methods) returns. The container must detect the case in which a transaction was started, but not completed, in the business method or interceptor method for the business method, and handle it as follows:

- Log this as an application error to alert the System Administrator.
- Roll back the started transaction.

- Discard the instance of the session bean.
- Throw the javax.ejb.EJBException. If the EJB 2.1 client view is used, the container should throw java.rmi.RemoteException if the client is a remote client, or throw the javax.ejb.EJBException if the client is a local client.

- **Identify correct and incorrect statements or examples about container-managed transaction demarcation, and given a list of transaction behaviors, match them with the appropriate transaction attribute.**

    - [EJB_3.0_CORE] 13.1.1; 13.2.5; 13.3.4

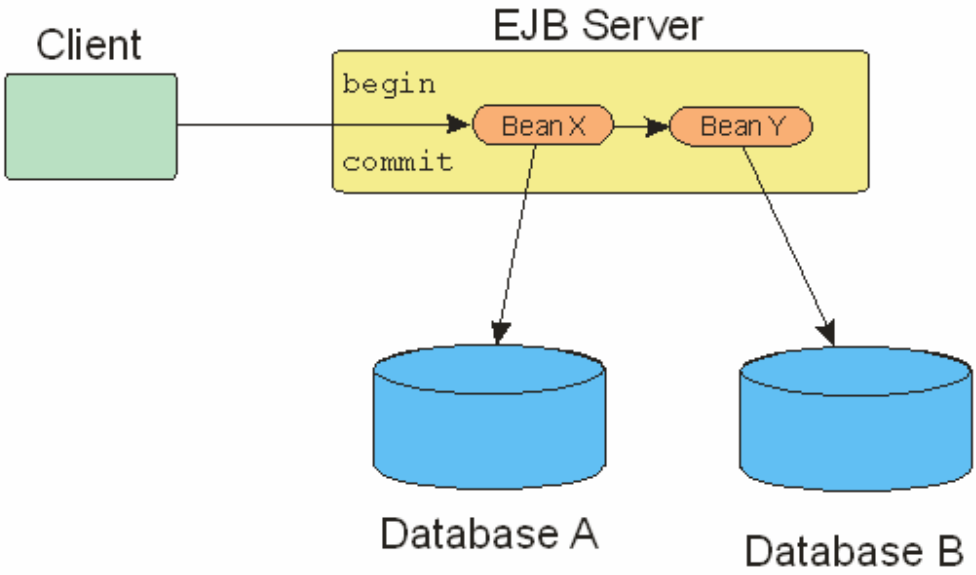
With **container-managed transaction** (CMT) demarcation, the CONTAINER demarcates transactions per instructions provided by the developer in metadata annotations or in the deployment descriptor. These instructions, called transaction attributes, tell the container whether it should include the work performed by an enterprise bean method in a client's transaction, run the enterprise bean method in a new transaction started by the container, or run the method with "no transaction".

Container-Managed Demarcation

Whenever a client invokes a method on an enterprise bean's business interface (or on the home or component interface of an enterprise bean), the container interposes on the method invocation. The interposition allows the container to control transaction demarcation declaratively through the transaction attribute set by the developer.

For example, if an enterprise bean method is configured with the REQUIRED transaction attribute, the container behaves as follows: If the client request is not associated with a transaction context, the container automatically initiates a transaction whenever a client invokes an enterprise bean method that requires a transaction context. If the client request contains a transaction context, the container includes the enterprise bean method in the client transaction.

The following figure illustrates such a scenario. A non-transactional client invokes the enterprise bean X, and the invoked method has the REQUIRED transaction attribute. Because the message from the client does not include a transaction context, the container starts a new transaction before dispatching the method on X. Bean XтАЩs 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.

If a message-driven bean's message listener method is configured with the REQUIRED transaction attribute, the container automatically starts a new transaction before the delivery of the message and, hence, before the invocation of the method.

JMS requires that the transaction be started before the dequeuing of the message.

The container automatically enlists the resource manager associated with the arriving message and all the resource managers accessed by the message listener method with the transaction.

Enterprise Beans Using Container-Managed Transaction Demarcation

The enterprise bean's business methods, message listener methods, business method interceptor methods, lifecycle callback interceptor methods, or timeout callback method MUST NOT use any resource-manager specific transaction management methods that would interfere with the container's demarcation of transaction boundaries. For example, the enterprise bean methods MUST NOT use the following methods of the java.sql.Connection interface: commit, setAutoCommit, and rollback; or the following methods of the javax.jms.Session interface: commit and rollback.

The enterprise bean's business methods, message listener methods, business method interceptor methods, lifecycle callback interceptor methods, or timeout callback method MUST NOT attempt to obtain or use the javax.transaction.UserTransaction interface.

The following is an example of a business method in an enterprise bean with container-managed transaction (CMT) demarcation. The business method updates two databases using JDBC connections. The container provides transaction demarcation as specified by the transaction

attribute:

```
@Stateless public class MySessionBean implements MySession {
    ...
    @TransactionAttribute(REQUIRED)
    public void someMethod(...) {
        java.sql.Connection con1;
        java.sql.Connection con2;
        java.sql.Statement stmt1;
        java.sql.Statement stmt2;

        // obtain con1 and con2 connection objects
        con1 = ...;
        con2 = ...;
        stmt1 = con1.createStatement();
        stmt2 = con2.createStatement();

        //
        // Perform some updates on con1 and con2. The container
        // automatically enlists con1 and con2 with the container-
        // managed transaction.
        //
        stmt1.executeQuery(...);
        stmt1.executeUpdate(...);
        stmt2.executeQuery(...);
        stmt2.executeUpdate(...);
        stmt1.executeUpdate(...);
        stmt2.executeUpdate(...);

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

➢Identify correct and incorrect statements or examples about transaction propagation semantics.

- [EJB_3.0_CORE] 13.6.2.1; 13.6.2.2; 13.6.2.3; 13.6.2.4; 13.6.2.5; 13.6.2.6; 13.6.2.7

NOT_SUPPORTED

The container invokes an enterprise bean method whose transaction attribute is set to the NOT_SUPPORTED value with an unspecified transaction context.

If a client calls with a transaction context, the container SUSPENDS the association of the transaction context with the current thread before invoking the enterprise bean's business method. The container RESUMES the suspended association when the business method has completed. The suspended transaction context of the client is not passed to the resource managers or other enterprise bean objects that are invoked from the business method.

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

REQUIRED

The container must invoke an enterprise bean method whose transaction attribute is set to the REQUIRED value with a valid transaction context.

If a client invokes the enterprise bean's method while the client is associated with a transaction context, the container invokes the enterprise bean's method in the client's transaction context.

If the client invokes the enterprise bean's method while the client is NOT ASSOCIATED with a transaction context, the container AUTOMATICALLY STARTS a new transaction before delegating a method call to the enterprise bean business method. The container automatically enlists all the resource managers accessed by the business method with the transaction. If the business method invokes other enterprise beans, the container passes the transaction context with the invocation. The container attempts to commit the transaction when the business method has completed. The container performs the commit protocol before the method result is sent to the client.

SUPPORTS

The container invokes an enterprise bean method whose transaction attribute is set to SUPPORTS as follows:

- If the client calls WITH a transaction context, the container performs the same steps as described in the REQUIRED case.
- If the client calls WITHOUT a transaction context, the container performs the same steps as described in the NOT_SUPPORTED case.

The SUPPORTS transaction attribute must be used with caution. This is because of the different transactional semantics provided by the two possible modes of execution. Only the enterprise beans that will execute correctly in both modes should use the SUPPORTS transaction attribute.

REQUIRES_NEW

The container MUST invoke an enterprise bean method whose transaction attribute is set to REQUIRES_NEW with a NEW TRANSACTION CONTEXT.

If the client invokes the enterprise bean's method while the client is not associated with a transaction context, the container automatically STARTS a new transaction before delegating a method call to the enterprise bean business method. The container automatically enlists all the resource managers accessed by the business method with the transaction. If the business method invokes other enterprise beans, the container passes the transaction context with the invocation. The container attempts to commit the transaction when the business method has

completed. The container performs the commit protocol before the method result is sent to the client.

If a client calls with a transaction context, the container SUSPENDS the association of the transaction context with the current thread before starting the new transaction and invoking the business method. The container RESUMES the suspended transaction association after the business method and the new transaction have been completed.

MANDATORY

The container must invoke an enterprise bean method whose transaction attribute is set to MANDATORY in a client's transaction context. The client is REQUIRED to call with a transaction context.

- If the client calls with a transaction context, the container performs the same steps as described in the REQUIRED case.
- If the client calls without a transaction context, the container throws the javax.ejb.EJBTransactionRequiredException. If the EJB 2.1 client view is used, the container throws the javax.transaction.TransactionRequiredException exception if the client is a remote client, or the javax.ejb.TransactionRequiredLocalException if the client is a local client.

NEVER

The container invokes an enterprise bean method whose transaction attribute is set to NEVER WITHOUT a transaction context defined by the EJB specification. The client is required to call WITHOUT a transaction context.

- If the client calls WITH a transaction context, the container throws the javax.ejb.EJBException. If the EJB 2.1 client view is used, the container throws the java.rmi.RemoteException exception if the client is a remote client, or the javax.ejb.EJBException if the client is a local client.
- If the client calls WITHOUT a transaction context, the container performs the same steps as described in the NOT_SUPPORTED case.

Transaction Attribute Summary

The following table provides a summary of the transaction context that the container passes to the business method and resource managers used by the business method, as a function of the transaction attribute and the client's transaction context. T1 is a transaction passed with the client request, while T2 is a transaction initiated by the container.

Table 9.1. Transaction Attribute Summary

Transaction attribute	Client's transaction	Transaction associated with business method	Transaction associated with resource managers
NOT_SUPPORTED	none	none	none
	T1	none	none
REQUIRED	none	T2	T2
	T1	T1	T1
SUPPORTS	none	none	none
	T1	T1	T1
REQUIRES_NEW	none	T2	T2
	T1	T2	T2
MANDATORY	none	ERROR	N/A
	T1	T1	T1
NEVER	none	none	none
	T1	ERROR	N/A

➢**Identify correct and incorrect statements or examples about specifying transaction information via annotations and/or deployment descriptors.**

- [EJB_3.0_CORE] 13.3.6; 13.3.7; 13.3.7.1; 13.3.7.2; 13.3.7.2.1

[EJB_3.0_SIMPLIFIED] 10.4; 10.5

Specification of a Bean's Transaction Management Type

By DEFAULT, a session bean or message-driven bean has **container managed transaction** (CMT) demarcation if the transaction management type IS NOT SPECIFIED. The Bean Provider of a session bean or a message-driven bean can use the TransactionManagement annotation to declare whether the session bean or message-driven bean uses bean-managed or container-managed transaction demarcation:

```
@Target(TYPE) @Retention(RUNTIME)
public @interface TransactionManagement {
    TransactionManagementType value()
        default TransactionManagementType.CONTAINER;
}
```

The value of the TransactionManagement annotation is either CONTAINER or BEAN.:

```
public enum TransactionManagementType {
	CONTAINER,
	BEAN
}
```

The TransactionManagement annotation is applied to the ENTERPRISE BEAN CLASS.

Example (BMT via annotation):

```
@Stateful
@TransactionManagement(TransactionManagementType.BEAN)
public class CalculatorBean implements Calculator {
	...
}
```

Alternatively, the Bean Provider can use the transaction-type deployment descriptor element to specify the bean's transaction management type. If the deployment descriptor is used, it is only necessary to explicitly specify the bean's transaction management type if bean-managed transaction is used.

Example (CMT via Deployment Descriptor):

```
<enterprise-beans>
	<session>
		<ejb-name>CalculatorBean</ejb-name>
		<business-remote>by.iba.ejb.Calculator</business-remote>
		<ejb-class>by.iba.ejb.CalculatorBean</ejb-class>
		<session-type>Stateless</session-type>
		<transaction-type>Container</transaction-type>
	</session>
</enterprise-beans>
```

The transaction management type of a bean is determined by the Bean Provider. The application assembler is NOT permitted to use the deployment descriptor to override a bean's transaction management type regardless of whether it has been explicitly specified or defaulted by the Bean Provider.

Specification of the Transaction Attributes for a Bean's Methods

The Bean Provider of an enterprise bean with container-managed transaction (CMT) demarcation MAY specify the transaction attributes for the enterprise bean's methods. By DEFAULT, the value of the **transaction attribute for a method of a bean with container-managed transaction** demarcation is the REQUIRED transaction attribute, and the transaction attribute does not need to be explicitly specified in this case.

A transaction attribute is a value associated with each of the following methods:

- a method of a bean's business interface
- a message listener method of a message-driven bean
- a timeout callback method
- a stateless session bean's web service endpoint method
- for beans written to the EJB 2.1 and earlier client view, a method of a session or entity bean's home or component interface

The transaction attribute specifies how the container must manage transactions for a method when a client invokes the method.

Transaction attributes are specified for the following methods:

- For a session bean written to the EJB 3.0 client view API, the transaction attributes are specified for those methods of the session bean class that correspond to the bean's business interface, the direct and indirect superinterfaces of the business interface, and for the timeout callback method, if any.
- For a stateless session bean that provides a web service client view, the transaction attributes are specified for the bean's web service endpoint methods, and for the timeout callback method, if any.
- For a message-driven bean, the transaction attributes are specified for those methods on the message-driven bean class that correspond to the bean's message listener interface and for the timeout callback method, if any.
- For a session bean written to the EJB 2.1 and earlier client view, the transaction attributes are specified for the methods of the component interface and all the direct and indirect superinterfaces of the component interface, excluding the methods of the javax.ejb.EJBObject or javax.ejb.EJBLocalObject interface; and for the timeout callback method, if any. Transaction attributes MUST NOT be specified for the methods of a session bean's home interface.
- For a EJB 2.1 (and earlier) entity bean, the transaction attributes are specified for the methods defined in the bean's component interface and all the direct and indirect superinterfaces of the component interface, excluding the getEJBHome, getEJBLocalHome, getHandle, getPrimaryKey, and isIdentical methods; for the methods defined in the bean's home interface and all the direct and indirect superinterfaces of the home interface, excluding the getEJBMetaData and getHomeHandle methods specific to the remote home interface; and for the timeout callback method, if any.

By DEFAULT, if a TransactionAttribute annotation is not specified for a method of an enterprise bean with container-managed transaction (CMT) demarcation, the value of the transaction attribute for the method is defined to be REQUIRED.

The Bean Provider may use the deployment descriptor as an alternative to metadata annotations to specify the transaction attributes (or as a means to supplement or override metadata annotations for transaction attributes). Transaction attributes specified in the deployment descriptor are assumed to OVERRIDE or supplement transaction attributes specified in annotations. If a transaction attribute value is not specified in the deployment descriptor, it is

assumed that the transaction attribute specified in annotations applies, or - in the case that no annotation has been specified - that the value is Required.

The application assembler is permitted to override the transaction attribute values using the bean's deployment descriptor. The deployer is also permitted to override the transaction attribute values at deployment time. Caution should be exercised when overriding the transaction attributes of an application, as the transactional structure of an application is typically intrinsic to the semantics of the application.

Enterprise JavaBeans defines the following values for the TransactionAttribute metadata annotation:

- MANDATORY
- REQUIRED
- REQUIRES_NEW
- SUPPORTS
- NOT_SUPPORTED
- NEVER

The deployment descriptor values that correspond to these annotation values are the following:

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

NOTE: For a message-driven bean's message listener methods (or interface), only the REQUIRED and NOT_SUPPORTED TransactionAttribute values may be used.

NOTE: For an enterprise bean's timeout callback method only the REQUIRES, REQUIRES_NEW and NOT_SUPPORTED transaction attributes may be used.

NOTE: If an enterprise bean implements the javax.ejb.SessionSynchronization interface, only the following values may be used for the transaction attributes of the bean's methods: REQUIRED, REQUIRES_NEW, MANDATORY.

The above restriction is necessary to ensure that the enterprise bean is invoked only in a transaction. If the bean were invoked without a transaction, the container would not be able to send the transaction synchronization calls.

Specification of Transaction Attributes with Metadata Annotations

The TransactionAttribute annotation is used to specify a transaction attribute.

```
@Target({METHOD, TYPE}) @Retention(RUNTIME)
public @interface TransactionAttribute {
    TransactionAttributeType value()
```

```
        default TransactionAttributeType.REQUIRED;
}
```

The value of the transaction attribute annotation is given by the enum TransactionAttributeType:

```
public enum TransactionAttributeType {
    MANDATORY,
    REQUIRED,
    REQUIRES_NEW,
    SUPPORTS,
    NOT_SUPPORTED,
    NEVER
}
```

The transaction attributes for the methods of a bean class may be specified on the class, the business methods of the class, or both.

Specifying the TransactionAttribute annotation on the bean class means that it applies to all applicable business interface methods of the class. If the transaction attribute type is not specified, it is assumed to be REQUIRED. The absence of a transaction attribute specification on the bean class is equivalent to the specification of TransactionAttribute(REQUIRED) on the bean class:

```
@Stateless
@TransactionAttribute(NOT_SUPPORTED)
public class PersistentCalculatorBean implements PersistentCalculator {

    public double add(double a, double b) {
        ...
    }
    ...
}
```

A transaction attribute may be specified on a method of the bean class to override the transaction attribute value explicitly or implicitly specified on the bean class:

```
@Stateless
@TransactionAttribute(NOT_SUPPORTED)
public class PersistentCalculatorBean  implements PersistentCalculator {

    public double add(double a, double b) {
        ...
    }

    @TransactionAttribute(REQUIRED)
    public void clearHistory() {
        ...
    }
    ...
}
```

If the bean class has superclasses, the following additional rules apply:

- A transaction attribute specified on a superclass S applies to the business methods defined by S. If a class-level transaction attribute is not specified on S, it is equivalent to specification of TransactionAttribute(REQUIRED) on S.
- A transaction attribute may be specified on a business method M defined by class S to override for method M the transaction attribute value explicitly or implicitly specified on the class S.
- If a method M of class S overrides a business method defined by a superclass of S, the transaction attribute of M is determined by the above rules as applied to class S.

Example:

```
@TransactionAttribute(SUPPORTS)
public class SomeClass {
	public void aMethod () {...}
	public void bMethod () {...}
	...
}

@Stateless public class ABean extends SomeClass implements A {
	public void aMethod () {...}

	@TransactionAttribute(REQUIRES_NEW)
	public void cMethod () {...}
	...
}
```

Assuming aMethod, bMethod, cMethod are methods of interface A, their transaction attributes are REQUIRED, SUPPORTS, and REQUIRES_NEW respectively.

Specification of Transaction Attributes in the Deployment Descriptor

Note that even in the absence of the use of annotations, it is not necessary to explicitly specify transaction attributes. If a transaction attribute is not specified for a method in an EJB 3.0 deployment descriptor, the transaction attribute DEFAULTS to Required.

If the deployment descriptor is used to override annotations, and transaction attributes are not specified for some methods, the values specified in annotations (whether explicit or defaulted) will apply for those methods.

The container-transaction element may be used to define the transaction attributes for business, home, component, and message-listener interface methods; web service endpoint methods; and timeout callback methods. Each container-transaction element consists of a list of ONE OR MORE method elements, and the trans-attribute element. The container-transaction element specifies that all the listed methods are assigned the specified transaction attribute value. It is required that all the methods specified in a single container-transaction element be methods of the SAME enterprise bean.

The method element uses the ejb-name, method-name, and method-params elements to denote one or more methods. There are three legal styles of composing the method element:

```
<!-- Style 1 -->

<method>
      <ejb-name>EJBNAME</ejb-name>
      <method-name>*</method-name>
</method>
```

This style is used to specify a default value of the transaction attribute for the methods for which there is no Style 2 or Style 3 element specified. There must be at most one container-transaction element that uses the Style 1 method element for a given enterprise bean.

```
<!-- Style 2 -->

<method>
      <ejb-name>EJBNAME</ejb-name>
      <method-name>METHOD</method-name>
</method>
```

This style is used for referring to a specified method of a business, home, component or message listener interface method; web service endpoint method; or timeout callback method of the specified enterprise bean. If there are multiple methods with the same overloaded name, this style refers to ALL the methods with the same name. There must be at most one container-transaction element that uses the Style 2 method element for a given method name. If there is also a container-transaction element that uses Style 1 element for the same bean, the value specified by the Style 2 element TAKES PRECEDENCE.

```
<!-- Style 3 -->

<method>
      <ejb-name>EJBNAME</ejb-name>
      <method-name>METHOD</method-name>
      <method-params>
            <method-param>PARAMETER_1</method-param>
            ...
            <method-param>PARAMETER_N</method-param>
      </method-params>
</method>
```

This style is used to refer to a single method within a set of methods with an overloaded name. If there is also a container-transaction element that uses the Style 2 element for the method name, or the Style 1 element for the bean, the value specified by the Style 3

element TAKES PRECEDENCE.

The optional method-intf element can be used to differentiate between methods with the same name and signature that are multiply defined across the business, component, and home interfaces, and/or web service endpoint. However, if the same method is a method of both a local business interface and the local component interface, the same transaction attribute applies to the method for BOTH interfaces. Likewise, if the same method is a method of both a remote business interface and the remote component interface, the same transaction attribute applies to the method for BOTH interfaces.

The following is an example of the specification of the transaction attributes in the deployment descriptor. The updatePhoneNumber method of the EmployeeRecord enterprise bean is assigned the transaction attribute Mandatory; all other methods of the EmployeeRecord bean are assigned the attribute Required. All the methods of the enterprise bean AardvarkPayroll are assigned the attribute RequiresNew:

```
<ejb-jar>
    ...
    <assembly-descriptor>
        ...
        <container-transaction>
            <method>
                <ejb-name>EmployeeRecord</ejb-name>
                <method-name>*</method-name>
            </method>
            <trans-attribute>Required</trans-attribute>
        </container-transaction>

        <container-transaction>
            <method>
                <ejb-name>EmployeeRecord</ejb-name>
                <method-name>updatePhoneNumber</method-name>
            </method>
            <trans-attribute>Mandatory</trans-attribute>
        </container-transaction>

        <container-transaction>
            <method>
                <ejb-name>AardvarkPayroll</ejb-name>
                <method-name>*</method-name>
            </method>
            <trans-attribute>RequiresNew</trans-attribute>
        </container-transaction>
    </assembly-descriptor>
</ejb-jar>
```

➢Identify correct and incorrect statements or examples about the use of the EJB API for transaction management, including getRollbackOnly, setRollbackOnly and the SessionSynchronization interfaces.

- [EJB_3.0_CORE] 13.3.4.1; 4.3.7; 13.3.4.2; 13.3.4.3; 13.6.2.8; 13.6.2.9

javax.ejb.SessionSynchronization Interface

A stateful session bean with **container-managed transaction** (CMT) demarcation can optionally implement the javax.ejb.SessionSynchronization interface.

The SessionSynchronization interface is defined as follows:

```
package javax.ejb;

public interface javax.ejb.SessionSynchronization {

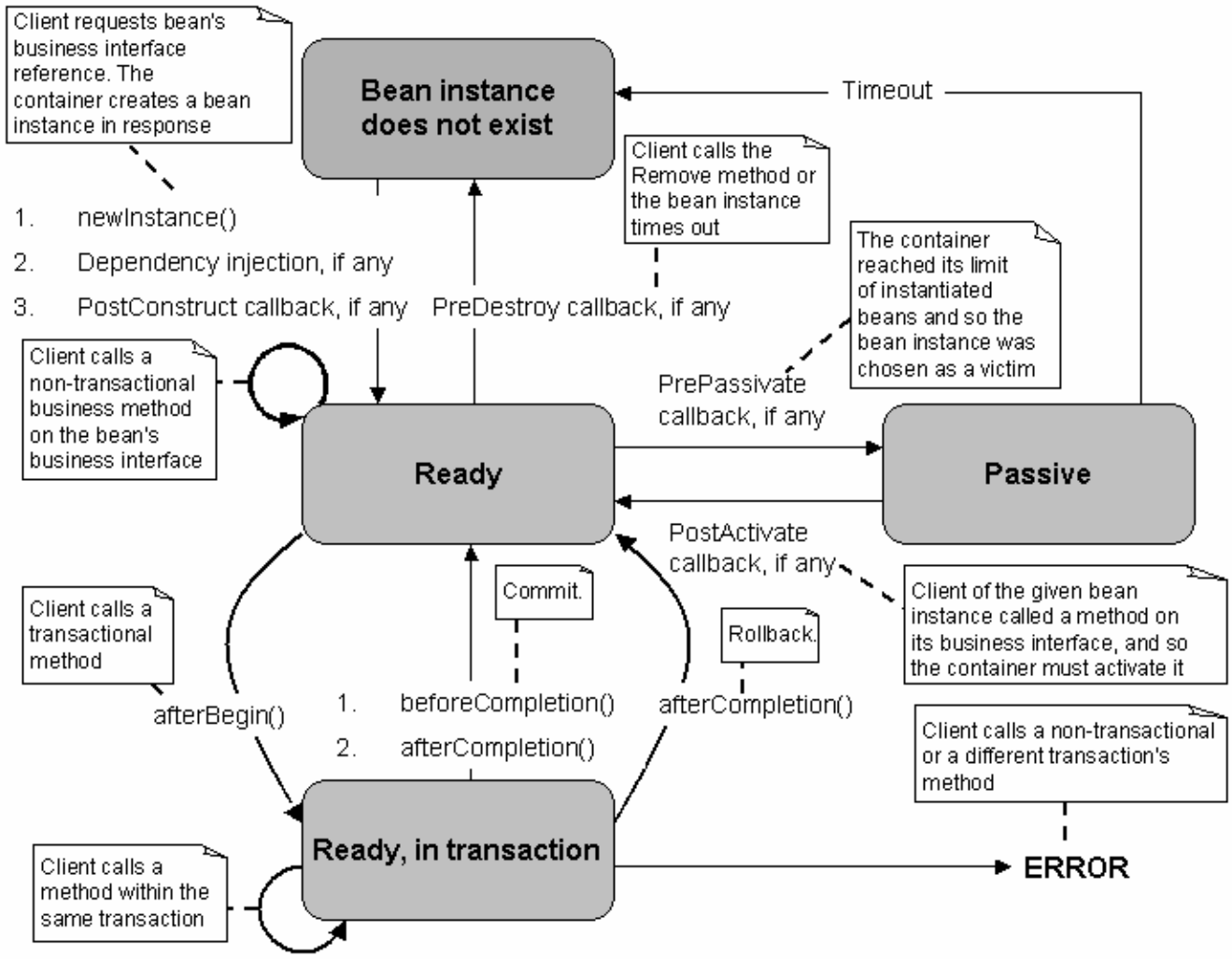
    void afterBegin() throws RemoteException;

    void beforeCompletion() throws RemoteException;

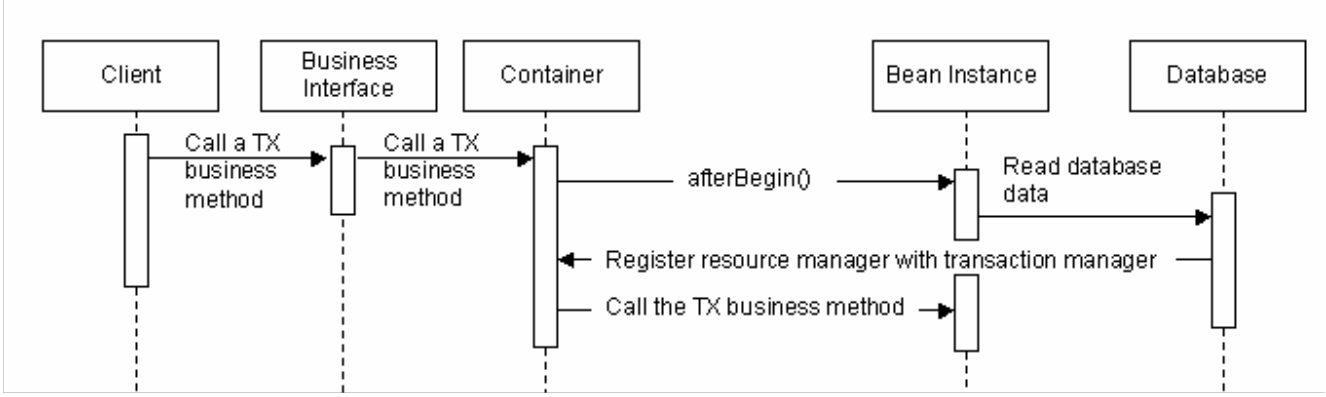
    void afterCompletion(boolean committed) throws RemoteException;

}
```

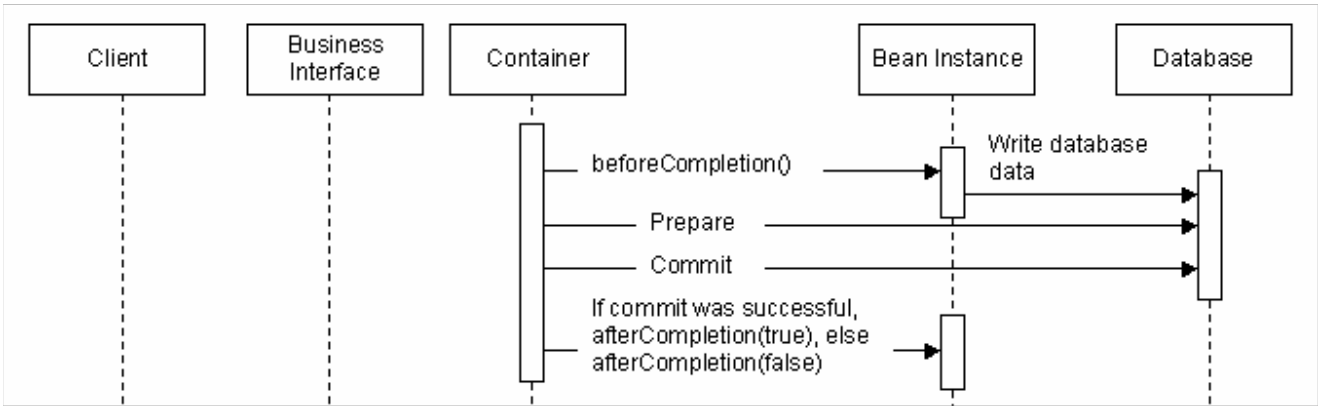
This interface provides the session bean instances with transaction synchronization notifications. The instances can use these notifications, for example, to manage database data they may cache within transactions - e.g., if the Java Persistence API is not used.



- The afterBegin notification signals a session bean instance that a new transaction has begun. The container invokes this method BEFORE the first business method within a transaction (which is not necessarily at the beginning of the transaction). The afterBegin notification is invoked with the transaction context. The instance may do any database work it requires within the scope of the transaction.
- The beforeCompletion notification is issued when a session bean instance's client has completed work on its current transaction but PRIOR to committing the resource managers used by the instance. At this time, the instance should write out any database updates it has cached. The instance can cause the transaction to roll back by invoking the setRollbackOnly method on its session context.
- The afterCompletion notification signals that the current transaction has completed. A completion status of true indicates that the transaction has committed. A status of false indicates that a rollback has occurred. Since a session bean instance's conversational state is not transactional, it may need to manually reset its state if a rollback occurred.



Begin transaction sequence:



Commit transaction sequence:

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

Only a **Stateful Session Bean with container-managed transaction (CMT) demarcation** MAY implement the SessionSynchronization interface. A stateless session bean MUST NOT implement the SessionSynchronization interface.

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

javax.ejb.EJBContext.setRollbackOnly Method

An enterprise bean with container-managed transaction (CMT) demarcation can use the setRollbackOnly method of its EJBContext object to mark the transaction such that the

transaction can NEVER COMMIT. Typically, an enterprise bean marks a transaction for rollback to protect data integrity before throwing an application exception, if the application exception class has not been specified to automatically cause the container to rollback the transaction.

For example, an AccountTransfer bean which debits one account and credits another account could mark a transaction for rollback if it successfully performs the debit operation, but encounters a failure during the credit operation.

The container must handle the EJBContext.setRollbackOnly method invoked from a business method executing with the REQUIRED, REQUIRES_NEW, or MANDATORY transaction attribute as follows:

- The container must ensure that the transaction will never commit. Typically, the container instructs the transaction manager to mark the transaction for rollback.
- If the container initiated the transaction immediately before dispatching the business method to the instance (as opposed to the transaction being inherited from the caller), the container must note that the instance has invoked the setRollbackOnly method. When the business method invocation completes, the container must roll back rather than commit the transaction. If the business method has returned normally or with an application exception, the container must pass the method result or the application exception to the client after the container performed the rollback.

The container MUST throw the java.lang.IllegalStateException if the EJBContext.setRollbackOnly method is invoked from a business method executing with the SUPPORTS, NOT_SUPPORTED, or NEVER transaction attribute.

javax.ejb.EJBContext.getRollbackOnly method

An enterprise bean with container-managed transaction demarcation can use the getRollbackOnly method of its EJBContext object to test if the current transaction has been marked for rollback. The transaction might have been marked for rollback by the enterprise bean itself, by other enterprise beans, or by other components (outside of the EJB specification scope) of the transaction processing infrastructure.

The container must handle the EJBContext.getRollbackOnly method invoked from a business method executing with the REQUIRED, REQUIRES_NEW, or MANDATORY transaction attribute.

The container MUST throw the java.lang.IllegalStateException if the EJBContext.getRollbackOnly method is invoked from a business method executing with the SUPPORTS, NOT_SUPPORTED, or NEVER transaction attribute.

➢**Chapter 10. Exceptions**

➢Identify correct and incorrect statements or examples about exception handling in EJB.

- [EJB_3.0_CORE] 14.1.1; 14.1.2

An *application exception* is an exception defined by the Bean Provider as part of the business logic of an application. Application exceptions are distinguished from *system exceptions* in EJB 3.0 specification.
Enterprise bean business methods use *application exceptions* to inform the client of abnormal application-level conditions, such as unacceptable values of the input arguments to a business method. A client can typically RECOVER from an application exception. Application exceptions are NOT intended for reporting system-level problems.

For example, the Account enterprise bean may throw an application exception to report that a debit operation cannot be performed because of an insufficient balance. The Account bean should not use an application exception to report, for example, the failure to obtain a database connection.

An application exception MAY be defined in the throws clause of a method of an enterprise bean's business interface, home interface, component interface, message listener interface, or web service endpoint.

An application exception MAY be a subclass (direct or indirect) of java.lang.Exception (i.e., a "*checked exception*"), or an application exception class may be defined as a subclass of the java.lang.RuntimeException (an "*unchecked exception*"). An application exception MAY NOT be a subclass of the java.rmi.RemoteException. The java.rmi.RemoteException and its subclasses are RESERVED for **system** exceptions.

The javax.ejb.CreateException, javax.ejb.RemoveException, javax.ejb.FinderException, and subclasses thereof are considered to be **application** exceptions. These exceptions are used as standard application exceptions to report errors to the client from the create, remove, and finder methods of the EJBHome and/or EJBLocalHome interfaces of components written to the EJB 2.1 client view.

Goals for Exception Handling

- An *application exception* thrown by an enterprise bean instance should be reported to the client PRECISELY (i.e., the client gets the SAME exception).
- An *application exception* thrown by an enterprise bean instance should NOT automatically rollback a client's transaction unless the application exception was defined to cause transaction rollback. The client should typically be given a chance to recover a transaction from an application exception.
- An unexpected exception that may have left the instance's state variables and/or underlying persistent data in an inconsistent state can be handled safely.

➢ Identify correct and incorrect statements or examples about application exceptions and system exceptions in session beans and message-driven beans, and defining a runtime exception as an application exception.

- [EJB_3.0_CORE] 14.2.1; 14.2.2

  [EJB_3.0_SIMPLIFIED] 10.8

Application Exceptions

The Bean Provider defines application exceptions. Application exceptions that are checked exceptions may be defined as such by being listed in the throws clauses of the methods of the bean's business interface, home interface, component interface, and web service endpoint. An application exception that is an unchecked exception is defined as an application exception by annotating it with the ApplicationException metadata annotation, or denoting it in the deployment descriptor with the application-exception element.

Example of runtime application exception (using annotation):

```
package by.iba;

import javax.ejb.ApplicationException;

@ApplicationException
// NOTE: This is the same as @ApplicationException(rollback=false)
public class InvalidEmployeeException extends RuntimeException {

	InvalidEmployeeException() {
	}

	InvalidEmployeeException(String s) {
		super(s);
	}
}
```

NOTE: ApplicationException may be applied to both CHECKED and UNCHECKED exceptions.

Example of runtime application exception (using deployment descriptor):

```
package by.iba;

public class InvalidEmployeeException extends RuntimeException {

	InvalidEmployeeException() {
	}

	InvalidEmployeeException(String s) {
		super(s);
	}
}
```

```
<?xml version="1.0" encoding="UTF-8"?>
<ejb-jar
        xmlns="http://java.sun.com/xml/ns/javaee"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
            http://java.sun.com/xml/ns/javaee/ejb-jar_3_0.xsd"
        version="3.0">

    <description>Sample of application exception</description>
    <assembly-descriptor>
        <application-exception>
            <exception-class>by.iba.InvalidEmployeeException</exception-class>
        </application-exception>
    </assembly-descriptor>
</ejb-jar>
```

The session bean may use the application exception now:

```
package by.iba.ejb;

@Stateless
public class PayrollBean implements Payroll {

    public int getTaxDeductions(Employee emp) {
        // ...
        throw new InvalidEmployeeException("Employee " + emp + " is invalid !");
    }
}
```

Because application exceptions are intended to be handled by the CLIENT, and not by the System Administrator, they should be used only for reporting business logic exceptions, not for reporting system level problems.

The Bean Provider is responsible for throwing the appropriate application exception from the business method to report a business logic exception to the client.

An application exception DOES NOT automatically result in marking the transaction for rollback unless the ApplicationException annotation is applied to the exception class and is specified with the rollback element value true or the application-exception deployment descriptor element for the exception specifies the rollback element as true. The rollback subelement of the application-exception deployment descriptor element may be explicitly specified to OVERRIDE the rollback value specified or defaulted by the ApplicationException annotation.

Example of runtime application exception which causes transaction ROLLBACK (using annotation):

```
package by.iba;

import javax.ejb.ApplicationException;
```

```
@ApplicationException(rollback=true)
public class InvalidEmployeeException extends RuntimeException {

	InvalidEmployeeException() {
	}

	InvalidEmployeeException(String s) {
		super(s);
	}
}
```

NOTE: ApplicationException may be applied to both CHECKED and UNCHECKED exceptions.

Example of runtime application exception which causes transaction ROLLBACK (using deployment descriptor):

```
package by.iba;

public class InvalidEmployeeException extends RuntimeException {

	InvalidEmployeeException() {
	}

	InvalidEmployeeException(String s) {
		super(s);
	}
}
```

```
<?xml version="1.0" encoding="UTF-8"?>
<ejb-jar
		xmlns="http://java.sun.com/xml/ns/javaee"
		xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
		xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
			http://java.sun.com/xml/ns/javaee/ejb-jar_3_0.xsd"
		version="3.0">

	<description>Sample of application exception causing ROLLBACK</description>
	<assembly-descriptor>
		<application-exception>
			<exception-class>by.iba.InvalidEmployeeException</exception-class>
			<rollback>true</rollback>
		</application-exception>
	</assembly-descriptor>
</ejb-jar>
```

The Bean Provider MUST do one of the following to ensure data integrity before throwing an application exception from an enterprise bean instance:

- Ensure that the instance is in a state such that a client's attempt to continue and/or commit the transaction does not result in loss of data integrity. For example, the instance

throws an application exception indicating that the value of an input parameter was invalid before the instance performed any database updates.

- If the application exception is not specified to cause transaction rollback, mark the transaction for rollback using the EJBContext.setRollbackOnly method before throwing the application exception. Marking the transaction for rollback will ensure that the transaction can never commit.

The Bean Provider is also responsible for using the standard EJB application exceptions (javax.ejb.CreateException, javax.ejb.RemoveException, javax.ejb.FinderException, and subclasses thereof) for beans written to the EJB 2.1 and earlier client view.

Bean Providers may define subclasses of the standard EJB application exceptions and throw instances of the subclasses in the enterprise bean methods. A subclass will typically provide more information to the client that catches the exception.

System Exceptions

A **system** exception is an exception that is a java.rmi.RemoteException (or one of its subclasses) or a RuntimeException that is not an application exception.

An enterprise bean business method, message listener method, business method interceptor method, or lifecycle callback interceptor method may encounter various exceptions or errors that prevent the method from successfully completing. Typically, this happens because the exception or error is unexpected, or the exception is expected but the EJB Provider does not know how to recover from it. Examples of such exceptions and errors are: failure to obtain a database connection, JNDI exceptions, unexpected RemoteException from invocation of other enterprise beans, unexpected RuntimeException, JVM errors, and so on.

If the enterprise bean method encounters a system-level exception or error that does not allow the method to successfully complete, the method should throw a suitable non-application exception that is compatible with the method?Hs throws clause. While the EJB specification does not prescribe the exact usage of the exception, it encourages the Bean Provider to follow these guidelines:

- If the bean method encounters a system exception or error, it should simply propagate the error from the bean method to the container (i.e., the bean method does not have to catch the exception).
- If the bean method performs an operation that results in a checked (i.e. not a subclass of java.lang.RuntimeException) exception that the bean method cannot recover, the bean method should throw the javax.ejb.EJBException that wraps the original exception.
- Any other unexpected error conditions should be reported using the javax.ejb.EJBException.

Note that the javax.ejb.EJBException is a subclass of the java.lang.RuntimeException, and therefore it does not have to be listed in the throws clauses of the business methods.

The container catches a non-application exception; logs it (which can result in alerting the System Administrator); and, unless the bean is a message-driven bean, throws the javax.ejb.EJBException or, if the web service client view is used, the java.rmi.RemoteException. If the EJB 2.1 client view is used, the container throws the java.rmi.RemoteException (or

subclass thereof) to the client if the client is a REMOTE client, or throws the javax.ejb.EJBException (or subclass thereof) to the client if the client is a LOCAL client. In the case of a message-driven bean, the container logs the exception and then throws a javax.ejb.EJBException that wraps the original exception to the resource adapter.

The Bean Provider can rely on the container to perform the following tasks when catching a non-application exception:

- The transaction in which the bean method participated will be rolled back.
- No other method will be invoked on an instance that threw a non-application exception.

This means that the Bean Provider DOES NOT have to perform any cleanup actions before throwing a non-application exception. It is the **container** that is responsible for the CLEANUP.

Exceptions Metadata Annotations

The ApplicationException annotation is applied to an exception to denote that it is an **application exception** and should be reported to the client DIRECTLY (i.e., unwrapped). The ApplicationException annotation may be applied to both **checked** and **unchecked** exceptions. The rollback element is used to indicate whether the container must cause the transaction to rollback when the exception is thrown.

```
@Target(TYPE) @Retention(RUNTIME)
public @interface ApplicationException {
    boolean rollback() default false;
}
```

➢**Given a list of responsibilities related to exceptions, identify those which are the bean provider's, and those which are the responsibility of the container provider. Be prepared to recognize responsibilities for which neither the bean nor container provider is responsible.**

- [EJB_3.0_CORE] See previous sections; 14.3.11; 4.4.3; 5.4.18

Release of Resources

When the container discards an instance because of a system exception, the container should release all the resources held by the instance that were acquired through the resource factories declared in the enterprise bean environment. For example, an object that implements the javax.sql.DataSource interface is a resource manager connection factory for java.sql.Connection objects that implement connections to a database management system.

NOTE: While the container should release the connections to the resource managers that the instance acquired through the resource factories declared in the enterprise bean environment, the container cannot, in general, release "unmanaged" resources that the instance may have acquired through the JDK APIs. For example, if the instance has opened a TCP/IP connection, most container implementations will not be able to release the connection. The connection will be eventually released by the JVM garbage collector mechanism.

Missed PreDestroy Calls for Session Beans

The Bean Provider cannot assume that the container will always invoke the PreDestroy lifecycle

callback interceptor method(s) (or ejbRemove method) for a session bean instance. The following scenarios result in the PreDestroy lifecycle callback interceptor method(s) not being called for an instance:

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

If resources are allocated in a PostConstruct lifecycle callback interceptor method (or ejbCreate<METHOD> method) and/or in the business methods, and normally released in a PreDestroy lifecycle callback interceptor method, these resources will NOT be automatically released in the above scenarios. The application using the session bean should provide some clean up mechanism to periodically clean up the unreleased resources.

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

Missed PreDestroy Callbacks for Message-Driven Beans

The Bean Provider cannot assume that the container will always invoke the PreDestroy callback method (or ejbRemove method) for a message-driven bean instance. The following scenarios result in the PreDestroy callback method not being called on an instance:

- A crash of the EJB container.
- A **system exception** thrown from the instance's method to the container.

If the message-driven bean instance allocates resources in the PostConstruct lifecycle callback method and/or in the message listener method, and releases normally the resources in the PreDestroy method, these resources will NOT be automatically released in the above scenarios. The application using the message-driven bean should provide some clean up mechanism to periodically clean up the unreleased resources.

➢Identify correct and incorrect statements or examples about the client's view of exceptions received from an enterprise bean invocation.

- [EJB_3.0_CORE] 14.4; 14.4.1.1; 14.4.1.2; 14.4.2

Client's View of Exceptions

A client accesses an enterprise bean either through the enterprise bean's business interface (whether local or remote), through the enterprise bean's remote home and remote interfaces, through the enterprise bean's local home and local interfaces, or through the enterprise bean's web service client view depending on whether the client is written to the EJB 3.0 API or earlier API and whether the client is a remote client, a local client, or a web service client.

The methods of the BUSINESS interface typically DO NOT throw the java.rmi.RemoteException, regardless of whether the interface is a remote or local interface.

```
/**
 * Local Business Interface
 */

@Local
public interface Calculator {
	public double add(double a, double b);
}


/**
 * Remote Business Interface
 */

@Remote
public interface RemoteCalculator {
	public double add(double a, double b);
	public String getServerInfo();
}


@Stateless
public class StatelessCalculator implements Calculator, RemoteCalculator {

	public double add(double a, double b) {
		return a + b;
	}

	public String getServerInfo () {
		return "This is SCBCD 5.0 Study Guide !";
	}
}
```

The REMOTE HOME interface, the REMOTE interface, and the WEB SERVICE ENDPOINT interface are Java RMI interfaces, and therefore the throws clauses of all their methods (including those inherited from superinterfaces) include the MANDATORY java.rmi.RemoteException. The throws clauses MAY include an arbitrary number of *application exceptions*.

```
/**
 * JAX-RPC Service Endpoint Interface
 */
public interface Calculator extends java.rmi.Remote {
	public double add(double a, double b) throws java.rmi.RemoteException;
	public String getServerInfo() throws java.rmi.RemoteException;
}
```

The LOCAL HOME and LOCAL interfaces are both Java local interfaces, and the throws clauses of all their methods (including those inherited from superinterfaces) MUST NOT include the java.rmi.RemoteException. The throws clauses MAY include an arbitrary number of *application exceptions*.

Application Exception to Local and Remote Clients

If a client program receives an APPLICATION exception from an enterprise bean invocation, the client CAN CONTINUE calling the enterprise bean. An **application** exception DOES NOT result in the removal of the EJB object.

Although the container does not automatically mark for rollback a transaction because of a thrown application exception, the transaction MIGHT have been marked for rollback by the enterprise bean instance before it threw the application exception OR the application exception MAY have been specified to require the container to rollback the transaction. There are two ways to learn if a particular application exception results in transaction rollback or not:

- Statically. Programmers can check the documentation of the enterprise bean's client view interface. The Bean Provider may have specified (although he or she is not required to) the application exceptions for which the enterprise bean marks the transaction for rollback before throwing the exception.
- Dynamically. Clients that are enterprise beans with container-managed transaction (CMT) demarcation can use the getRollbackOnly method of the javax.ejb.EJBContext object to learn if the current transaction has been marked for rollback; other clients (BMT) may use the getStatus method of the javax.transaction.UserTransaction interface to obtain the transaction status.

Application Exception to Web Service Clients

If a stateless session bean throws an application exception from one of its web service methods, it is the responsibility of the container to map the exception to the SOAP fault specified in the WSDL that describes the port type that the stateless session bean implements.

System Exception (java.rmi.RemoteException and javax.ejb.EJBException)

A client receives the javax.ejb.EJBException or the java.rmi.RemoteException as an indication of a **failure to invoke** an enterprise bean method or to properly complete its invocation. The exception can be thrown by the CONTAINER or by the COMMUNICATION SUBSYSTEM between the client and the container.

If the client receives the javax.ejb.EJBException or the java.rmi.RemoteException exception from a method invocation, the client, in general, DOES NOT KNOW if the enterprise bean's method has been completed or not.

If the client executes in the context of a transaction, the client's transaction may, OR may not, have been marked for rollback by the communication subsystem or target bean's container.

For example, the transaction would be marked for rollback if the underlying transaction service or the target bean's container doubted the integrity of the data because the business method may have been partially completed. Partial completion could happen, for example, when the target bean's method returned with a RuntimeException exception, or if the remote server crashed in the middle of executing the business method.

The transaction may not necessarily be marked for rollback. This might occur, for example, when the communication subsystem on the client-side has not been able to send the request to the server.

When a client executing in a transaction context receives an EJBException or a RemoteException from an enterprise bean invocation, the client may use either of the following strategies to deal with the exception:

- **Discontinue the transaction**. If the client is the transaction originator, it may simply rollback its transaction. If the client is not the transaction originator, it can mark the transaction for rollback or perform an action that will cause a rollback. For example, if the client is an enterprise bean, the enterprise bean may throw a RuntimeException which will cause the container to rollback the transaction.
- **Continue the transaction**. The client may perform additional operations on the same or other enterprise beans, and eventually attempt to commit the transaction. If the transaction was marked for rollback at the time the EJBException or RemoteException was thrown to the client, the commit will fail.

If the client chooses to continue the transaction, the client can first inquire about the transaction status to avoid fruitless computation on a transaction that has been marked for rollback. A client that is an enterprise bean with container-managed transaction (CMT) demarcation can use the EJBContext.getRollbackOnly method to test if the transaction has been marked for rollback; a client that is an enterprise bean with bean-managed transaction (BMT) demarcation, and other client types, can use the UserTransaction.getStatus method to obtain the status of the transaction.

Some implementations of EJB servers and containers may provide more detailed exception reporting by throwing an appropriate subclass of the javax.ejb.EJBException or java.rmi.RemoteException to the client.

➢Given a particular method condition, identify the following: whether an exception will be thrown, the type of exception thrown, the container's action, and the client's view.

- [EJB_3.0_CORE] 14.3.1; 14.3.4

Exceptions from a Session Bean's Business Interface Methods

Table below specifies how the container must handle the exceptions thrown by the methods of the business interface for beans with container-managed transaction (CMT) demarcation, including the exceptions thrown by business method interceptor methods which intercept the invocation of business methods. The table specifies the container's action as a function of the condition under which the business interface method executes and the exception thrown by the method. The table also illustrates the exception that the client will receive and how the client can recover from the exception.

Table 10.1. Handling of Exceptions Thrown by a Business Interface Method of a Bean with Container-Managed Transaction Demarcation

Method condition	Method exception	Container's action	Client's view

Method condition	Method exception	Container's action	Client's view
	all other exceptions	Log the exception or error (so that the System Administrator is alerted of the problem). Mark the transaction for rollback. Discard instance (so that the container must not invoke any business methods or container callbacks on the instance). Throw javax.ejb.EJBTransactionRolledbackException to client. If the business interface is a remote business interface that extends java.rmi.Remote, the javax.transaction.TransactionRolledbackException is thrown to the client, which will receive this exception.	Receives javax.ejb.EJBTransactionRolledbackException. Continuing transaction is fruitless.
	all other exceptions	Log the exception or error. Rollback the container-started transaction. Discard instance. Throw EJBException to client. If the business interface is a remote business interface that extends java.rmi.Remote, the java.rmi.RemoteException is thrown to the client, which will receive this exception.	Receives EJBException. If the client executes in a transaction, the client's transaction may or may not be marked for rollback.
	all other exceptions	Log the exception or error. Discard instance. Throw EJBException to client. If the business interface is a remote business interface that extends java.rmi.Remote, the java.rmi.RemoteException is thrown to the client, which will receive this exception.	Receives EJBException. If the client executes in a transaction, the client's transaction may OR may not be marked for rollback.

Table below specifies how the container must handle the exceptions thrown by the methods of

the business interface for beans with bean-managed transaction (BMT) demarcation, including the exceptions thrown by business method interceptor methods which intercept the invocation of business methods. The table specifies the container's action as a function of the condition under which the business interface method executes and the exception thrown by the method. The table also illustrates the exception that the client will receive and how the client can recover from the exception.

Table 10.2. Handling of Exceptions Thrown by a Business Interface Method of a Session Bean with Bean-Managed Transaction Demarcation

Bean method condition	Bean method exception	Container action	Client receives
	all other exceptions	Log the exception or error. Mark for rollback a transaction that has been started, but not yet completed, by the instance. Discard instance. Throw EJBException to client. If the business interface is a remote business interface that extends java.rmi.Remote, the java.rmi.RemoteException is thrown to the client, which will receive this exception.	Receives EJBException.

Exceptions from Message-Driven Bean Message Listener Methods

Table below specifies how the container must handle the exceptions thrown by a message listener method of a message-driven bean with container-managed transaction (CMT) demarcation, including the exceptions thrown by business method interceptor methods which intercept the invocation of message listener methods. The table specifies the container's action as a function of the condition under which the method executes and the exception thrown by the method.

Table 10.3. Handling of Exceptions Thrown by a Message Listener Method of a Message-Driven Bean with Container-Managed Transaction Demarcation

Method condition	Method exception	Container's action
	system exceptions	Log the exception or error (so that the System Administrator is alerted of the problem). Rollback the container-started transaction. Discard instance (so that the container must

Method condition	Method exception	Container's action
		not invoke any methods on the instance). Throw EJBException that wraps the original exception to resource adapter.
	system exceptions	Log the exception or error. Discard instance. Throw EJBException that wraps the original exception to resource adapter.

Table below specifies how the container must handle the exceptions thrown by a message listener method of a message-driven bean with bean-managed transaction (BMT) demarcation. The table specifies the container's action as a function of the condition under which the method executes and the exception thrown by the method.

Table 10.4. Handling of Exceptions Thrown by a Message Listener Method of a Message-Driven Bean with Bean-Managed Transaction Demarcation

Bean method condition	Bean method exception	Container action
	system exceptions	Log the exception or error. Mark for rollback a transaction that has been started, but not yet completed, by the instance. Discard instance. Throw EJBException that wraps the original exception to resource adapter.

➢**Chapter 11. Security Management**

➢Match security behaviors to declarative security specifications (default behavior, security roles, security role references, and method permissions).

- [EJB_3.0_CORE] 17.2.5.3 ; 17.3.1 ; 17.3.2 ; 17.3.2.3 ; 17.3.3

Default Behavior

- If the Bean Provider and Application Assembler do not define security roles, the Deployer will have to define security roles at deployment time.
- **Unspecified Method Permissions**
- It is possible that some methods are NOT assigned to any security roles nor annotated as DenyAll or contained in the exclude-list element. In this case, the Deployer should assign method permissions for all of the unspecified methods, either by assigning them to security roles, or by marking them as unchecked.
- If the Deployer does not assigned method permissions to the unspecified methods, those methods MUST be treated by the container as unchecked.
- **Security Roles**
- The Bean Provider or Application Assembler can define one or more security roles in the bean's metadata annotations or deployment descriptor. The Bean Provider or Application Assembler then assigns groups of methods of the enterprise beans' business, home, and component interfaces, and/or web service endpoints to the security roles to define the security view of the application.
- Because the Bean Provider and Application Assembler do not, in general, know the security environment of the operational environment, the security roles are meant to be logical roles (or actors), each representing a type of user that should have the same access rights to the application.
- The Deployer then assigns user groups and/or user accounts defined in the operational environment to the security roles defined by the Bean Provider and Application Assembler.
- Defining the security roles in the metadata annotations and/or deployment descriptor is optional. Their omission means that the Bean Provider and Application Assembler chose not to pass any security deployment related instructions to the Deployer.
- If Java language metadata annotations are used, the Bean Provider uses the DeclareRoles and RolesAllowed annotations to define the security roles. The set of security roles used by the application is taken to be the aggregation of the security roles defined by the security role names used in the DeclareRoles

and RolesAllowed annotations. The Bean Provider may augment the set of security roles defined for the application by annotations in this way by means of the security-role deployment descriptor element.

- If the deployment descriptor is used, the The Bean Provider and/or Application Assembler uses the security-role deployment descriptor element as follows:
    - Define each security role using a security-role element.
    - Use the role-name element to define the name of the security role.
    - Optionally, use the description element to provide a description of a security role.

The following example illustrates security roles definition in a deployment descriptor:

```
<assembly-descriptor>
    <security-role>
        <description>
            This role includes the employees of the
            enterprise who are allowed to access the
            employee self-service application. This role
            is allowed only to access his/her own
            information.
        </description>
        <role-name>employee</role-name>
    </security-role>

    <security-role>
        <description>
            This role includes the employees of the human
            resources department. The role is allowed to
            view and update all employee records.
        </description>
        <role-name>hr-department</role-name>
    </security-role>

    <security-role>
        <description>
            This role includes the employees of the payroll
            department. The role is allowed to view and
            update the payroll entry for any employee.
        </description>
        <role-name>payroll-department</role-name>
    </security-role>

    <security-role>
        <description>
            This role should be assigned to the personnel
            authorized to perform administrative functions
            for the employee self-service application.
            This role does not have direct access to
            sensitive employee and payroll information.
        </description>
        <role-name>admin</role-name>
```

```
    </security-role>
</assembly-descriptor>
```

- **Security Role References**
- **Declaration of Security Roles Referenced from the Bean's Code**
- The Bean Provider is responsible for using the DeclareRoles annotation or the security-role-ref elements of the deployment descriptor to declare all the security role names used in the enterprise bean CODE. The DeclareRoles annotation is specified on a bean class, where it serves to declare roles that may be tested by calling isCallerInRole from within the methods of the annotated class. Declaring the security roles allows the Bean Provider, Application Assembler, or Deployer to link these security role names used in the code to the security roles defined for an assembled application. In the absence of this linking step, any security role name as used in the code will be assumed to correspond to a security role of the SAME name.
- The Bean Provider declares the security roles referenced in the code using the DeclareRoles metadata annotation. When declaring the name of a role used as a parameter to the isCallerInRole(String roleName) method, the declared name MUST be the same as the parameter value. The Bean Provider may optionally provide a description of the named security roles in the description element of the DeclareRoles annotation.
- In the following example, the DeclareRoles annotation is used to indicate that the enterprise bean AardvarkPayroll makes the security check using isCallerInRole("payroll") in its business method.

```
@DeclareRoles("payroll")
@Stateless public class PayrollBean implements Payroll {

    @Resource SessionContext ctx;

    public void updateEmployeeInfo(EmplInfo info) {

        oldInfo = ... read from database;

        // The salary field can be changed only by callers
        // who have the security role "payroll"
        if (info.salary != oldInfo.salary && !ctx.isCallerInRole("payroll")) {
            throw new SecurityException(...);
        }
        ...
    }
    ...
}
```

- If the DeclareRoles annotation is not used, the Bean Provider MUST use the security-role-ref elements of the deployment descriptor to declare the security roles referenced in the code. The security-role-ref elements are defined as follows:
    - Declare the name of the security role using the role-name element. The name MUST be the security role name that is used as a parameter to the isCallerInRole(String roleName) method.
    - Optionally provide a description of the security role in the description element.

The following example illustrates how an enterprise bean's references to security roles are declared in the deployment descriptor.

```
<enterprise-beans>
    ...
    <session>
        <ejb-name>AardvarkPayroll</ejb-name>
        <ejb-class>com.aardvark.payroll.PayrollBean</ejb-class>
        ...
        <security-role-ref>
            <description>
                This security role should be assigned to the
                employees of the payroll department who are
                allowed to update employees' salaries.
            </description>
            <role-name>payroll</role-name>
        </security-role-ref>
        ...
    </session>
    ...
</enterprise-beans>
```

- The deployment descriptor above indicates that the enterprise bean AardvarkPayroll makes the security check using isCallerInRole("payroll") in its business method.
- A security role reference, including the name defined by the reference, is scoped to the COMPONENT whose bean class contains the DeclareRoles metadata annotation or whose deployment descriptor element contains the security-role-ref deployment descriptor element.
- The Bean Provider (or Application Assembler) may also use the security-role-ref elements for those references that were declared in annotations and which the Bean Provider wishes to have linked to a security-role whose name differs from the reference value. If a security role reference is NOT linked to a security role in this way, the container MUST map the reference name to the security role of the SAME name.
- **Linking Security Role References to Security Roles**

- The security role references used in the COMPONENTS of the application are linked to the security roles defined for the APPLICATION. In the absence of any explicit linking, a security role reference will be linked to a security role having the SAME name.
- The Application Assembler may explicitly link all the security role references declared in the DeclareRoles annotation or security-role-ref elements for a component to the security roles defined by the use of annotations and/or in the security-role elements.
- The Application Assembler links each security role reference to a security role using the role-link element. The value of the role-link element MUST be the name of one of the security ROLES defined in a security-role element or by means of the DeclareRoles annotations or RolesAllowed annotations, but need not be specified when the role-name used in the code is the same as the name of the security-role (to be linked).

The following deployment descriptor example shows how to link the security role reference named payroll to the security role named payroll-department:

```
<enterprise-beans>
    ...
    <session>
        <ejb-name>AardvarkPayroll</ejb-name>
        <ejb-class>com.aardvark.payroll.PayrollBean</ejb-class>
        ...
        <security-role-ref>
            <description>
                This role should be assigned to the
                employees of the payroll department.
                Members of this role have access to
                anyone's payroll record.
                The role has been linked to the
                payroll-department role.
            </description>
            <role-name>payroll</role-name>
            <role-link>payroll-department</role-link>
        </security-role-ref>
        ...
    </session>
    ...
</enterprise-beans>
```

- **Method Permissions**
- If the Bean Provider and/or Application Assembler have defined security roles for the enterprise beans in the ejb-jar file, they can also specify the methods of the business, home, and component interfaces, and/or web service endpoints that each security role is allowed to invoke.
- Metadata annotations and/or the deployment descriptor can be used for this purpose.

- Method permissions are defined as a binary relation from the set of security roles to the set of methods of the business interfaces, home interfaces, component interfaces, and/or web service endpoints of session and entity beans, including all their superinterfaces (including the methods of the EJBHome and EJBObject interfaces and/or EJBLocalHome and EJBLocalObject interfaces). The method permissions relation includes the pair (R, M) if and only if the security role R is allowed to invoke the method M.

- **From a list of responsibilities, identify which roles are responsible for which aspects of security: application assembler, bean provider, deployer, container provider, system administrator, or server provider.**

  - [EJB_3.0_CORE] 17.3; 17.4; 17.4.2

Responsibilities of the Bean Provider and/or Application Assembler

The Bean Provider and Application Assembler (which could be the same party as the Bean Provider) may define a *security view* of the enterprise beans contained in the ejb-jar file. Providing the security view is optional for the Bean Provider and Application Assembler.

The main reason for providing the security view of the enterprise beans is to simplify the Deployer's job. In the absence of a security view of an application, the Deployer needs detailed knowledge of the application in order to deploy the application securely. For example, the Deployer would have to know what each business method does to determine which users can call it. The security view defined by the Bean Provider or Application Assembler presents a more consolidated view to the Deployer, allowing the Deployer to be less familiar with the application.

The security view consists of a set of *security roles*. A security role is a semantic grouping of permissions that a given type of users of an application must have in order to successfully use the application.

The Bean Provider or Application Assembler defines *method permissions* for each security role. A method permission is a permission to invoke a specified group of methods of the enterprise beans' business interface, home interface, component interface, and/or web service endpoint.

It is important to keep in mind that the security roles are used to define the LOGICAL security view of an application. They should not be confused with the user groups, users, principals, and other concepts that exist in the target enterprise's operational environment.

In special cases, a qualified Deployer may change the definition of the security roles for an application, or completely ignore them and secure the application using a different mechanism that is specific to the operational environment.

Deployer's Responsibilities

The Deployer is responsible for ensuring that an assembled application is secure after it has been deployed in the target operational environment. This section defines the Deployer's responsibility with respect to EJB security management.

The Deployer uses deployment tools provided by the EJB Container Provider to read the security view of the application supplied by the Bean Provider and/or Application Assembler in the metadata annotations and/or deployment descriptor. The Deployer's job is to map the security view that was specified by the Bean Provider and/or Application Assembler to the mechanisms and policies used by the security domain in the target operational environment. The output of the Deployer's work includes an application security policy descriptor that is specific to the operational environment. The format of this descriptor and the information stored in the descriptor are specific to the EJB container.

The Deployer assigns principals and/or groups of principals (such as individual users or user groups) used for managing security in the operational environment to the security roles defined by means of the DeclareRoles and RolesAllowed metadata annotations and/or security-role elements of the deployment descriptor.

The Deployer does not assign principals and/or principal groups to the security role references - the principals and/or principals groups assigned to a security role apply also to all the linked security role references.

For example, the Deployer of the AardvarkPayroll enterprise bean would assign principals and/or principal groups to the security-role payroll-department, and the assigned principals and/or principal groups would be implicitly assigned also to the linked security role payroll:

```
<session>
    <ejb-name>AardvarkPayroll</ejb-name>
    <ejb-class>com.aardvark.payroll.PayrollBean</ejb-class>
    ...
    <security-role-ref>
        <description>
            This role should be assigned to the
            employees of the payroll department.
            Members of this role have access to
            anyone's payroll record.
            The role has been linked to the
            payroll-department role.
        </description>
        <role-name>payroll</role-name>
        <role-link>payroll-department</role-link>
    </security-role-ref>
    ...
</session>
```

The EJB architecture DOES NOT specify how an enterprise should implement its security architecture. Therefore, the process of assigning the logical security roles defined in the application's deployment descriptor to the operational environment's security concepts is specific to that operational environment. Typically, the deployment process consists of assigning to each security role one or more user groups (or individual users) defined in the operational environment. This assignment is done on a per-application basis. (That is, if multiple independent ejb-jar files use the same security role name, each may be assigned differently.) If the deployer does not assign the logical security roles defined by the application to groups in the operational environment, it must be assumed that a logical role maps to a principal or principal

group of the same name.

EJB Container Provider's Responsibilities

The EJB Container Provider is responsible for providing the deployment tools that the Deployer can use to perform the deployment tasks.

➢Identify correct and incorrect statements or examples about use of the isCallerInRole and getCallerPrincipal EJB programmatic security APIs.

- EJB_3.0_CORE] 17.2.5; 17.2.5.1; 17.2.5.2;

Note: In general, security management should be enforced by the container in a manner that is transparent to the enterprise beans' business methods. The security API described in this section should be used only in the less frequent situations in which the enterprise bean business methods need to access the security context information.
The javax.ejb.EJBContext interface provides two methods (plus two deprecated methods that were defined in EJB 1.0) that allow the Bean Provider to access security information about the enterprise bean's caller:

```
public interface javax.ejb.EJBContext {
    ...
    //
    // The following two methods allow the EJB class
    // to access security information.
    //
    java.security.Principal getCallerPrincipal();
    boolean isCallerInRole(String roleName);

    //
    // The following two EJB 1.0 methods are DEPRECATED.
    //
    java.security.Identity getCallerIdentity();
    boolean isCallerInRole(java.security.Identity role);
    ...
}
```

The Bean Provider can invoke the getCallerPrincipal and isCallerInRole methods ONLY in the enterprise bean's business methods. If they are otherwise invoked when no security context exists, they should throw the java.lang.IllegalStateException runtime exception.

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

Use of getCallerPrincipal

The purpose of the getCallerPrincipalmethod is to allow the enterprise bean methods to obtain the current caller principal's name. The methods might, for example, use the name as a key to information in a database.

An enterprise bean can invoke the getCallerPrincipal method to obtain a java.security.Principal interface representing the current caller. The enterprise bean can then obtain the distinguished name of the caller principal using the getName method of the java.security.Principal interface. If the security identity has not been established, getCallerPrincipal returns the container's representation of the UNAUTHENTICATED identity.

Note that getCallerPrincipal returns the principal that represents the CALLER of the enterprise bean, NOT the principal that corresponds to the run-as security identity for the bean, if any.

The meaning of the *current caller*, the Java class that implements the java.security.Principal interface, and the realm of the principals returned by the getCallerPrincipal method depend on the operational environment and the configuration of the application.

The following code sample illustrates the use of the getCallerPrincipal() method:

```
@Stateless public class EmployeeServiceBean implements EmployeeService {

    @Resource SessionContext ctx;
    @PersistenceContext EntityManager em;

    public void changePhoneNumber(...) {
        ...
        // obtain the caller principal.
        callerPrincipal = ctx.getCallerPrincipal();

        // obtain the caller principalтАЩs name.
        callerKey = callerPrincipal.getName();

        // use callerKey as primary key to find EmployeeRecord
        EmployeeRecord myEmployeeRecord = em.find(EmployeeRecord.class, callerKey);

        // update phone number
        myEmployeeRecord.setPhoneNumber(...);
        ...
    }
}
```

In the previous example, the enterprise bean obtains the principal name of the current caller and uses it as the primary key to locate an EmployeeRecord entity. This example assumes that application has been deployed such that the current caller principal contains the primary key used for the identification of employees (e.g., employee number).

Use of isCallerInRole

The main purpose of the isCallerInRole(String roleName) method is to allow the Bean Provider to code the security checks that cannot be easily defined declaratively in the deployment descriptor using method permissions. Such a check might impose a role-based limit on a request, or it might depend on information stored in the database.

The enterprise bean code can use the isCallerInRole method to test whether the current caller has been assigned to a given security role. Security roles are defined by the Bean Provider or the Application Assembler, and are assigned to principals or principal groups that exist in the operational environment by the Deployer.

Note that isCallerInRole(String roleName) tests the principal that represents the CALLER of the enterprise bean, NOT the principal that corresponds to the run-as security identity for the bean, if any.

The following code sample illustrates the use of the isCallerInRole(String roleName) method:

```
@Stateless public class PayrollBean implements Payroll {

    @Resource SessionContext ctx;

    public void updateEmployeeInfo(EmplInfo info) {
        oldInfo = ... read from database;

        // The salary field can be changed only by callers
        // who have the security role "payroll"
        if (info.salary != oldInfo.salary && !ctx.isCallerInRole("payroll")) {
            throw new SecurityException(...);
        }
        ...
    }
    ...
}
```

➢Given a security-related deployment descriptor tag or annotation, identify correct and incorrect statements and/or code related to that tag.

- [EJB_3.0_SIMPLIFIED] 10.9; 10.9.1; 10.9.2; 10.9.3; 10.9.4; 10.9.5

[EJB_3.0_CORE] 17.3.2.1 *; 17.3.2.2 *; 17.3.4; 17.3.4.1

Security and Method Permissions Metadata Annotations

The following security-related annotations are in the package javax.annotation.security.

- **Security Role References**

  The DeclareRoles annotation is used to declare the references to security roles in the enterprise bean code:

```
package javax.annotation.security;

@Target({TYPE}) @Retention(RUNTIME)
public @interface DeclareRoles {
    String[] value();
}
```

- **Method Permissions**
- The RolesAllowed annotation specifies the security roles that are allowed to invoke the methods of the bean. The value of the RolesAllowed annotation is a list of security role names.

- This annotation can be specified on the bean class and/or it can be specified on methods of the class that are methods of the business interface. Specifying the RolesAllowed annotation on the bean class means that it applies to ALL applicable interface methods of the class. Specifying the annotation on a method applies it to that method ONLY. If the annotation is applied at both the class and the method level, the method value OVERRIDES if the two disagree. If the PermitAll annotation is applied to the bean class, and RolesAllowed is specified on an individual method, the value of the RolesAllowed annotation OVERRIDES for the given method.

```
package javax.annotation.security;

@Target({TYPE, METHOD}) @Retention(RUNTIME)
public @interface RolesAllowed {
      String[] value();
}
```

- **PermitAll**

The PermitAll annotation specifies that all security roles are allowed to invoke the specified method(s) - i.e., that the specified method(s) are "unchecked". This annotation can be specified on the bean class and/or it can be specified on the business methods of the class. Specifying the PermitAll annotation on the bean class means that it applies to ALL applicable business methods of the class. Specifying the annotation on a method applies it to that method ONLY, overriding any class-level setting for the particular method.

```
package javax.annotation.security;

@Target ({TYPE, METHOD}) @Retention(RUNTIME)
public @interface PermitAll {}
```

- **DenyAll**

The DenyAll annotation specifies that no security roles are allowed to invoke the specified method - i.e. that the specified method is to be EXCLUDED from execution.

```
package javax.annotation.security;

@Target (METHOD) @Retention(RUNTIME)
public @interface DenyAll {}
```

- **RunAs**

The RunAs annotation is used to specify the bean's run-as property. This annotation is applied to the bean class. Its value is the name of a security role.

```
package javax.annotation.security;
```

```
@Target(TYPE) @Retention(RUNTIME)
public @interface RunAs {
    String value();
}
```

Specification of Method Permissions with Metadata Annotations

The following is the description of the rules for the specification of method permissions using Java language metadata annotations.

The method permissions for the methods of a bean class may be specified on the class, the business methods of the class, or both.

The RolesAllowed, PermitAll, and DenyAll annotations are used to specify method permissions. The value of the RolesAllowed annotation is a list of security role names to be mapped to the security roles that are permitted to execute the specified method(s). The PermitAll annotation specifies that all security roles are permitted to execute the specified method(s). The DenyAll annotation specifies that no security roles are permitted to execute the specified method(s).

Specifying the RolesAllowed or PermitAll annotation on the bean class means that it applies to all applicable business methods of the class.

Method permissions may be specified on a method of the bean class to OVERRIDE the method permissions value specified on the bean class.

If the bean class has superclasses, the following additional rules apply:

- A method permissions value specified on a superclass S applies to the business methods defined by S.
- A method permissions value may be specified on a business method M defined by class S to override for method M the method permissions value explicitly or implicitly specified on the class S.
- If a method M of class S overrides a business method defined by a superclass of S, the method permissions value of M is determined by the above rules as applied to class S.

Example:

```
@RolesAllowed("admin")
public class SomeClass {
    public void aMethod () {...}
    public void bMethod () {...}
    ...
}

@Stateless public class MyBean implements A extends SomeClass {
    @RolesAllowed("HR")
    public void aMethod () {...}

    public void cMethod () {...}
    ...
}
```

Assuming aMethod, bMethod, cMethod are methods of business interface A, the method permissions values of methods aMethod and bMethod are RolesAllowed("HR") and RolesAllowed("admin") respectively. The method permissions for method cMethod have not been specified.

Specification of Method Permissions in the Deployment Descriptor

The Bean Provider may use the deployment descriptor as an alternative to metadata annotations to specify the method permissions (or as a means to supplement or override metadata annotations for method permission values). The application assembler is permitted to override the method permission values using the bean's deployment descriptor.

Any values explicitly specified in the deployment descriptor override any values specified in annotations. If a value for a method has not be specified in the deployment descriptor, and a value has been specified for that method by means of the use of annotations, the value specified in annotations will apply. The granularity of overriding is on the per-method basis.

The Bean Provider or Application Assembler defines the method permissions relation in the deployment descriptor using the method-permission elements as follows:

- Each method-permission element includes a list of one or more security roles and a list of one or more methods. All the listed security roles are allowed to invoke all the listed methods. Each security role in the list is identified by the role-name element, and each method (or a set of methods, as described below) is identified by the method element. An optional description can be associated with a method-permission element using the description element.
- The method permissions relation is defined as the union of all the method permissions defined in the individual method-permission elements.
- A security role or a method may appear in multiple method-permission elements.

The Bean Provider or Application Assembler can indicate that all roles are permitted to execute one or more specified methods (i.e., the methods should not be "checked" for authorization prior to invocation by the container). The unchecked element is used instead of a role name in the method-permission element to indicate that all roles are permitted.

If the method permission relation specifies both the unchecked element for a given method and one or more security roles, all roles are permitted for the specified methods.

The exclude-list element can be used to indicate the set of methods that should NOT be called. The Deployer should configure the enterprise bean's security such that no access is permitted to any method contained in the exclude-list.

If a given method is specified both in the exclude-list element and in the method permission relation, the Deployer should configure the enterprise bean's security such that NO access is permitted to the method.

The method element uses the ejb-name, method-name, and method-params elements to denote one or more methods of an enterprise bean's business , home, and component interface, and/or web service endpoint. There are three legal styles for composing the method element:

```
<method>
      <ejb-name>EJBNAME</ejb-name>
      <method-name>*</method-name>
</method>
```

This style is used for referring to ALL of the methods of the business, home, and component interfaces, and web service endpoint of a specified enterprise bean.

```
<method>
      <ejb-name>EJBNAME</ejb-name>
      <method-name>METHOD</method-name>
</method>
```

This style is used for referring to a specified method of the business, home, or component interface, or web service endpoint of the specified enterprise bean. If there are multiple methods with the same overloaded name, this style refers to ALL of the overloaded methods.

```
<method>
      <ejb-name>EJBNAME</ejb-name>
      <method-name>METHOD</method-name>
      <method-params>
            <method-param>PARAMETER_1</method-param>
            ...
            <method-param>PARAMETER_N</method-param>
      </method-params>
</method>
```

This style is used to refer to a specified method within a set of methods with an overloaded name. The method must be defined in the specified enterprise bean's business, home, or component interface, or web service endpoint. If there are multiple methods with the same overloaded name, however, this style refers to ALL of the overloaded methods.

The optional method-intf element can be used to differentiate between methods with the same name and signature that are multiply defined across the business, component, or home interfaces, and/or web service endpoint. If the same method is a method of both the local business interface and local component interface, the same method permission values apply to the method for BOTH interfaces. Likewise, if the same method is a method of both the remote business interface and remote component interface, the same method permission values apply to the method for BOTH interfaces.

The following example illustrates how security roles are assigned method permissions in the deployment descriptor:

```
<method-permission>
```

```
        <role-name>employee</role-name>
        <method>
            <ejb-name>EmployeeService</ejb-name>
            <method-name>*</method-name>
        </method>
</method-permission>

<method-permission>
        <role-name>employee</role-name>
        <method>
            <ejb-name>AardvarkPayroll</ejb-name>
            <method-name>findByPrimaryKey</method-name>
        </method>
        <method>
            <ejb-name>AardvarkPayroll</ejb-name>
            <method-name>getEmployeeInfo</method-name>
        </method>
        <method>
            <ejb-name>AardvarkPayroll</ejb-name>
            <method-name>updateEmployeeInfo</method-name>
        </method>
</method-permission>

<method-permission>
        <role-name>payroll-department</role-name>
        <method>
            <ejb-name>AardvarkPayroll</ejb-name>
            <method-name>findByPrimaryKey</method-name>
        </method>
        <method>
            <ejb-name>AardvarkPayroll</ejb-name>
            <method-name>getEmployeeInfo</method-name>
        </method>
        <method>
            <ejb-name>AardvarkPayroll</ejb-name>
            <method-name>updateEmployeeInfo</method-name>
        </method>
        <method>
            <ejb-name>AardvarkPayroll</ejb-name>
            <method-name>updateSalary</method-name>
        </method>
</method-permission>

<method-permission>
        <role-name>admin</role-name>
        <method>
            <ejb-name>EmployeeServiceAdmin</ejb-name>
            <method-name>*</method-name>
        </method>
</method-permission>
```

Specification of Security Identities in the Deployment Descriptor

The Bean Provider or Application Assembler typically specifies whether the caller's security identity should be used for the execution of the methods of an enterprise bean OR whether a specific run-as identity should be used.

By default the caller's security identity is used. The Bean Provider can use the RunAs metadata annotation to specify a run-as identity for the execution of the bean's methods. If the deployment descriptor is used, the Bean Provider or the Application Assembler can use the security-identity deployment descriptor element for this purpose OR to override a security identity specified in metadata. The value of the security-identity element is either use-caller-identity or run-as.

Defining the security identities in the deployment descriptor is optional for the Application Assembler. Their omission in the deployment descriptor means that the Application Assembler chose not to pass any instructions related to security identities to the Deployer in the deployment descriptor.

If a run-as security identity is not specified by the Deployer, the container should use the caller's security identity for the execution of the bean's methods.

Run-as

The Bean Provider can use the RunAs metadata annotation or the Bean Provider or Application Assembler can use the run-as deployment descriptor element to define a run-as identity for an enterprise bean in the deployment descriptor. The run-as identity applies to the enterprise bean as a WHOLE, that is, to ALL methods of the enterprise bean's business, home, and component interfaces, and/or web service endpoint; to the message listener methods of a message-driven bean; and to the timeout callback method of an enterprise bean; and all internal methods of the bean that they might in turn call.

Establishing a run-as identity for an enterprise bean DOES NOT affect the identities of its callers, which are the identities tested for permission to access the methods of the enterprise bean. The run-as identity establishes the identity the enterprise bean will use when it makes calls.

Because the Bean Provider and Application Assembler do not, in general, know the security environment of the operational environment, the run-as identity is designated by a *logical* role-name, which corresponds to one of the security roles defined by the Bean Provider or Application Assembler in the metadata annotations or deployment descriptor.

The Deployer then ASSIGNS a security principal defined in the operational environment to be used as the principal for the run-as identity. The security principal assigned by the Deployer should be a principal that has been assigned to the security role specified by RunAs annotation or by the role-name element of the run-as deployment descriptor element.

The Bean Provider and/or Application Assembler is responsible for the following in the specification of run-as identities:

- Use the RunAs metadata annotation or role-name element of the run-as deployment descriptor element to define the name of the security role.
- Optionally, use the description element to provide a description of the principal that is

expected to be bound to the run-as identity in terms of its security role.

The following example illustrates the definition of a run-as identity using metadata annotations:

```
@RunAs("admin")
@Stateless public class EmployeeServiceBean implements EmployeeService {
	...
}
```

Using the deployment descriptor, this can be specified as follows:

```
<enterprise-beans>
	...
	<session>
		<ejb-name>EmployeeService</ejb-name>
		...
		<security-identity>
			<run-as>
				<role-name>admin</role-name>
			</run-as>
		</security-identity>
		...
	</session>
	...
</enterprise-beans>
```

1Appendix A. Enterprise Application Development Environment - Installing

1Overview

This section investigates the viability of doing EJB 3.0 development using software that is available for FREE.

In order to investigate the options, very popular packages are chosen, and used to develop a simple EJB 3.0 application.

Software Components

Developing enterprise applications requires software, below is a list of the software used:

- An operating system: Windows XP SP2 (Microsoft Windows XP [Version 5.1.2600])
- Java Virtual Machine (JVM): Sun Microsystems JSE 5.0 version (Java(TM) 2 Runtime Environment, Standard Edition (build 1.5.0_09-b01))
- An application server (AS): JBoss version 4.0.5 GA
- Integrated development environment (IDE):

- Eclipse SDK version 3.2.1 (eclipse-SDK-3.2.1-win32)
- JBoss IDE version 2.0.0 Beta2

  NOTE: For IDE installing you may use either separate Eclipse and JBoss IDE software packages (this approach is described below), or Eclipse + JBoss IDE *bundled package* which is available for download from JBoss site.

1Installing Application Server

JBoss needs an environment variable called "JAVA_HOME" to be set to the root installation of the JDK. This must be set before JBoss is run. The previous section explains how this is achieved. The JBoss installation used to test EJB 3.0 requires that the JDK be version **5.0** (or higher).

1. JBoss AS can be downloaded as a JAR file from http://labs.jboss.com/portal/jbossas/download.

   NOTE: Since you want to use EJB 3.0, you **MUST** download the installer file.

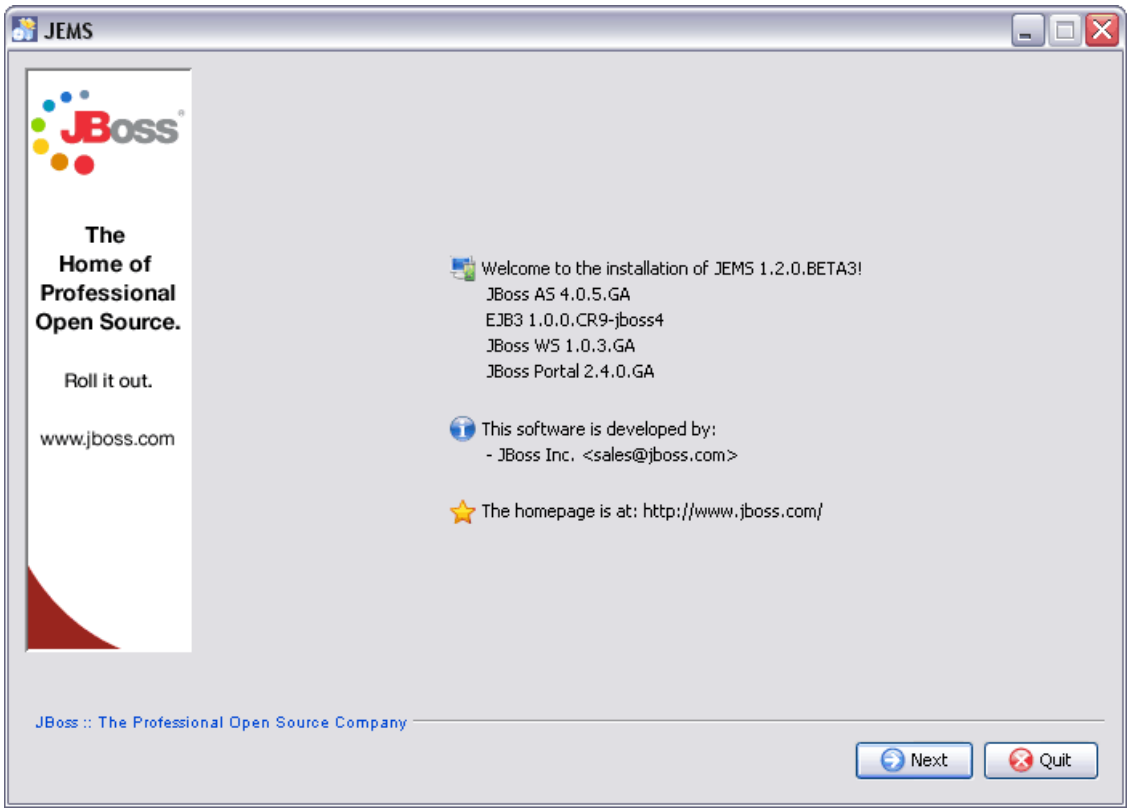
   NOTE: The latest avaiable version of JBoss standalone *installer* (which includes EJB 3.0) is 4.0.4, while the latest version of JBoss AS is **4.0.5**.

   You need to use the JEMS installer (http://labs.jboss.com/portal/jemsinstaller/downloads) to install the ejb3 profile in order to run EJB 3.0 applications on JBoss AS version **4.0.5**.

   The reason is that JBoss Inc is not legally allowed by Sun to bundle EJB 3.0 runtime in a certified J2EE 1.4 server (i.e., the JBoss AS 4.x).

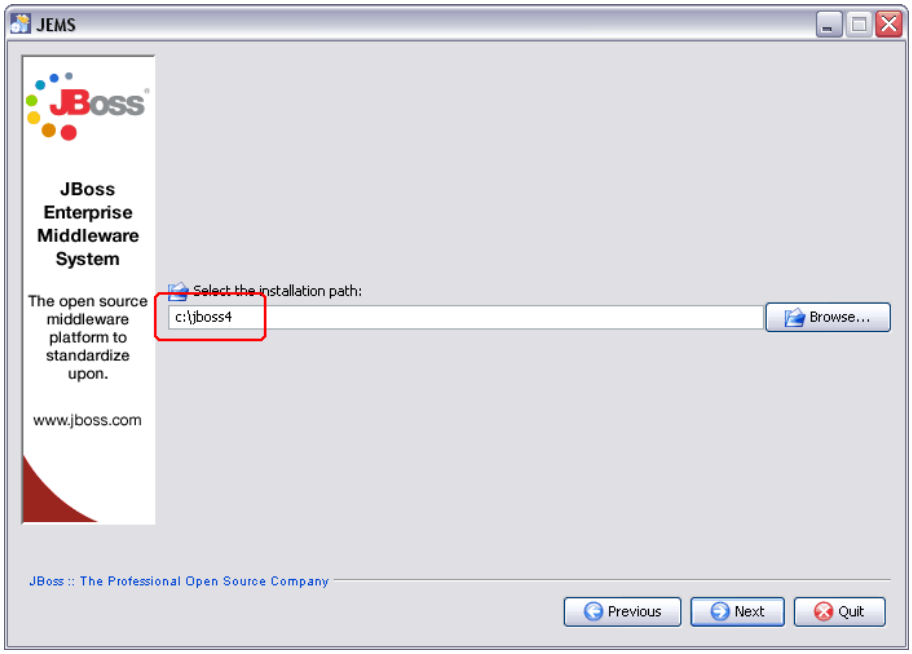
   NOTE: JBoss AS version 4.0.4 has problem with missing DeclareRoles security annotation. For this reason, the recommended JBoss AS version to use is **4.0.5**.

2. The archive is an executable Java ARchive, when run it launches the JBoss installer Java Application. Depending on how Java has been installed it is possible to launch the installer by double clicking the JAR file. Otherwise a command line window is opened in the directory where the JBoss JAR file was downloaded, and Java is launched with the JBoss installer file as a parameter:
3. C:\temp> java -jar jems-installer-1.2.0.BETA3.jar



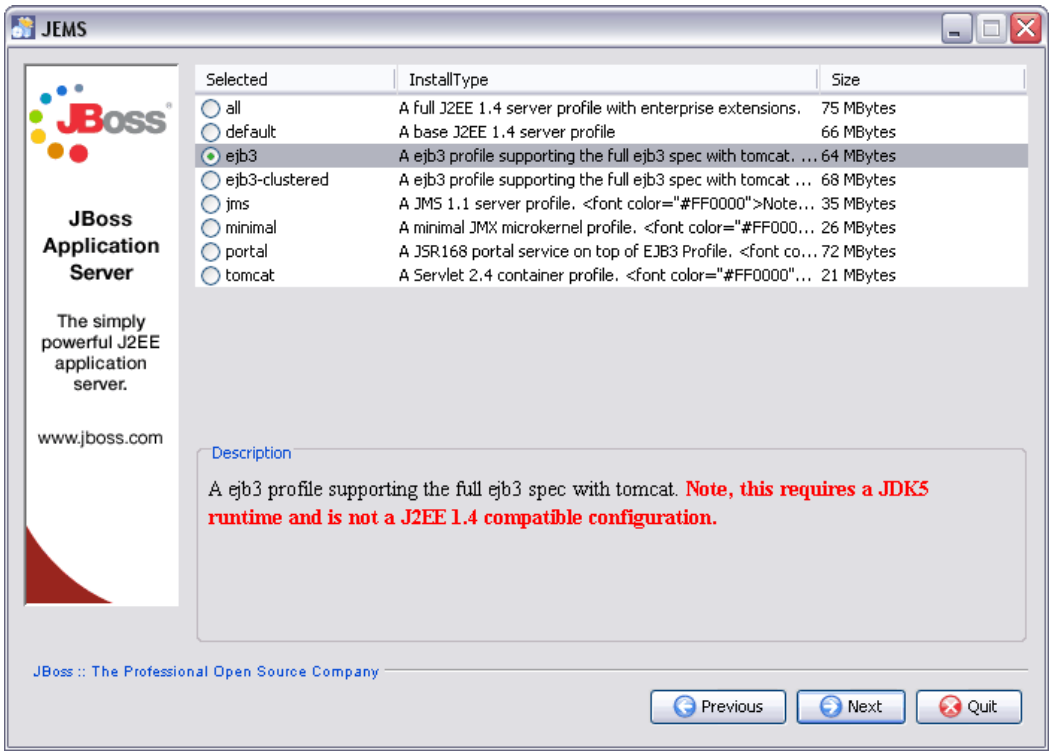
Click the **Next** button.

4. The installation folder path of JBoss should NOT contain spaces. On Windows Systems, the path "C:\Program Files\jboss-4.0.5.GA" is not advisable. Enter the "c:\jboss4" installation path:



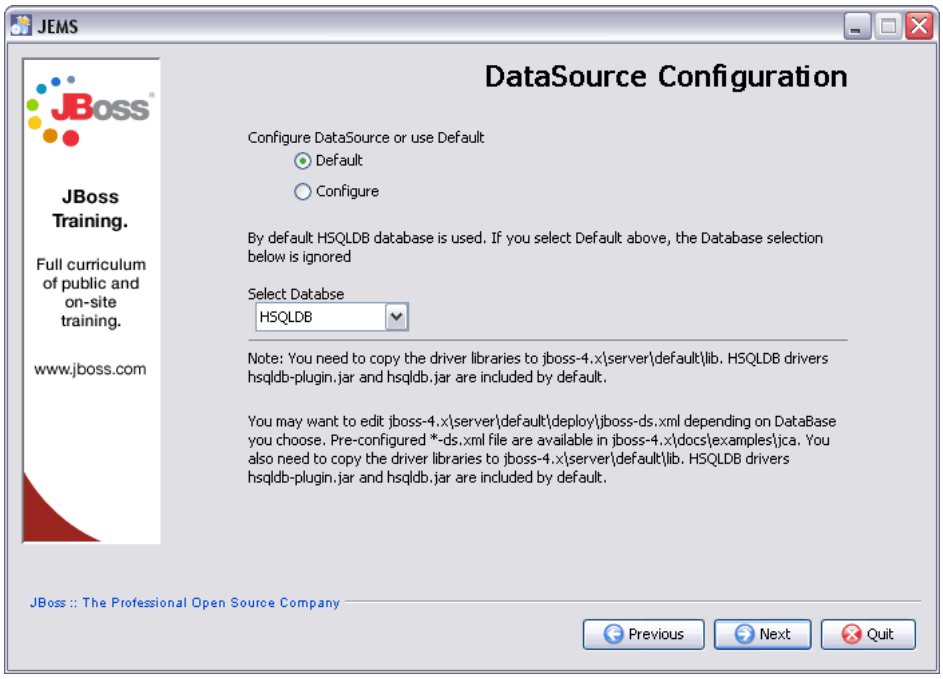
5. It is important to have EJB 3.0 support installed. At the time of writing EJB 3.0 is still very new, and so MUST be explicitly selected instead of the "all" install type server option.

   Choose the **ejb3** profile when asked:



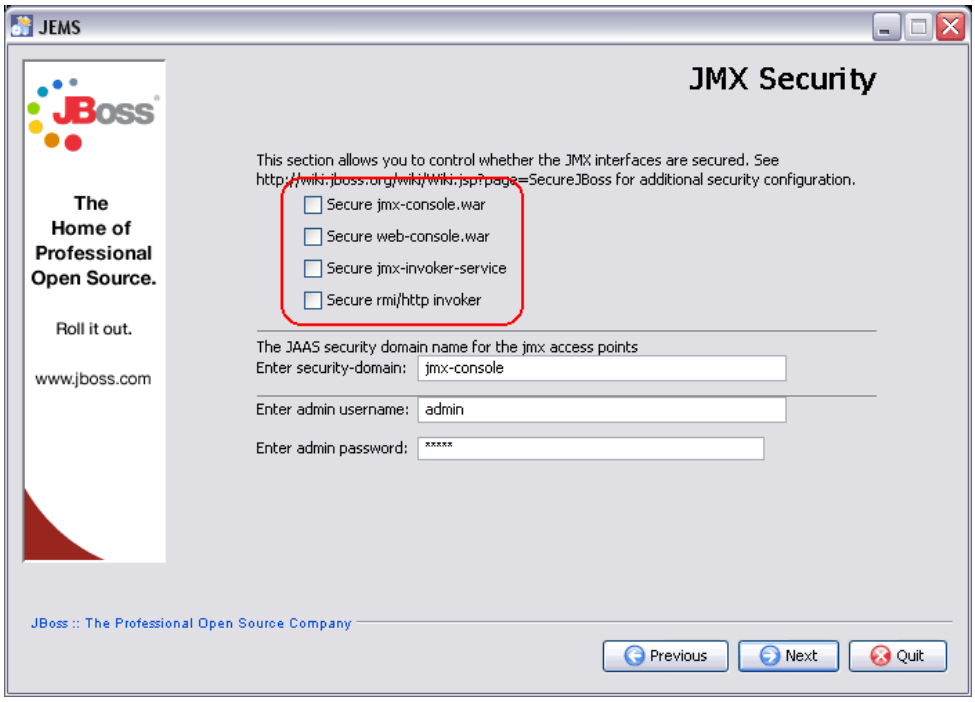
   Click the **Next** button.

6. Configure DataSource to use default database (HSQLDB)



Click the **Next** button.

7. In the **Configure JMX Security** dialog enter the following:
    - Uncheck ALL checkboxes.
    - **Enter admin username** : admin
    - **Enter admin password** : admin



Click the **Next** button.

8. When the install is done, click the **Next** button and the **Done** button.

Running JBoss For The First Time

JBoss should be started, just to make sure the JDK and JBoss are installed correctly, and to allow JBoss to make any final installation steps it requires. This is done by entering "run.bat" at the command prompt, in the JBoss bin folder.

```
C:\jboss4\bin> run.bat
=========================================================================
========

  JBoss Bootstrap Environment

  JBOSS_HOME: C:\jboss4\bin\\..

  JAVA: C:\Program Files\Java\jdk1.5.0_09\bin\java

  JAVA_OPTS:  -Dprogram.name=run.bat -server -Xms128m -Xmx512m -Dsun.rmi.dgc.client.gcInterval=3600000 -
Dsun.rmi.dgc.server.gcInterval=3600000

  CLASSPATH: C:\Program Files\Java\jdk1.5.0_09\lib\tools.jar;C:\jboss4\bin\\run.jar

=========================================================================
========

13:15:29,623 INFO  [Server] Starting JBoss (MX MicroKernel)...
13:15:29,623 INFO  [Server] Release ID: JBoss [Zion] 4.0.5.GA (build: CVSTag=Branch_4_0 date=200610162339)
```

```
13:15:29,639 INFO  [Server] Home Dir: C:\jboss4
13:15:29,639 INFO  [Server] Home URL: file:/C:/jboss4/

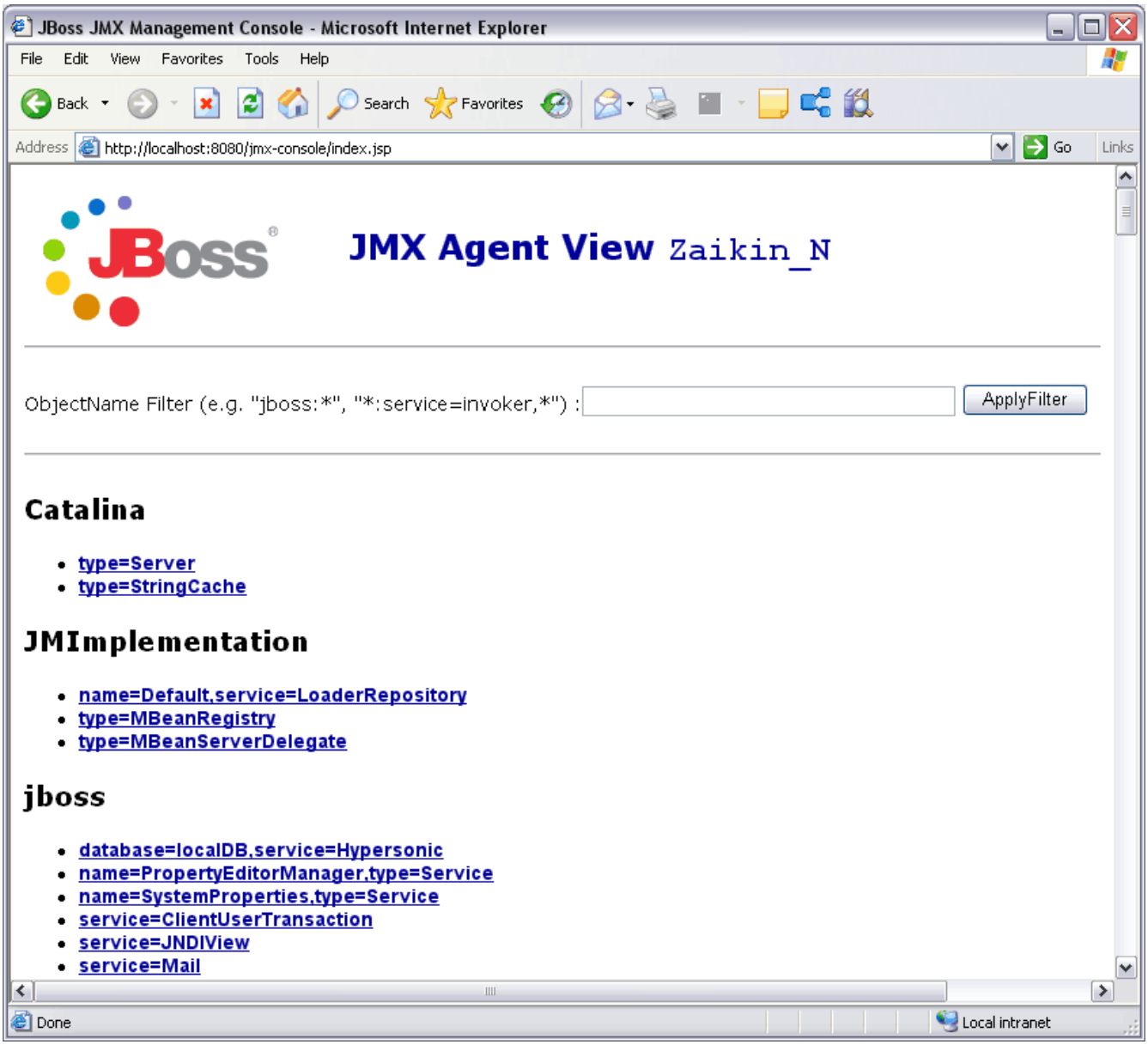
[skipped]

13:16:10,013 INFO  [TomcatDeployer] deploy, ctxPath=/jmx-console, warUrl=.../deploy/jmx-console.war/
13:16:10,779 INFO  [Http11BaseProtocol] Starting Coyote HTTP/1.1 on http-0.0.0.0-8080
13:16:11,044 INFO  [ChannelSocket] JK: ajp13 listening on /0.0.0.0:8009
13:16:11,076 INFO  [JkMain] Jk running ID=0 time=0/94  config=null
13:16:11,107 INFO  [Server] JBoss (MX MicroKernel) [4.0.5.GA (build: CVSTag=Branch_4_0 date=200610162339)]
Started in 41s:468ms
```

If the console window displayed some Java Exception stack trace(s), they need to be resolved.

To test that the server started correctly, a web browser is opened, and the following URL entered: http://localhost:8080/jmx-console/index.jsp, where localhost can be the name of the server on which JBoss was installed.

A successful installation displays the "**JMX Agent View**" page:



To stop the server, either **Ctrl + C** is pressed in the command prompt used to launch JBoss, or, a new command prompt is opened, and "shutdown.bat -S" is typed. In the JBoss command window, the output shows that JBoss AS has shut down:

```
C:\jboss4\bin> shutdown.bat -S
Shutdown message has been posted to the server.
Server shutdown may take a while - check logfiles for completion

13:17:40,152 INFO  [Http11BaseProtocol] Stopping Coyote HTTP/1.1 on http-0.0.0.0-8080
13:17:40,168 INFO  [TomcatDeployer] undeploy, ctxPath=/, warUrl=.../deploy/jbossweb-tomcat55.sar/ROOT.war/
13:17:40,168 INFO  [TomcatDeployer] undeploy, ctxPath=/jbossws, warUrl=.../tmp/deploy/tmp22708jbossws-
context-exp.war/
13:17:40,168 INFO  [TomcatDeployer] undeploy, ctxPath=/web-console, warUrl=.../deploy/management/console-
mgr.sar/web-console.war/
```

13:17:41,058 INFO [Server] Shutdown complete
Shutdown complete
Halting VM

1Installing IDE

Installing Eclipse

Eclipse SDK, the Java programmer's IDE used in this *SCBCD 5.0 Study Guide*, can be downloaded from Eclipse downloads home.

The latest release (3.2.1) MUST be downloaded. It is possible to download other, less stable development (also called *Milestone*) releases. As a rule, if there is no known reason not to download the release version, it is safer to go for the release version.

1. Eclipse SDK (**Release Build: 3.2.1**) is downloaded as a ZIP file from any mirror.

   NOTE: You should select eclipse-SDK-3.2.1-win32.zip file to download.

   NOTE: Previous Eclipse SDK Releases (3.2, 3.1) will NOT work with JBoss IDE 2.0.0 well.

   Save the file to some temporary folder (C:\temp).

2. The IDE is installed on the development machine by extracting the contents of the downloaded archive (the JDK 5.0 should already be installed and ready to run).

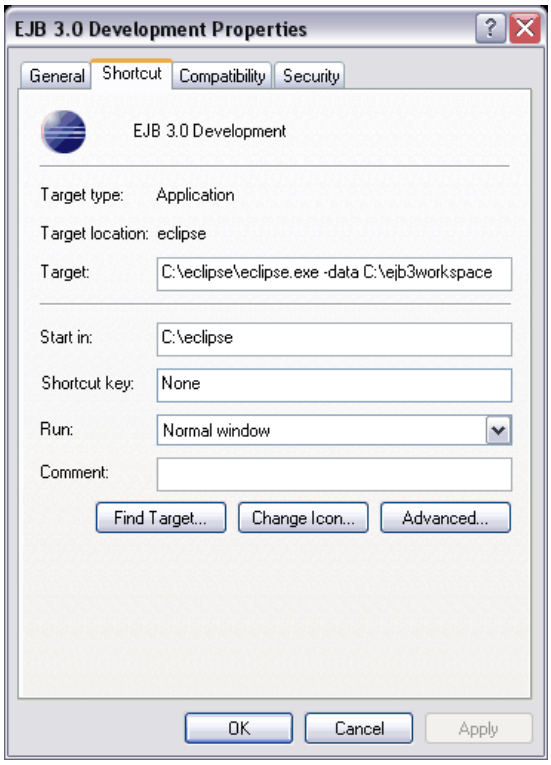
   Use jar utility (WinZip or any other unzip program) to unzip the contents of the file to C:\temp:

   C:\temp> jar xvf eclipse-SDK-3.2-win32.zip

   This will create "C:\temp\eclipse" folder.

3. Then move Eclipse folder to C:\
4. C:\temp> move eclipse c:\

5. Create a shortcut (or command file) named "**EJB 3.0 Development**" to run Eclipse SDK 3.2.1 with workspace on disk C:\ using -data command line argument and folder name C:\ejb3workspace

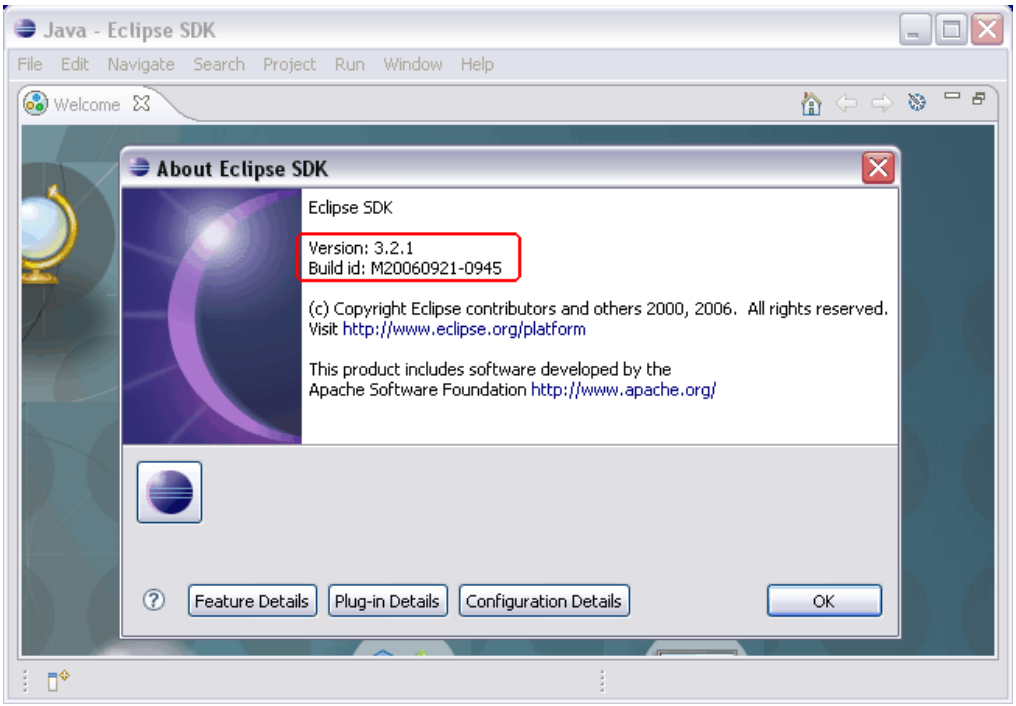


You may also create command file with the following contents:

```
C:\eclipse\eclipse.exe -data C:\ejb3workspace
```

6. Start Eclipse SDK 3.2.1 with "C:\ejb3workspace" workspace using new shortcut (or command file).

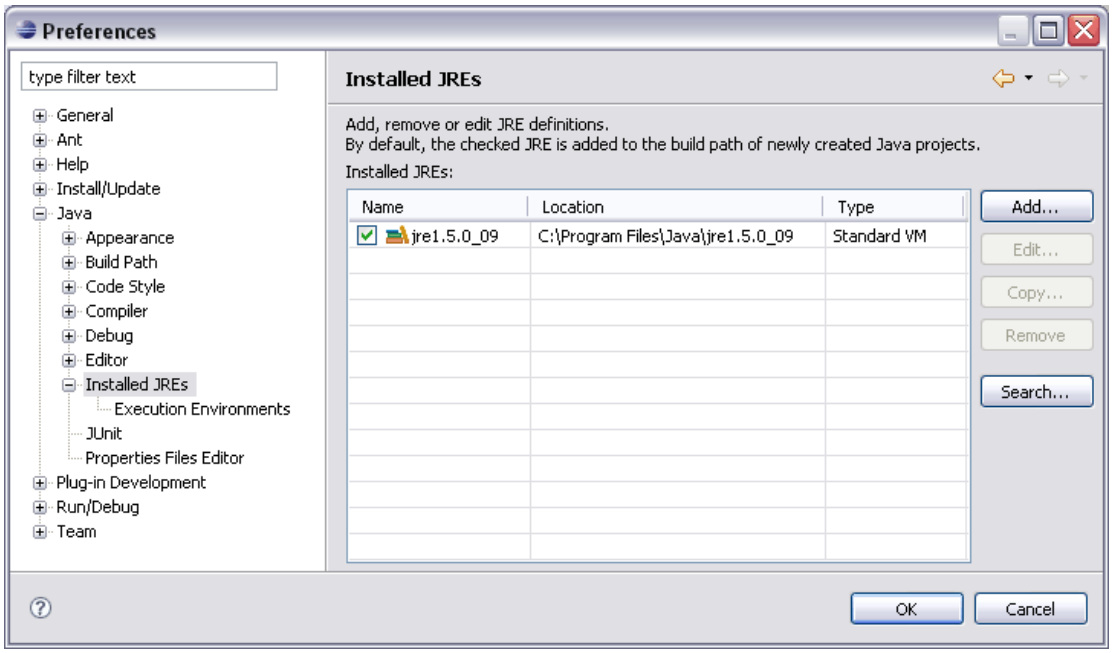
   Open **Help > About Eclipse SDK** menu to check the version:



7. Make sure that Eclipse is configured to use the correct JRE 5.0.

   Open menu **Window > Preferences... > Java > Installed JREs**

   The menu item opens a dialog to configure Eclipse:



   Additional Java Runtime Environments can be added and selected.

   Click the **OK** button, then exit the Eclipse.

Installing JBoss IDE Plugin

Eclipse is written to allow third parties to write a "*plug-in*". A plug-in can be installed to add functionality to Eclipse. Some plug-ins are available from the Eclipse site, another good source is http://www.eclipseplugincentral.com, this last site lists many plug-ins for Eclipse. Some are commercial, some are free.

Plug-ins are extremely popular, it is well worth spending some time to get acquainted with what is on offer. Plug-ins are essentially third party "Wizards" that are used to simplify a particular programming or configuration task. They save time, and reduce the number of bugs in an application.

In this *SCBCD 5.0 Study Guide* we use JBoss Application Server. The JBoss website lists a plug-in for Eclipse called **JBoss Eclipse IDE**. This plug-in facilitates the creation and debugging of Java applications which run inside JBoss. The plug-in is installed as follows:

NOTE: You MUST close Eclipse (if it is running) prior to plug-in installing.

1. Download package **JBoss IDE for Eclipse** version 2.0.0 Beta2 from JBoss web-site http://labs.jboss.com/portal/jbosside/download/index.html.

   Save the JBossIDE-2.0.0.Beta2-ALL.zip file to some temporary folder (C:\temp).

2. The plug-in is installed on the development machine by extracting the contents of the downloaded plug-in archive into Eclipse folder.

   Use jar utility (WinZip or any other unzip program) to unzip the contents of the file to C:\temp:

   ```
   C:\temp> jar -xvf JBossIDE-2.0.0.Beta2-ALL.zip
   ```

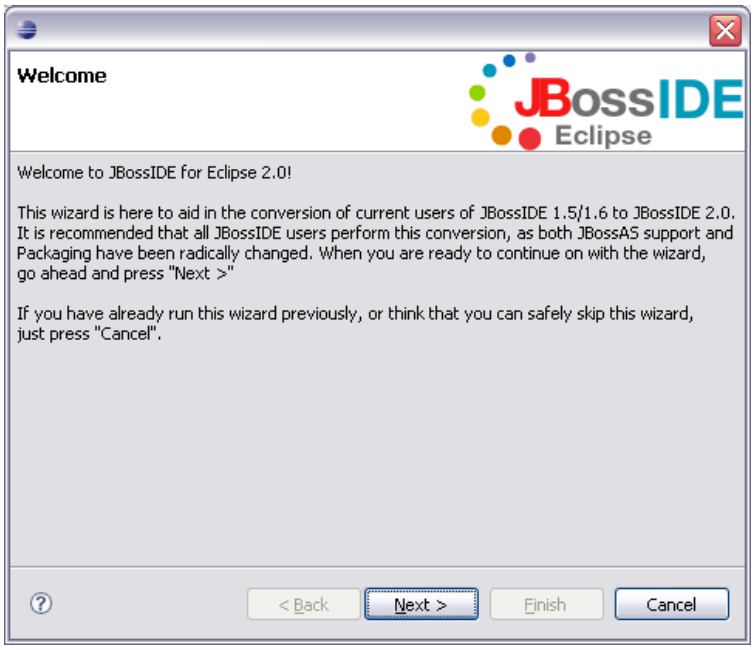
   This will create "C:\temp\eclipse" folder with 2 plug-in subfolders: "features" and "plugins".

3. Copy features plug-in folder to C:\eclipse\features
4. ```
   C:\temp> xcopy eclipse\features c:\eclipse\features /s /e /y
   ```

5. Copy plugins plug-in folder to C:\eclipse\plugins
6. ```
   C:\temp> xcopy eclipse\plugins c:\eclipse\plugins /s /e
   ```

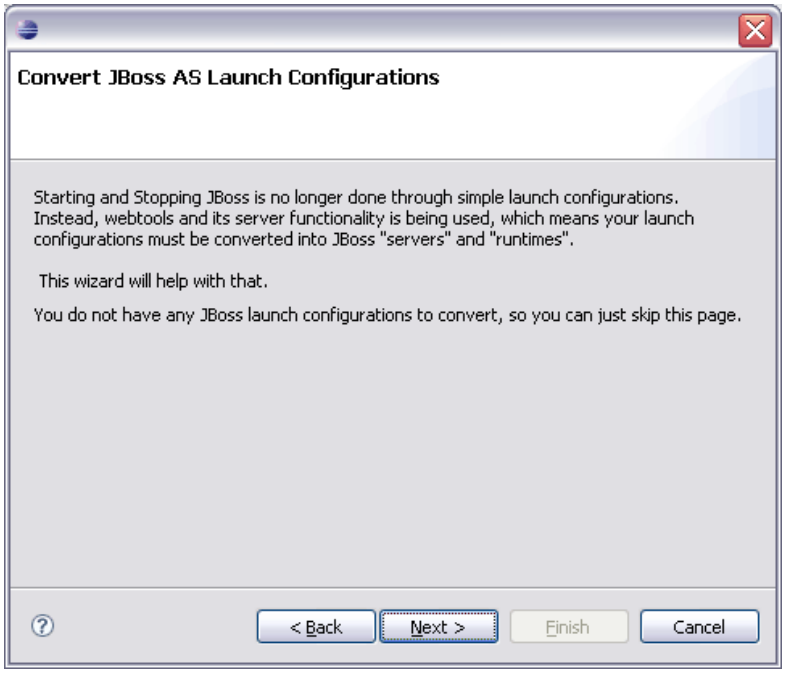
7. Optionally: cleanup temporary directory
8. ```
   C:\temp> rmdir eclipse /s /q
   ```

9. Start Eclipse SDK using shortcut or command file.
10. The **Welcome JBoss IDE** screen appears:
   ```



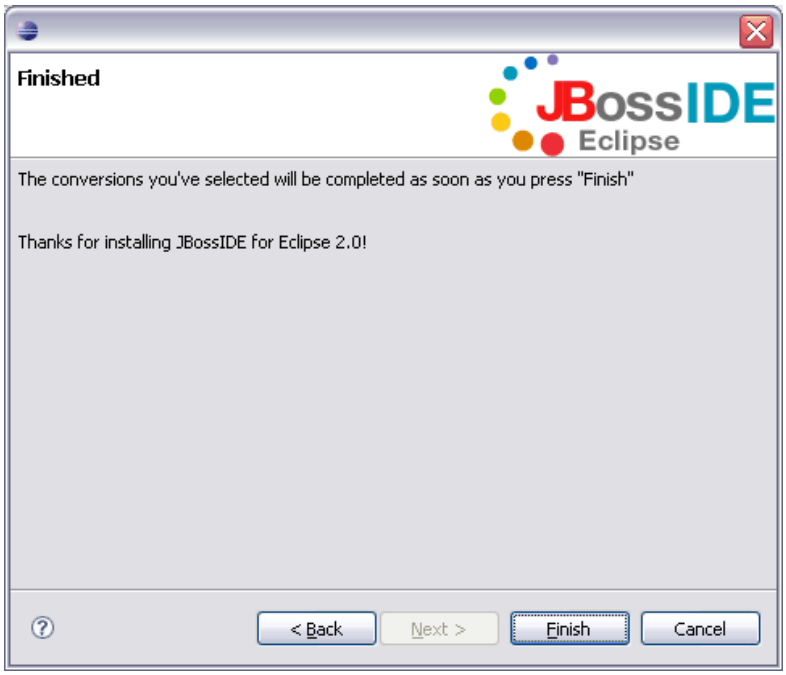
Click the **Next** button.

11. The **Convert JBoss AS Launch Configurations** screen appears:



Click the **Next** button.

12. The **Finished conversion** screen appears:



Click the **Finish** button.

Congratulations ! You have installed the EJB 3.0 Development Environment !!!

1Appendix B. Enterprise Application Development Environment - Testing - Part 1 (Stateless Session Bean)

1Overview

Installing and configuring JDK 5.0, JBoss 4.0.5, and Eclipse SDK 3.2.1 (including plug-in) required many steps.

During the installation, each individual component was installed and run. So it is known that each component runs OK. In this section, the components are tested together. In this way installation problems can be found and rectified.

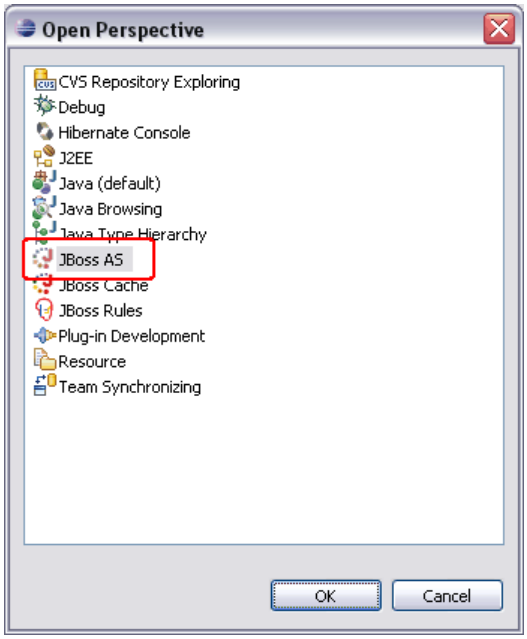
The easiest way to test the installation, is to write a small project which uses all the components that have been installed.

A bonus of this step is that each component is used, and therefore some idea of how these components are used and configured begins to be learnt.

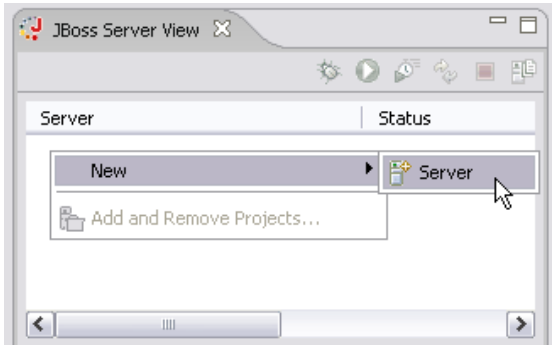
NOTE: The source code for installation tests can be downloaded as a zip archive.

1Creating The Server

1. Start the Eclipse IDE.
2. Select **Window > Open Perspective > Other...** from the Eclipse menu.
3. Select **JBoss AS** perspective and click the **OK** button:

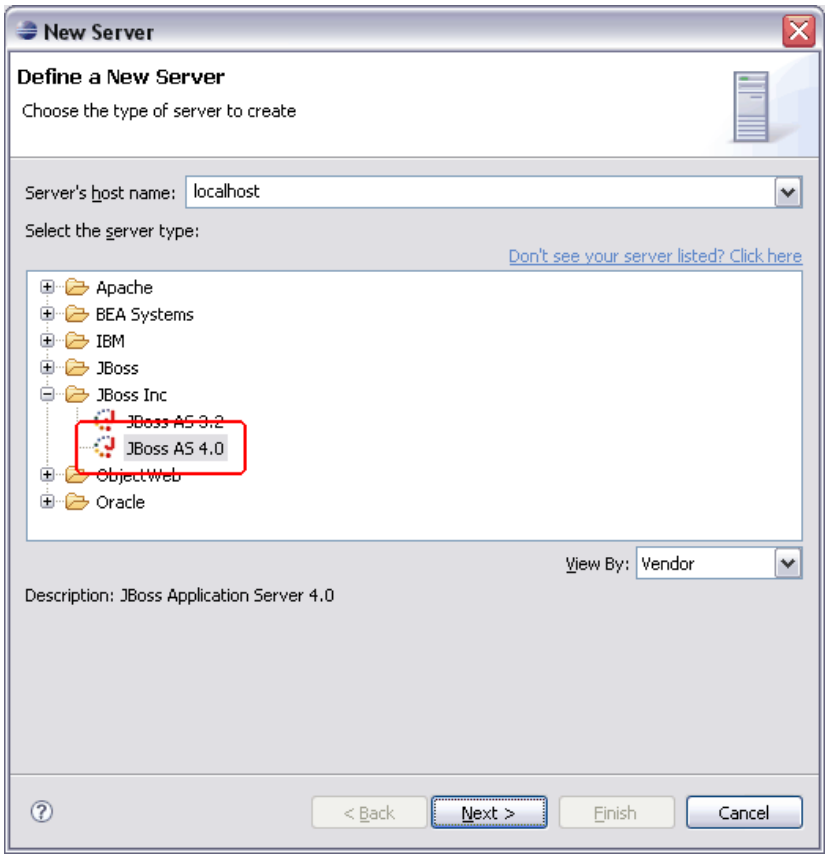


4. In the **JBoss Server View**, right click and select **New > Server** context menu:



5. In the **New Server** dialog, select **JBoss Inc > JBoss AS 4.0** server type and click the **Next** button.

   NOTE: Do NOT select **JBoss > JBoss v4.0** server type.

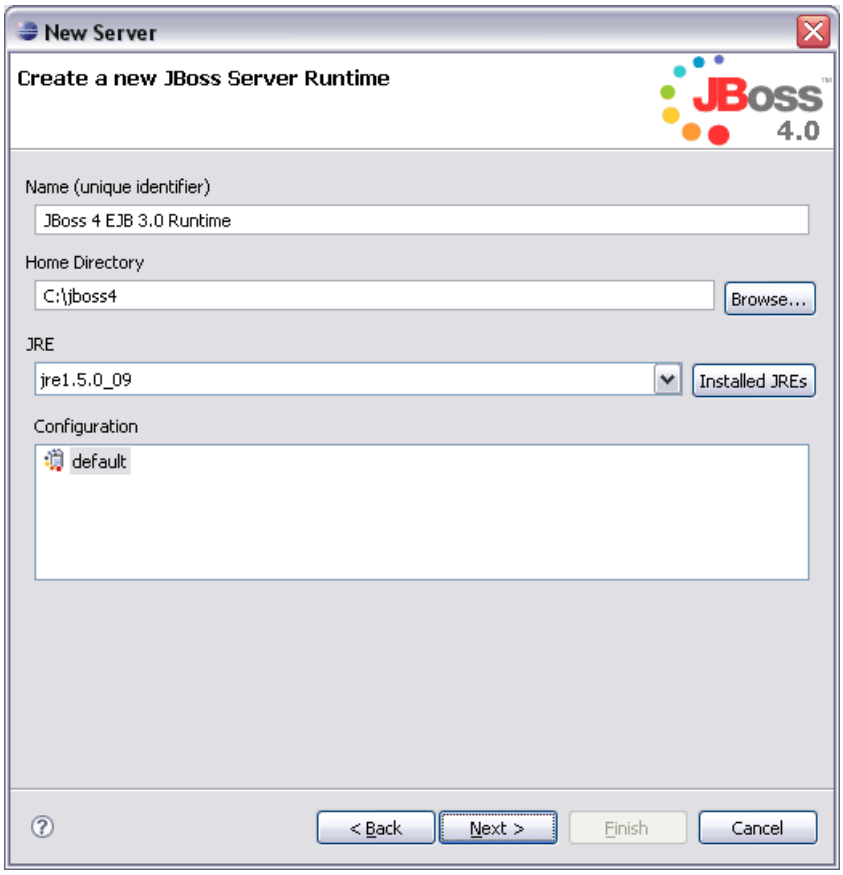


6. In the **Create a new JBoss Server Runtime** dialog, enter:

   Name (unique identifier): JBoss 4 EJB 3.0 Runtime

   Home Directory: C:\jboss4

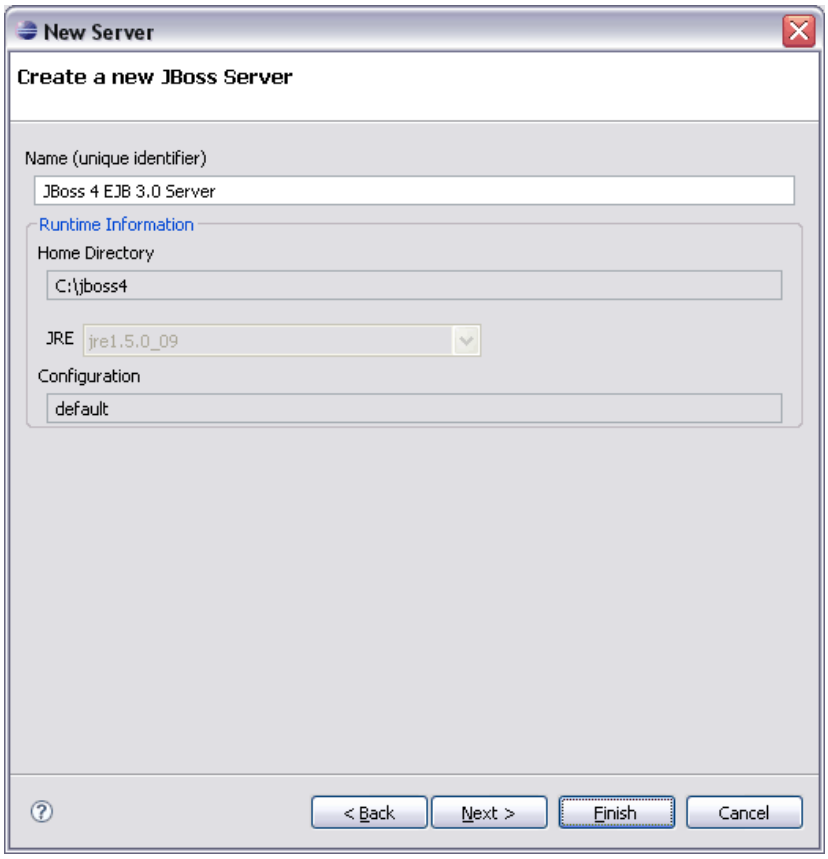
   Click the **Next** button.



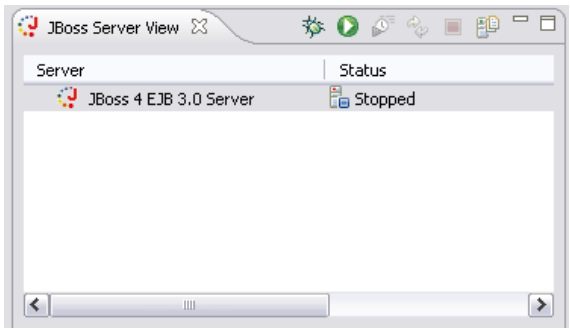
7. In the **Create a new JBoss Server** dialog, enter:

   Name (unique identifier): JBoss 4 EJB 3.0 Server

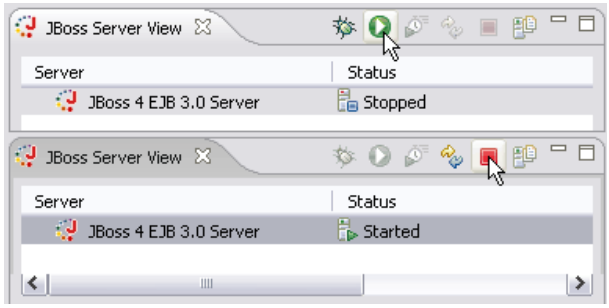
   Click the **Finish** button.



8. New server will be created:

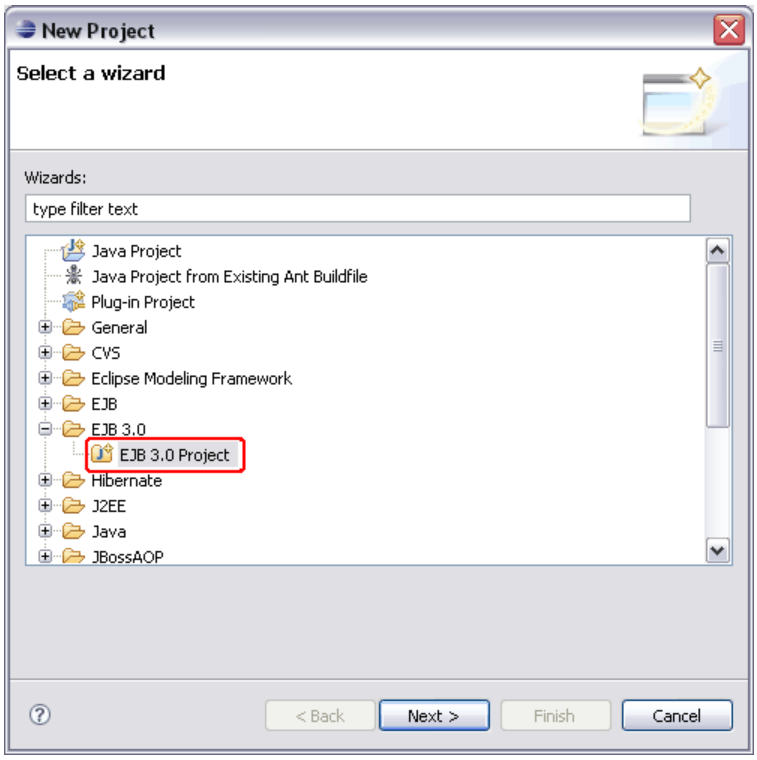


9. OPTIONALLY: Try to start and stop new server using toolbar buttons:

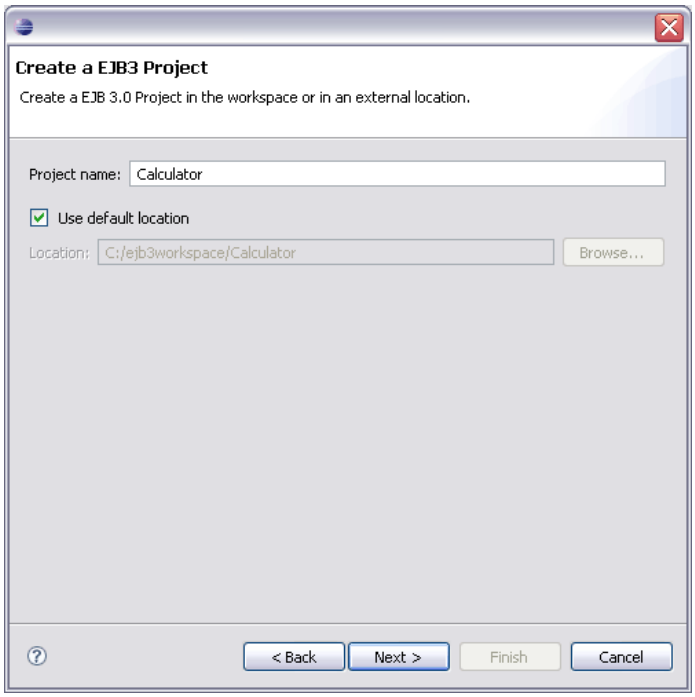


1Creating The Project

1. Select **File > New > Project...** from the Eclipse menu.
2. Browse to **EJB 3.0 > EJB 3.0 Project** and click the **Next** button.

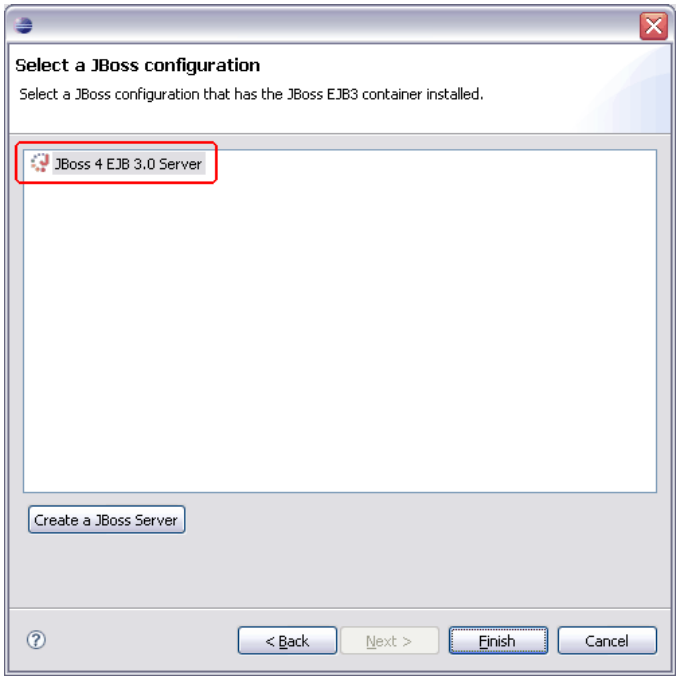


3. Set the project name to Calculator and click the **Next** button.

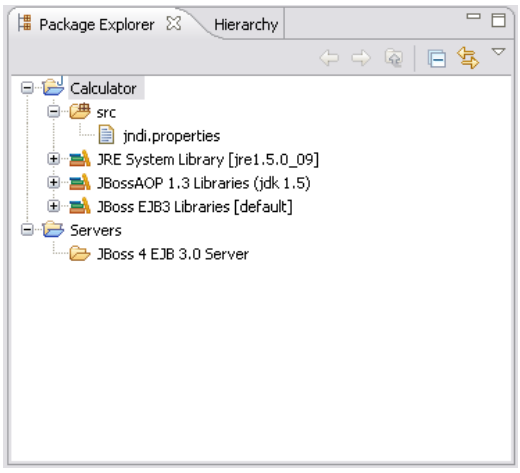


4. In the **Select JBoss configuration** dialog, select the JBoss 4 EJB 3.0 Server configuration which was created in the previous section.

   Click the **Finish** button:

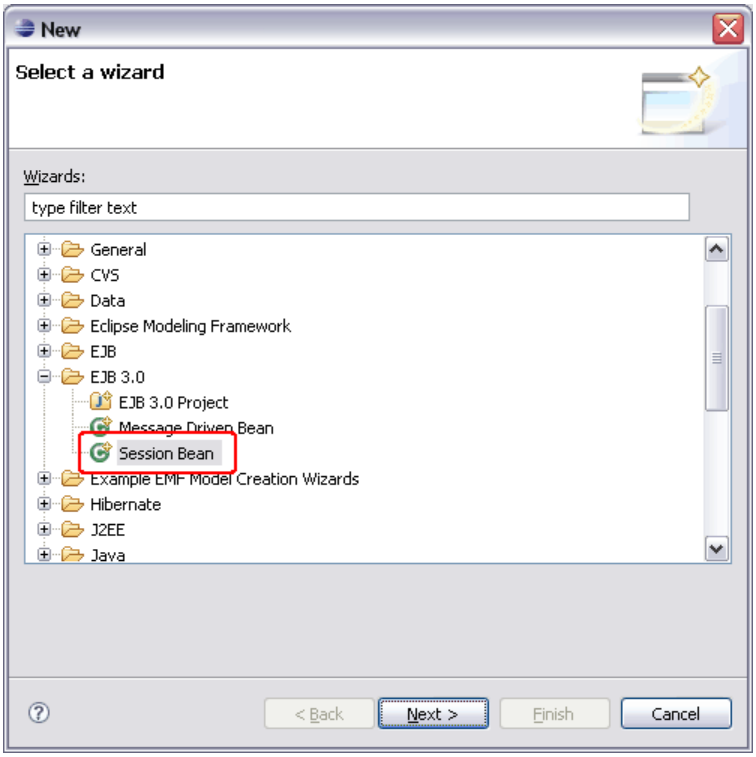


5. New Calculator EJB 3.0 project was created:



1Creating A Stateless EJB

1. Select **File > New > Other...** from the Eclipse menu.
2. Browse to **EJB 3.0 > Session Bean** and click the **Next** button.



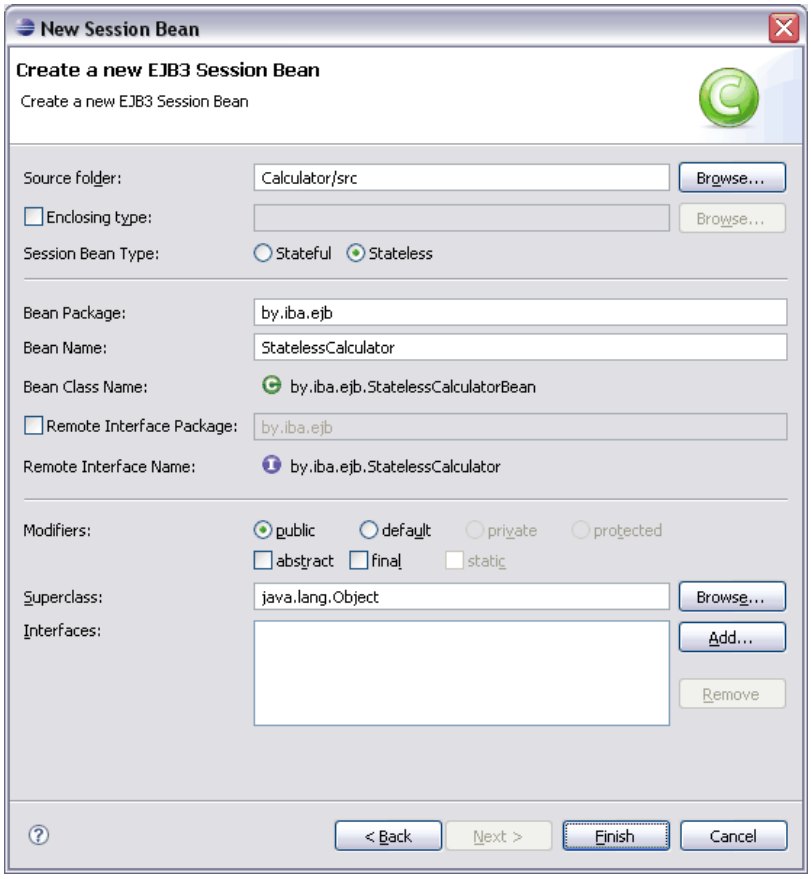
3. In the **Create a new EJB 3 Session Bean** dialog, enter the following:

   The **Bean Package** field: by.iba.ejb

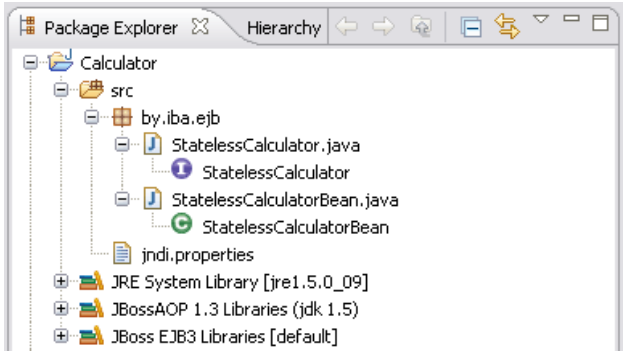
The **Bean Name** field: StatelessCalculator

NOTE: make sure Stateless **Session Bean Type** radio button is selected.

Click the **Finish** button.



4. The wizard has created two files: the Bean Implementation Class (StatelessCalculatorBean.java) and the Bean Remote Interface (StatelessCalculator.java):



StatelessCalculatorBean class:

```
package by.iba.ejb;

import javax.ejb.Stateless;
```

```
import by.iba.ejb.StatelessCalculator;

public @Stateless class StatelessCalculatorBean implements StatelessCalculator {

}
```

StatelessCalculator interface:

```
package by.iba.ejb;

import javax.ejb.Remote;

@Remote
public interface StatelessCalculator {

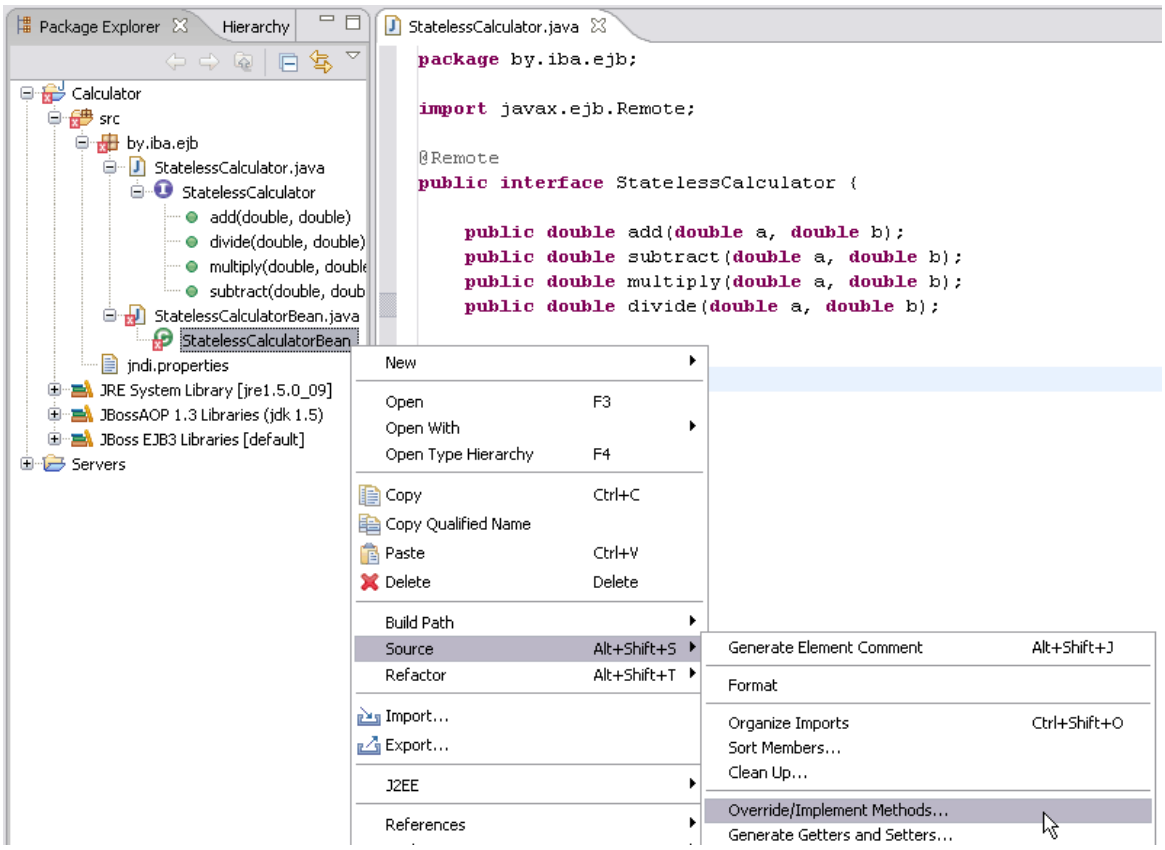
}
```

1Defining A Stateless EJB

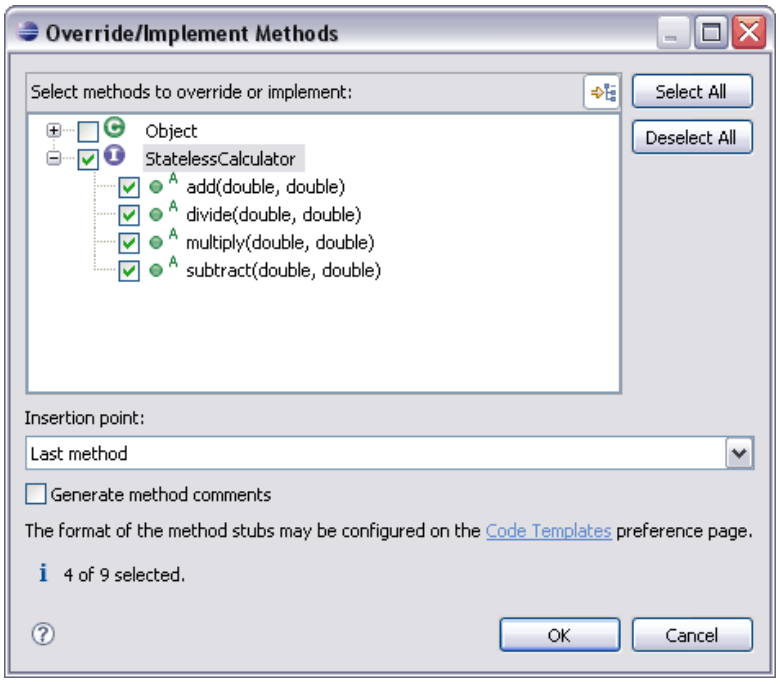
1. Add business method signatures to the StatelessCalculator interface:

```
2.    public double add(double a, double b);
3.    public double subtract(double a, double b);
4.    public double multiply(double a, double b);
5.    public double divide(double a, double b);
```

6. In the **Package Explorer** View, right click on the StatelessCalculatorBean class and select **Source > Override/Implement Methods...**.



7. Accept the default selection and press the **OK** button:



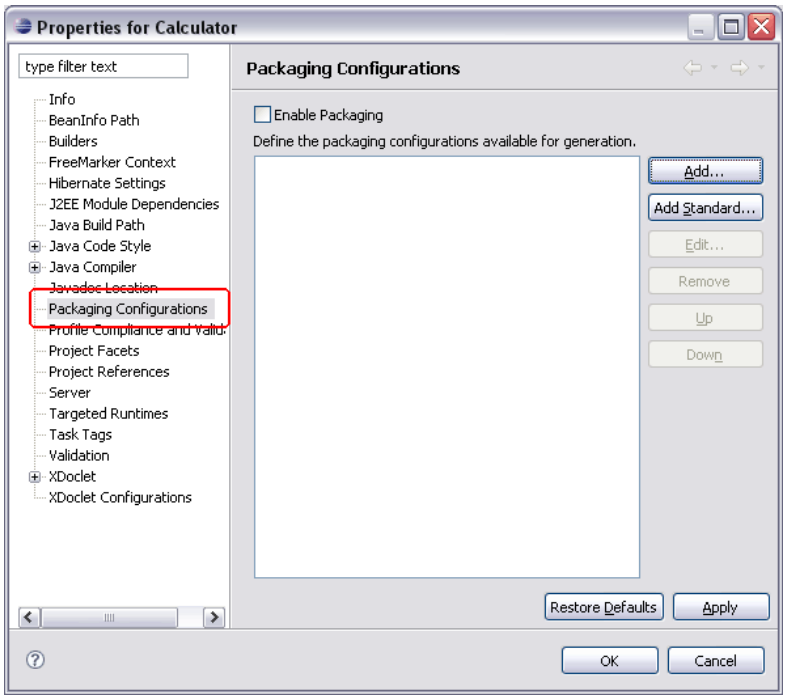
8. Open the StatelessCalculatorBean.java file (if not already open) and modify the newly created methods as shown below:

```
9.
10.   public double add(double a, double b) {
11.           return a + b;
12.   }
13.
14.   public double subtract(double a, double b) {
15.           return a - b;
16.   }
17.
18.   public double multiply(double a, double b) {
19.           return a * b;
20.   }
21.
22.   public double divide(double a, double b) {
23.           if (b == 0.0) {
24.                   throw new javax.ejb.EJBException("Divide by zero error !");
25.           }
26.           return a / b;
27.   }
```

28. The stateless EJB is complete. WIth EJB 3, there is no need to code deployment descriptors or the home interface. The next step is to package the EJB in the EJB JAR file.

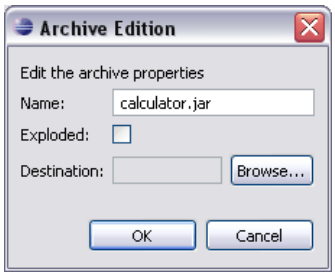
1Packaging A Stateless EJB

1. The Stateless EJB needs to be packaged in a JAR file. Right click the Calculator project in the **Package Explorer** View and select the **Properties** context menu.
2. Select the **Packaging Configurations** option:

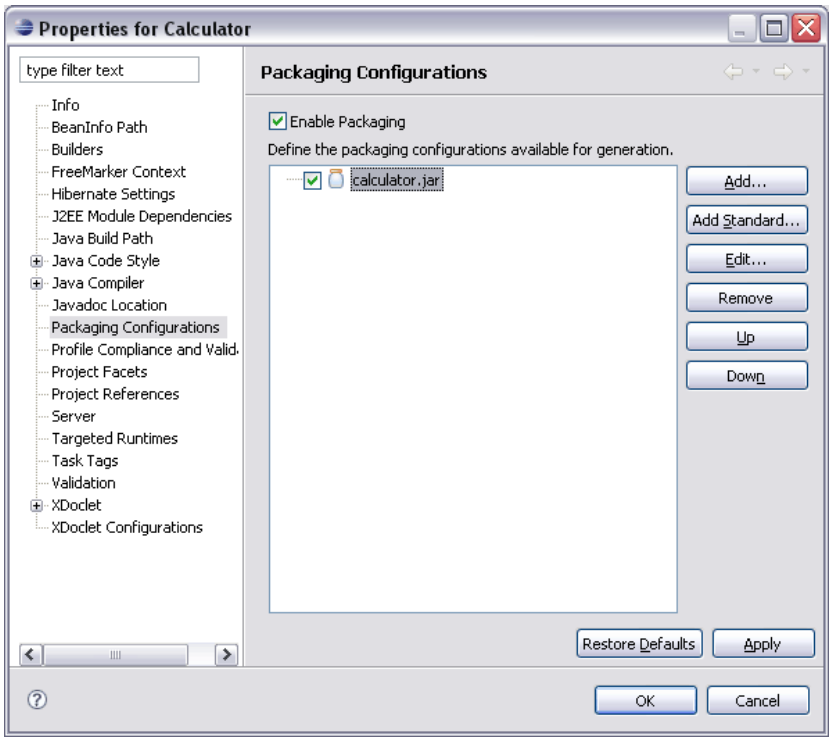


3. At the top of the page there is a check-box labeld **Enable Packaging**. Check this check-box.

4. Click the **Add...** button. Type calculator.jar in the dialog and click the **OK** button.



5. You have created a new packaging configuration that will produce the calculator.jar file.



6. We want to add the Stateless EJB class and interface. Eclipse has generated the compiled classes into the bin folder (declared as the default output dir of the project).

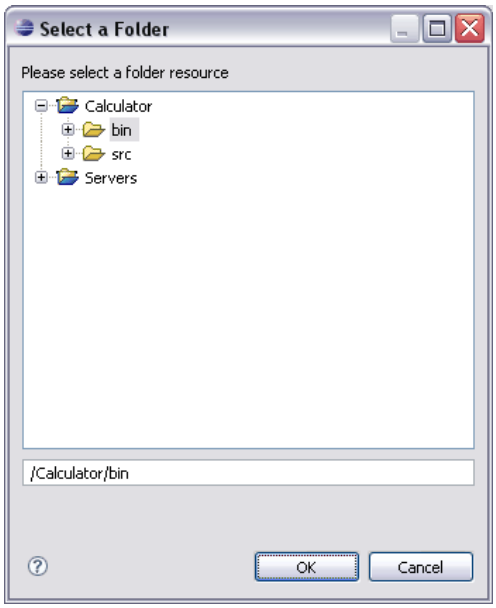
   Select the calculator.jar item and right-click in the area to pop-up the menu and choose **Add Folder**. A **Folder Selection** dialog appears.

   This dialog allows to select which folder (local to workspace or in the file system) to include into the package, to specify include and exclude filters and to set a prefix that will be append when building the package.

7. Click on the **Project Folder...** button. A **Folder Chooser** dialog appears.

This dialog allows selecting which folder to include. This folder can be choosen among all the opened projects.

Select the /Calculator/bin folder and click the **OK** button.



8. The folder is now /Calculator/bin.

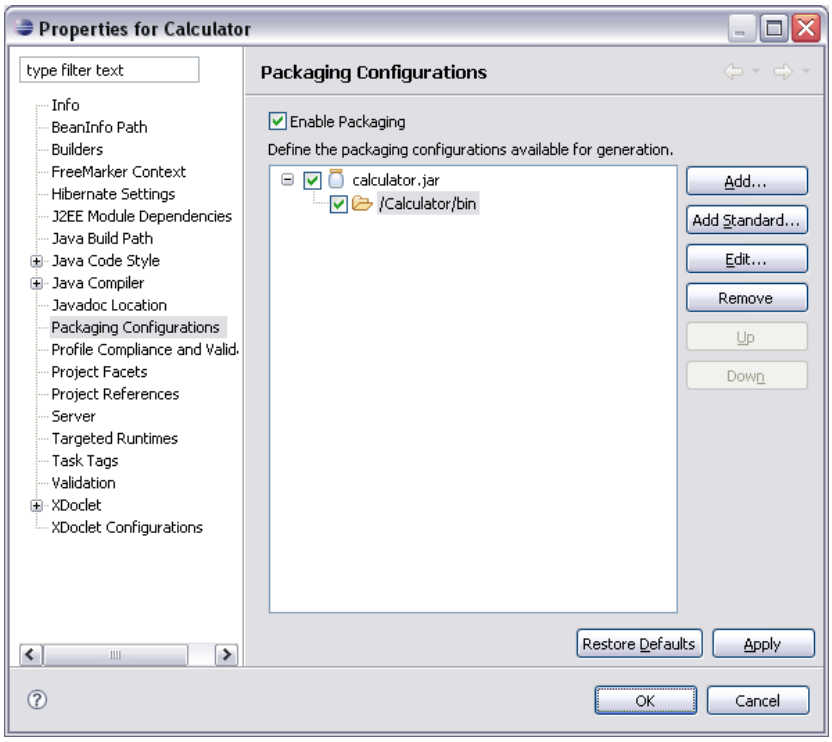
   As we only want the EJB class and interface, specify the following as an include filter:

   by/iba/ejb/*.class

   Click the **OK** button.

9. The packaging configuration for the calculator.jar is now complete.

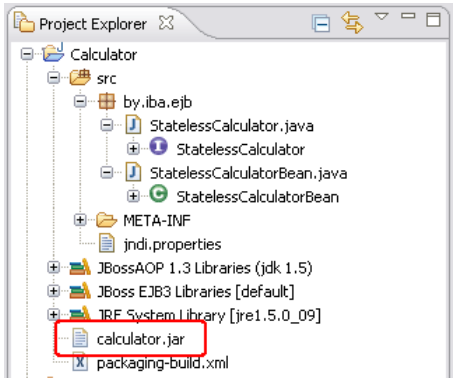
   Click the **OK** button to save the packaging configuration.



10. Right-click on the Calculator project and select **Run Packaging**. The packaging will display its output in the console. The output should look like this:

11. Buildfile: C:\ejb3workspace\Calculator\packaging-build.xml
12. N65540:
13. [jar] Building jar: C:\ejb3workspace\Calculator\calculator.jar
14. _packaging_generation_:
15. BUILD SUCCESSFUL
16. Total time: 875 milliseconds

17. After the execution, you should have a project that looks like this:

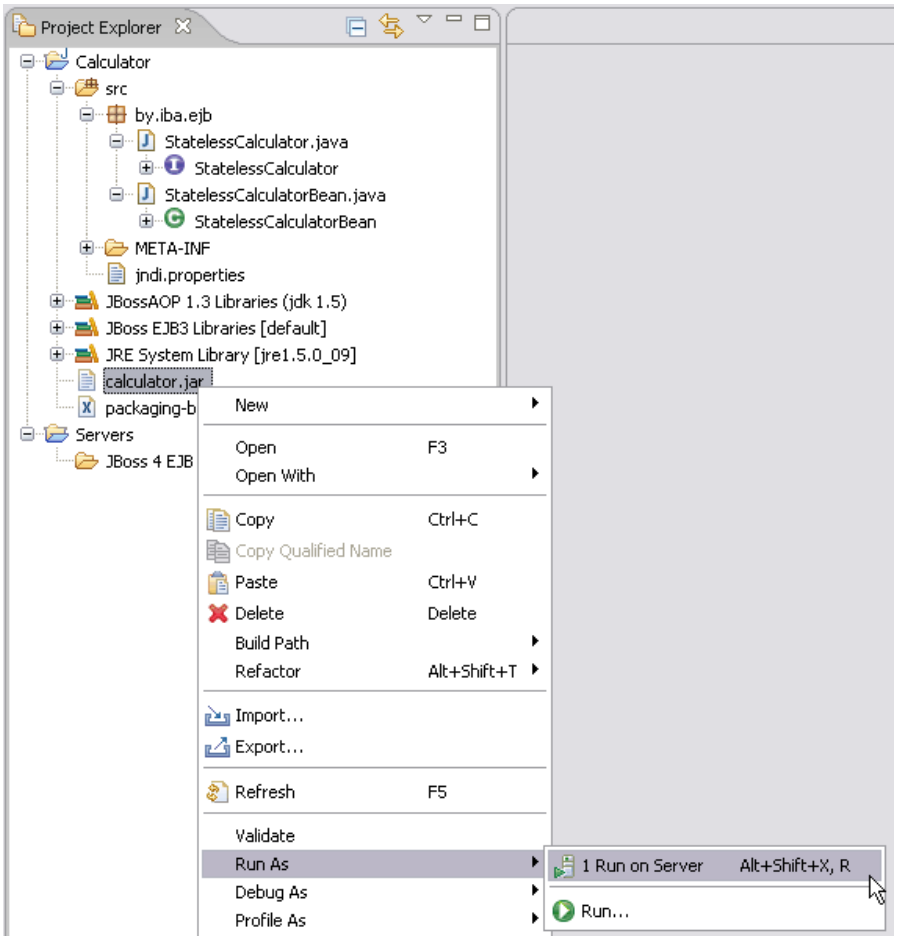


1Deploying An EJB JAR file

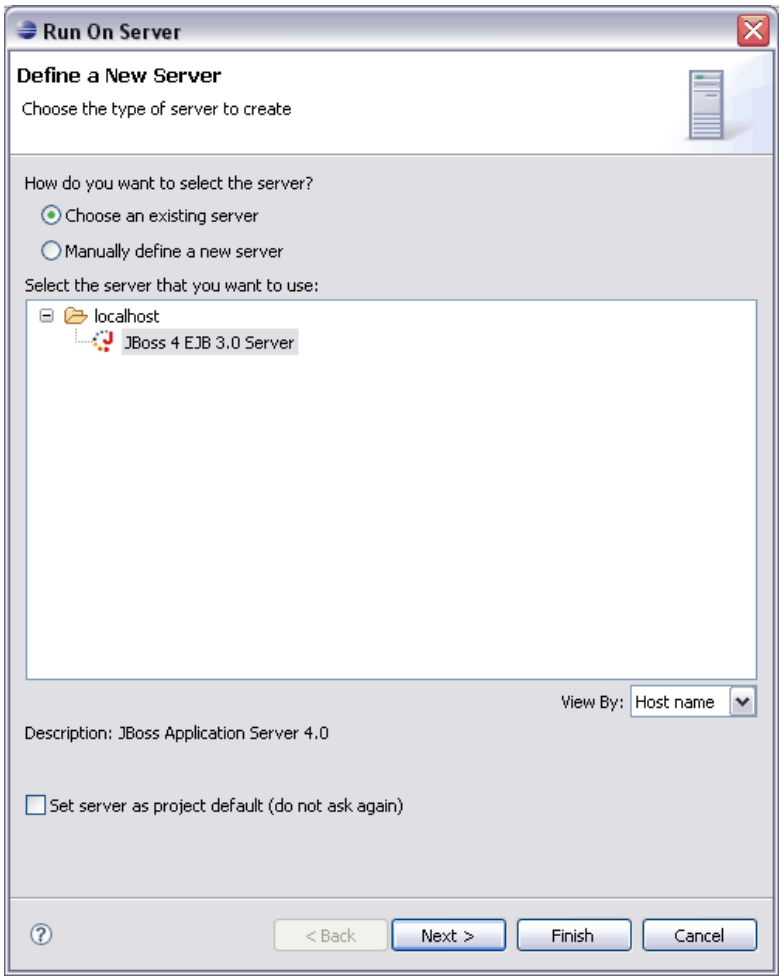
In a conventional JBoss server, deploying any packages is simple: the archive file to deploy should be copied to the relevant server folder in JBoss. For example, simply copy an archive file to the C:\jboss4\server\default\deploy folder to deploy the module. If JBoss is running it detects the change and dynamically deploys the contents.

When programming using JBoss IDE, there is another way to deploy packages.

1. In the **Project Explorer** View right click the calculator.jar file and select **Run As > Run on Server** context menu:

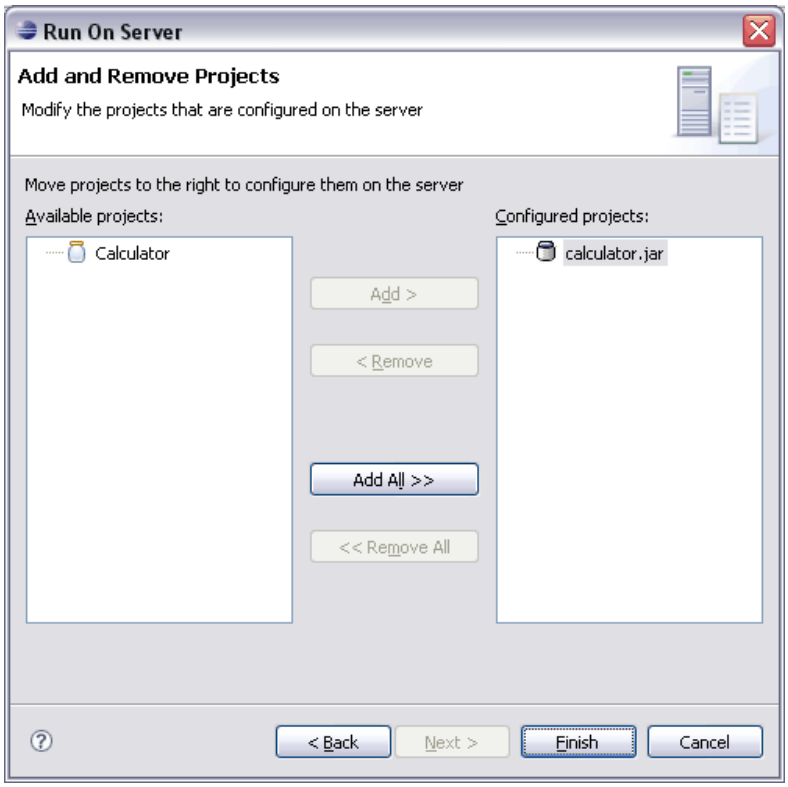


2. Select the JBoss 4 EJB 3.0 Server server and click the **Next** button.



3. Make sure the calculator.jar is in the **Configured projects** pane.

   Click the **Finish** button.

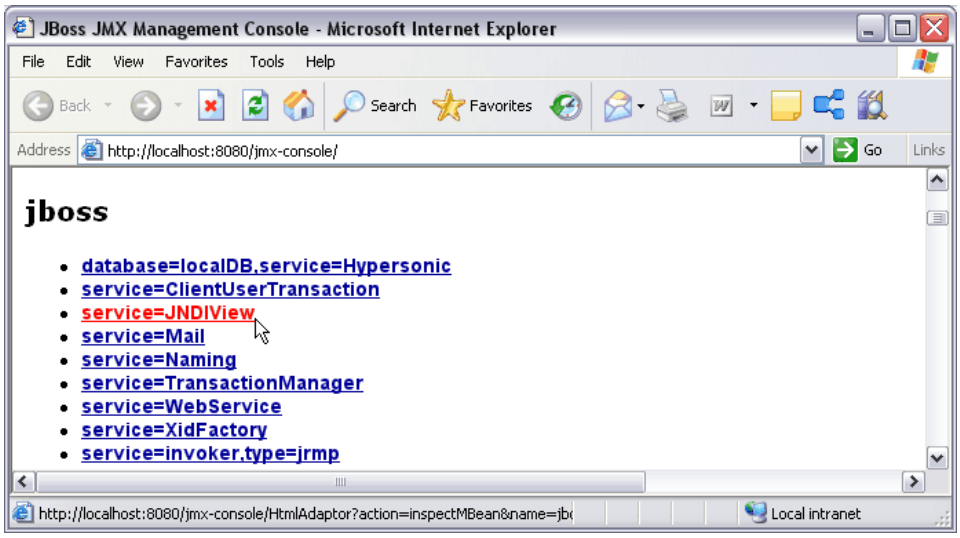


4. In the **Console** View, you should see some deployment activity. The Session EJB is now deployed:

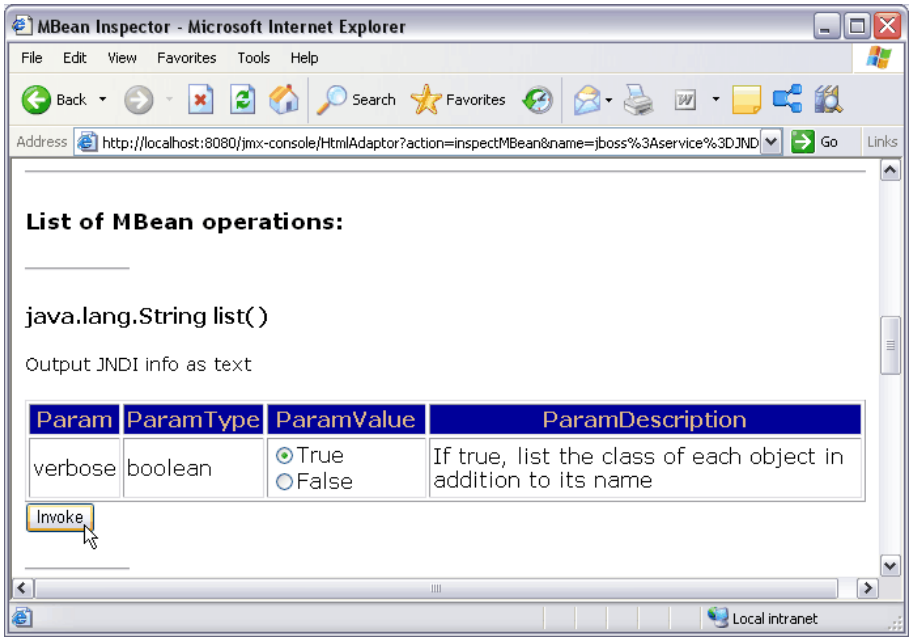
5. 13:34:47,242 INFO [Ejb3Deployment] EJB3 deployment time took: 297
6. 13:34:47,461 INFO [JmxKernelAbstraction] installing MBean: jboss.j2ee:jar=calculator.jar,name=StatelessCalculatorBean,service=EJB3 with dependencies:
7. 13:34:47,757 INFO [EJBContainer] STARTED EJB: by.iba.ejb.StatelessCalculatorBean ejbName: StatelessCalculatorBean
8. 13:34:47,851 INFO [EJB3Deployer] Deployed: file:/C:/jboss4/server/default/deploy/calculator.jar

9. Verify the Remote Bean interface is available now via JNDI.

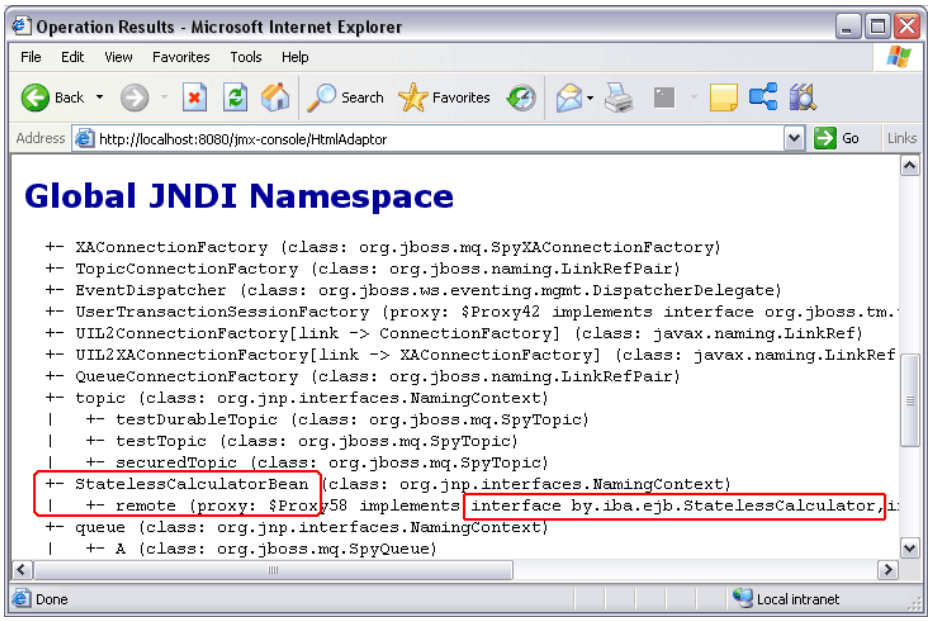
   Open JMX Console in your browser and select the service=JNDIView link:



10. Invoke the java.lang.String list() method to output JNDI info as text by clicking HTML form button:

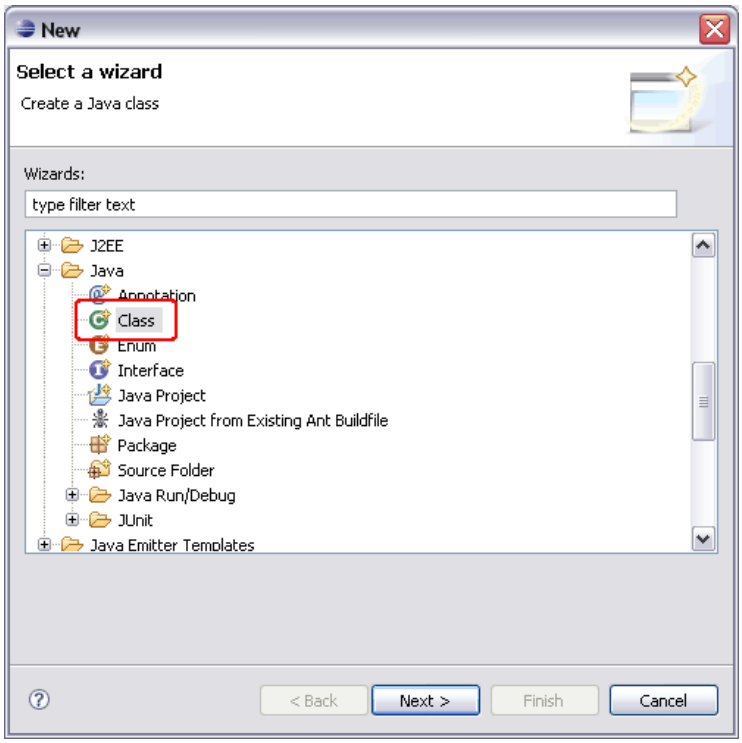


11. You can see in the global JNDI namespace the Remote Interface (StatelessCalculator) is available at the global JNDI name StatelessCalculatorBean/remote.



1Writing A Standalone Java Test Client

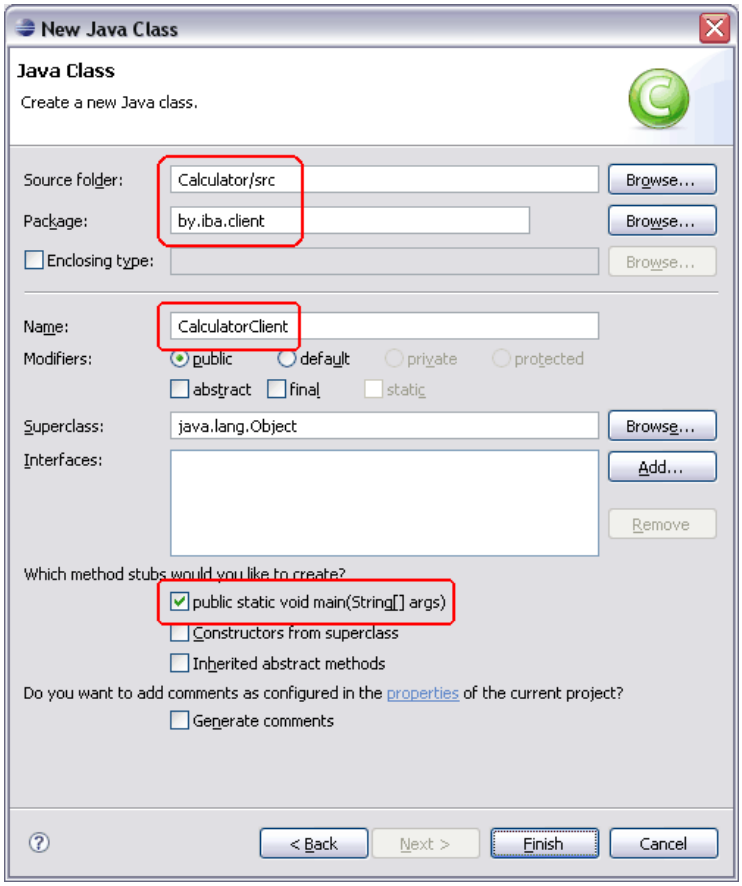
1. In the **Package Explorer** View make sure the Calculator project is selected.
2. Create a new Java class. Open menu **File > New > Other...** and choose **Java > Class**.



3. The package will be by.iba.client and the class name CalculatorClient.

   Leave the default options selected and be sure that public static void main(...) method is checked.

   Click the **Finish** button.



4. Add the following code to the public static void main method body:

```
5.    try {
6.            Context jndiContext = new InitialContext();
7.            Object ref = jndiContext.lookup("StatelessCalculatorBean/remote");
8.            StatelessCalculator calc = (StatelessCalculator) PortableRemoteObject
9.                    .narrow(ref, StatelessCalculator.class);
10.
11.           System.out.println("4 + 3 = " + calc.add(4,3));
12.           System.out.println("4 - 3 = " + calc.subtract(4,3));
13.           System.out.println("4 * 3 = " + calc.multiply(4,3));
14.           System.out.println("4 / 3 = " + calc.divide(4,3));
15.
16.   } catch (NamingException ne) {
17.           ne.printStackTrace();
18.   }
```

NOTE: To resolve imports problems press **Shift + Ctrl + O**. Press **Ctrl + S** to save the Java class.

19. Run the Java class by selecting menu **Run > Run As > Java Application**.

    NOTE: Make sure the CalculatorClient.java is selected in **Package Explorer** View.

    You should see the following result in the **Console** View:

    4 + 3 = 7.0
    4 - 3 = 1.0
    4 * 3 = 12.0
    4 / 3 = 1.3333333333333333

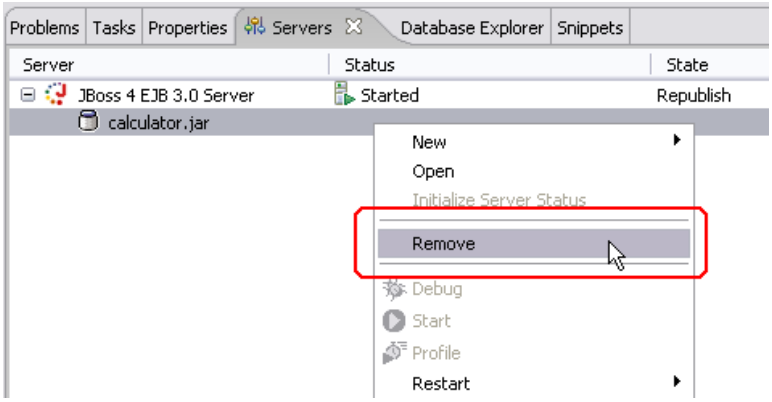
Congratulations ! You have succesfully created and tested your first EJB 3.0 application !!!

1Undeploying An EJB JAR file

In a conventional JBoss server, undeploying any packages is simple: the deployed archive file should be deleted from the relevant server folder in JBoss. For example, simply delete an archive file from the C:\jboss4\server\default\deploy folder to undeploy the module. If JBoss is running it detects the change and dynamically undeploys the contents.

When programming using JBoss IDE, there are several other ways to undeploy package.

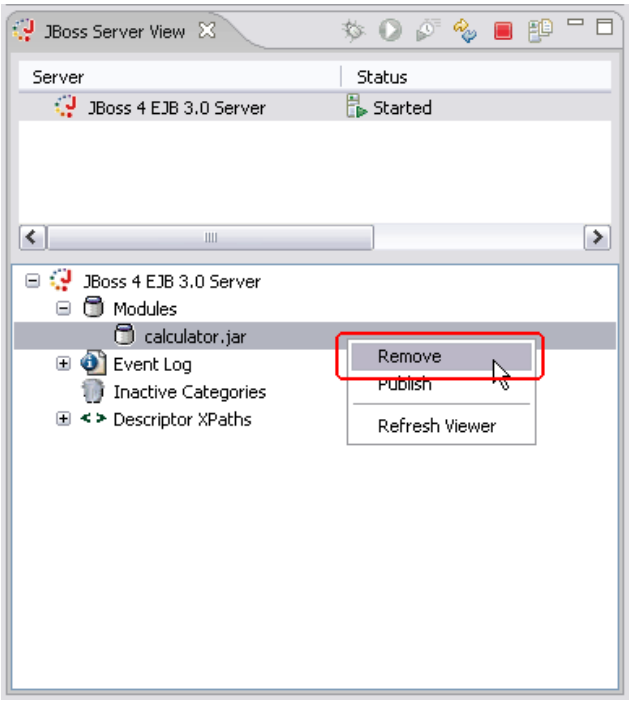
1. Open **J2EE** Perspective: **Window > Open Perspective > Other... > J2EE**.
2. In the **Servers** View, expand the JBoss 4 EJB 3.0 Server and right click on calculator.jar EJB module. Select **Remove**. This will undeploy calculator.jar EJB module.



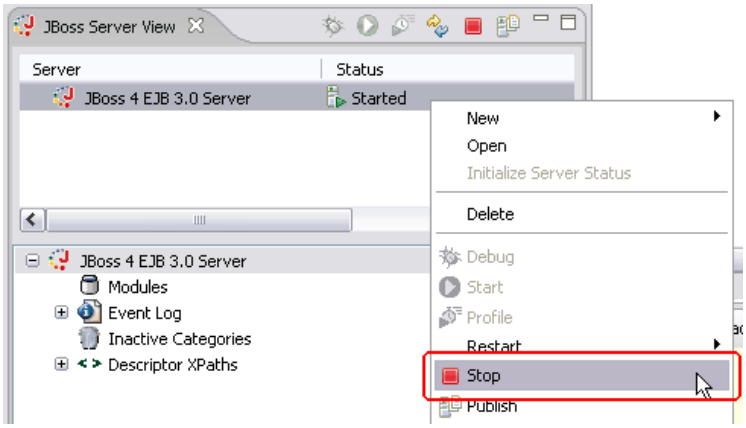
Another way to undeploy package from JBoss server.

1. Open **JBoss AS** Perspective: **Window > Open Perspective > Other... > JBoss AS**.

2. In the **JBoss Server View**, expand the JBoss 4 EJB 3.0 Server / Modules and right click on calculator.jar EJB module. Select **Remove**. This will undeploy calculator.jar EJB module.



OPTIONALLY: Stop the JBoss server by right clicking and selecting **Stop** menu item:



1Appendix C. Enterprise Application Development Environment - Testing - Part 2 (Stateful Session Bean)

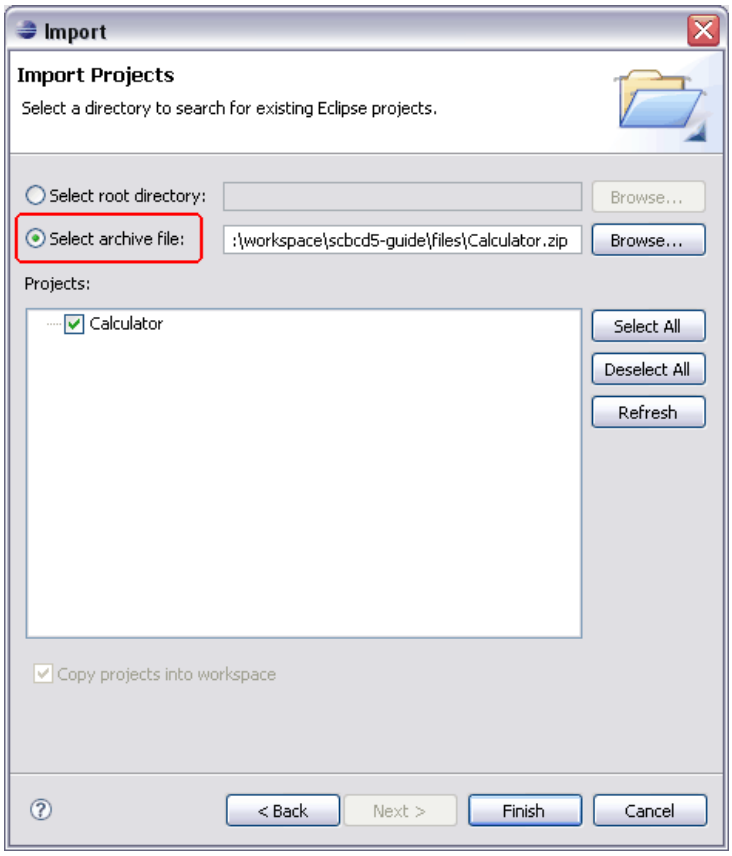
1Overview

In this part of testing we will create a Stateful Session EJB which will utilize the Stateless EJB created in the Part 1 of the Environment Testing series.

NOTE: The source code for this part of testing can be downloaded as a zip archive.

The Part 2 is using the created in Part 1 Calculator project with StatelessCalculator bean. If you did not go through Part 1, you can import the Calculator project from the archive:

1. Click the **File > Import > General > Existing Projects into Workspace** menu.
2. Make sure the **Select archive file** radio button is selected:



1Creating A Stateful Session Bean

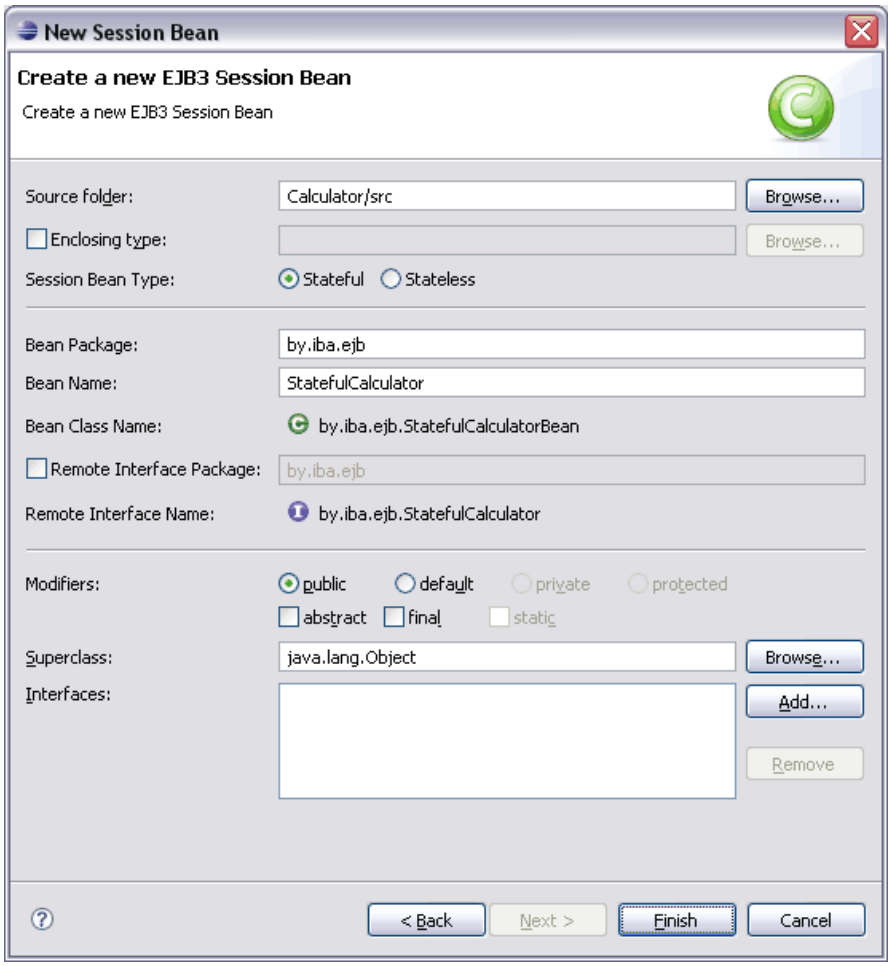
1. Select **File > New > Other...** from the Eclipse menu.
2. Browse to **EJB 3.0 > Session Bean** and click the **Next** button.
3. In the **Create a new EJB 3 Session Bean** dialog, enter the following:

   The **Bean Package** field: by.iba.ejb

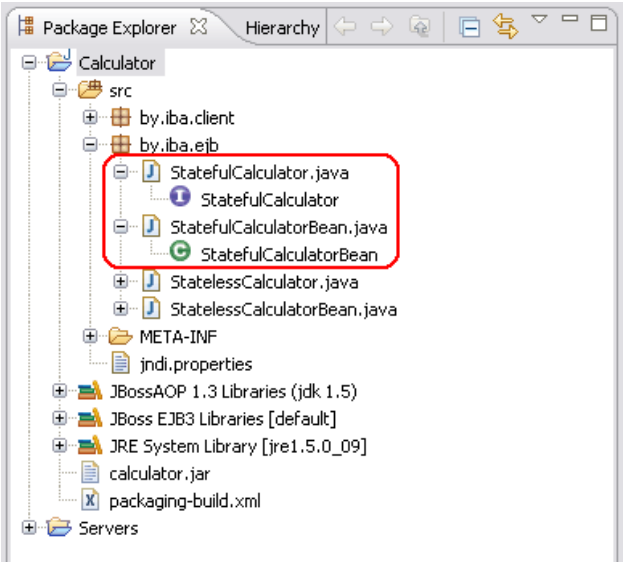
   The **Bean Name** field: StatefulCalculator

   NOTE: make sure Stateful **Session Bean Type** radio button is selected.

   Click the **Finish** button.



4. The wizard has created two files: the Bean Implementation Class (StatefulCalculatorBean.java) and the Bean [Business] Remote Interface (StatefulCalculator.java):



StatefulCalculatorBean class:

```
package by.iba.ejb;

import javax.ejb.Stateful;
import by.iba.ejb.StatefulCalculator;

public @Stateful class StatefulCalculatorBean implements StatefulCalculator {

}
```

StatefulCalculator interface:

```
package by.iba.ejb;

import javax.ejb.Remote;

@Remote
public interface StatefulCalculator {

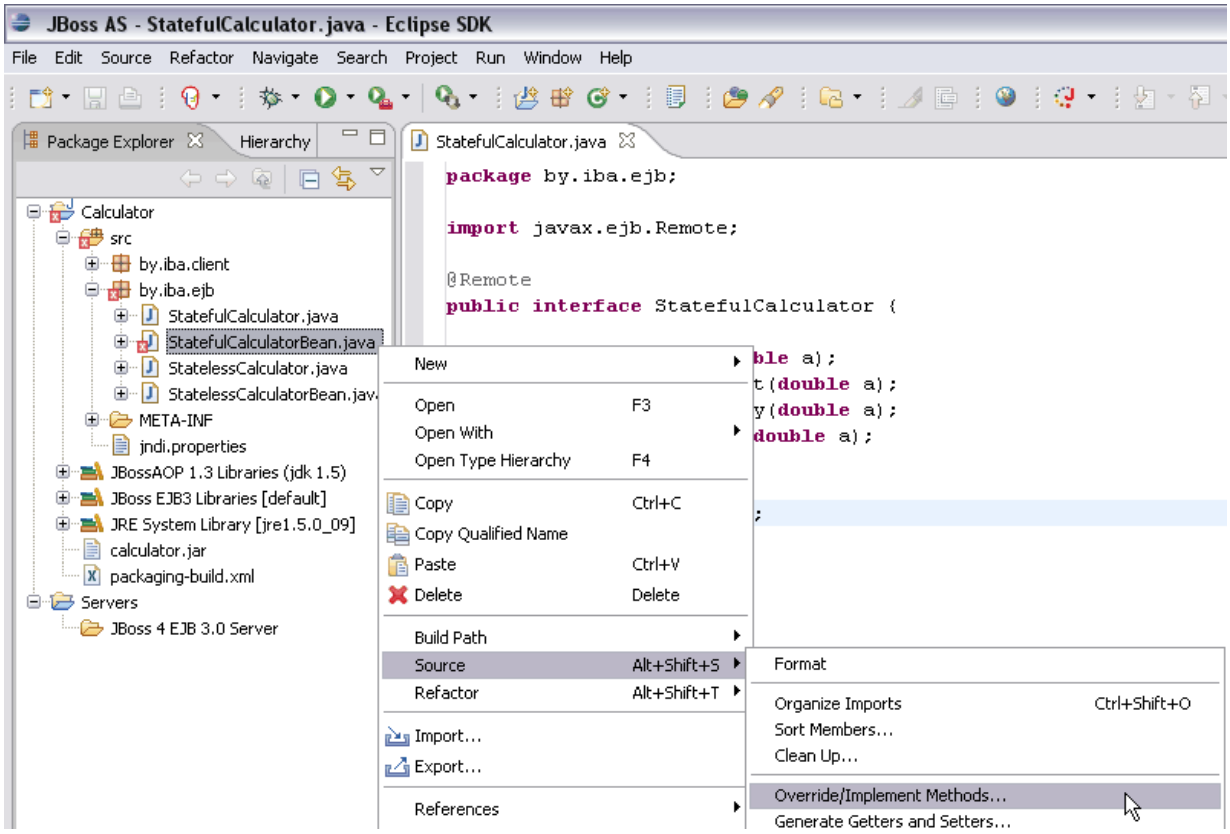
}
```

1Defining A Stateful Session Bean

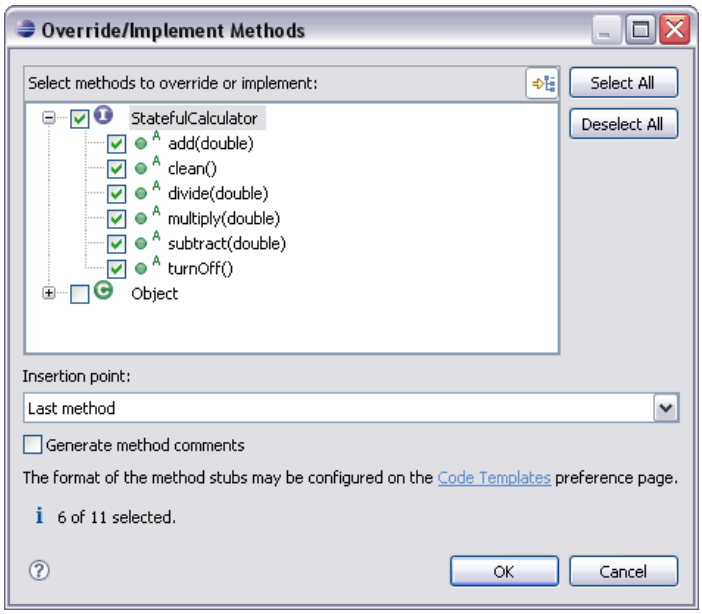
1. Add business method signatures to the StatefulCalculator interface:

```
2.     public double add(double a);
3.     public double subtract(double a);
4.     public double multiply(double a);
5.     public double divide(double a);
6.     public void clean();
7.
8.     public void turnOff();
```

9. In the **Package Explorer** View, right click on the StatelessCalculatorBean class and select **Source > Override/Implement Methods...**.



10. Accept the default selection and press the **OK** button:



11. Open the StatefulCalculatorBean.java file (if not already open), add instance variables and modify the newly created methods as shown below:

```
12.  @EJB private StatelessCalculator calculator;
13.
14.  double register = 0;
15.
16.  public double add(double a) {
17.          register = calculator.add(register, a);
18.          return register;
19.  }
20.
21.  public double subtract(double a) {
22.          register = calculator.subtract(register, a);
23.          return register;
24.  }
25.
26.  public double multiply(double a) {
27.          register = calculator.multiply(register, a);
28.          return register;
29.  }
30.
31.  public double divide(double a) {
32.          register = calculator.divide(register, a);
33.          return register;
34.  }
35.
36.  public void clean() {
37.          register = 0;
38.  }
39.
40.  @Remove
41.  public void turnOff() {
42.          System.out.println("[StatefulCalculatorBean] Good bye ! I gotta split !");
43.  }
44.
45.  @SuppressWarnings("unused")
46.  @PostConstruct
47.  private void afterCreated() {
48.          System.out.println("[StatefulCalculatorBean] PostConstruct callback !");
49.  }
50.
51.  @SuppressWarnings("unused")
52.  @PreDestroy
53.  private void beforeRemoved() {
54.          System.out.println("[StatefulCalculatorBean] PreDestroy callback !");
55.  }
```

NOTE: To resolve imports problems press **Shift + Ctrl + O**. Press **Ctrl + S** to save the Java class.

56. The Statful Session EJB is complete. With EJB 3, there is no need to code deployment descriptors or the home interface. The next step is to package the EJB in the EJB JAR file and deploy on the server.

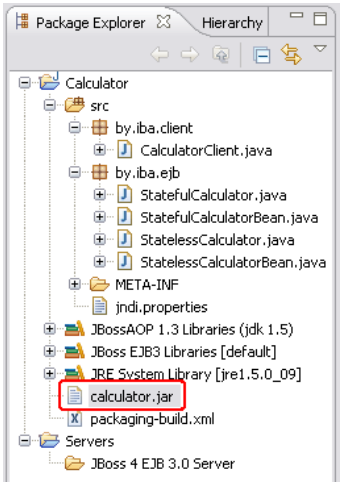
1Packaging and Deploying A Stateful EJB

1. The EJB project needs to be packaged in a JAR file. We will be re-using the packaging configuration which was created in Part 1: Packaging A Stateless EJB.

2. Right-click on the Calculator project and select **Run Packaging**. The packaging will display its output in the console. The output should look like this:

3. Buildfile: C:\ejb3workspace\Calculator\packaging-build.xml
4. N65540:
5. _packaging_generation_:
6. BUILD SUCCESSFUL
7. Total time: 906 milliseconds

8. After the execution, you should have a project that looks like this:



9. OPTIONALLY: Undeploy the previous calculator.jar EJB module as described here: Undeploying An EJB JAR file.

10. In the **Project Explorer** View right click the calculator.jar file and select **Run As > Run on Server** context menu.

11. In the **Console** View, you should see some deployment activity. The Session EJBs are now deployed:

12.
13. 16:00:32,527 INFO [Ejb3Deployment] EJB3 deployment time took: 47
14. 16:00:32,558 INFO [JmxKernelAbstraction] installing MBean: jboss.j2ee:jar=calculator.jar,name=StatefulCalculatorBean,service=EJB3 with dependencies:
15. 16:00:32,558 INFO [JmxKernelAbstraction] jboss.j2ee:jar=calculator.jar,name=StatelessCalculatorBean,service=EJB3
16. 16:00:32,558 INFO [JmxKernelAbstraction] installing MBean: jboss.j2ee:jar=calculator.jar,name=StatelessCalculatorBean,service=EJB3 with dependencies:
17. 16:00:32,574 INFO [EJBContainer] STARTED EJB: by.iba.ejb.StatelessCalculatorBean ejbName: StatelessCalculatorBean

18. 16:00:32,652 INFO [EJBContainer] STARTED EJB: by.iba.ejb.StatefulCalculatorBean ejbName: StatefulCalculatorBean
19. 16:00:32,699 INFO [SimpleStatefulCache] Initializing SimpleStatefulCache with maxSize: 100000 timeout: 300 for jboss.j2ee:jar=calculator.jar,name=StatefulCalculatorBean,service=EJB3
20. 16:00:32,699 INFO [EJB3Deployer] Deployed: file:/C:/jboss4/server/default/deploy/calculator.jar

21. OPTIONALLY: Verify the Stateful Bean Remote [Business] Interface is available now via JNDI.

    Open JMX Console in your browser and select the service=JNDIView link.

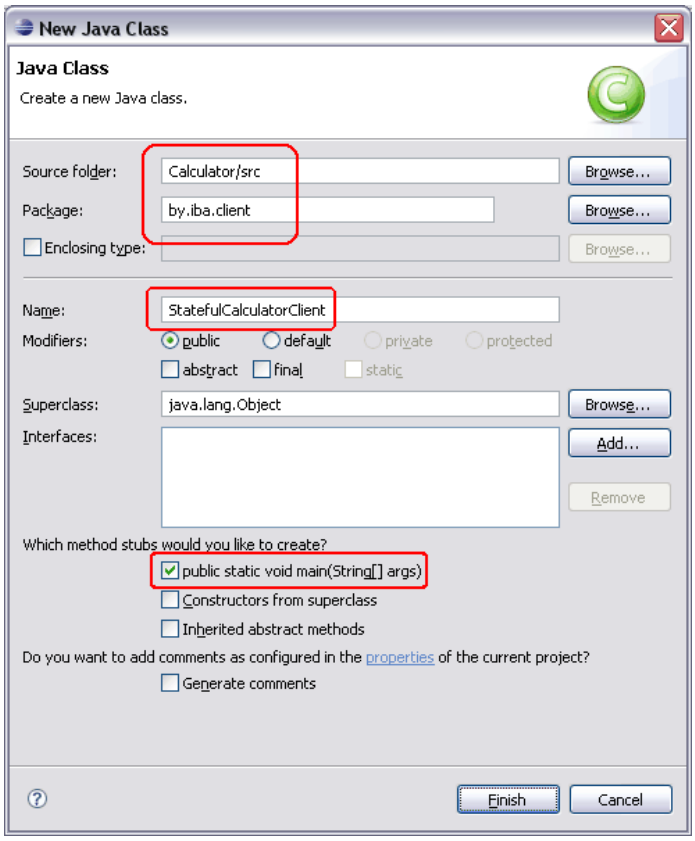
    Invoke the java.lang.String list() method to output JNDI info as text.

1Writing A Standalone Java Test Client for Stateful EJB

1. In the **Package Explorer** View make sure the Calculator project is selected.
2. Create a new Java class. Open menu **File > New > Other...** and choose **Java > Class**.
3. The package will be by.iba.client and the class name StatefulCalculatorClient.

   Leave the default options selected and be sure that public static void main(...) method is checked.

   Click the **Finish** button.



4. Add the following code to the public static void main method body:

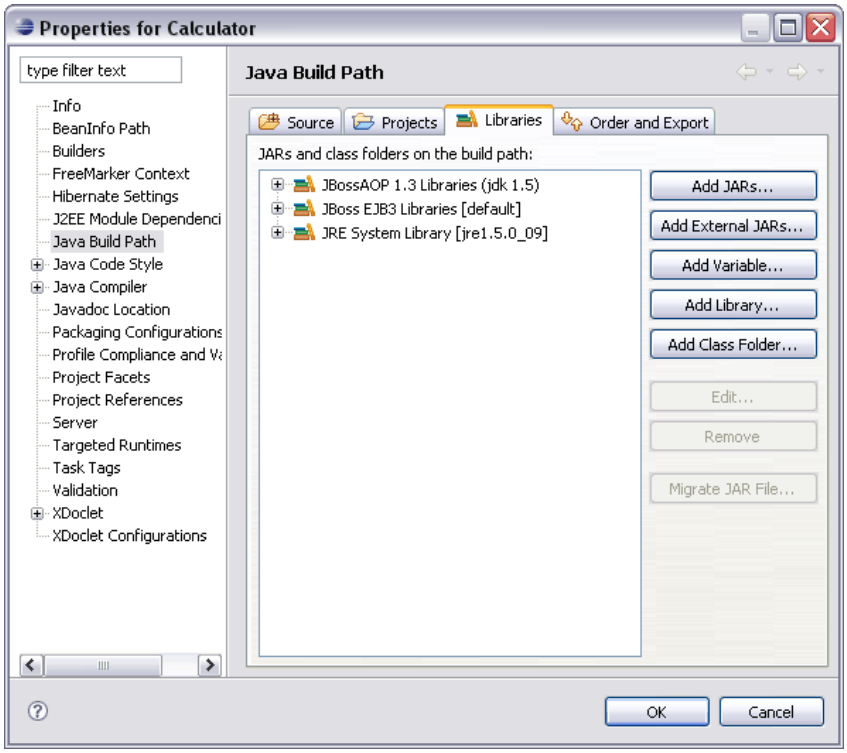
```
5.    try {
6.            Context jndiContext = new InitialContext();
7.            Object ref = jndiContext.lookup("StatefulCalculatorBean/remote");
8.            StatefulCalculator calc = (StatefulCalculator) PortableRemoteObject
9.                    .narrow(ref, StatefulCalculator.class);
10.
11.           calc.clean();
12.           double register = calc.add(4);
13.           System.out.println(register + " + 3 = " + (register = calc.add(3)));
14.           System.out.println(register + " - 3 = " + (register = calc.subtract(3)));
15.           System.out.println(register + " * 3 = " + (register = calc.multiply(3)));
16.           System.out.println(register + " / 3 = " + (register = calc.divide(3)));
17.
18.           calc.turnOff();
19.
20.           System.out.println("Trying to access removed bean...");
21.           calc.clean();
22.
23.   } catch (Exception e) {
24.           System.out.println(e.getMessage());
25.   }
```

NOTE: To resolve imports problems press **Shift + Ctrl + O**. Press **Ctrl + S** to save the Java class.

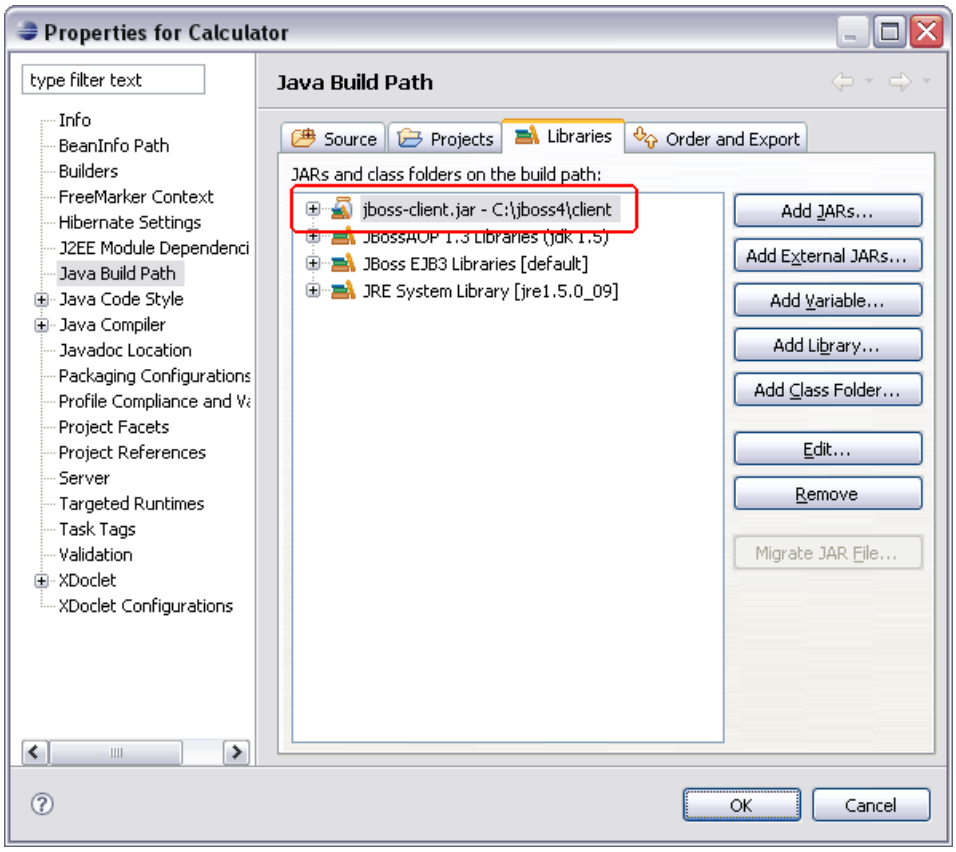
26. Update client's CLASSPATH with required libraries.

Right click the Calculator project in the **Package Explorer** View and select the **Properties** context menu.

Select the **Java Build Path** option, go to **Libraries** tab:



Click the **Add External JARs...** button and add the C:\jboss4\client\jboss-client.jar file to the CLASSPATH:



Click the **OK** button.

27. Run the Java class by selecting menu **Run > Run As > Java Application**.

    NOTE: Make sure the StatefulCalculatorClient.java is selected in **Package Explorer** View.

    You should see the following result in the Java client's **Console** View:

```
4.0 + 3 = 7.0
7.0 - 3 = 4.0
4.0 * 3 = 12.0
12.0 / 3 = 4.0
Trying to access removed bean...
Could not find Stateful bean: 93u4t67-2seebc-eu6srhdl-1-eu70q7f5-k
```

    If you switch to the Servers's **Console** View (use **Display Selected Console** button), you will see the server's output:


```
17:17:33,329 INFO  [STDOUT] [StatefulCalculatorBean] PostConstruct callback !
17:17:33,391 INFO  [STDOUT] [StatefulCalculatorBean] Good bye ! I gotta split !
17:17:33,407 INFO  [STDOUT] [StatefulCalculatorBean] PreDestroy callback !
```

Congratulations ! You have succesfully tested EJB 3.0 application with Stateful Session Bean !!!

1Appendix D. Enterprise Application Development Environment - Testing - Part 3 (EJB 3.0 Entity)

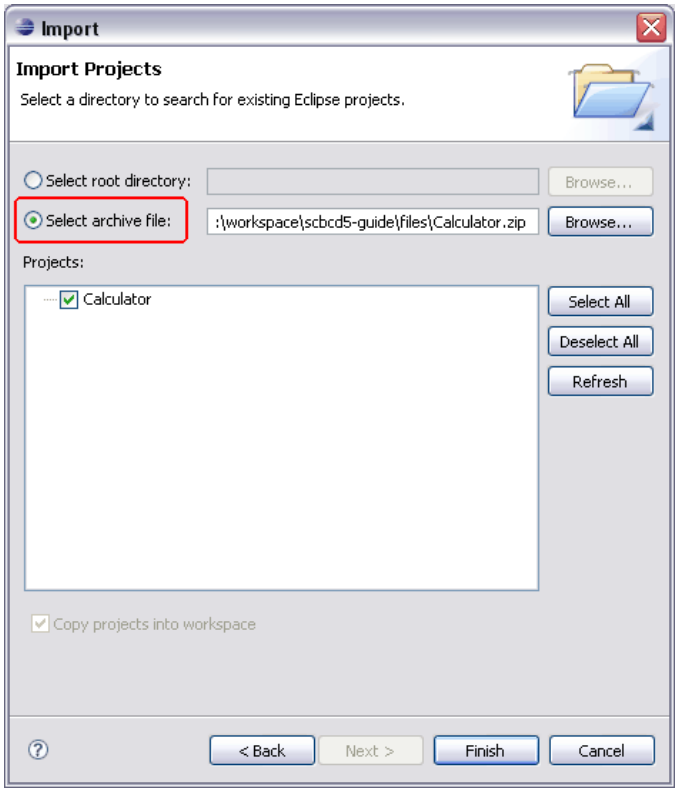
1Overview

In the previous parts, Eclipse, and JBoss were tested by building Stateless and Stateful Session EJBs. In this part an EJB 3.0 Entity will be created to test the database connectivity. Once this part has been completed, the infrastructure is known to be complete and functioning.

NOTE: The source code for this part of testing can be downloaded as a zip archive.

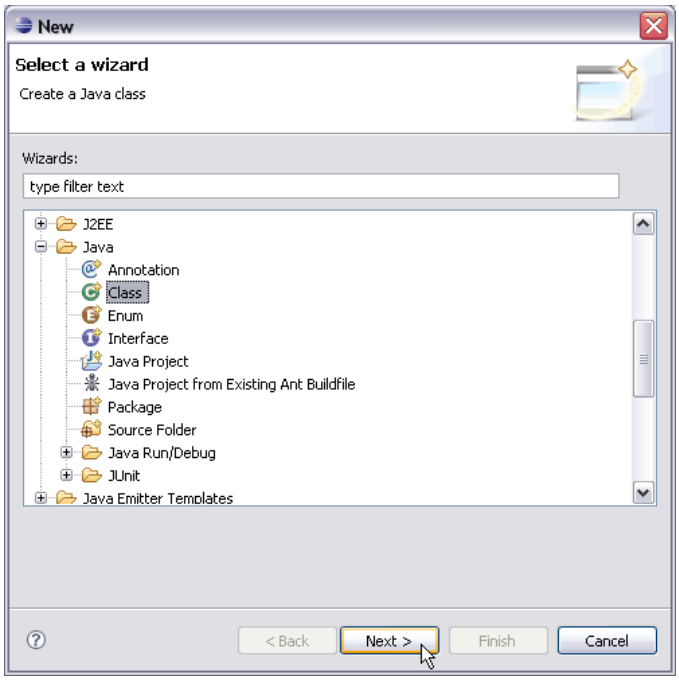
The Part 3 is using the created in Part 1 Calculator project with StatelessCalculator bean. If you did not go through Part 1, you can import the Calculator project from the Part 1 archive or from the Part 2 archive:

1. Click the **File > Import > General > Existing Projects into Workspace** menu.
2. Make sure the **Select archive file** radio button is selected:



1Creating An Entity Object (POJO)

1. Select **File > New > Other...** from the Eclipse menu.
2. Browse to **Java > Class** and click the **Next** button.



3. In the **New Java Class** dialog, enter the following:

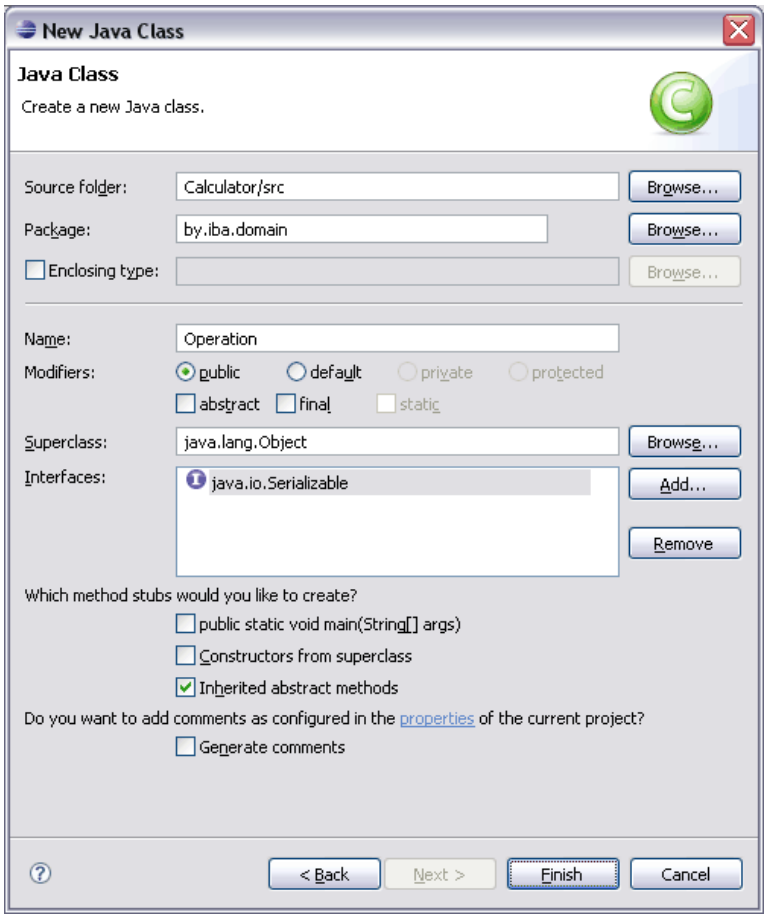
   The **Package** field: by.iba.domain

   The **Name** field: Operation

   The **Interfaces** list: java.io.Serializable (click the **Add...** button to add the interface).

   NOTE: make sure **Source folder** is Calculator/src.

   Click the **Finish** button.



4. Open the new Operation.java file and add the @Entity annotation to the class:

```
@Entity
public class Operation implements Serializable {
                ...
```

The class is annotated with @Entity to denote that it is an Entity Object.

8. Add the following instance variables (future persistent fields) to the class:

```
private int id;

private Timestamp timestamp;

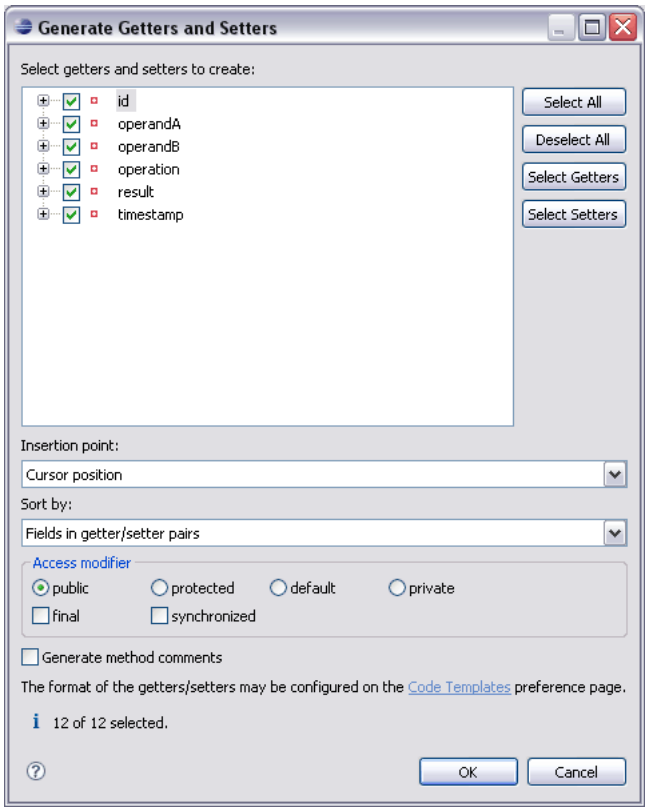
private double operandA;

private double operandB;

private String operation;

private double result;
```

20. Click the **Source > Generate Getters and Setters...** to generate getters and setters.



21. Add the following annotations to the **getter methods**:

    The getId() method is annotated with @Id to denote that id is the primary key of the entity (by default, the mapped column for the primary key of the entity is assumed to be the primary key of the primary table), the method is also annotated with @GeneratedValue to denote that the persistence provider must assign primary keys for the entity.

```
@Id
@GeneratedValue
@Column(name="OPERATION_ID")
public int getId() {
   return id;
}

@Column(name="OPERAND_A")
public double getOperandA() {
   return operandA;
}

@Column(name="OPERAND_B")
public double getOperandB() {
   return operandB;
}

@Column(length=8)
public String getOperation() {
   return operation;
```

```
}
```

Override the toString() method:

```
public String toString() {
   return " \n" + getTimestamp() + " : " +
                getOperandA() + " " + getOperation() + " " + getOperandB() +  " is " + getResult();
}
```

Click the **Ctrl + Shift + O** to organize imports and **Ctrl + S** to save.

1Use An Entity Object (POJO)

1. Create the new Stateless Session EJB.

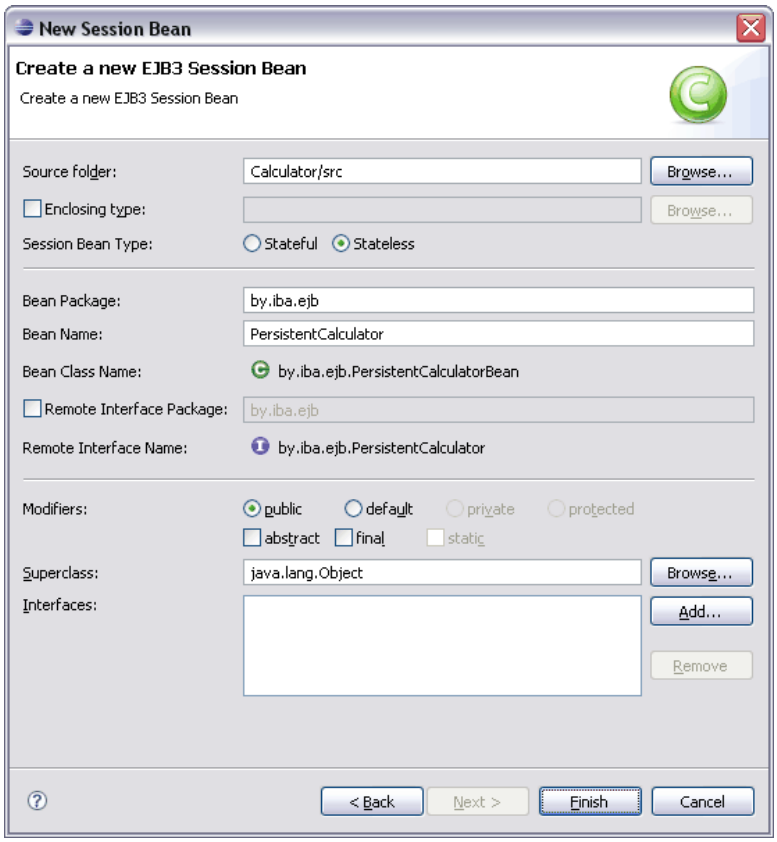
   In the **Create a new EJB 3 Session Bean** dialog, enter the following:

   The **Bean Package** field: by.iba.ejb

   The **Bean Name** field: PersistentCalculator

   NOTE: make sure Stateless **Session Bean Type** radio button is selected.

   Click the **Finish** button.



2. Add business method signatures to the PersistentCalculator interface:

```
3.
4.      public double add(double a, double b);
5.      public double subtract(double a, double b);
6.      public double multiply(double a, double b);
7.      public double divide(double a, double b);
8.
9.      public List<Operation> getHistory();
10.     public void clearHistory();
11.
```

Click the **Ctrl + Shift + O** to organize imports and **Ctrl + S** to save.

12. Open the PersistentCalculatorBean.java file (if not already open) and modify the newly created class as shown below:

```
13.
14.
15.             @PersistenceContext(unitName = "calculator")
16.             private EntityManager manager;
17.
18.             @EJB private StatelessCalculator calculator;
19.
20.             public double add(double a, double b) {
21.                     return calculator.add(a, b);
```

```
22.             }
23.
24.             public double subtract(double a, double b) {
25.                     return calculator.subtract(a, b);
26.             }
27.
28.             public double multiply(double a, double b) {
29.                     return calculator.multiply(a, b);
30.             }
31.
32.             public double divide(double a, double b) {
33.                     return calculator.divide(a, b);
34.             }
35.
36.             public List<Operation> getHistory() {
37.                     ArrayList<Operation> list = new ArrayList<Operation>();
38.                     Query q = manager.createQuery("FROM Operation o ORDER BY o.timestamp DESC");
39.                     q.setMaxResults(5);
40.                     for (Object o : q.getResultList()) {
41.                             list.add((Operation) o);
42.                     }
43.                     return list;
44.             }
45.
46.             public void clearHistory() {
47.                     Query q = manager.createQuery("FROM Operation o");
48.                     for (Object o : q.getResultList()) {
49.                             manager.remove(o);
50.                     }
51.                     manager.flush();
52.             }
53.
54.             @AroundInvoke
55.             public Object persistOperation(InvocationContext ctx) throws Exception {
56.                     Object res = null;
57.                     try {
58.                             res = ctx.proceed();
59.                     } catch (Exception e) {
60.                     } finally {
61.                             if ((ctx.getParameters() != null) && (ctx.getParameters().length > 0)) {
62.                                     Operation o = new Operation();
63.                                     double operandA = new
    Double(ctx.getParameters()[0].toString());
64.                                     double operandB = new
    Double(ctx.getParameters()[1].toString());
65.                                     double result   = new Double(res.toString());
66.                                     o.setOperandA(operandA);
67.                                     o.setOperandB(operandB);
68.                                     o.setResult(result);
69.                                     o.setOperation(ctx.getMethod().getName());
70.                                     o.setTimestamp(new Timestamp(new Date().getTime()));
71.                                     manager.persist(o);
72.                             }
```

```
73.                     }
74.                     return res;
75.             }
76.
```

Click the **Ctrl + Shift + O** to organize imports and **Ctrl + S** to save the file.

1Configuring Persistence Unit Definition

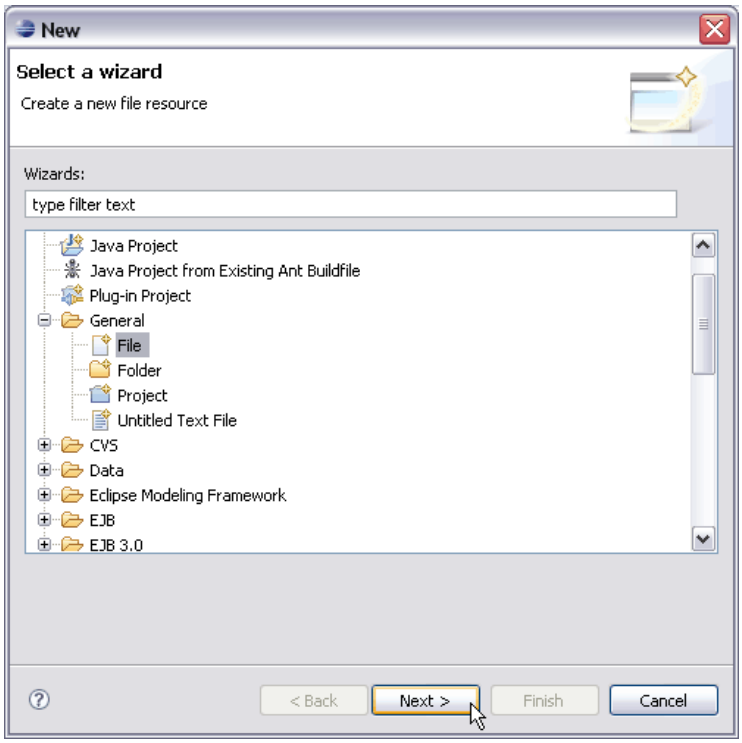
1. The EntityManager in the PersistentCalculatorBean.java source (see the previous section), uses the name "calculator" to identify the **persistence unit** to use:

```
2.     @PersistenceContext(unitName = "calculator")
3.     private EntityManager manager;
```

This persistence unit is defined in a new file called persistence.xml, in the src/META-INF folder of the Calculator project.

Select **File > New > Other...** from the Eclipse menu.

Browse to **General > File** and click the **Next** button.

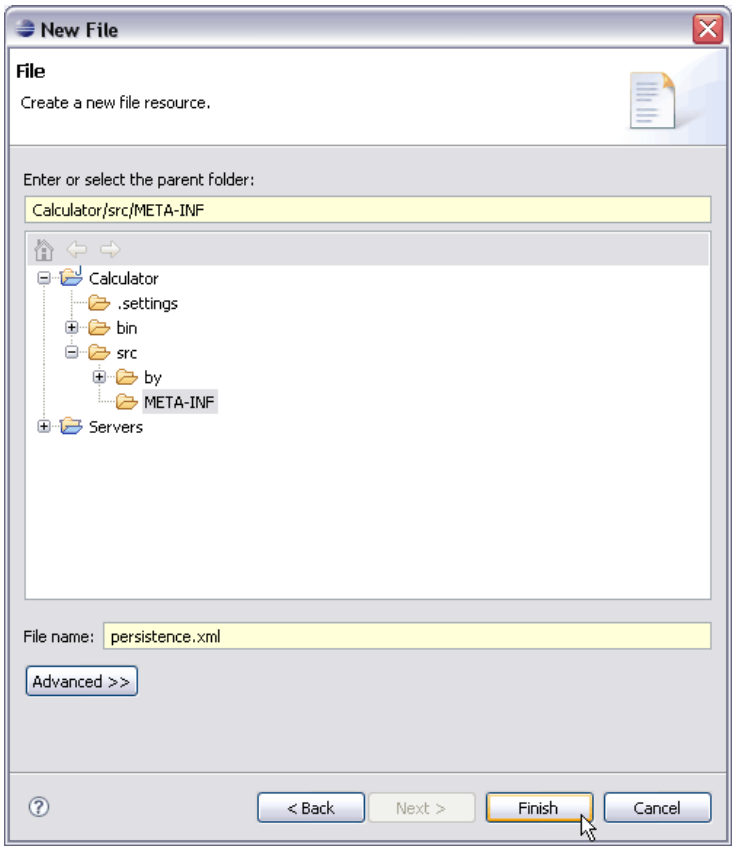


In the **New File** dialog, enter the following:

The **Enter or select the parent folder** field: Calculator/src/META-INF

The **File name** field: persistence.xml

Click the **Finish** button.



4. Switch to the **Source** tab of the file editor and add the persistence.xml file contents as follows:

```
<?xml version="1.0" encoding="UTF-8"?>

<persistence>
	<persistence-unit name="calculator">
		<jta-data-source>java:/DefaultDS</jta-data-source>
		<properties>
			<property name="hibernate.hbm2ddl.auto" value="create-drop" />
		</properties>
	</persistence-unit>
</persistence>

```

The persistence.xml file makes a link between the persistence unit and the datasource. JBoss comes with the **Hypersonic SQL** database embedded within it and a default datasource available in JNDI under java:/DefaultDS.

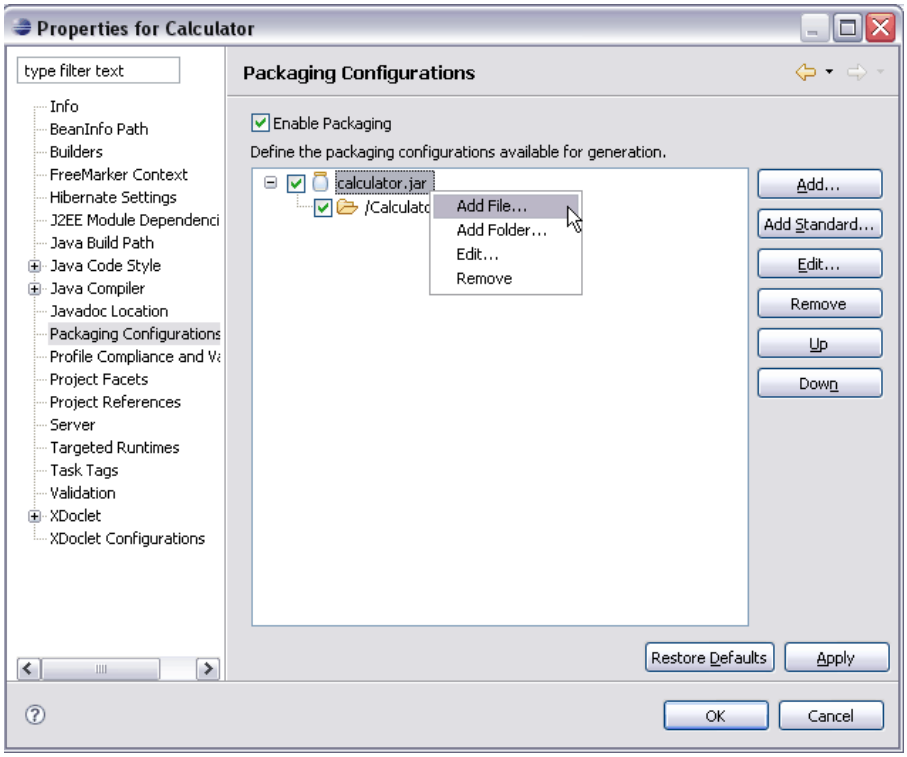
The persistence configuration says to create new tables each time the application is deployed and to drop tables when it's undeployed:

```
<property name="hibernate.hbm2ddl.auto" value="create-drop" />
```

This means that your data will be lost each time you redeploy your application.

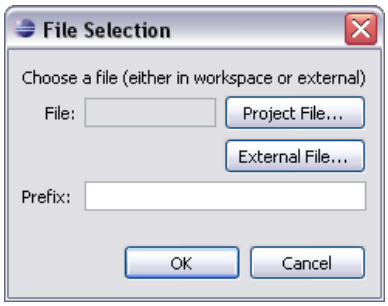
1Packaging Persistence Unit Definition

1. The file persistence.xml must be packaged along with the EJB class files. This is done by accessing the project properties, and selecting the **Packaging Configurations** section:



2. Select the calculator.jar item and right-click in the area to pop-up the menu and choose **Add File...**. A **File Selection** dialog appears.

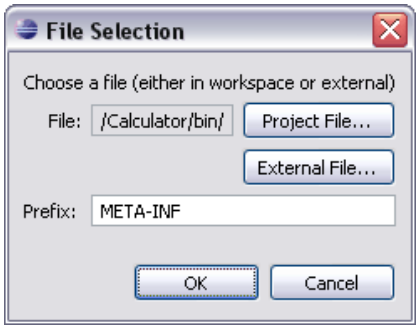
   This dialog allows to select which file (local to workspace or in the file system) to include into the package:



3. Click on the **Project File...** button. A **Select a File** dialog appears.

   Select the /Calculator/bin/META-INF/persistence.xml file and click the **OK** button.

   In the **Prefix** field enter META-INF. This is the location within the archive where persistence.xml should be placed:



   Click the **OK** button.

4. We need to add the Entity Object class too. Eclipse has generated the compiled classes into the bin folder (declared as the default output dir of the project).

   Select the calculator.jar item and right-click in the area to pop-up the menu and choose **Add Folder...**. A **Folder Selection** dialog appears.

   This dialog allows to select which folder (local to workspace or in the file system) to include into the package, to specify include and exclude filters and to set a prefix that will be append when building the package.

5. Click on the **Project Folder...** button. A **Folder Chooser** dialog appears.

   This dialog allows selecting which folder to include. This folder can be choosen among all the opened projects.

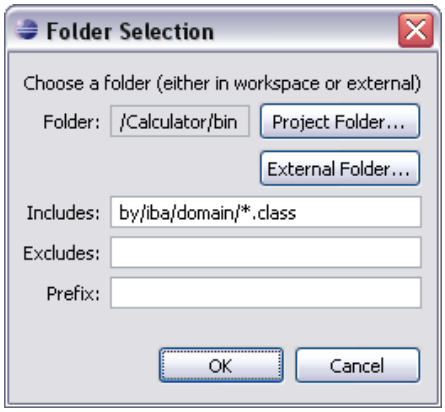
   Select the /Calculator/bin folder and click the **OK** button.

6. The folder is now /Calculator/bin.

   As we only want the Entity Object class, specify the following as an include filter:

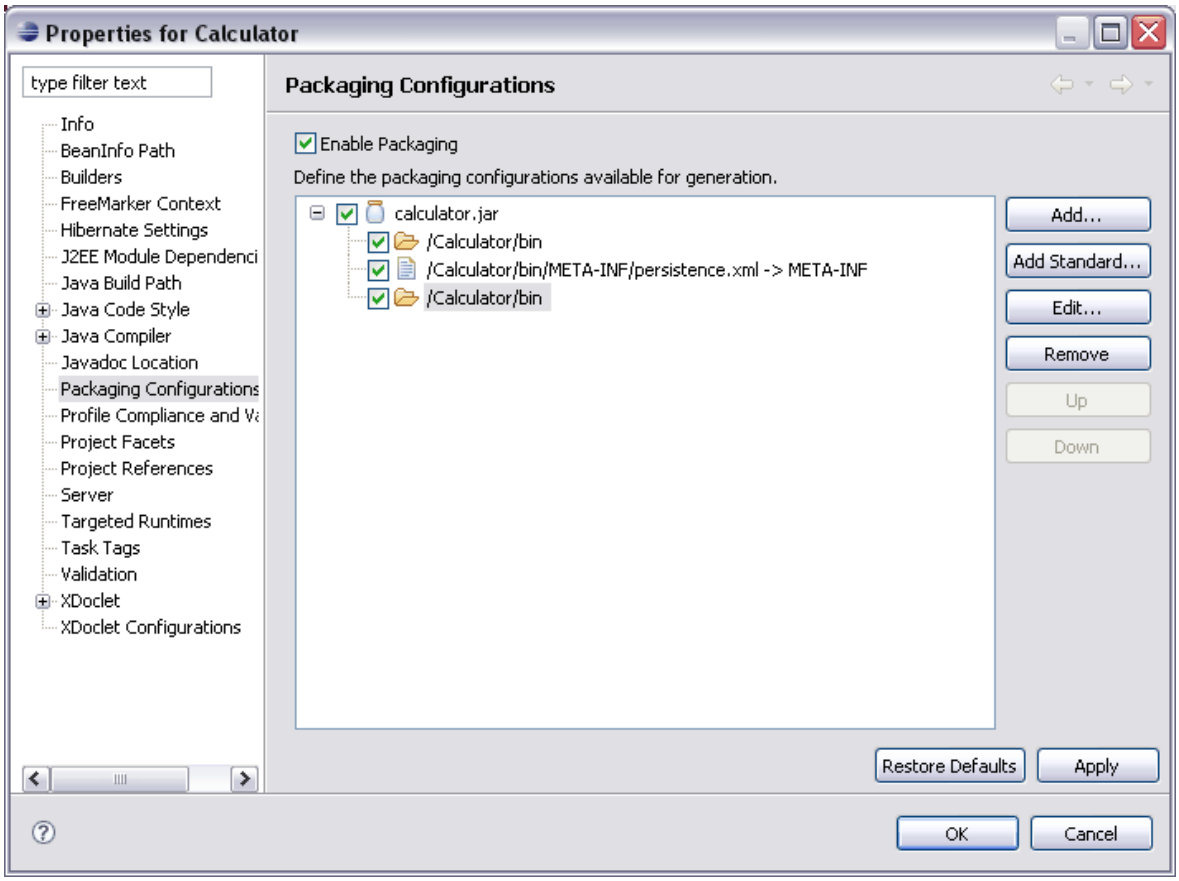
   by/iba/domain/*.class

Click the **OK** button.



7. The packaging configuration for the calculator.jar is now complete.

   Click the **OK** button to save the packaging configuration.

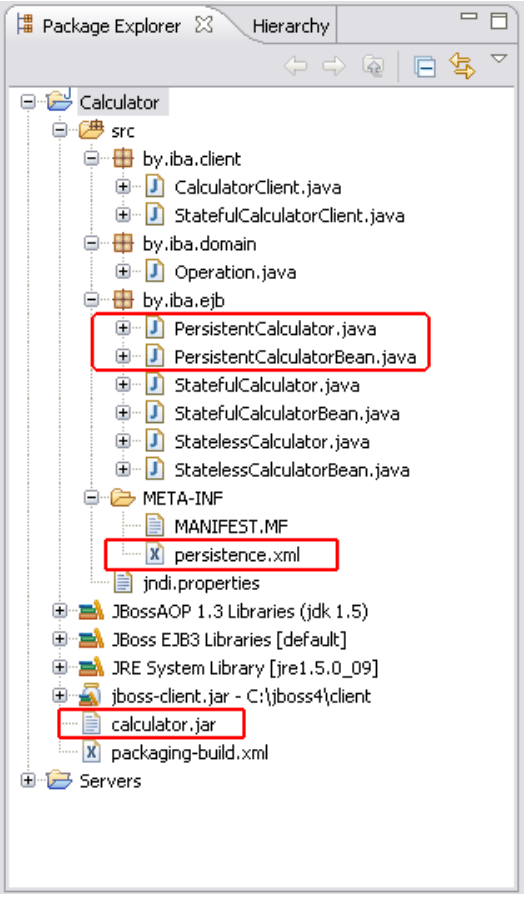


8. Right-click on the Calculator project and select **Run Packaging**. The packaging will display its output in the console. The output should look like this:

9. Buildfile: C:\ejb3workspace\Calculator\packaging-build.xml
10. N65540:
11. [jar] Building jar: C:\ejb3workspace\Calculator\calculator.jar

12. _packaging_generation_:
13. BUILD SUCCESSFUL
14. Total time: 1 second

15. After the execution, you should have a project that looks like this:



The contents of the calculator.jar should be as follows:

```
C:\ejb3workspace\Calculator>jar tvf calculator.jar
     0 Wed Nov 15 12:28:52 EET 2006 META-INF/
   106 Wed Nov 15 12:28:50 EET 2006 META-INF/MANIFEST.MF
   286 Wed Nov 15 11:10:48 EET 2006 META-INF/persistence.xml
     0 Mon Nov 06 17:17:18 EET 2006 by/
     0 Tue Nov 14 14:15:52 EET 2006 by/iba/
     0 Wed Nov 15 09:42:44 EET 2006 by/iba/ejb/
   409 Wed Nov 15 09:46:04 EET 2006 by/iba/ejb/PersistentCalculator.class
  4067 Wed Nov 15 12:09:54 EET 2006 by/iba/ejb/PersistentCalculatorBean.class
   305 Mon Nov 06 17:17:18 EET 2006 by/iba/ejb/StatefulCalculator.class
  1800 Mon Nov 06 17:17:18 EET 2006 by/iba/ejb/StatefulCalculatorBean.class
   268 Mon Nov 06 17:17:18 EET 2006 by/iba/ejb/StatelessCalculator.class
   907 Mon Nov 06 17:17:18 EET 2006 by/iba/ejb/StatelessCalculatorBean.class
     0 Tue Nov 14 14:15:54 EET 2006 by/iba/domain/
  2368 Wed Nov 15 11:50:22 EET 2006 by/iba/domain/Operation.class
```

16. OPTIONALLY: Undeploy the previous calculator.jar EJB module as described here: Undeploying An EJB JAR file.

17. In the **Project Explorer** View right click the calculator.jar file and select **Run As > Run on Server** context menu.

18. In the **Console** View, you should see some deployment activity. The EJB JAR is now deployed:

19. 12:31:07,162 INFO [JmxKernelAbstraction] installing MBean: jboss.j2ee:jar=calculator.jar,name=PersistentCalculatorBean,service=EJB3 with dependencies:
20. 12:31:07,162 INFO [JmxKernelAbstraction] jboss.j2ee:jar=calculator.jar,name=StatelessCalculatorBean,service=EJB3
21. 12:31:07,162 INFO [JmxKernelAbstraction] persistence.units:jar=calculator.jar,unitName=calculator
22. 12:31:07,162 INFO [JmxKernelAbstraction] installing MBean: jboss.j2ee:jar=calculator.jar,name=StatefulCalculatorBean,service=EJB3 with dependencies:
23. 12:31:07,162 INFO [JmxKernelAbstraction] jboss.j2ee:jar=calculator.jar,name=StatelessCalculatorBean,service=EJB3
24. 12:31:07,162 INFO [JmxKernelAbstraction] installing MBean: jboss.j2ee:jar=calculator.jar,name=StatelessCalculatorBean,service=EJB3 with dependencies:
25. 12:31:07,178 INFO [EJBContainer] STARTED EJB: by.iba.ejb.StatelessCalculatorBean ejbName: StatelessCalculatorBean
26. 12:31:07,240 INFO [EJBContainer] STARTED EJB: by.iba.ejb.PersistentCalculatorBean ejbName: PersistentCalculatorBean
27. 12:31:07,271 INFO [EJBContainer] STARTED EJB: by.iba.ejb.StatefulCalculatorBean ejbName: StatefulCalculatorBean
28. 12:31:07,287 INFO [SimpleStatefulCache] Initializing SimpleStatefulCache with maxSize: 100000 timeout: 300 for jboss.j2ee:jar=calculator.jar,name=StatefulCalculatorBean,service=EJB3
29. 12:31:07,287 INFO [EJB3Deployer] Deployed: file:/C:/jboss4/server/default/deploy/calculator.jar

1Writing A Standalone Java Test Client for Entity Object

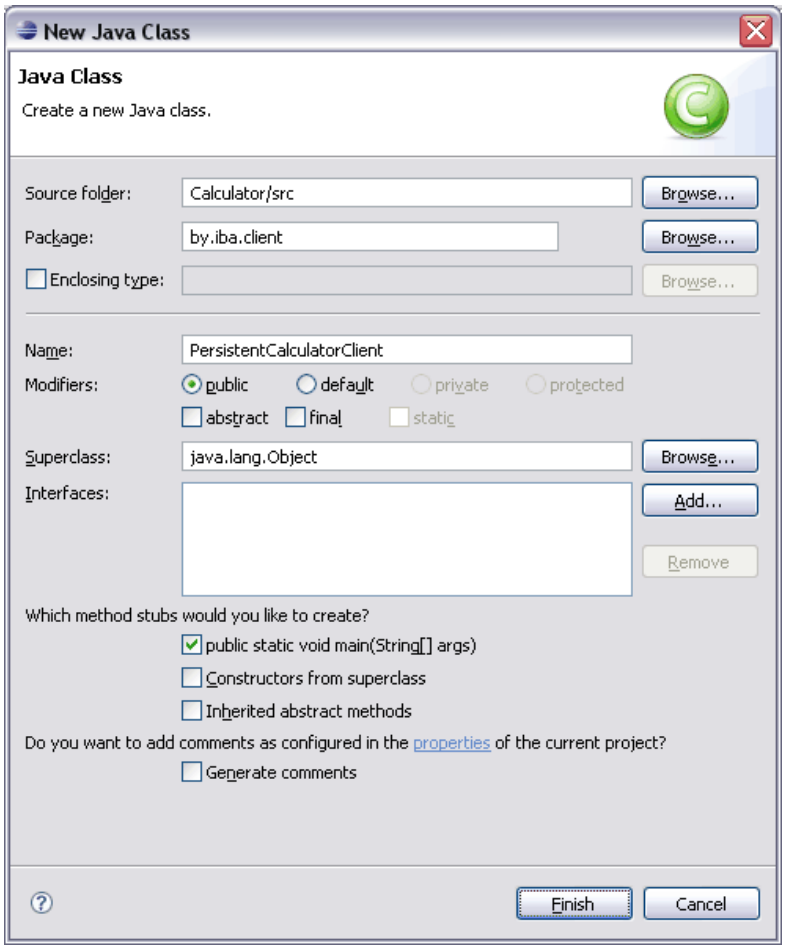
1. In the **Package Explorer** View make sure the Calculator project is selected.

2. Create a new Java class. Open menu **File > New > Other...** and choose **Java > Class**.

3. The package will be by.iba.client and the class name PersistentCalculatorClient.

   Leave the default options selected and be sure that public static void main(...) method is checked.

   Click the **Finish** button.



4. Add the following code to the public static void main method body:

```
5.      try {
6.              Context jndiContext = new InitialContext();
7.              Object ref = jndiContext.lookup("PersistentCalculatorBean/remote");
8.              PersistentCalculator calc = (PersistentCalculator) PortableRemoteObject
9.                      .narrow(ref, PersistentCalculator.class);
10.
11.             System.out.println("4 + 3 = " + calc.add(4, 3));
12.             System.out.println("4 - 3 = " + calc.subtract(4, 3));
13.             System.out.println("4 * 3 = " + calc.multiply(4, 3));
14.             System.out.println("4 / 3 = " + calc.divide(4, 3));
15.             System.out.println("6 * 3 = " + calc.multiply(6, 3));
16.             System.out.println("7 / 3 = " + calc.divide(7, 3));
17.
18.             System.out.println("   History before cleanup:");
19.             System.out.println(calc.getHistory());
20.
21.             calc.clearHistory();
22.
23.             System.out.println("   History after cleanup:");
24.             System.out.println(calc.getHistory());
25.
```

```
26.    } catch (NamingException ne) {
27.                ne.printStackTrace();
28.    }
```

NOTE: To resolve imports problems press **Shift + Ctrl + O**. Press **Ctrl + S** to save the Java class.

29. Run the Java class by selecting menu **Run > Run As > Java Application**.

    NOTE: Make sure the PersistentCalculatorClient.java is selected in **Package Explorer** View.

    You should see the following result in the Java client's **Console** View:

```
4 + 3 = 7.0
4 - 3 = 1.0
4 * 3 = 12.0
4 / 3 = 1.3333333333333333
6 * 3 = 18.0
7 / 3 = 2.3333333333333335
  History before cleanup:
[
2006-11-15 13:32:52.159 : 7.0 divide 3.0 is 2.3333333333333335,
2006-11-15 13:32:52.143 : 6.0 multiply 3.0 is 18.0,
2006-11-15 13:32:52.143 : 4.0 divide 3.0 is 1.3333333333333333,
2006-11-15 13:32:52.127 : 4.0 multiply 3.0 is 12.0,
2006-11-15 13:32:52.112 : 4.0 subtract 3.0 is 1.0]
  History after cleanup:
[]
```

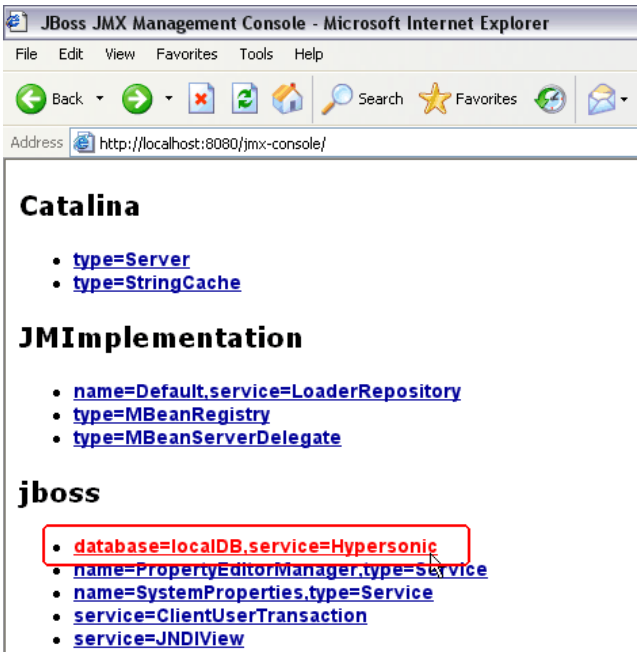
30. **OPTIONALLY**: You can see the DB table created by JBoss for the Entity Object.

    a. Comment out the history cleanup code line in the client:

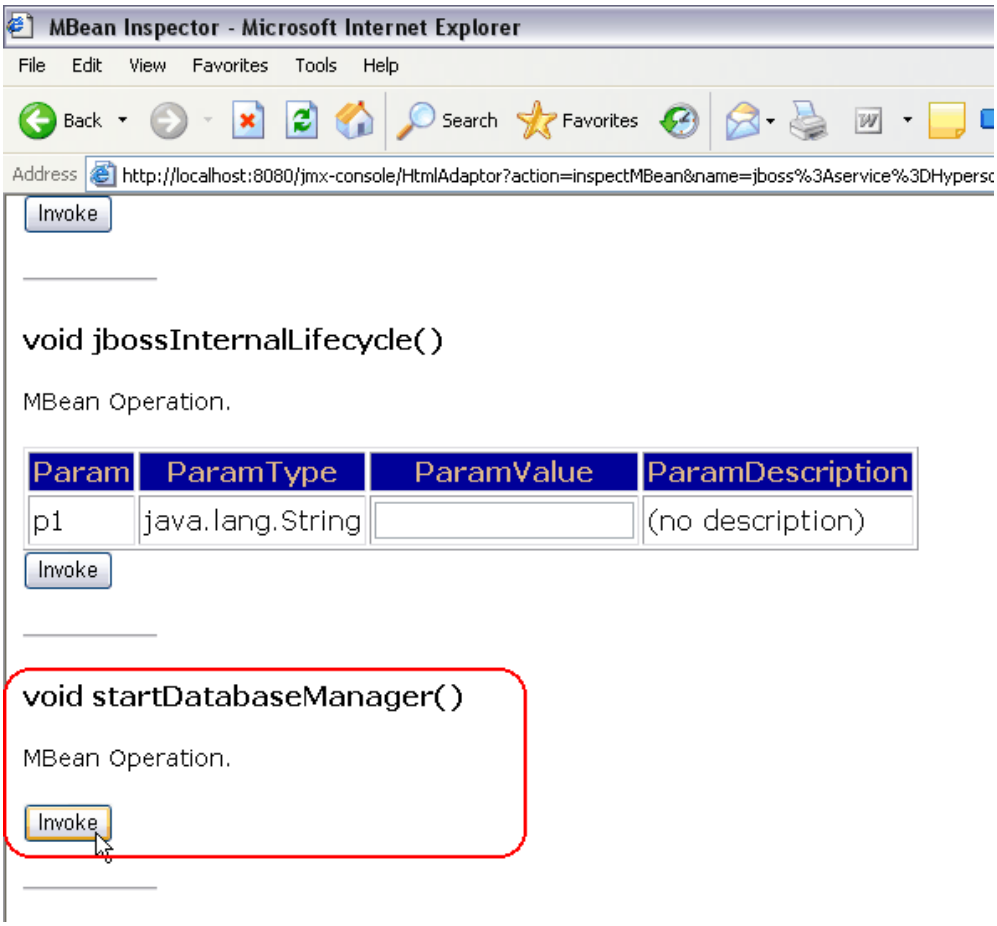
```
b.        System.out.println("    History before cleanup:");
c.        System.out.println(calc.getHistory());
d.
e.        // calc.clearHistory();
f.        ...
```

    g. Re-run PersistentCalculatorClient client (to populate the table).

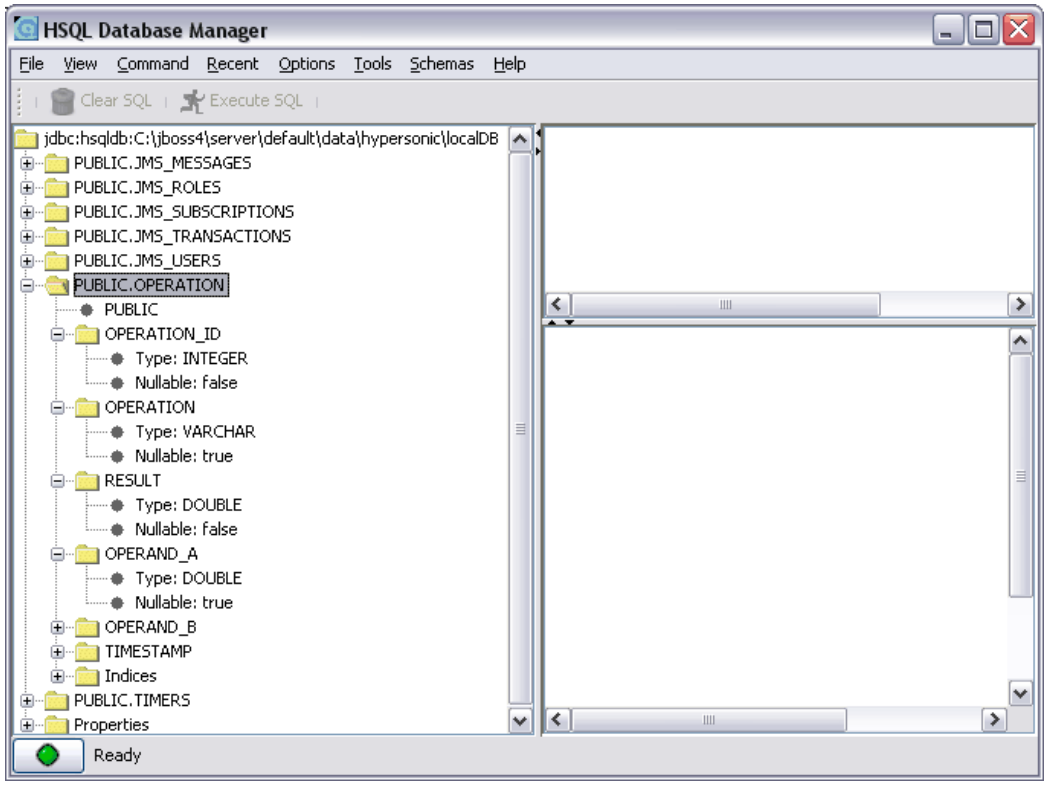
    h. Open a browser, go to **JMX Console** http://localhost:8080/jmx-console:



i. Go to Hypersonic SQL service and click on startDatabaseManager()

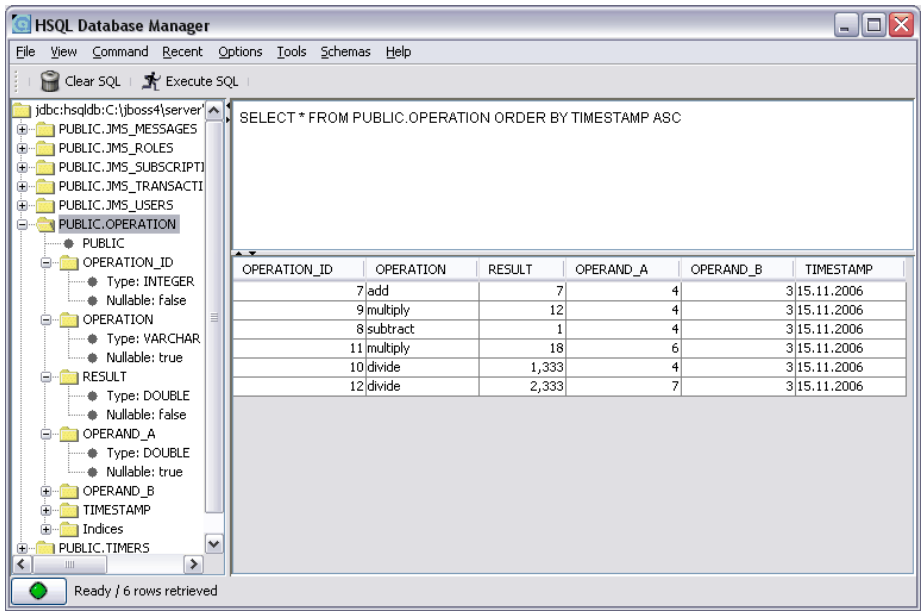


j. The **HSQL Database Manager** will come up, displaying the database table OPERATION.



k. If you execute an SQL query on the table, the data from the OPERATION table will be displayed as shown below:

l. SELECT * FROM PUBLIC.OPERATION ORDER BY TIMESTAMP ASC



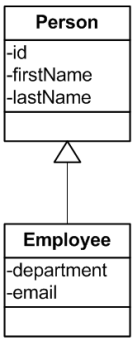
Congratulations ! You have succesfully tested EJB 3.0 application with Entity Object !!!

1Appendix E. EJB 3.0 Entity Inheritance Hierarchies

1Overview

JPA provides declarative support for three main implementation strategies that dictate how the entities in a hierarchy map to underlying tables.

To illustrate how these three strategies are manifested in code, we will use the sample entity hierarchy that demonstrates inheritance:



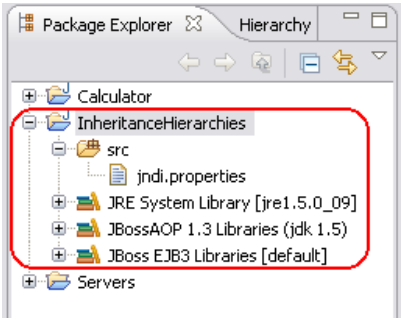
The Person entity serves as the root class in an entity hierarchy, and is extended by the Employee entity.

NOTE: The source code for this tutorial can be downloaded as a zip archive.

1Creating The EJB 3.0 Project

Create InheritanceHierarchies EJB 3.0 project as described in Creating The Project section.

New InheritanceHierarchies EJB 3.0 project should look like this:



1Creating Entity Objects for SINGLE_TABLE Inheritance Type

Inheritance strategy is defined by the InheritanceType enum:

```
public enum InheritanceType {
        SINGLE_TABLE,
        JOINED,
        TABLE_PER_CLASS
}
```

- SINGLE_TABLE: Single table per class inheritance hierarchy. This is the default strategy. The entity hierarchy is essentially flattened into the sum of its fields, and these fields are mapped down to a single table.
- JOINED: Common base table, with joined subclass tables. In this approach, each entity in the hierarchy maps to its own dedicated table that maps only the fields declared on that entity. The root entity in the hierarchy is known as the base table, and the tables for all other entities in the hierarchy join with the base table.
- TABLE_PER_CLASS: Single table per outermost concrete entity class. This strategy maps each leaf (i.e., outermost, concrete) entity to its own dedicated table. Each such leaf entity branch is flattened, combining its declared fields with the declared fields on all of its super-entities, and the sum of these fields is mapped onto its table.

The default inheritance mapping strategy is SINGLE_TABLE in which all the entities in the class hierarchy map onto a single table. A dedicated **discriminator** column on this table identifies the specific entity type associated with each row, and each entity in the hierarchy is given a unique value to store in this column. By default, the discriminator value for an entity is its entity name, although an entity may override this value using the @DiscriminatorValue annotation. This approach performs well for querying, since only a single table is involved.

Create Person EJB 3.0 Entity as described in Creating An Entity Object (POJO) section:

- Source folder: InheritanceHierarchies/src
- Package: by.iba.ejb3.singletable
- Name: Person

Copy-paste code below in the class body:

```
package by.iba.ejb3.singletable;

import javax.persistence.Column;
import javax.persistence.DiscriminatorColumn;
```

```
import javax.persistence.DiscriminatorType;
import javax.persistence.DiscriminatorValue;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.Inheritance;
import javax.persistence.InheritanceType;
import javax.persistence.Table;

@Entity(name="Person_SINGLE_TABLE")
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name = "TYPE", discriminatorType=DiscriminatorType.STRING)
@DiscriminatorValue("PERSON")
@Table(name = "PERSON_SINGLE_TABLE")
public class Person {

	@Id
	@Column(nullable = false)
	@GeneratedValue
	private Long id;

	@Column(name = "FIRST_NAME")
	private String firstName;

	@Column(name = "LAST_NAME")
	private String lastName;

	public String getFirstName() {
		return firstName;
	}

	public void setFirstName(String firstName) {
		this.firstName = firstName;
	}

	public Long getId() {
		return id;
	}

	public void setId(Long id) {
		this.id = id;
	}

	public String getLastName() {
		return lastName;
	}

	public void setLastName(String lastName) {
		this.lastName = lastName;
	}
}
```

By default, the persistence manager looks for a column named DTYPE in the root entity's table (PERSON, in this case). In our example, we have named the discriminator column TYPE, so we explicitly annotate this setting, using the @DiscriminatorColumn(name = "TYPE") annotation. Were we to use a column named DTYPE, we could have skipped this annotation altogether and accepted the default value.

Now create Employee EJB 3.0 Entity as described in Creating An Entity Object (POJO) section:

- Source folder: InheritanceHierarchies/src
- Package: by.iba.ejb3.singletable
- Name: Employee
- Superclass: by.iba.ejb3.singletable.Person

Copy-paste code below in the class body:

```
package by.iba.ejb3.singletable;

import javax.persistence.DiscriminatorValue;
import javax.persistence.Entity;

@Entity(name="Employee_SINGLE_TABLE")
@DiscriminatorValue("EMPLOYEE")
public class Employee extends Person {

	private String department;

	private String email;

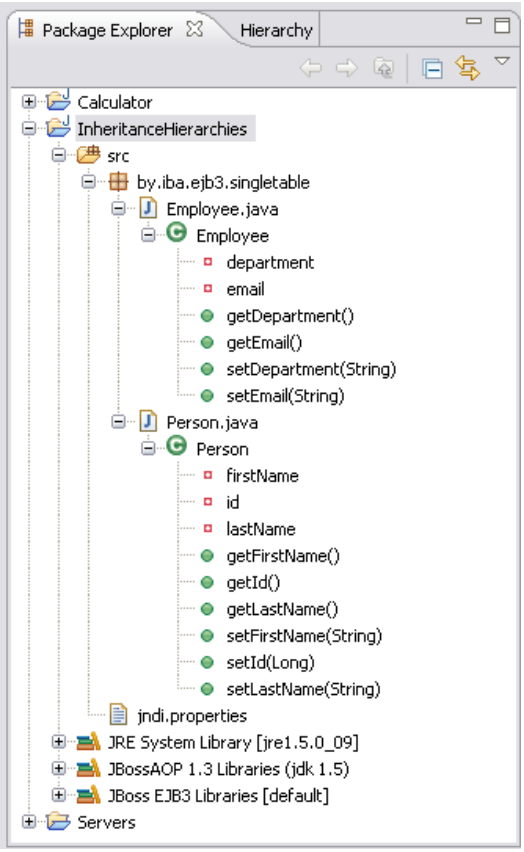
	public String getDepartment() {
		return department;
	}

	public void setDepartment(String department) {
		this.department = department;
	}

	public String getEmail() {
		return email;
	}

	public void setEmail(String email) {
		this.email = email;
	}
}
```

Now InheritanceHierarchies EJB 3.0 project should look like this:



1Creating Entity Objects for JOINED Inheritance Type

In the JOINED strategy, each entity in the hierarchy introduces its own table but only to map fields that are declared on that entity type. The root entity in the hierarchy maps to a root table that defines the primary key structure to be used by all tables in the entity hierarchy, as well as the discriminator column and optionally a version column. Each of the other tables in the hierarchy defines a primary key that matches the root table's primary key, and optionally adds a foreign key constraint from their ID column(s) to the root table's ID column(s). The non-root tables don't hold a discriminator type or version columns. Since each entity instance in the hierarchy is represented by a virtual row that spans its own table as well as the tables for all of its super-entities, it eventually joins with a row in the root table that captures this discriminator type and version information. Querying all the fields of any type requires a join across all the tables in the supertype hierarchy.

Create Person EJB 3.0 Entity as described in Creating An Entity Object (POJO) section:

- Source folder: InheritanceHierarchies/src
- Package: by.iba.ejb3.joined

- Name: Person

Copy-paste code below in the class body:

```
package by.iba.ejb3.joined;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.Inheritance;
import javax.persistence.InheritanceType;
import javax.persistence.Table;

@Entity(name="Person_JOINED")
@Inheritance(strategy = InheritanceType.JOINED)
@Table(name = "PERSON_JOINED")
public class Person {

	@Id
	@Column(nullable = false)
	@GeneratedValue
	private Long id;

	@Column(name = "FIRST_NAME")
	private String firstName;

	@Column(name = "LAST_NAME")
	private String lastName;

	public String getFirstName() {
		return firstName;
	}

	public void setFirstName(String firstName) {
		this.firstName = firstName;
	}

	public Long getId() {
		return id;
	}

	public void setId(Long id) {
		this.id = id;
	}

	public String getLastName() {
		return lastName;
	}

	public void setLastName(String lastName) {
		this.lastName = lastName;
	}
```

```
}
```

Now create Employee EJB 3.0 Entity as described in Creating An Entity Object (POJO) section:

- Source folder: InheritanceHierarchies/src
- Package: by.iba.ejb3.joined
- Name: Employee
- Superclass: by.iba.ejb3.joined.Person

Copy-paste code below in the class body:

```
package by.iba.ejb3.joined;

import javax.persistence.Entity;
import javax.persistence.Table;

@Entity(name="Employee_JOINED")
@Table(name = "EMPLOYEE_JOINED")
public class Employee extends Person {

	private String department;

	private String email;

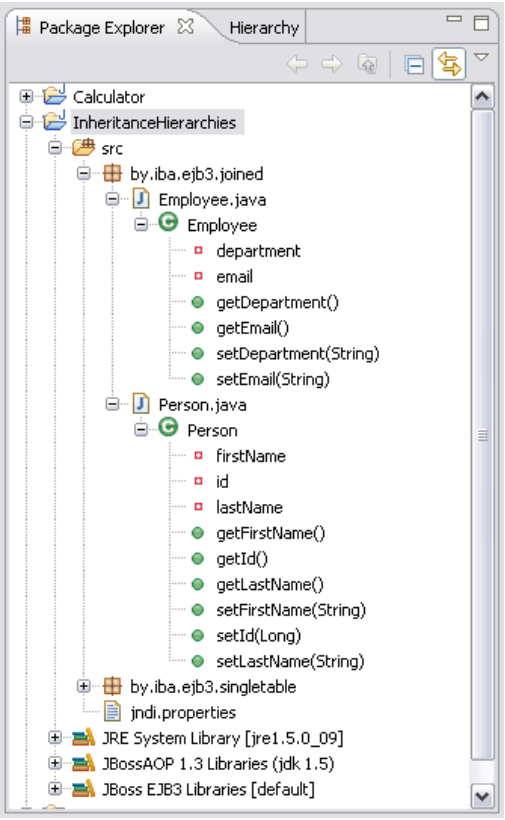
	public String getDepartment() {
		return department;
	}

	public void setDepartment(String department) {
		this.department = department;
	}

	public String getEmail() {
		return email;
	}

	public void setEmail(String email) {
		this.email = email;
	}
}
```

Now InheritanceHierarchies EJB 3.0 project should look like this:



1Creating Entity Objects for TABLE_PER_CLASS Inheritance Type

This inheritance mapping option maps each outermost **concrete** entity to its own dedicated table. Each table maps all of the fields in that entity's entire type hierarchy; since there's no shared table, no columns are shared. The only table structure requirement is that all tables must share a common primary key structure, meaning that the name(s) and type(s) of the primary key column(s) must match across all tables in the hierarchy.

The tables are required to share nothing in common except the structure of their primary key, and since the table implicitly identifies the entity type, **no discriminator column is required**. With this inheritance mapping strategy, **only concrete entities require tables**.

Create Person EJB 3.0 Entity as described in Creating An Entity Object (POJO) section:

- Source folder: InheritanceHierarchies/src
- Package: by.iba.ejb3.tableperclass
- Name: Person

Copy-paste code below in the class body:

```
package by.iba.ejb3.tableperclass;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Inheritance;
import javax.persistence.InheritanceType;
import javax.persistence.Table;

@Entity(name="Person_TABLE_PER_CLASS")
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
@Table(name = "PERSON_TABLE_PER_CLASS")
public class Person {

	@Id
	@Column(nullable = false)
	@GeneratedValue(strategy=GenerationType.TABLE)
	private Long id;

	@Column(name = "FIRST_NAME")
	private String firstName;

	@Column(name = "LAST_NAME")
	private String lastName;

	public String getFirstName() {
		return firstName;
	}

	public void setFirstName(String firstName) {
		this.firstName = firstName;
	}

	public Long getId() {
		return id;
	}

	public void setId(Long id) {
		this.id = id;
	}

	public String getLastName() {
		return lastName;
	}

	public void setLastName(String lastName) {
		this.lastName = lastName;
	}
}
```

Now create Employee EJB 3.0 Entity as described in Creating An Entity Object (POJO) section:

- Source folder: InheritanceHierarchies/src
- Package: by.iba.ejb3.tableperclass
- Name: Employee
- Superclass: by.iba.ejb3.tableperclass.Person

Copy-paste code below in the class body:

```
package by.iba.ejb3.tableperclass;

import javax.persistence.Entity;
import javax.persistence.Table;

@Entity(name="Employee_TABLE_PER_CLASS")
@Table(name = "EMPLOYEE_TABLE_PER_CLASS")
public class Employee extends Person {

	private String department;

	private String email;

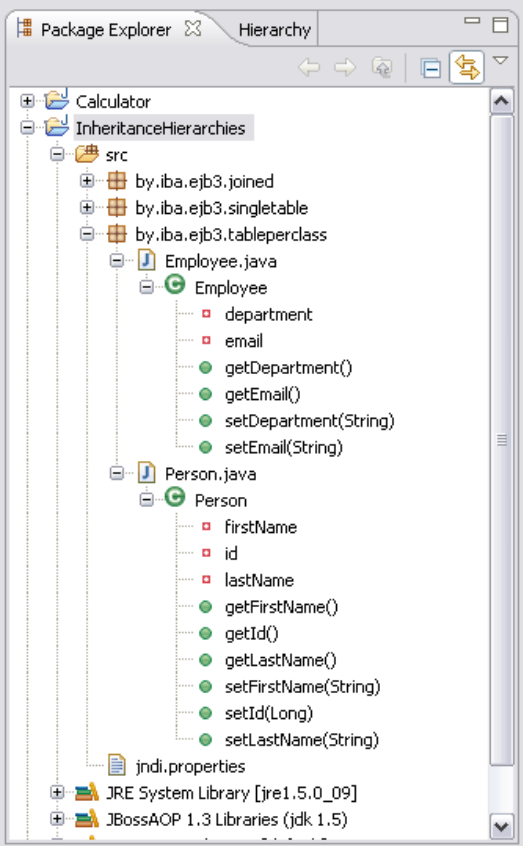
	public String getDepartment() {
		return department;
	}

	public void setDepartment(String department) {
		this.department = department;
	}

	public String getEmail() {
		return email;
	}

	public void setEmail(String email) {
		this.email = email;
	}
}
```

Now InheritanceHierarchies EJB 3.0 project should look like this:



1Configuring Persistence Unit Definition

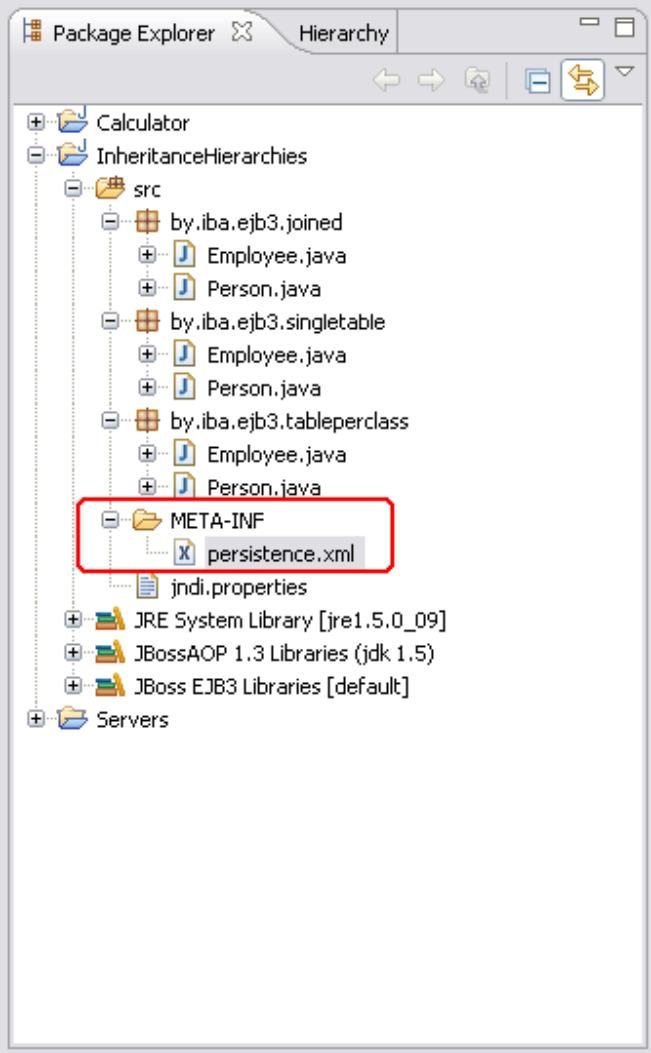
Create persistence unit definition as described in Configuring Persistence Unit Definition section:

Copy-paste the following content in the InheritanceHierarchies/src/META-INF/persistence.xml file :

```
<?xml version="1.0" encoding="UTF-8"?>

<persistence>
	<persistence-unit name="inheritance">
		<jta-data-source>java:/DefaultDS</jta-data-source>
		<properties>
			<property name="hibernate.hbm2ddl.auto" value="create-drop" />
		</properties>
	</persistence-unit>
</persistence>
```

Now InheritanceHierarchies EJB 3.0 project should look like this:



1Creating A Stateless EJB

New session bean will be used to create and persist entities, later we will explore tables content for the created entities.

Create new stateless session EJB as described in Use An Entity Object (POJO) section.

Source folder: InheritanceHierarchies/src

Session Bean Type: Stateless

Bean Package: by.iba.ejb3.session

Bean Name: Manager

Open Manager.java file (Remote [Business] Interface) and add the following code:

```
public void populate();
```

Open ManagerBean.java file (Bean Class) and add the following code:

```
@PersistenceContext(unitName = "inheritance")
private EntityManager manager;

public void populate() {
        populateJoined();
        populateSingleTable();
        populateTablePerClass();
}

private void populateJoined() {
        by.iba.ejb3.joined.Person p = new by.iba.ejb3.joined.Person();
        p.setFirstName("Volha");
        p.setLastName("Zaikina");

        by.iba.ejb3.joined.Employee e = new by.iba.ejb3.joined.Employee();
        e.setFirstName("Mikalai");
        e.setLastName("Zaikin");
        e.setDepartment("IS");
        e.setEmail("nzaikin[at]iba.by");

        manager.persist(p);
        manager.persist(e);
}

private void populateSingleTable() {
        by.iba.ejb3.singletable.Person p = new by.iba.ejb3.singletable.Person();
        p.setFirstName("Volha");
        p.setLastName("Zaikina");

        by.iba.ejb3.singletable.Employee e = new by.iba.ejb3.singletable.Employee();
        e.setFirstName("Mikalai");
        e.setLastName("Zaikin");
        e.setDepartment("IS");
        e.setEmail("nzaikin[at]iba.by");

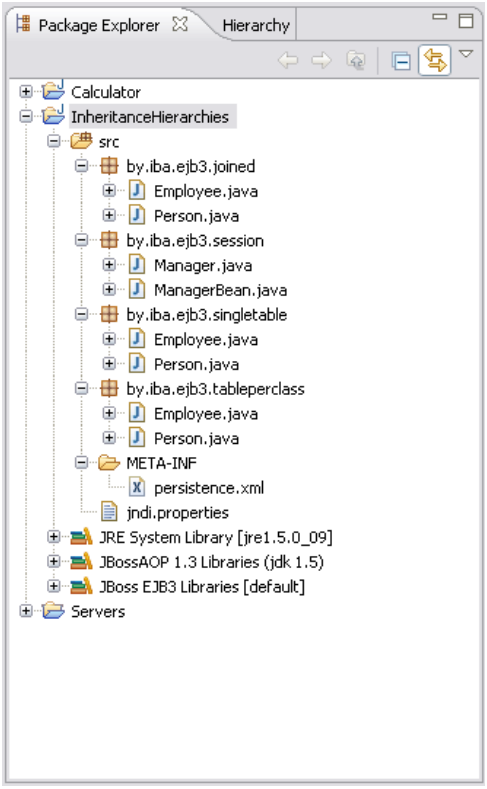
        manager.persist(p);
        manager.persist(e);
}

private void populateTablePerClass() {
        by.iba.ejb3.tableperclass.Person p = new by.iba.ejb3.tableperclass.Person();
        p.setFirstName("Volha");
        p.setLastName("Zaikina");

        by.iba.ejb3.tableperclass.Employee e = new by.iba.ejb3.tableperclass.Employee();
        e.setFirstName("Mikalai");
        e.setLastName("Zaikin");
        e.setDepartment("IS");
        e.setEmail("nzaikin[at]iba.by");
```

```
            manager.persist(p);
            manager.persist(e);
    }
```

Now InheritanceHierarchies EJB 3.0 project should look like this:



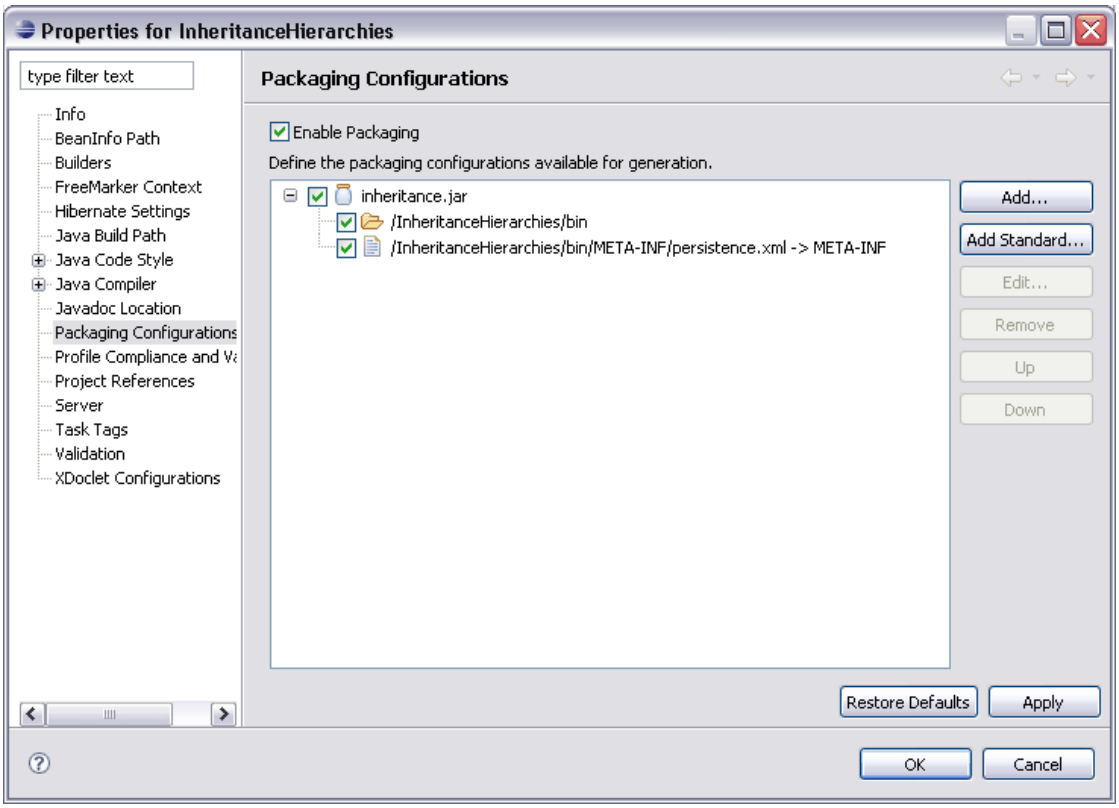
1Packaging An EJB JAR File

Package InheritanceHierarchies project in EJB JAR file as described in Packaging A Stateless EJB and Packaging Persistence Unit Definition sections:

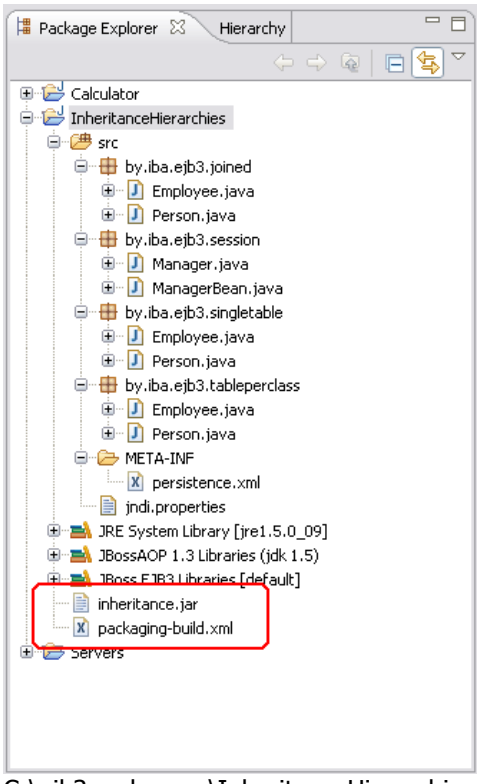
- Select the archive name: inheritance.jar
- Add folder: /InheritanceHierarchies/bin

  Includes: by/iba/ejb3/**/*.class
- Add file: /InheritanceHierarchies/bin/META-INF/persistence.xml

  Prefix: META-INF



Run packaging. New inheritance.jar file will be created in the InheritanceHierarchies folder:



C:\ejb3workspace\InheritanceHierarchies>jar -tf inheritance.jar

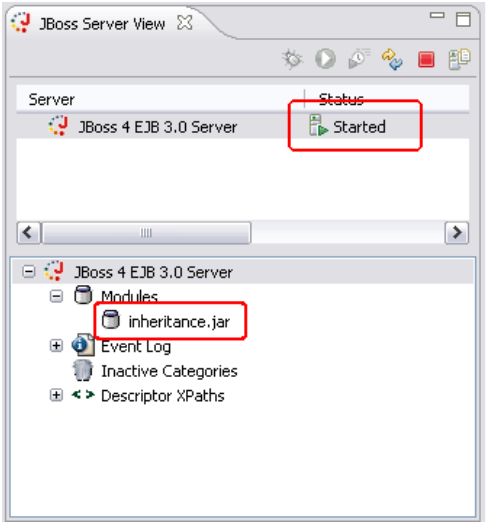
META-INF/
META-INF/MANIFEST.MF
META-INF/persistence.xml
by/
by/iba/
by/iba/ejb3/
by/iba/ejb3/joined/
by/iba/ejb3/joined/Employee.class
by/iba/ejb3/joined/Person.class
by/iba/ejb3/session/
by/iba/ejb3/session/Manager.class
by/iba/ejb3/session/ManagerBean.class
by/iba/ejb3/singletable/
by/iba/ejb3/singletable/Employee.class
by/iba/ejb3/singletable/Person.class
by/iba/ejb3/tableperclass/
by/iba/ejb3/tableperclass/Employee.class
by/iba/ejb3/tableperclass/Person.class

1Deploying An EJB JAR file

Deploy the inheritance.jar EJB JAR file as described in Deploying An EJB JAR file section:

23:48:03,703 INFO [AnnotationBinder] Binding entity from annotated class: by.iba.ejb3.joined.Person
23:48:03,703 INFO [EntityBinder] Bind entity by.iba.ejb3.joined.Person on table PERSON_JOINED
23:48:03,703 INFO [AnnotationBinder] Binding entity from annotated class: by.iba.ejb3.joined.Employee
23:48:03,703 INFO [EntityBinder] Bind entity by.iba.ejb3.joined.Employee on table EMPLOYEE_JOINED
23:48:03,703 INFO [AnnotationBinder] Binding entity from annotated class: by.iba.ejb3.singletable.Person
23:48:03,718 INFO [EntityBinder] Bind entity by.iba.ejb3.singletable.Person on table PERSON_SINGLE_TABLE
23:48:03,718 INFO [AnnotationBinder] Binding entity from annotated class: by.iba.ejb3.singletable.Employee
23:48:03,734 INFO [AnnotationBinder] Binding entity from annotated class: by.iba.ejb3.tableperclass.Person
23:48:03,734 INFO [EntityBinder] Bind entity by.iba.ejb3.tableperclass.Person on table PERSON_TABLE_PER_CLASS
23:48:03,734 INFO [AnnotationBinder] Binding entity from annotated class: by.iba.ejb3.tableperclass.Employee
23:48:03,734 INFO [EntityBinder] Bind entity by.iba.ejb3.tableperclass.Employee on table EMPLOYEE_TABLE_PER_CLASS

JBoss Server View should look like that:



1Writing And Running A Standalone Java Test Client (EJB 3.0 Entities Creating)

Create the InheritanceClient a standalone Java test client as described in Writing A Standalone Java Test Client section:

Source folder: InheritanceHierarchies/src

Package: by.iba.client

Name: InheritanceClient

Check the public static void main method checkbox.

Copy-paste the following code in the method:

```
try {
		Context jndiContext = new InitialContext();
		Object ref = jndiContext.lookup("ManagerBean/remote");
		Manager manager = (Manager) PortableRemoteObject.narrow(ref, Manager.class);
		manager.populate();
} catch (NamingException ne) {
		ne.printStackTrace();
}
```

NOTE: To resolve imports problems press **Shift + Ctrl + O**. Press **Ctrl + S** to save the Java class.

Run the Java class (as Java application). New entites had been created and persisted in database.

1Exploring Entity Inheritance Hierarchies in Database

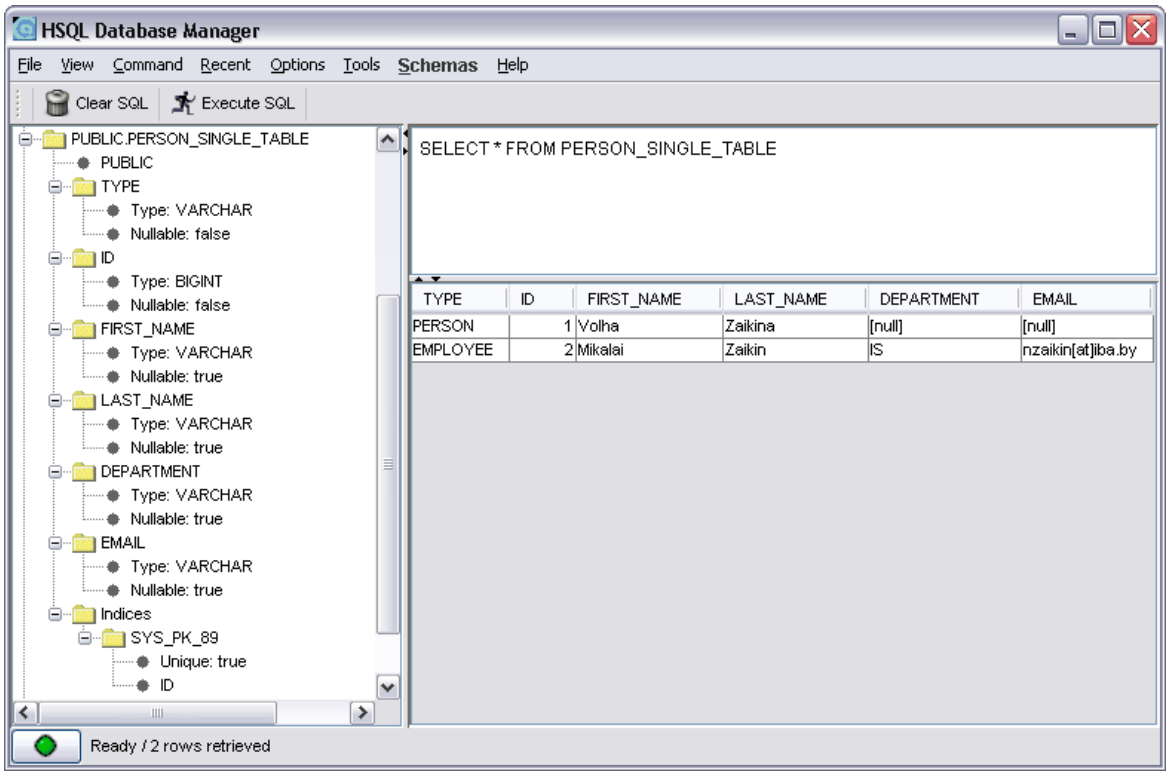
Open a browser, go to **JMX Console** http://localhost:8080/jmx-console.

Go to Hypersonic SQL service and click on startDatabaseManager() as described here: Writing A Standalone Java Test Client for Entity Object.

Explore table contents for different inheritance strategies:

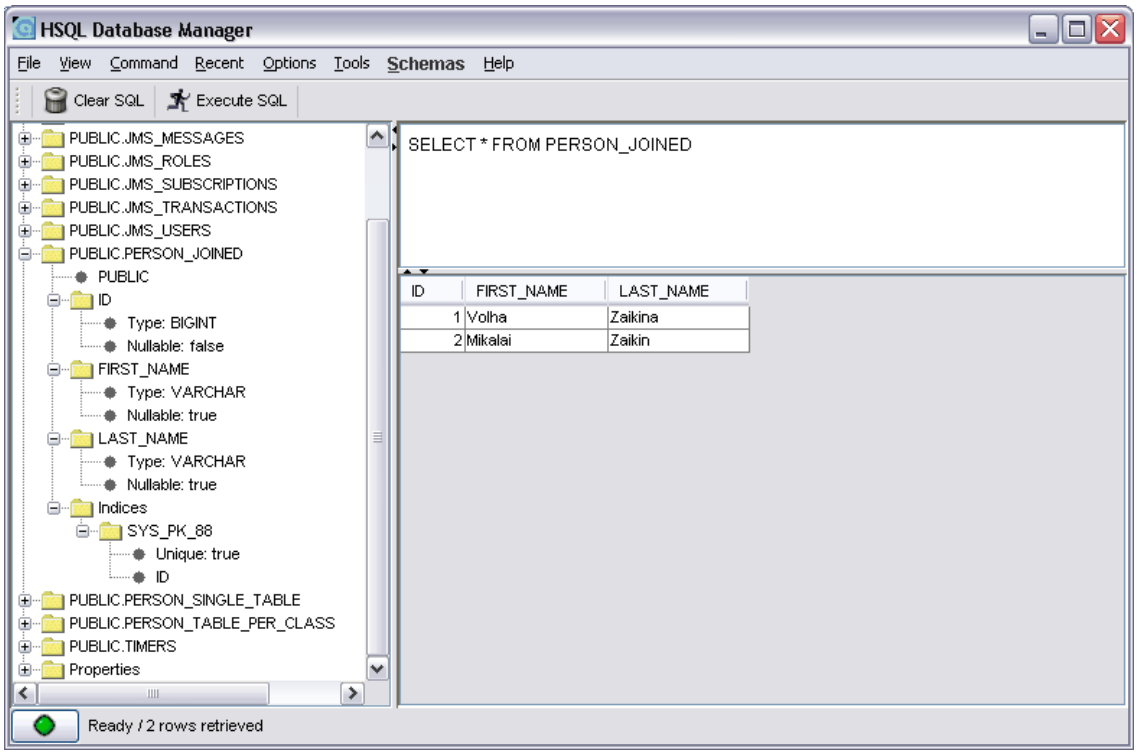
- SINGLE_TABLE Inheritance Strategy.

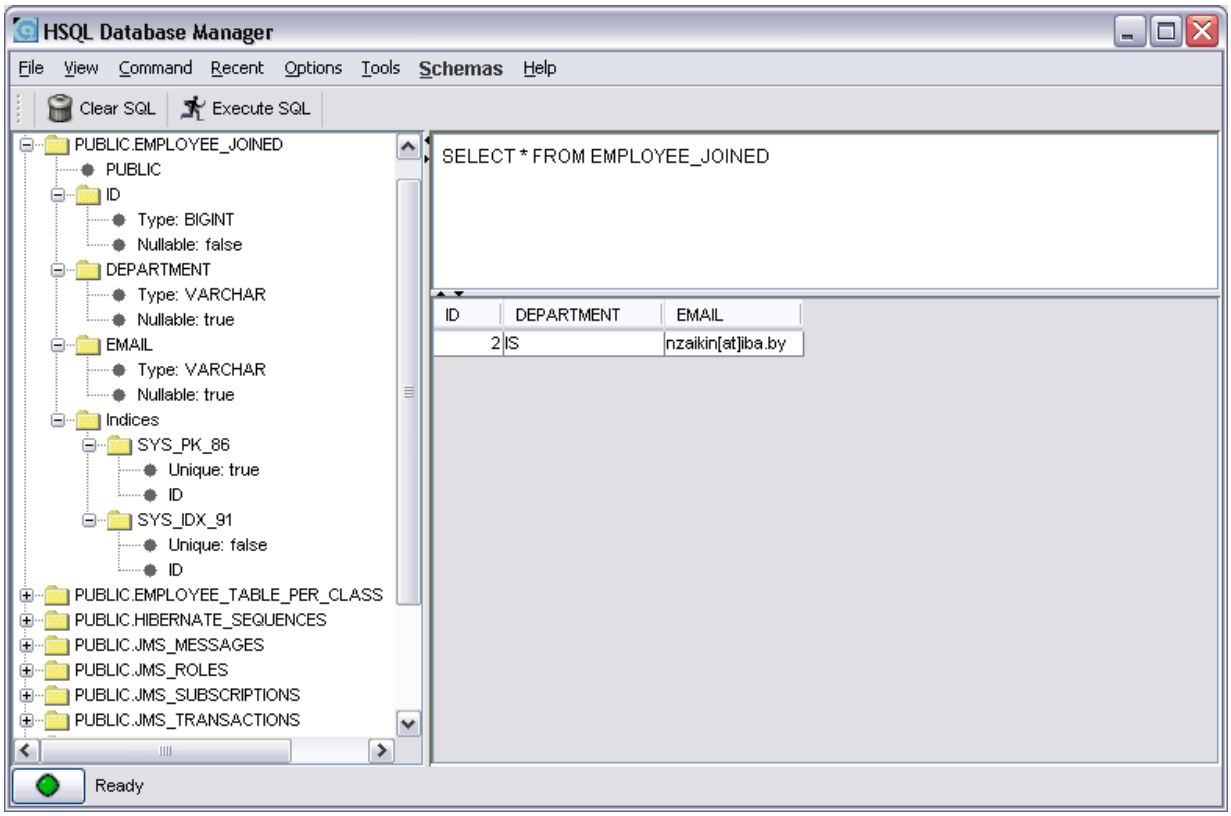
  Table definition and content:



- JOINED Inheritance Strategy.

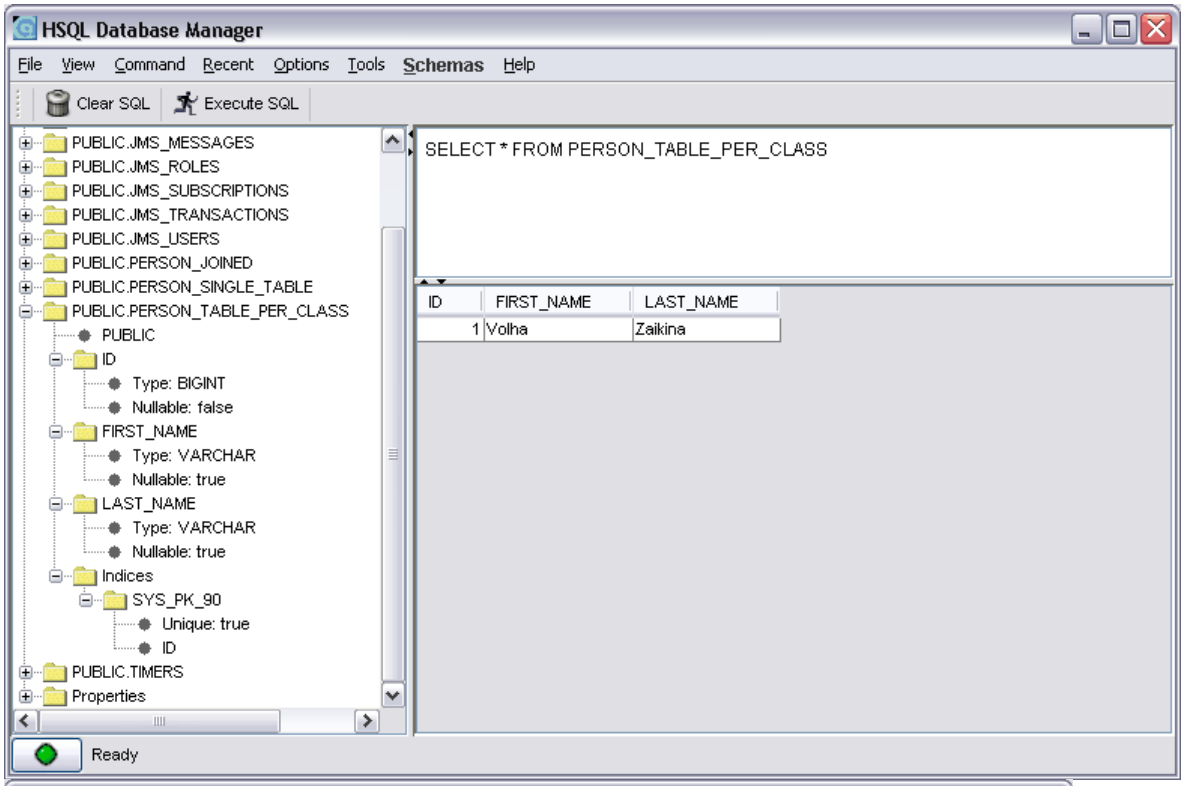
  Tables definition and content:

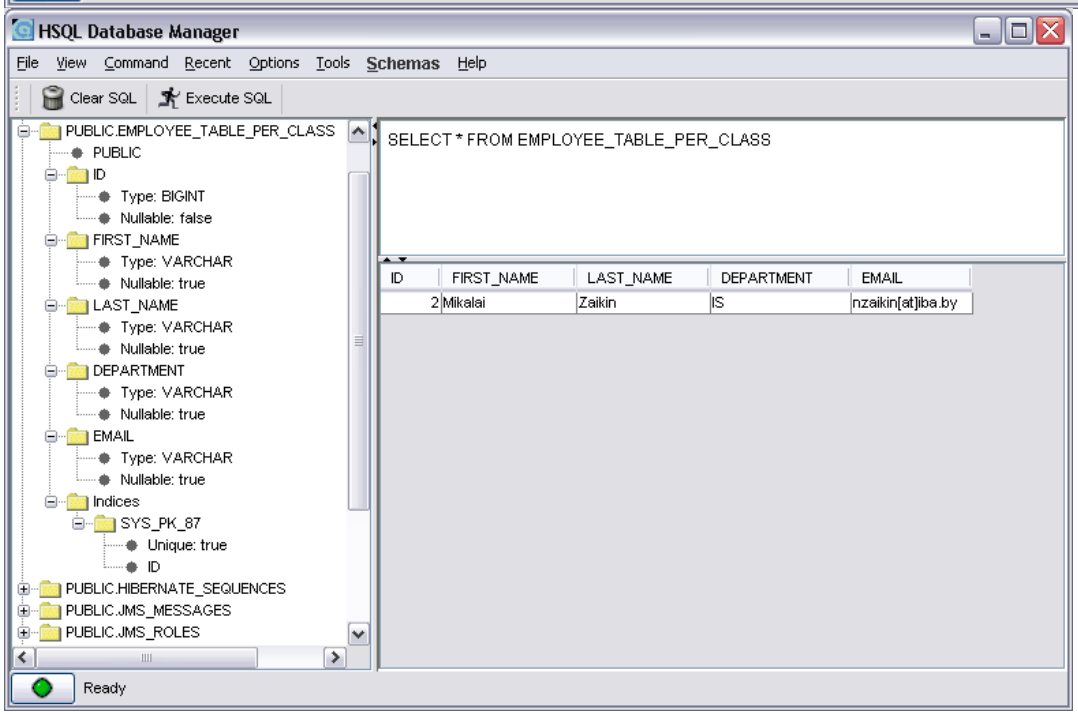




- TABLE_PER_CLASS Inheritance Strategy.

Tables definition and content:





Congratulations ! In this tutorial we explored the ability of EJB 3.0 Entities to handle inheritance in a standard object-oriented way and still easily map to a relational database.

1 Appendices

➢Bibliography

[EJB_3.0_CORE] *JSR-000220 Enterprise JavaBeans 3.0 Final Release (ejbcore)*. http://jcp.org/en/jsr/detail?id=220.

[EJB_3.0_PERSISTENCE] *JSR-000220 Enterprise JavaBeans 3.0 Final Release (persistence)*. http://jcp.org/en/jsr/detail?id=220.

[EJB_3.0_SIMPLIFIED] *JSR-000220 Enterprise JavaBeans 3.0 Final Release (simplified)*. http://jcp.org/en/jsr/detail?id=220.

[JAVARANCH_SCBCD] *JavaRanch.com forum for SCBCD Certification*. http://saloon.javaranch.com/cgi-bin/ubb/ultimatebb.cgi?ubb=forum&f=70.

[JEE_5_TUTORIAL] *The Java EE 5 Tutorial* . http://java.sun.com/javaee/5/docs/tutorial/doc/index.html.

[OREILLY_EJB_3.0] *Enterprise JavaBeans 3.0, Fifth Edition, by by Bill Burke and Richard Monson-Haefel, published by O'Reilly, 2006* .

[JAX_WS_2.0] *Java API for XML-Based Web Services (JAX-WS 2.0)* . http://jcp.org/en/jsr/detail?id=224.

[WS_METADATA] *Web Services Metadata for the Java Platform* . http://jcp.org/en/jsr/detail?id=181.