7

Enterprise JavaBeans and the EJB Container Model

CERTIFICATION OBJECTIVES

7.01	List the Required Classes/Interfaces That Must Be Provided for an Enterprise JavaBeans Component
7.02	Distinguish Between Session and Entity Beans
7.03	Recognize Appropriate Uses for Entity, Stateful Session, and Stateless Session Beans
7.04	Distinguish Between Stateful and Stateless Session Beans
7.05	State the Benefits and Costs of Container-Managed Persistence
7.06	State the Transactional Behavior in a Given Scenario for an Enterprise Bean Method with a Specified Transactional Deployment Descriptor
7.07	Given a Requirement Specification Detailing Security and Flexibility Needs, Identify Architectures That Would Fulfill Those Requirements
7.08	Identify Costs and Benefits of Using an Intermediate Data Access Object Between an Entity Bean and the Data Resource
7.09	State the Benefits of Bean Pooling in an EJB Container
7.10	State the Benefits of Passivation in an EJB container
7.11	Explain How the Enterprise JavaBeans Container Does Life Cycle Management and Has the Capability to Increase Scalability
✓	Two-Minute Drill
Q&A	Self Test

The Enterprise JavaBeans (EJB) specification is an industry initiative led and driven by Sun Microsystems with participation from many supporting vendors in the industry. Sun owns the process of defining, creating, and publishing the specification while ensuring the incorporation of input and feedback from the industry and the general public.

The EJB requirements enable communication with Java Platform Enterprise Edition (JEE) clients including JavaServer Pages (JSP), servlets, and application clients as well as with EJBs in other EJB containers. The goal of these features is to allow EJB invocations to work even when client components and EJBs are deployed in Java Platform EE products from different vendors. Support for interoperability among components includes transaction propagation, naming services, and security services.

The term *enterprise* implies that an application will be scalable, available, reliable, secure, transactional, and distributed. To provide these types of features, an enterprise application requires access to a variety of infrastructure services, such as distributed communication services, naming and directory services, transaction services, messaging services, data access and persistence services, and resource-sharing services.

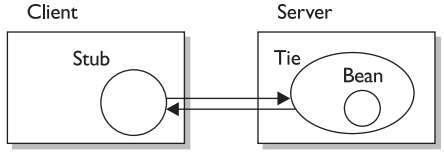
When a *distributed framework* is used, a client makes a call to what appears to be the interface of a business object. What it actually calls, however, is a *stub* that mimics the interface of that business object. This layer between clients and business objects is added because it is more practical to place stubs in the remote and distributed locations of clients than to place complete copies in the location of business objects.

In a distributed framework, the client calls a business method on a stub as if it were the real object. The stub then communicates this request to a *tie*. The tie calls the method on the real business object. A result is returned to the stub and the client (see Figure 7-1).

The Java Platform EE application programming interfaces (API) provide common interfaces that supply easy access to these underlying infrastructure services,

FIGURE 7-1

The distributed processing framework



regardless of their actual implementation. The Java Platform EE APIs and their vendor implementations provide additional services that are not supplied directly by the Java Virtual Machine (JVM), such as database access, transaction support, security enforcement, caching, and concurrency.

The following benefits are gained by adhering to the Java Platform EE standard:

- **Reusable application components** The productivity benefits of writing components in the Java programming language include component reuse and outsourcing, declarative customization (not programmatic), and the ability for the developer to focus on business logic only.
- **Portability** The portability characteristics of EJB components are made possible by Java Platform EE. This platform consists of several standard Java APIs that provide access to a core set of enterprise-class infrastructure services. These standardized APIs ensure that the Java code can be run on any vendor's application server.
- **Broad industry adoption** A wide selection of vendor tools and components allows choice and flexibility in server selection.
- **Application portability** Code is more than just platform independent; it is also middleware independent.
- **No vendor lock-in** Architecture decisions are made at deployment, not at the development phase. Interserver *portability* allows code to be deployed on any EJB server; interserver *scalability* allows servers to be transparently replaced to accommodate changing needs for service level, performance, or security.
- **Protection of IT investment** Wraps and integrates with the existing infrastructure, application, and data stores; is portable across multiple servers and databases; serves multilingual clients, such as browsers, Java technology, ActiveX, or Common Object Request Broker (CORBA) clients; EJB framework simplifies and enhances CORBA; and existing middleware solutions are being adapted by the well-established vendors to support the EJB framework via a thin portability layer.

JSR 220: Enterprise JavaBeans 3.0

Earlier versions of the Enterprise JavaBeans (EJB) specification have been criticized by many developers for its complexity. For this reason, the Java Community Process

(JCP) worked on Java Specification Request (JSR) 220 in order to improve the EJB architecture by reducing its complexity from the enterprise application developer's point of view.

JSR 220 (EJB 3.0) focused on the following goals:

- Definition of the Java language metadata annotations that can be used to annotate EJB applications, reducing the number of program classes and interfaces required and at eliminating the need for an EJB deployment descriptor.
- Where possible a "configuration by exception" approach is to be taken. Setup of many more defaults to specify common, expected behaviors and requirements on the EJB container. Encapsulation of environmental dependencies and JNDI access through the use of annotations, dependency injection mechanisms, and simple lookup mechanisms.
- Simplification of the enterprise bean types.
- Session beans: the elimination of the requirement for component and home interfaces. The required business interface for a session bean can now be a plain old Java interface (POJI) rather than an *EJBObject*, *EJBLocalObject*, or *java.rmi.Remote* interface.
- Simplification of entity persistence through the Java Persistence API. Support for lightweight domain modeling, including inheritance and polymorphism.
- Elimination of all required interfaces for persistent entities (entity beans).
- Annotations and XML deployment descriptor elements for the object/relational mapping of persistent entities (entity beans).
- A query language for Java Persistence that is an extension to EJB QL, with addition of projection, explicit inner and outer join operations, bulk update and delete, subqueries, and group-by. Addition of a dynamic query capability and support for native SQL queries.
- An interceptor facility for session beans and message-driven beans.
- Reduction of the requirements for usage of checked exceptions.
- Elimination of the requirement for the implementation of callback interfaces (This reduces the number of stub methods the developer has to add, e.g., `ejbLoad`, `ejbStore`, `ejbPassivate`, `ejbActivate` in a stateless session bean).

More on Java language Annotation feature

In Java enterprise versions prior to EJB 3.0, metadata was placed inside deployment descriptors (files containing XML). In Java EE 5 that implements the EJB 3.0 specification, metadata can now be placed in application code using a new Java language feature called *annotations*. This feature does not make the deployment descriptors obsolete; they continue to provide a way to add to or override metadata (annotations) at deployment time.

The Java annotation feature is an enhancement that was introduced in version 5 of the Java Platform Standard Edition (Java SE). Annotations are open ended in terms of functionality that can be offered, and they can enhance other areas within Java apart from deployment descriptors. Table 7-1 shows a list of many of the Java Language metadata (annotations) that can be used in EJB 3.0.

TABLE 7-1 Sample Java Language Annotations Used in EJB 3.0

Annotation	Description
`@Entity`	Specifies an entity bean component.
`@MessageDriven`	Specifies a message-driven bean component.
`@Stateful`	Used to annotate a class as a stateful session bean component.
`@Stateless`	Used to annotate a class as a stateless session bean component.
`@EJB`	Used on the client to reference the business interfaces of other beans and for EJB 2.1 or older beans, the home interfaces.
`@PostConstruct, @PreDestroy, @PostActivate, @PrePassivate`	All used to annotate a life cycle event callback method.
`@Resource`	Used for resource injection, examples are: `int`, `SessionContext`, `DataSource`, `QueueConnectionFactory`, `Queue`
`@RolesAllowed, @PermitAll, @DenyAll`	Declare method permissions.
`@RunAs`	The principal identity the enterprise bean will use when it makes calls.
`@Timeout`	Specifies a time-out method on a component that uses container-managed timer services.
`@TransactionAttribute`	Applies a transaction attribute to all methods of a business interface or to individual business methods on a bean class. Can be `MANDATORY`, `REQUIRED`, `REQUIRES_NEW`, `SUPPORTS`, `NOT_SUPPORTED`, `NEVER`.

(*Continued*)

TABLE 7-1 Sample Java Language Annotations Used in EJB 3.0

Annotation	Description
`@TransactionManagement`	Declares whether a bean will have container-managed (`CONTAINER`) or bean-managed (`BEAN`) transactions.
`@Column`	Specifies a mapped column for a persistent property or field.
`@Id`	Specifies primary key for an entity.
`@JoinColumn`	Specifies a mapped column for joining an entity association.
`@ManyToMany`	Defines a many-valued association with many-to-many multiplicity.
`@ManyToOne`	Defines a single-valued association to another entity class that has a many-to-one relationship.
`@OneToMany`	Defines a many-valued association with a one-to-many relationship.
`@OneToOne`	Defines a single-valued association to another entity that has a one-to-one relationship.
`@PersistenceContext`	Used to inject a dependency on an `EntityManager`.
`@PersistenceUnit`	Used to inject a dependency on an `EntityManagerFactory`.
`@SecondaryTable`	Used to specify a secondary table, which indicates that the data for the entity class is stored across multiple tables.
`@SecondaryTables`	Specifies multiple secondary tables for an entity.
`@Table`	Specifies the primary table for the entity.
`@UniqueConstraint`	Used to specify that a unique constraint is to be included in the generated DDL for a primary or secondary table.

For more information with respect to EJB 3.0 (JSR 220), look at the specification documents on the Java Community Process web site at

```
http://jcp.org/en/jsr/detail?id=220
```

In the download section of the preceding web site link, the following documents can be located:

1. Enterprise JavaBeans 3.0 Final Release (ejbcore) *ejb-3_0-fr-spec-ejbcore.pdf*
2. Enterprise JavaBeans 3.0 Final Release (persistence) *ejb-3_0-fr-spec-persistence.pdf*
3. Enterprise JavaBeans 3.0 Final Release (simplified) *ejb-3_0-fr-spec-simplified.pdf*

These specification documents are fairly large, but it makes sense to give them a read at some point before taking the exam.

This chapter covers two versions of the EJB specification, namely 3.0 and prior to 3.0. This book contains material that covers multiple versions of the examination, so it is important that you know which version you will be taking in order to concentrate on the correct answers for the exam objectives. Within the text of this chapter, when necessary we will differentiate between the two specifications by marking them with a notation of either EJB 3.0 or prior to EJB 3.0.

CERTIFICATION OBJECTIVE 7.01

List the Required Classes/Interfaces That Must Be Provided for an Enterprise JavaBeans Component

Here, we review the component architecture of EJBs. We also cover the required classes and interfaces for EJB, which include the home and remote interfaces, the XML deployment descriptor, the business logic (bean) class, and the context objects. While these names may or may not be meaningful to you at this point, you will soon understand how each of these pieces fits into the EJB component model.

Classes and Interfaces Prior to EJB 3.0

The components used to create and access EJBs facilitate the creation and execution of business logic on an enterprise system. Each EJB session and entity bean must have the following classes and interfaces:

- Home *(EJBHome)* interface
- Remote *(EJBObject)* interface
- XML deployment descriptor
- Bean class
- Context objects

Home (EJBHome) Interface (Prior to EJB 3.0)

The `EJBHome` object provides the life cycle operations (`create()`, `remove()`, `find()`) for an EJB. The container's deployment tools generate the class for the

`EJBHome` object. The `EJBHome` object implements the *EJBsHome* interface. The client references an `EJBHome` object to perform life cycle operations on an *EJBObject* interface. The Java Naming and Directory Interface (JNDI) is used by the client to locate an `EJBHome` object.

The *EJBHome* interface provides access to the bean's life cycle services and can be utilized by a client to create or destroy bean instances. For entity beans, it provides finder methods that allow a client to locate an existing bean instance and retrieve it from its persistent data store.

Remote (EJBObject) Interface (Prior to EJB 3.0)

The remote *(EJBObject)* interface provides access to the business methods within the EJB. An *EJBObject* represents a client view of the EJB. The *EJBObject* exposes all of the application-related interfaces for the object, but not the interfaces that allow the EJB container to manage and control the object. The *EJBObject* wrapper allows the EJB container to intercept all operations made on the EJB. Each time a client invokes a method on the *EJBObject*, the request goes through the EJB container before being delegated to the EJB. The EJB container implements state management, transaction control, security, and persistence services transparently to both the client and the EJB.

XML Deployment Descriptor (Prior to EJB 3.0)

The deployment descriptor is an XML (Extensible Markup Language) file provided with each module and application that describes how the parts of a Java Platform EE application should be deployed. The deployment descriptor configures specific container options in your deployment tool of choice.

The rules associated with the EJB that govern life cycle, transactions, security, and persistence are defined in an associated XML deployment descriptor object. These rules are defined declaratively at the time of deployment rather than programmatically at the time of development. At runtime, the EJB container automatically performs the services according to the values specified in the deployment descriptor object associated with the EJB.

Business Logic (Bean) Class

The bean class is developed by the bean developer and contains the implementation and the methods defined in the remote interface. In other words, the bean class

has the basic business logic. For entity and session beans, the bean class extends either *javax.ejb.SessionBean* or *javax.ejb.EntityBean*, depending upon the type of EJB required.

Context Objects for Session and Entity (Prior to EJB 3.0)

For each active EJB instance, the EJB container generates an instance context object to maintain information about the management rules and the current state of the instance. A session bean uses a `SessionContext` object, while an entity bean uses an `EntityContext` object. Both the EJB and the EJB container use the context object to coordinate transactions, security, persistence, and other system services.

Classes and Interfaces for EJB 3.0

All EJB 3.0 classes are Plain Old Java Objects (POJOs) and all EJB 3.0 interfaces are Plain Old Java Interfaces (POJIs). Each EJB 3.0 bean has the following classes and interfaces:

- Bean class
- Business interface (can be generated by default)
- The XML deployment descriptor, which is now optional and largely unnecessary for simple EJBs

Bean Class (EJB 3.0)

The bean class is developed by the bean developer and continues to house the business logic.

Business Interface (EJB 3.0)

The business interface continues to define the access to the business methods within the EJB. This interface can be defined by default simply by specifying the `@remote` annotation within the bean class itself.

XML Deployment Descriptor (EJB 3.0)

The EJB 3.0 deployment descriptor is an optional XML file that provides the EJB deployer with the ability to add to or override metadata (annotations) contained in the application code (bean class).

CERTIFICATION OBJECTIVE 7.02

Distinguish Between Session and Entity Beans

Here, we review two of the three types of EJBs: session and entity beans. See Chapter 8 for information regarding the third type of EJB, the message-driven beans.

Session and Entity Beans

The EJB specification supports both *transient* and *persistent* objects. A transient object is referred to as a *session* bean, and a persistent object is known as an *entity* bean.

Session Beans

A session bean is an EJB that is created by a client and usually exists only for the duration of a single client/server session. A session bean usually performs operations such as calculations or database access on behalf of the client. While a session bean may be transactional, it is not recoverable if a system crash occurs. Session bean objects can be stateless, or they can maintain a conversational state across methods and transactions. If a session bean maintains a state, the EJB container manages this state if the object must be removed from memory. However, persistent data must be managed by the session bean object itself.

The tools for a container typically generate additional classes for a session bean at deployment time. These tools obtain information from the EJB architecture by examining its classes and interfaces. This information is utilized to generate two classes dynamically that implement the home and remote interfaces of the bean. These classes enable the container to intercede in all client calls on the session bean. The container generates a serializable *Handle* class as well, which provides a way to identify a session bean instance within a specific life cycle. These classes can be implemented to perform customized operations and functionality when mixed in with container-specific code.

In addition to these custom classes, each container provides a class that provides metadata to the client and implements the *SessionContext* interface. This provides access to information about the environment in which a bean is invoked.

Entity Beans

An *entity bean* is an object representation of persistent data maintained in a permanent data store such as a database. A primary key identifies each instance of an entity bean. Entity beans are transactional and are recoverable in the event of a system crash.

Entity beans are representations of explicit data or collections of data, such as a row in a relational database. Entity bean methods provide procedures for acting on the data representation of the bean. An entity bean is persistent and survives as long as its data remains in the database.

An entity bean can be created in two ways: by direct action of the client in which a `create()` method is called on the bean's home interface, or by some other action that adds data to the database that the bean type represents. In fact, in an environment with legacy data, entity objects may exist before an EJB is even deployed.

An entity bean can implement either bean-managed or container-managed persistence. In the case of bean-managed persistence, the implementer of an entity bean stores and retrieves the information managed by the bean through direct database calls. The bean may utilize either Java Database Connectivity (JDBC) or SQL-Java (SQLJ) for this method. (Session beans may also access the data they manage using JDBC or SQLJ.) A disadvantage to this approach is that it makes it more difficult to adapt bean-managed persistence to alternative data sources.

In the case of container-managed persistence, the container provider may implement access to the database using standard APIs. The container provider can offer tools to map instance variables of an entity bean to calls to an underlying database. This approach makes it easier to use entity beans with different databases.

SCENARIO & SOLUTION	
You need to maintain nonenterprise data across method invocations for the duration of a session. What kind of EJB would you use?	You should use a session bean, an EJB that is created by a client and usually exists only for the duration of a single client/server session.
You need to create an EJB to represent enterprise data. What kind of EJB should you use?	You should use an entity bean, which is an object representation of persistent data maintained in a permanent data store such as a database.

Above are some possible scenario questions that will help you review the differences between session and entity beans.

CERTIFICATION OBJECTIVE 7.03

Recognize Appropriate Uses for Entity, Stateful Session, and Stateless Session Beans

In an enterprise environment, application use may grow to a point at which systems based on such things as Java servlets and Hypertext Markup Language (HTML) are not scalable to provide the required performance. At this point, a distributed solution can provide the scalability needed to meet changing demands. EJB allows the application to be distributed onto as many servers as required.

When to Use Entity and Session JavaBeans

The following details some appropriate scenarios for using EJBs:

- Use entity beans to persist data. An entity bean is a sharable enterprise data resource that can be accessed and updated by multiple users.
- Use stateful session beans when any one of the following conditions is true; otherwise use stateless session beans:
- The session bean must retain data in its member variables across method invocations.
- The state of the bean needs to be initialized when the session bean is instantiated.
- The session bean must retain information about the client across multiple method invocations.
- The session bean is servicing an interactive client whose presence must be known to the applications server or EJB container.

CERTIFICATION OBJECTIVE 7.04

Distinguish Between Stateful and Stateless Session Beans

Now that you know a little bit about the EJB architecture, session beans, and entity beans, this objective breaks down each of these components in detail and covers using session beans and the differences between stateless session beans and stateful

session beans. You'll learn how to define stateless and stateful session bean classes, add methods to them, define the session bean interface, create a remote interface, create a home interface, and create deployment descriptors. We'll talk about the steps required to compile, assemble, and deploy stateless session beans and how to call them from a client.

Using Session Beans

Building a session bean can be quite simple once you have mastered a few basic steps. These steps are explained later in this section by walking you through an example of a session bean that provides validation for fields passed to it in a hash table.

As mentioned, session beans can either be stateful or stateless. With stateful beans, the EJB container saves internal bean data during and in-between method calls on the client's behalf. With stateless beans, the clients may call any available instance of an instantiated bean for as long as the EJB container has the ability to pool stateless beans. This enables the number of instantiations of a bean to be reduced, thereby reducing required resources.

Stateless Session Beans

A session bean represents work performed by a *single* client. That work can be performed within a single method invocation, or it may span multiple method invocations. If the work does span more than one method, the object must retain the user's object state across the method calls, and a stateful session bean would therefore be required.

Generally, stateless beans are intended to perform individual operations automatically and don't maintain state across method invocations. They're also *amorphous*, in that any client can use any instance of a stateless bean at any time at the container's discretion. They are the lightest in weight and easiest to manage of the various EJB component configurations.

Stateful Session Beans

Stateful session beans maintain state both within and between transactions. Each stateful session bean is therefore associated with a specific client. Containers are able to save and retrieve a bean's state automatically while managing instance pools (as opposed to bean pools) of stateful session beans.

Stateful session beans maintain data consistency by updating their fields each time a transaction is committed. To keep informed of changes in transaction status, a stateful session bean implements the *SessionSynchronization* interface. The container calls methods of this interface while it initiates and completes transactions involving the bean.

Session beans, whether stateful or stateless, are not designed to be persistent. The data maintained by stateful session beans is intended to be transitional. It is used solely for a particular session with a particular client. A stateful session bean instance typically can't survive system failures and other destructive events. While a session bean has a container-provided identity (called its *handle*), that identity passes when the client removes the session bean at the end of a session. If a client needs to revive a stateful session bean that has disappeared, it must provide its own means to reconstruct the bean's state.

Stateful vs. Stateless Session Beans

A stateful session bean *will* maintain a conversational state with a client. The state of the session is maintained for the duration of the conversation between the client and the stateful session bean. When the client removes the stateful session bean, its session ends and the state is destroyed. The transient nature of the state of the stateful session bean should not be problematic for either the client or the bean, because once the conversation between the client and the stateful session bean ends, neither the client nor the stateful session bean should have any use for the state.

A stateless session bean *will not* maintain conversational states for specific clients longer than the period of an individual method invocation. Instance variables used by a method of a stateless bean may have a state, but only for the duration of the method invocation. After a method has finished running either successfully or unsuccessfully, the states of all its instance variables are dropped. The transient nature of this state gives the stateless session bean beneficial attributes, such as the following:

- **Bean pooling** Any stateless session bean method instance that is not currently invoked is equally available to be called by an EJB container or application server to service the request of a client. This allows the EJB container to pool stateless bean instances and increase performance.
- **Scalability** Because stateless session beans are able to service multiple clients, they tend to be more scalable when applications have a large number of clients. When compared to stateful session beans, stateless session beans usually require less instantiation.

- **Performance** An EJB container will never move a stateless session bean from RAM out to a secondary storage, which it may do with a stateful session bean; therefore, stateless session beans may offer greater performance than stateful session beans.

Since no explicit mapping exists between multiple clients and stateless bean instances, the EJB container is free to service any client's request with any available instance. Even though the client calls the `create()` and `remove()` methods of the stateless session bean, making it appear that the client is controlling the life cycle of an EJB, it is actually the EJB container that is handling the `create()` and `remove()` methods without necessarily instantiating or destroying an EJB instance.

Defining the Session Bean Class (Prior to EJB 3.0)

The session bean class must be declared with the `public` attribute. This attribute enables the container to obtain access to the session bean. Java gives a developer the ability to extend a base class and inherit its properties. This ability pertains to session beans as well, allowing developers to take full advantage of any object-oriented legacy code that they may wish to reuse.

The following is an example of a session bean (prior to EJB 3.0) extending a base class:

```
public class ValidateInputBean extends TradingBaseClass implements SessionBean
{
   ...
}
```

Session Bean Interface (Prior to EJB 3.0)

Session beans are held to the Java Platform EE (prior to EJB 3.0) specification that requires all session beans to implement the *javax.ejb.SessionBean* interface. This requirement forces session beans to contain the following methods:

- `ejbActivate()`
- `ejbPassivate()`
- `ejbRemove()`
- `setSessionContext(SessionContext)`

A minimum sample of how a bean class must look is shown here:

```
public class ValidateInputBean extends TradingBaseClass implements SessionBean
{
   public void ejbActivate  () throws EJBException {..}
   public void ejbPassivate () throws EJBException {..}
   public void ejbRemove    () throws EJBException {..}
   protected   SessionContext m_context;
   public void setSessionContext (SessionContext sc)
                  throws EJBException {
      m_context = sc;
   }
}
```

Clients of a session bean may either be *remote* or *local*, depending on what interfaces are implemented.

Remote clients access a session bean via their remote and remote home interfaces (*javax.ejb.EJBObject* and *javax.ejb.EJBHome*, respectively). Remote clients have the advantage of being location independent. They can access a session bean in an EJB container from any Remote Method Invocation-Internet Inter-ORB Protocol (RMI-IIOP)–compliant application, including non-Java programs such as CORBA-based applications. Because remote objects are accessed through standard Java RMI APIs, objects that are passed as method arguments are passed by value. This means that a copy of the object being passed is created and sent between the client and the session bean.

Local clients access a session bean via their local and local home interfaces (*javax .ejb.EJBLocalObject* and *javax.ejb.EJBLocalHome*, respectively). A local client is location dependent. It must reside inside the same JVM as the session bean with which it interfaces. Local clients can have objects passed as arguments to methods by reference. Doing this avoids the overhead of creating copies of objects sent between clients and session beans. Certain applications will perform considerably better without this overhead. The enterprise bean provider should be aware that both the client and the session bean can change common objects.

Both local and remote home interfaces (*javax.ejb.LocalHome* and *javax.ejb.EJBHome*, respectively) provide an interface to the client, allowing the client to create and remove session objects. However, session objects are more commonly removed by using the `remove()` method in the *EJBObject* interface.

Neither local nor remote clients access session beans directly. To gain access to session bean methods, they use a *component* interface to the session bean. Instances of a session bean's remote interface are called session *EJBObjects*, while instances of a session bean's local interface are called session *EJBLocalObjects*.

Both local and remote interfaces provide the following services to a client:

- Delegate business method invocations on a session bean instance.
- Return the session object's home interface.
- Test to determine whether a session object is identical to another session object.
- Remove a session object.

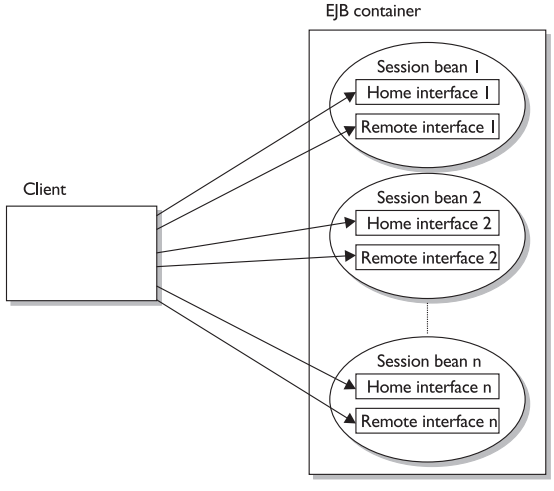
Any method on a session bean class that is to be made visible to a client must be added to the bean's remote interface. This makes it possible to hide session bean methods from clients as well as make different methods of a session bean available using different interfaces. Figure 7-2 illustrates how the client will see the EJB session bean interfaces.

When the application is deployed, the container or application server will use the interfaces defined by the enterprise bean provider and create *EJBHome*, *EJBObject*, stub, and tie classes:

- The *EJBHome* class is used to create instances of the session bean class and the *EJBObject* class.
- The *EJBObject* class provides access to the desired methods of the session bean.
- The stub classes act as proxies to the remote *EJBObjects*.
- The tie classes provide the call and dispatch mechanisms that bind the proxy to the *EJBObject*.

FIGURE 7-2

EJB session bean interface exposed to a client



Creating a Remote Interface (Prior to EJB 3.0) All remote interfaces must extend the class *javax.ejb.EJBObject*. The following is an example:

```
public interface ValidateInputRemote extends EJBObject { .. }
```

Methods in Remote Interfaces (Prior to EJB 3.0) The enterprise bean provider provides the session bean's remote interface, which extends *javax.ejb.EJBObject*, and the EJB container implements this interface. An enterprise bean's remote interface provides the client's view of a session object and defines the business methods that are callable by the client.

All business methods declared in the remote interface must have the same parameters and same return value types as the bean class. It is not necessary for all bean class methods to be exposed to the remote client.

All business methods declared in the remote interface must also throw at least the same exceptions as those in the bean class. They must also throw the *java.rmi .RemoteException* exception, because EJBs are dependent on the RMI package, specifically the *java.rmi.Remote* package, for distributed processing.

Normally, the container or application server being used will generate the necessary remote interface code. The container should also update this code when changes are made to the bean class.

The method names and the signatures in the remote interface must be identical to the method names and signatures of the business methods defined by the enterprise bean. This is different from the home interface, where method signatures must match, as method names can be different.

In addition to business methods that may be defined in the remote interface, the methods listed here and shown in Figure 7-3 must be contained inside the remote interface:

- **getEJBHome()** This method returns a reference to the session bean's home interface.
- **getHandle()** This method returns a handle for the *EJBObject*. This handle can be used at a later time to reobtain a reference to the *EJBObject*. A session object handle can be serialized to a persistent data store to enable the retrieval of a session object even beyond the lifetime of a client process. This is assuming that the EJB container does not crash or time out the session object, thereby destroying it.

- **getPrimaryKey()** This method is not to be used for session beans. It returns the session bean object's primary key, but since individual session objects are to be used only by the specific client that creates them, they are intended to appear anonymous. If `getPrimaryKey()` is called looking for the identity of a session object, the method will throw an exception. This is different from entity objects, which expose their identity as primary keys.
- **isIdentical()** This method is used to test whether the *EJBObject* passed is identical to the invoked *EJBObject*.
- **remove()** This method is used to remove a session bean object.

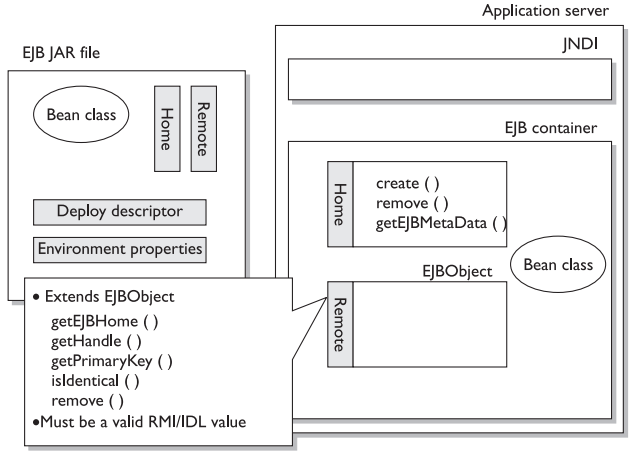
Note that these methods are included automatically by virtue of inheritance. A typical remote interface definition for a session bean looks something like this:

```
import javax.ejb.*;
import java.rmi.*;
import java.util.*;
public interface ValidateInputRemote extends EJBObject
{
     public void isInt (double amount) throws RemoteException;
     public void isNum (double amount) throws RemoteException;
}
```

Creating a Home Interface (Prior to EJB 3.0) Session beans are instantiated when a client makes a call to one of the `create()` methods defined in the home interface. The home interface contains a `create()` method for every corresponding `ejbCreate()` method in the bean class.

FIGURE 7-3

Methods in the session bean remote interface



The home interface is implemented in a container through an object called the *home* object. The container makes visible an instance of the home object to clients that want to instantiate a session bean.

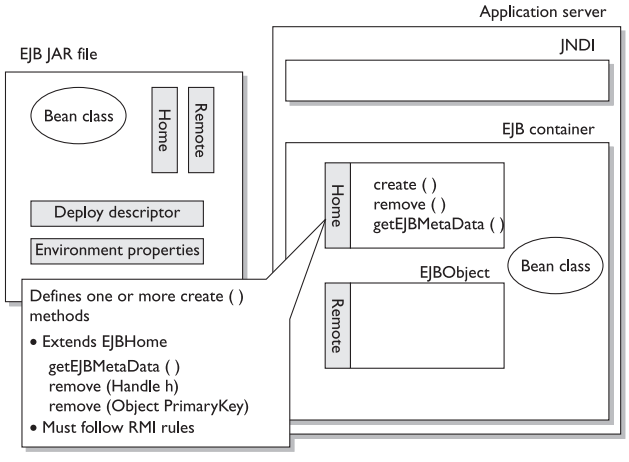
In addition to `create()` methods that are to be defined in the home interface, the methods listed here and shown in Figure 7-4 are to be contained in the home interface:

- **getEJBMetaData()** This method is used to obtain the *EJBMetaData* interface of an EJB. EJB deployment tools are responsible for implementing classes that provide metadata to the remote client. The *EJBMetaData* interface enables the client to get information about the enterprise bean. This metadata may be used to give access to the enterprise bean clients that use a scripting language to access these enterprise beans. Development tools may also use this metadata. The *EJBMetaData* interface is not a remote interface, so its class must be serializable.
- **getHomeHandle()** This method is used to obtain a handle of a home object. The EJB specification allows a client to obtain a handle for the remote home interface. The client can use the home handle to store a reference to an entity bean's remote home interface in stable storage and recreate the reference later. This handle functionality may be useful to a client who needs to use the remote home interface in the future but does not know the JNDI name of the remote home interface.
- **remove(Handle h)** This method is used to remove *EJBObjects* that are identified by their handles. A handle may be retrieved by using the `getHandle()` method.
- **remove(Object primaryKey)** This method should not be used for session beans. It is used to remove *EJBObjects* that are identified by their primary key. Because session objects do not have primary keys that are accessible to clients, invoking this method on a session bean will result in a *RemoveException*. A container may also remove the session object automatically when the session object's lifetime expires.

Since all session objects keep their identity anonymous, `finder()` methods for session beans should not be defined. The `finder()` methods for entity beans are covered later on in this chapter, in the section "Home Interfaces and finder( ) Methods."

FIGURE 7-4

Methods in the session bean home interface



Again, note that the methods in the preceding list are included automatically by virtue of inheritance. Here is an example of a home interface definition for an EJB:

```
import javax.ejb.*;
import java.rmi.*;
public interface AccountRemoteHome extends EJBHome
{
  Account create() throws CreateException, RemoteException;
}
```

Session Bean Class Methods (Prior to EJB 3.0)

Session bean classes are used as the "molds" for instantiating session bean instances. These instances are indirectly called as local and remote clients via home and remote interfaces. `EJBCreate()` methods correspond with the `create()` methods of the session bean's home interface and are used for initializing the session bean's instance. The business methods created in a session bean class are a superset of those defined for the session bean's local or remote interface. These business methods implement the core business logic for session beans.

create() Methods (Prior to EJB 3.0) The EJB specification requires the session bean class to contain one or more `ejbCreate()` methods. These `ejbCreate()` methods are normally used to initialize the bean.

As many `ejbCreate()` methods as necessary may be added to the bean (only one `ejbCreate()` method for stateless session beans, however), as long as their signatures meet the following requirements:

- They must have a public access control modifier.
- They must have a return type of *void*.
- They must have RMI compliant arguments, in that they are serializable objects.
- They must not have a *static* or *void* modifier.

Several exceptions may be thrown including the *javax.ejb.CreateException* and other application-specific exceptions. The `ejbCreate()` method will usually throw a *CreateException* if an input parameter is not valid.

When a client invokes a `create()` method of a home interface, the `ejbCreate()` method of the session bean is called and the session bean and *EJBObject* are instantiated. After the session bean and *EJBObject* have been instantiated, the `create()` method returns a remote object reference of the *EJBObject* instance associated with the session bean instance to the client. The client can then invoke all of the business methods of this reference.

Because the `ejbCreate()` method is able to throw both *javax.ejb.CreateException* when there is a problem creating an object and the *javax.ejb.EJBException* when there is a system problem, the `create()` methods must be declared to throw both of these exceptions as well. In addition to the two aforementioned exceptions, the `create()` method must also throw any programmer-defined applications exception that may be thrown in the `ejbCreate()` method.

Business Methods The primary purpose of a session bean is to execute business methods that implement business logic for use by a client. The `create()` (prior to EJB 3.0) or the *EJBContext* `lookup()` (EJB 3.0) methods return object references from which clients may invoke business methods. To the client, these business methods appear to be running locally; however, they actually run remotely in the session bean container.

Session bean business methods, like any other Java method, are defined with the following procedures:

- Add a method to the bean.
- Write and then save the code.
- Debug the code.
- Finish the bean.

It is important to note that the bean class should *not* implement the remote interface. The bean class code is where all of the actual business code exists. This code is not supposed to be called without a proxy; therefore, it cannot be viewed directly by the client.

EXERCISE 7-1

Review Code for a Stateless Session Bean (Prior to EJB 3.0)

In the following exercise, the classes and interfaces are provided for a stateless session bean. The exercise is presented step-wise in the order that the classes and interfaces would be created. Your completed code should look something like the code contained in the following sections.

1. Create the Stateless Session Bean Class Here is an example of a stateless session bean class. As you will see, the only purpose of this bean is to pass back a simple message to the caller. The inline documentation points out the required methods along with the business methods.

```
package javaee.architect.SLSession;
import javax.ejb.*;
// A stateless session bean.
public class SLSessionBean implements SessionBean {
  SessionContext sessionContext;
  // Bean's methods required by EJB specification.
  public void ejbCreate() throws CreateException {
    log("ejbCreate()");
  }
  public void ejbRemove() {
    log("ejbRemove()");
  }
  public void ejbActivate() {
    log("ejbActivate()");
  }
  public void ejbPassivate() {
    log("ejbPassivate()");
  }
  public void setSessionContext(SessionContext parm) {
    this.sessionContext = parm;
  }
  // Bean's business methods.
```

```
  public String getMsg() {
    log("getMsg()");
    return
      "This is a message from an stateless session bean!";
  }
  public void log(String parm) {
    System.out.println(new java.util.Date()
      +":SLSessionBean:"+this.hashCode()+" "+parm);
  }
}
```

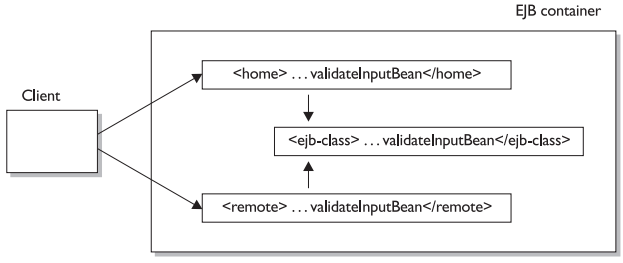
2. Create the Stateless Session Bean Home Interface Here is a home interface for our stateless session bean example:

```
package javaee.architect.SLSession;
import javax.ejb.*;
import java.rmi.*;
// This is the remote home interface, used by clients as a
// factory for EJB objects (remote references). The EJB
// container vendor implements this extended interface.
// In this home interface there is a create() method that
// corresponds to the ejbCreate() method in bean code.
public interface SLSessionRemoteHome extends EJBHome {
  // Creates/returns the EJB Object (remote reference).
  public SLSessionRemote create() throws CreateException, RemoteException;
}
```

3. Create the Stateless Session Bean Remote Interface Here is a remote interface for our stateless session bean example:

```
package javaee.architect.SLSession;
import javax.ejb.*;
import java.rmi.*;
// This is the remote interface, used by clients when
// they need to call an EJB objects. The EJB container
// vendor implements this extended interface, which is
// responsible for delegating subsequent calls to the
// bean code.
public interface SLSessionRemote extends javax.ejb.EJBObject {
  // Returns a String to caller.
  public String getMsg() throws RemoteException;
}
```

The following illustration shows how a client will view the remote interface.



4. Create Deployment Descriptors A deployment descriptor, located within a Java Archive (JAR) file, allows the properties of an EJB to be maintained outside of Java code. It allows the bean developer to make information about the bean available to the application assembler and the bean deployer. A deployment descriptor also provides runtime information used by the EJB container. The EJB specification is specific with regard to the content and format of deployment descriptors.

The deployment descriptor, written in XML, contains the structural information about the EJB, such as the relative path and name of the bean class file, remote interface, and home interface, as well as the state management type and the transaction management type.

The deployment descriptor file may also contain optional information pertaining to multiple role names, environment entries, and data-source references. Note that all of the attributes of the bean are contained within XML tags.

Here is our deployment descriptor for the stateless session bean:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE ejb-jar PUBLIC
"-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans 2.0//EN"
"http://java.sun.com/dtd/ejb-jar_2_0.dtd">
<ejb-jar>
  <enterprise-beans>
    <session>
      <display-name>SLSession</display-name>
      <ejb-name>SLSession</ejb-name>
      <home>javaee.architect.SLSession.SLSessionRemoteHome</home>
      <remote>javaee.architect.SLSession.SLSessionRemote</remote>
      <ejb-class>javaee.architect.SLSession.SLSessionBean</ejb-class>
      <session-type>Stateless</session-type>
      <transaction-type>Container</transaction-type>
    </session>
```

```
  </enterprise-beans>
  <assembly-descriptor>
    <container-transaction>
      <method>
        <ejb-name>SLSession</ejb-name>
        <method-name>*</method-name>
      </method>
      <trans-attribute>Required</trans-attribute>
    </container-transaction>
  </assembly-descriptor>
</ejb-jar>
```

Some of the elements in the preceding deployment descriptor sample are described here:

- **ejb-jar** The root element of the EJB deployment descriptor. These files are discussed later in the chapter, in the section "The Life Cycle of an EJB."
- **enterprise-beans** Declares the session, entity, and/or message-driven beans.
- **session** Defines the enterprise bean to be a session bean as opposed to an entity or message-driven bean.
- **ejb-name** A unique name of a session, entity, or message-driven bean in an *ejb-jar* file; this element is used to tie EJBs together and for constructing a URL (note that there is no relationship between the element *ejb-name* and the JNDI name that is assigned to an enterprise bean's home).
- **home** The fully qualified name of an enterprise bean's home interface.
- **remote** The fully qualified name of enterprise bean's remote interface.
- **ejb-class** The fully qualified name of the enterprise bean's class.
- **session-type** The session-type element is either stateful or stateless.
- **transaction-type** Declares whether transaction demarcation is performed by the enterprise bean or the EJB container.

5. Compile, Assemble, and Deploy Stateless Session Bean The next steps in the process are to compile, assemble, and then deploy the session bean. The following set of steps is used to complete this process. Although this book is designed for the architect and actual implementation steps are not required, we have added the details of these steps for completeness.

As part of the deployment process, references in the deployment descriptor need to be resolved to actual resources in the container. How these resources are resolved is,

at the moment, container specific. In the WebLogic environment, the following file can be used for the stateless session bean example:

```
<?xml version="1.0"?>
<!DOCTYPE weblogic-ejb-jar PUBLIC
"-//BEA Systems, Inc.//DTD WebLogic 8.1.0 EJB//EN"
"http://www.bea.com/servers/wls810/dtd/weblogic-ejb-jar.dtd" >
<weblogic-ejb-jar>
    <weblogic-enterprise-bean>
        <ejb-name>SLSession</ejb-name>
        <jndi-name>SLSessionRemoteHome</jndi-name>
    </weblogic-enterprise-bean>
</weblogic-ejb-jar>
```

Here are the remaining steps to complete the compile/package/deployment process:

1. Compile the Java classes.
2. Package the classes and deployment descriptors into a JAR file.
3. Generate stub and tie code for the container and add them to the JAR file.
4. Deploy the JAR file to the application server.
5. Package the required classes for a remote client of the bean.

Here is an example client of the stateless session bean:

```
package javaee.architect.SLSession;
import javax.naming.Context;
import javax.naming.InitialContext;
import java.util.Properties;
// This client calls a method on a stateless session bean.
public class SLSessionClient {
  public static void main(String[] args) throws Exception {
    // Get the JNDI initial context.
    Context ctx = getInitialContext();
    // Get a reference to the home object
    Object obj = ctx.lookup("SLSessionRemoteHome");
    // Narrow (cast) the returned RMI-IIOP object.
    SLSessionRemoteHome home = (SLSessionRemoteHome)
      javax.rmi.PortableRemoteObject.narrow(obj,
      SLSessionRemoteHome.class);
    // Use the home object (factory) to create the
    // SLSB EJB Object (the remote reference).
    SLSessionRemote mySLSessionRemote = home.create();
```

```
    // Call the getMsg() method on the EJB object.
    // The remote reference will delegate the call
    // to the bean code, receive the response and
    // then pass it to this client.
    System.out.println(mySLSessionRemote.getMsg());
    // When finished with the remote reference,
    // remove it and the EJB container will then
    // destroy the EJB object.
    mySLSessionRemote.remove();
  }
  private static Context getInitialContext() throws Exception {
    // This implementation is specific to the Weblogic
    // server and will differ for other server vendors.
    String providerUrl = "t3://localhost:7001";
    String icFactory = "weblogic.jndi.WLInitialContextFactory";
    String user = null;
    String password = null;
    Properties properties = null;
    try {
      properties = new Properties();
      properties.put(Context.INITIAL_CONTEXT_FACTORY, icFactory);
      properties.put(Context.PROVIDER_URL, providerUrl);
      if (user != null) {
        properties.put(Context.SECURITY_PRINCIPAL, user);
        properties.put(Context.SECURITY_CREDENTIALS,
          password == null ? "" : password);
      }
      return new InitialContext(properties);
    }
    catch(Exception e) {
      System.out.println(
        "Unable to connect to JNDI server at " + providerUrl);
      throw e;
    }
  }
}
```

Here is the output information provided by the client:

```
This is a message from a stateless session bean!
```

Here is the output information provided by the client and the application server console:

```
...when the jar is deployed to the server...
Sat Jan 20 12:58:55 EST 2007:SLSessionBean:4536570 ejbCreate()
```

```
Sat Jan 20 12:58:55 EST 2007:SLSessionBean:2076276 ejbCreate()
...when the client application is executed...
Sat Jan 20 12:59:15 EST 2007:SLSessionBean:2076276 getMsg()
...when the server is shut down...
Sat Jan 20 13:01:33 EST 2007:SLSessionBean:2076276 ejbRemove()
Sat Jan 20 13:01:33 EST 2007:SLSessionBean:4536570 ejbRemove()
```

The next section will cover the client side of the calls in a little more detail.

Calling Stateless Session Beans from a Client

When a client calls an EJB, the ultimate goal is to gain the benefits derived from executing the business methods on a bean class. Before the EJB can get access to these methods, it must first find the *EJBHome* interface necessary to make an instance of the EJB class. The first step in the process of finding the *EJBHome* starts with creating an *InitialContext* class.

InitialContext The *InitialContext* class acts as the client's interface to the JNDI interface. The *InitialContext* may contain information that will allow a client to bind to many naming services such as JNDI, CORBA Common Object Service ("COS"), and Domain Naming Service (DNS). Using the *InitialContext* class allows a client to have to maintain only a single interface to any naming service in the client's environment that supports JNDI. If any problems are encountered with the creation of the *InitialContext* object or with calling one of its methods, the *javax.naming .NamingException* will be thrown.

More on Type Narrowing In a stand-alone Java application, if a Java object such as *Object* is returned from a *Hashtable*, the return type of the method `get()` will be the supertype *Object* instead of the derived type *String*. Here's an example:

```
Hashtable hash = ..;
..;
hash.get("keyToAStringElement");
```

It is up to the developer to cast, or narrow, the return value of the method `get()` to the proper object type. For example:

```
String aString = (String) hash.get("keyToAStringElement");
```

In the EJB application, once an object reference is obtained by a client, the method `javax.rmi.PortableRemoteObject.narrow()` must be used to perform type-narrowing for its client-side representation of its home and remote interfaces.

The *javax.rmi.PortableRemoteObject* class is part of the RMI-IIOP standard extension. Type-narrowing ensures that the client programs are interoperable with different EJB containers.

Once the *InitialContext* has been used as the starting point for looking up a specific JNDI registered object, the `javax.rmi.PortableRemoteObject.narrow()` method should be called to perform type-narrowing of the client-side representations of the home interface.

Finding Objects and Interfaces: The JNDI Clients that have access to the JNDI API may use this API to look up enterprise beans, resources such as databases, and data in environment variables. From the earlier example, the client application locates an EJB with the following:

```
// Get a reference to the home object
Object obj = ctx.lookup("SLSessionRemoteHome");
```

The name that a Java Platform EE client uses to refer to an EJB does not necessarily have to be identical to the JNDI name of the EJB deployed in the EJB container or application server. The level of indirection provided by the ability to map Java Platform EE client names to JNDI registered EJBs gives great flexibility to distributed applications by allowing the client to use names that reference EJBs that make logical sense to the client. The client even has the ability to reference a single EJB with different names, when it makes sense to do so. This flexibility comes in handy when either the client code or the server code changes dynamically. The name that a stand-alone Java client uses to refer to an EJB using a JNDI lookup method must be identical to the EJBs JNDI name in the EJB container or application server.

Creating an Instance Using EJBHome Finally, after the home reference has been found, narrowed, and called, the home reference's `create()` method can be called, returning the remote reference upon which the desired business methods can be invoked. The syntax for the calling of the `create()` method may look somewhat like this:

```
SLSessionRemote mySLSessionRemote = home.create();
```

Remember that `create()` returns a reference to a remote interface, not the bean object itself. This means that certain programming practices that may be taken for granted in Java may not be used with EJBs. For example, objects are passed to EJBs via the arguments of the method calls and results are passed back to the client

via the EJBs return object. It is not possible to maintain a reference to an input argument of an EJB, change that argument's values inside of the EJB, and then have access to those changes at the client.

Calling Session Bean Methods from a Client After all the components of a session bean have been created and a remote reference to the EJB is made available to a client, the exposed business methods of the *EJBObject* and the life cycle methods of the *EJBHome* are available to the client as if the EJB were local to that client.

Coding Clients to Call EJBs After the session bean reference has been made available to a client, it is up to the client to instantiate the bean components through the bean's home interface. Only then can the business logic methods of the bean class be accessed via the exposed methods of the *EJBObject*. For Java Platform EE applications to run in an efficient and stable fashion, both client and server developers should adhere to standard programming policy practices and procedures, two of which are mentioned next.

Session Beans, Reentrance, and Loop-Back Calls If a bean is allowed to invoke methods on itself or another bean that invokes methods on the initial bean, the initial bean is said to be *reentrant*. This type of self-accessing call is referred to as a *loop-back call*. As opposed to an entity bean, a session bean is never allowed to be reentrant. If a session bean attempts to make a loop-back, the EJB container should throw a *java.rmi.RemoteException*.

Remove the Bean When Done When a client no longer has use for a session bean, it should remove the session object using the `javax.ejb.EJBObject.remove()` method or the `javax.ejb.EJBHome.remove(Handle handle)` method. If the `javax.ejb.EJBHome.remove(Object primaryKey)` method is called on a session by mistake, the *javax.ejb.RemoveException* will be thrown because session beans, unlike entity beans, do not have a primary key.

Different Types of Clients

The EJB framework allows many different types of clients to instantiate EJBs and takes advantage of the business logic that they implement. For the rest of this section, examples of different types of clients calling a session bean will be presented along with some of the necessary tasks that must be completed to support these different clients. This section concentrates on how clients can be integrated with EJBs.

Servlets Calling Session Beans A *servlet* is a Java program that runs within a Web or application server and implements the *javax.servlet.Servlet* interface. Servlets are designed to receive and respond to requests from Internet clients or browsers. The standard protocols used for communication between a browser and a servlet are usually Hypertext Transfer Protocol (HTTP) or secure Hypertext Transfer Protocol (HTTPS). Servlets receive and respond to requests from Internet clients using methods defined by the *javax.servlet.Servlet* interface. After the web or application server constructs the servlet, the servlet gets initialized by the life cycle method `init()`.

EXERCISE 7-2

Review Code for a Stateful Session Bean (Prior to EJB 3.0)

To become more familiar with developing and coding, let's review the code for a stateful session bean. Your code should look something like the code shown next. Again, we use a step-wise approach that covers all the steps required to create, package, deploy, and call the bean.

1. Create the Stateful Session Bean Class Following is an example of a stateful session bean class. The purpose of this bean is to create and initialize a counter and have a business method increment the counter. The in-line documentation points out the required methods along with the business methods.

```
package javaee.architect.SFSession;
import javax.ejb.*;
// A stateful session bean.
// When bean is created a counter is initialized with the
// parameter value. A business method increments the counter.
public class SFSessionBean implements SessionBean {
  private SessionContext sessionContext;
  // The counter.
  private int ctr;
  // Bean's methods required by EJB specification
  public void ejbCreate(int parm) throws CreateException {
    this.ctr = parm;
    log("ejbCreate("+parm+")");
  }
  public void ejbRemove() {
    log("ejbRemove() ctr="+ctr);
  }
```

```
  public void ejbActivate() {
    log("ejbActivate() ctr="+ctr);
  }
  public void ejbPassivate() {
    log("ejbPassivate() ctr="+ctr);
  }
  public void setSessionContext(SessionContext parm) {
    this.sessionContext = parm;
  }
  // Bean's business methods
  public int increment() {
    log("increment() ctr="+ctr);
    return ++ctr;
  }
  private void log(String parm) {
    System.out.println(new java.util.Date()
      +":SFSessionBean:"+this.hashCode()+" "+parm);
  }
}
```

2. Create the Stateful Session Bean Home Interface Here is a home interface for our stateful session bean example:

```
package javaee.architect.SFSession;
import javax.ejb.*;
import java.rmi.*;
// This is the home interface, used by clients as a
// factory for EJB objects (remote references). The EJB
// container vendor implements this extended interface.
// In this home interface there is a create() method that
// corresponds to the ejbCreate() method in actual bean code.
public interface SFSessionRemoteHome extends EJBHome {
  public SFSessionRemote create(int ct)
    throws CreateException, RemoteException;
}
```

3. Create the Stateful Session Bean Remote Interface Here is a remote interface for our stateful session bean example:

```
package javaee.architect.SFSession;
import javax.ejb.*;
import java.rmi.*;
// This is the remote interface, used by clients when
```

```
// they need to call an EJB object. The EJB container
// vendor implements this extended interface, which is
// responsible for delegating subsequent calls to the
// actual bean code.
public interface SFSessionRemote extends EJBObject {
  public int increment() throws RemoteException;
}
```

4. Create Deployment Descriptors As mentioned in the preceding exercise, the deployment descriptor, located within a JAR file, allows the properties of an EJB to be maintained outside of Java code.

Here is our deployment descriptor for the stateful session bean:

```
<!DOCTYPE ejb-jar PUBLIC
    "-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans 2.0//EN"
    "http://java.sun.com/dtd/ejb-jar_2_0.dtd">
<ejb-jar>
 <enterprise-beans>
  <session>
   <display-name>SFSession</display-name>
   <ejb-name>SFSession</ejb-name>
   <home>javaee.architect.SFSession.SFSessionRemoteHome</home>
   <remote>javaee.architect.SFSession.SFSessionRemote</remote>
   <ejb-class>javaee.architect.SFSession.SFSessionBean</ejb-class>
   <session-type>Stateful</session-type>
   <transaction-type>Container</transaction-type>
  </session>
 </enterprise-beans>
 <assembly-descriptor>
  <container-transaction>
   <method>
    <ejb-name>SFSession</ejb-name>
    <method-name>*</method-name>
   </method>
   <trans-attribute>Required</trans-attribute>
  </container-transaction>
 </assembly-descriptor>
</ejb-jar>
```

5. Compile, Assemble, and Deploy Stateful Session Bean The next steps in the process are to compile, assemble, and then deploy the stateful session bean. In our example, the following list of steps are used to complete the process.

Although this is book is designed for the architect and actual implementation steps are not required, we have added the details of these steps for completeness.

As part of the deployment process, references in the deployment descriptor need to be resolved to actual resources in the container. How these resources are resolved is, at the moment, container specific. In the WebLogic environment, the following file can be used for the stateful session bean example:

```
<?xml version="1.0"?>
<!DOCTYPE weblogic-ejb-jar PUBLIC
"-//BEA Systems, Inc.//DTD WebLogic 8.1.0 EJB//EN"
"http://www.bea.com/servers/wls810/dtd/weblogic-ejb-jar.dtd" >
<weblogic-ejb-jar>
  <weblogic-enterprise-bean>
    <ejb-name>SLSession</ejb-name>
    <stateful-session-descriptor>
      <stateful-session-cache>
        <max-beans-in-cache>3</max-beans-in-cache>
        <idle-timeout-seconds>120</idle-timeout-seconds>
        <cache-type>LRU</cache-type>
      </stateful-session-cache>
    </stateful-session-descriptor>
    <jndi-name>SLSessionRemoteHome</jndi-name>
  </weblogic-enterprise-bean>
</weblogic-ejb-jar>
```

Here are the remaining steps to complete the compile/package/deployment process:

1. Compile the Java classes.
2. Package the classes and deployment descriptors into a JAR file.
3. Generate stub and tie code for the container and add them to the JAR file.
4. Deploy the JAR to the application server.
5. Package the required classes for a remote client of the bean.

Here is an example client of the stateful session bean:

```
package javaee.architect.SFSession;
import javax.ejb.*;
import javax.naming.*;
import java.util.Properties;
// This client calls a method on 5 instances of a
// stateful session bean. The EJB container is setup
```

```
// to allow only 3 in memory.  When executed it will
// demonstrate how beans are passivated to and
// activated from storage.
public class SFSessionClient {
  private static final int NUMBEANS = 5;
  public static void main(String[] args) {
    try {
      // Get the JNDI initial context.
      Context ctx = getInitialContext();
      // Narrow (cast) the returned RMI-IIOP object.
      SFSessionRemoteHome home = (SFSessionRemoteHome)
        javax.rmi.PortableRemoteObject.narrow(
          ctx.lookup("SFSessionRemoteHome"),
          SFSessionRemoteHome.class);
      // Create array to hold EJB Objects
      SFSessionRemote mySFSession[] = new SFSessionRemote[NUMBEANS];
      // Populate array with remote references
      System.out.println("Instantiating beans");
      System.out.println("and calling increment()...");
      for (int i=0; i < NUMBEANS; i++) {
        // Create initialized remote reference
        mySFSession[i] = home.create(((i+1)*10)-1);
        // Call the increment method and print value
        System.out.println(new java.util.Date()
          +" loop1: mySFSession["
          +i+"].increment()="
          +mySFSession[i].increment());
        // Put this thread to sleep for a bit
        Thread.sleep(1000);
      }
      // Now call the increment method and see
      // what happens with passivation/activation
      System.out.println("Calling increment() again.");
      for (int i=0; i < NUMBEANS; i++) {
        // Again call increment and print value
        System.out.println(new java.util.Date()
          +" loop2: mySFSession["
          +i+"].increment()="
          +mySFSession[i].increment());
        // Sleep for a bit again
        Thread.sleep(1000);
      }
      // Finished with the beans, so remove them
      for (int i=0; i < NUMBEANS; i++) {
        mySFSession[i].remove();
      }
```

```
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
  private static Context getInitialContext() throws Exception {
    // This implementation is specific to the Weblogic
    // server and will differ for other server vendors.
    String providerUrl = "t3://localhost:7001";
    String icFactory = "weblogic.jndi.WLInitialContextFactory";
    String user = null;
    String password = null;
    Properties properties = null;
    try {
      properties = new Properties();
      properties.put(Context.INITIAL_CONTEXT_FACTORY, icFactory);
      properties.put(Context.PROVIDER_URL, providerUrl);
      if (user != null) {
        properties.put(Context.SECURITY_PRINCIPAL, user);
        properties.put(Context.SECURITY_CREDENTIALS,
          password == null ? "" : password);
      }
      return new InitialContext(properties);
    }
    catch(Exception e) {
      System.out.println(
        "Unable to connect to JNDI server at " + providerUrl);
      throw e;
    }
  }
}
```

Here is the output information provided by the client:

```
Instantiating beans
and calling increment()...
Sat Jan 20 13:04:09 EST 2007 loop1: mySFSession[0].increment()=10
Sat Jan 20 13:04:10 EST 2007 loop1: mySFSession[1].increment()=20
Sat Jan 20 13:04:11 EST 2007 loop1: mySFSession[2].increment()=30
Sat Jan 20 13:04:12 EST 2007 loop1: mySFSession[3].increment()=40
Sat Jan 20 13:04:13 EST 2007 loop1: mySFSession[4].increment()=50
Calling increment() again.
Sat Jan 20 13:04:14 EST 2007 loop2: mySFSession[0].increment()=11
Sat Jan 20 13:04:15 EST 2007 loop2: mySFSession[1].increment()=21
Sat Jan 20 13:04:16 EST 2007 loop2: mySFSession[2].increment()=31
Sat Jan 20 13:04:17 EST 2007 loop2: mySFSession[3].increment()=41
Sat Jan 20 13:04:18 EST 2007 loop2: mySFSession[4].increment()=51
```

Here is the output information provided by the client and the application server console:

```
Sat Jan 20 13:04:09 EST 2007:SFSessionBean:4039138 ejbCreate(9)
Sat Jan 20 13:04:09 EST 2007:SFSessionBean:4039138 increment() ctr=9
Sat Jan 20 13:04:10 EST 2007:SFSessionBean:5775816 ejbCreate(19)
Sat Jan 20 13:04:10 EST 2007:SFSessionBean:5775816 increment() ctr=19
Sat Jan 20 13:04:11 EST 2007:SFSessionBean:2897995 ejbCreate(29)
Sat Jan 20 13:04:11 EST 2007:SFSessionBean:2897995 increment() ctr=29
Sat Jan 20 13:04:12 EST 2007:SFSessionBean:6664484 ejbCreate(39)
Sat Jan 20 13:04:12 EST 2007:SFSessionBean:6664484 increment() ctr=39
Sat Jan 20 13:04:13 EST 2007:SFSessionBean:4034480 ejbCreate(49)
Sat Jan 20 13:04:13 EST 2007:SFSessionBean:4034480 increment() ctr=49
Sat Jan 20 13:04:14 EST 2007:SFSessionBean:4039138 increment() ctr=10
Sat Jan 20 13:04:15 EST 2007:SFSessionBean:5775816 increment() ctr=20
Sat Jan 20 13:04:16 EST 2007:SFSessionBean:2897995 increment() ctr=30
Sat Jan 20 13:04:17 EST 2007:SFSessionBean:6664484 increment() ctr=40
Sat Jan 20 13:04:18 EST 2007:SFSessionBean:4034480 increment() ctr=50
Sat Jan 20 13:04:19 EST 2007:SFSessionBean:4039138 ejbRemove() ctr=11
Sat Jan 20 13:04:19 EST 2007:SFSessionBean:5775816 ejbRemove() ctr=21
Sat Jan 20 13:04:19 EST 2007:SFSessionBean:2897995 ejbRemove() ctr=31
Sat Jan 20 13:04:19 EST 2007:SFSessionBean:6664484 ejbRemove() ctr=41
Sat Jan 20 13:04:19 EST 2007:SFSessionBean:4034480 ejbRemove() ctr=51
```

Using Entity Beans (Prior to EJB 3.0)

An entity bean is an EJB that models data that is typically stored in a relational database management system (RDBMS). Usually, it exists for as long as the data associated with it exists.

An entity bean is an *object* representation of data. An object is generally considered to be any entity with two specific attributes: *state* and *functionality*.

Entity beans implement this criteria and are therefore considered objects. An entity bean contains a copy of persisted data; therefore, it has a state. It also contains business logic; therefore, it has functionality. With these properties, entity beans can provide the same types of benefits as those of object databases.

Entity beans allow data to be persisted as Java objects as opposed to existing as rows in a table. They enable the enterprise bean provider to associate Java objects abstractly with relational database components (see Figure 7-5). Subsequently, the EJB deployer is able to map these abstract relational components to existing persistence devices.

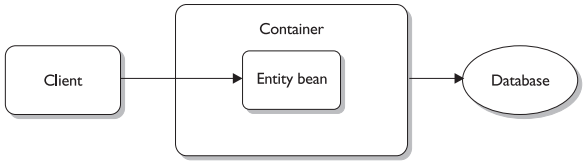


FIGURE 7-5

Entity bean representing RDBMS-based data

Entity beans access relational databases as well as enterprise systems or applications used to persist data (see Figure 7-6). Entity beans are able to be written in a robust and object-oriented fashion so that they can be used with a variety of data sources or data streams.

Entity beans are most often used for representing a set of data, such as the columns in a database table, with each entity bean instance containing one element of that data set, such as a row from a database table.

Methods of the entity bean's home interface, as defined in the EJB specification, allow clients to read, insert, update, and delete entities in a database.

Entity bean instances hold a copy of persisted data. If multiple clients execute the same find operation, for example, all of them will get handles to the same entity bean instance. Each client will get a handle to the same logical instance of the bean, but based on the transactional mode, each client could actually be talking to a different physical instance of the bean class. If contention exists between entity bean calls, it is handled by regarding each call as a separate transaction.

Uses of Entity Beans

The following are common uses of entity beans:

- Entity beans can be used to enforce the integrity of data that will be persisted as well as data that might potentially be persisted.
- Entity beans can be reused to cache data, therefore saving trips to the database.

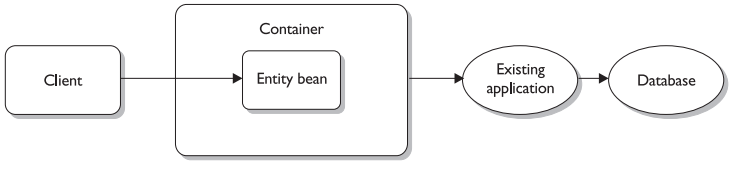


FIGURE 7-6

Entity beans representing legacy application–based data

- Entity beans can be used to model domain objects with unique identities that might be shared by multiple clients.
- Unlike session beans, entity beans are intended to model records in a data set, not to maintain conversations with clients.
- Entity beans can be used for wrapping JDBC code, hence giving the application an object-oriented interface for the data set.
- Entity beans can be wrapped by session beans, giving the developer more control in determining how clients can control data.
- Entity beans can be used in either bean-managed persistence (BMP) or container-managed persistence (CMP) mode. CMP mode should be used if at all possible, allowing the enterprise bean provider to concentrate on writing business logic instead of JDBC logic.

Entity Bean Life Cycle States

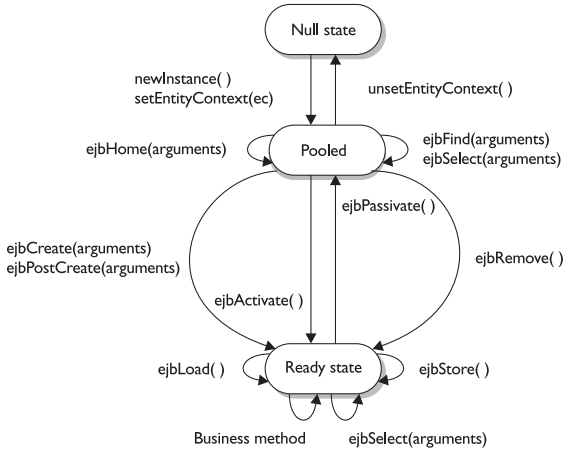
As they execute the bean methods described later in this objective, entity bean instances can be in one of three states (see Figure 7-7):

- **Null** The bean instance doesn't exist.
- **Pooled** The bean exists but isn't associated with any specific entity object.
- **Ready** The bean instance has been assigned an entity object identity.

The EJB container invokes the following entity bean interface methods when life cycle events occur. The enterprise bean provider is responsible for placing business

FIGURE 7-7

Various states of an entity bean



logic in this container to handle the events of the application. The container invokes the following methods:

- **ejbActivate()** An entity bean instance is chosen, removed from the available pool, and assigned to a specific *EJBObject*.
- **ejbLoad()** The container synchronizes its state by loading data from the underlying data source.
- **ejbPassivate()** An entity bean instance is about to be disassociated with a specific *EJBObject* and returned to the available pool.
- **ejbRemove()** An *EJBObject* that is associated with an entity bean instance is removed by a client-invoked remove operation on the entity bean's home or remote interface.
- **ejbStore()** The container needs to synchronize the underlying data source, or persistent state, with the entity bean instance by storing data to the underlying data source.

Developing Entity Beans

The preceding sections covered most of the components of an entity bean. The following exercise completes a checklist of tasks, annotated with helpful hints, required to develop an entity bean. Often, a development tool will be provided with the particular application or EJB server in use, such as SilverStream, *WebLogic*, or WebSphere. When using development tools, many of these tasks will be wrapped in a graphical user interface (GUI) integrated development environment (IDE) for creating interfaces and deployment descriptor files.

EXERCISE 7-3

Review Code for Entity Bean Using Container-Managed Persistence (Prior to EJB 3.0)

Provide the code for an entity bean. Your code should look similar to the code shown in the following sections. To increase your familiarity with this process, we review the code for an entity bean that uses CMP. We use a stepwise approach that covers all the things required to create, package, deploy, and call the bean.

1. Create the CMP Entity Bean Class The following CMP entity bean class example assumes that the following table definition exists in the database:

```
create table tb_product (
   productId   varchar(64),
   name        varchar(64),
   productPx   numeric(18),
   description varchar(64)
);
```

The in-line documentation points out the required methods along with the business methods:

```
package javaee.architect.EntityCMP;
import javax.ejb.*;
import java.util.*;
// CMP Entity Bean
abstract public class EntityCMPBean implements EntityBean {
  EntityContext entityContext;
  // Methods required by the EJB specification.
  // They are called by the container only.

  // This method corresponds to the create() method
  // found in the home interface.  Because this
  // is CMP, the method will return void and the
  // EJB Container will make the primary key.
  public java.lang.String ejbCreate(java.lang.String productId,
    java.lang.String name, java.lang.Double productPx,
    java.lang.String description) throws CreateException {
    log("ejbCreate()  [primary key="+productId+"]");
    // However, with CMP we must set the Bean's fields
    // with the parameters passed in, so that the EJB
    // Container is able to inspect our Bean and
    // insert the corresponding database entries.
    setProductId(productId);
    setName(name);
    setProductPx(productPx);
    setDescription(description);
    return null;
  }
  // Called after ejbCreate() method.
  public void ejbPostCreate(java.lang.String productId, java.lang.String name,
  java.lang.Double productPx,
ava.lang.String description) throws CreateException {
    log("ejbPostCreate() [primary key="+getProductId()+"]");
```

```
}
public void ejbRemove() throws RemoveException {
  log("ejbRemove() [primary key="+getProductId()+"]");
}
// Loads/re-loads the entity bean instance with
// the current value in database. We can leave
// this basically empty for CMP. The EJB container
// will set public fields to the current values.
public void ejbLoad() {
  log("ejbLoad() [primary key="+getProductId()+"]");
}
// Updates the database value with the current
// value of this entity bean instance. We can leave
// this basically empty for CMP. The EJB container
// will save public fields to the database.
public void ejbStore() {
  log("ejbStore() [primary key="+getProductId()+"]");
}
public void ejbActivate() {
  log("ejbActivate() [primary key="+getProductId()+"]");
}
public void ejbPassivate() {
  log("ejbPassivate() [primary key="+getProductId()+"]");
}
public void unsetEntityContext() {
  log("unsetEntityContext() [primary key="+getProductId()+"]");
  this.entityContext = null;
}
public void setEntityContext(EntityContext entityContext) {
  log("setEntityContext() called.");
  this.entityContext = entityContext;
}
// No finder methods are required because
// they are implemented by Container

// Abstract getters and setters
public abstract void setProductId(java.lang.String productId);
public abstract void setName(java.lang.String name);
public abstract void setProductPx(java.lang.Double productPx);
public abstract void setDescription(java.lang.String description);
public abstract java.lang.String getProductId();
public abstract java.lang.String getName();
public abstract java.lang.Double getProductPx();
public abstract java.lang.String getDescription();
```

```
 // Log message to console
  public void log(String msg) {
    System.out.println(Calendar.getInstance().getTime()
      +":EntityCMPBean:" + msg);
  }
}
```

2. Create the CMP Entity Bean Home Interface Here is a home interface for our CMP entity bean example:

```
package javaee.architect.EntityCMP;
import javax.ejb.*;
import java.util.*;
import java.rmi.*;
// This is the home interface, used by clients as a
// factory for EJB objects (remote references). The EJB
// container vendor implements this extended interface.
// In this home interface there is a create() method that
// corresponds to the ejbCreate() method in actual bean code.
public interface EntityCMPRemoteHome extends javax.ejb.EJBHome {
  // Finder methods that return one or
  // more EJB Objects (remote reference).
  // The functionality of these finder methods
  //  can be customized at deployment time
  public Collection findByName(String string)
    throws FinderException, RemoteException;
  public Collection findByDescription(String string)
    throws FinderException, RemoteException;
  public Collection findByProductPx(Double dbl)
    throws FinderException, RemoteException;
  public Collection findAllProducts()
    throws FinderException, RemoteException;
  public EntityCMPRemote findByPrimaryKey(String productId)
    throws FinderException, RemoteException;
  // Creates/returns the EJB Object (remote reference).
  public EntityCMPRemote create(String productId, String name,
    Double productPx, String description)
    throws CreateException, RemoteException;
}
```

3. Create the CMP Entity Bean Remote Interface Here is a remote interface for our CMP entity bean example. It is important to note that the following code is not supposed to demonstrate the *best practice* for the design of the remote interface.

An improved implementation would provide access via a session facade (see Chapter 5), which would simplify access and reduce potential network traffic.

```
package javaee.architect.EntityCMP;
import javax.ejb.*;
import java.util.*;
import java.rmi.*;
// This is the remote interface, used by clients when
// they need to call an EJB objects. The EJB container
// vendor implements this extended interface, which is
// responsible for delegating subsequent calls to the
// actual bean code.
public interface EntityCMPRemote extends EJBObject {
  // Getters and setters for Entity Bean fields.
  public String getProductId() throws RemoteException;
  public void setName(String name) throws RemoteException;
  public String getName() throws RemoteException;
  public void setProductPx(Double productPx) throws RemoteException;
  public Double getProductPx() throws RemoteException;
  public void setDescription(String description) throws RemoteException;
  public String getDescription() throws RemoteException;
}
```

4. Create Deployment Descriptors As mentioned in the first code review exercise in this chapter, the deployment descriptor, located within a JAR file, allows the properties of an EJB to be maintained outside of Java code.

Here is our deployment descriptor *(ejb-jar.xml)* for the CMP entity session bean. You will notice some EJB Query Language (EJB QL), which defines the queries for the finder and select methods of an entity bean with CMP.

```
<?xml version="1.0"?><!DOCTYPE ejb-jar PUBLIC
"-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans 2.0//EN"
"http://java.sun.com/dtd/ejb-jar_2_0.dtd">
<ejb-jar>
  <enterprise-beans>
    <entity>
      <display-name>EntityCMP</display-name>
      <ejb-name>EntityCMP</ejb-name>
      <home>javaee.architect.EntityCMP.EntityCMPRemoteHome</home>
      <remote>javaee.architect.EntityCMP.EntityCMPRemote</remote>
      <ejb-class>javaee.architect.EntityCMP.EntityCMPBean</ejb-class>
      <persistence-type>Container</persistence-type>
      <prim-key-class>java.lang.String</prim-key-class>
```

```
<reentrant>False</reentrant>
<cmp-version>2.x</cmp-version>
<abstract-schema-name>EntityCMP</abstract-schema-name>
<cmp-field>
  <field-name>productId</field-name>
</cmp-field>
<cmp-field>
  <field-name>name</field-name>
</cmp-field>
<cmp-field>
  <field-name>productPx</field-name>
</cmp-field>
<cmp-field>
  <field-name>description</field-name>
</cmp-field>
<primkey-field>productId</primkey-field>
<query>
  <query-method>
    <method-name>findByName</method-name>
    <method-params>
      <method-param>java.lang.String</method-param>
    </method-params>
  </query-method>
  <ejb-ql>WHERE name = ?1</ejb-ql>
</query>
<query>
  <query-method>
    <method-name>findByDescription</method-name>
    <method-params>
      <method-param>java.lang.String</method-param>
    </method-params>
  </query-method>
  <ejb-ql>WHERE description = ?1</ejb-ql>
</query>
<query>
  <query-method>
    <method-name>findByProductPx</method-name>
    <method-params>
      <method-param>java.lang.Double</method-param>
    </method-params>
  </query-method>
  <ejb-ql>WHERE productPx = ?1</ejb-ql>
</query>
```

```
      <query>
        <query-method>
          <method-name>findAllProducts</method-name>
          <method-params />
        </query-method>
        <ejb-ql>WHERE productId IS NOT NULL</ejb-ql>
      </query>
    </entity>
  </enterprise-beans>
  <assembly-descriptor>
    <container-transaction>
      <method>
        <ejb-name>EntityCMP</ejb-name>
        <method-name>*</method-name>
      </method>
      <trans-attribute>Required</trans-attribute>
    </container-transaction>
  </assembly-descriptor>
</ejb-jar>
```

5. Compile, Assemble, and Deploy the CMP Entity Bean The next steps in the process are to compile, assemble, and then deploy the entity bean. In our example, the following set of steps are used to complete the process. As part of the deployment process, references in the deployment descriptor need to be resolved to actual resources in the container. How these resources are resolved is, at the moment, container specific.

In the WebLogic environment, the following files can be used for the CMP Entity bean example:

No, "the following files" above refers to the *weblogic-ejb-jar.xml* and *weblogic-cmp-rdbms-jar.xml* files listed below—paHere's the code for the *weblogic-ejb-jar.xml* file:

```
<?xml version="1.0"?>
<!DOCTYPE weblogic-ejb-jar PUBLIC
"-//BEA Systems, Inc.//DTD WebLogic 8.1.0 EJB//EN"
"http://www.bea.com/servers/wls810/dtd/weblogic-ejb-jar.dtd" >
<weblogic-ejb-jar>
  <weblogic-enterprise-bean>
    <ejb-name>EntityCMP</ejb-name>
    <entity-descriptor>
      <entity-cache>
        <max-beans-in-cache>1000</max-beans-in-cache>
      </entity-cache>
      <persistence>
```

```
        <persistence-type>
          <type-identifier>WebLogic_CMP_RDBMS</type-identifier>
          <type-version>6.0</type-version>
          <type-storage>META-INF/weblogic-cmp-rdbms-jar.xml</type-storage>
        </persistence-type>
        <persistence-use>
          <type-identifier>WebLogic_CMP_RDBMS</type-identifier>
          <type-version>6.0</type-version>
        </persistence-use>
      </persistence>
    </entity-descriptor>
    <jndi-name>EntityCMPRemoteHome</jndi-name>
  </weblogic-enterprise-bean>
</weblogic-ejb-jar>
```

Here's the code for the *weblogic-cmp-rdbms-jar.xml* file:

```
<!DOCTYPE weblogic-rdbms-jar PUBLIC
 '-//BEA Systems, Inc.//DTD WebLogic 6.0.0 EJB RDBMS Persistence//EN'
 'http://www.bea.com/servers/wls600/dtd/weblogic-rdbms-persistence.dtd'>
<weblogic-rdbms-jar>
  <weblogic-rdbms-bean>
    <ejb-name>EntityCMP</ejb-name>
    <data-source-name>_dbPool</data-source-name>
    <table-name>tb_product</table-name>
    <field-map>
      <cmp-field>productId</cmp-field>
      <dbms-column>productId</dbms-column>
    </field-map>
    <field-map>
      <cmp-field>name</cmp-field>
      <dbms-column>name</dbms-column>
    </field-map>
    <field-map>
      <cmp-field>productPx</cmp-field>
      <dbms-column>productPx</dbms-column>
    </field-map>
    <field-map>
      <cmp-field>description</cmp-field>
      <dbms-column>description</dbms-column>
    </field-map>
  </weblogic-rdbms-bean>
</weblogic-rdbms-jar>
```

Here are the remaining steps for completing the compile/package/deployment process:

1. Compile the Java classes.
2. Package the classes and deployment descriptors into a JAR file.
3. Generate stub and tie code for the container and add them to the JAR file.
4. Deploy the JAR file to the application server.
5. Package the required classes for a remote client of the bean.

Here is an example of a client of the CMP entity bean:

```
package javaee.architect.EntityCMP;
import javax.ejb.*;
import javax.naming.*;
import javax.rmi.*;
import java.rmi.*;
import java.util.*;
// Client application for a CMP Entity Bean.
public class EntityCMPClient {
  public static void main(String[] args) throws Exception {
    EntityCMPRemoteHome home = null;
    try {
      // Lookup and get a reference to the home object
      Context ctx = getInitialContext();
      home = (EntityCMPRemoteHome) PortableRemoteObject.narrow(
         ctx.lookup("EntityCMPRemoteHome"), EntityCMPRemoteHome.class);
      // Create some new EJB Objects
      System.out.println("Adding new products...");
      home.create("10000", "Penne Pasta", new Double(15), "Bowl of pasta");
      home.create("10001", "Tomato Soup", new Double, "Bowl of Tomato soup");
      home.create("10002", "Apple Pie", new Double(3.5), "Large Apple pie");
      home.create("10003", "Milk", new Double(1), "Glass of milk");
      home.create("10004", "Juice", new Double(1), "Carton of apple juice");
      home.create("10005", "Juice", new Double(1), "Carton of cranberry juice");
      // Find a product and display its description
      Iterator i = home.findByName("Juice").iterator();
      System.out.println("Here are the products with name=Juice:");
      while (i.hasNext()) {
        EntityCMPRemote product = (EntityCMPRemote)
          PortableRemoteObject.narrow(i.next(), EntityCMPRemote.class);
        System.out.println(product.getDescription());
      }
      // Find all products that cost $1
      i = home.findByProductPx(new Double(1)).iterator();
      System.out.println("Here are the products that cost $1:");
```

```
    while (i.hasNext()) {
      EntityCMPRemote product = (EntityCMPRemote)
        PortableRemoteObject.narrow(i.next(), EntityCMPRemote.class);
      System.out.println(product.getDescription());
    }
  } catch (Exception e) {
    e.printStackTrace();
  } finally {
    if (home != null) {
      // Remove products added by this client
      System.out.println("Deleting products added..");
      Iterator i = home.findAllProducts().iterator();
      while (i.hasNext()) {
        try {
          EntityCMPRemote product = (EntityCMPRemote)
            PortableRemoteObject.narrow(i.next(), EntityCMPRemote.class);
          if (product.getProductId().startsWith("1000")) {
            product.remove();
          }
        }
        catch (Exception e) {
          e.printStackTrace();
        } // end try/catch
      } // end while
    } // end if
  } // end finally
} // end main
private static Context getInitialContext() throws Exception {
  // This implementation is specific to the Weblogic
  // server and will differ for other server vendors.
  String providerUrl = "t3://localhost:7001";
  String icFactory = "weblogic.jndi.WLInitialContextFactory";
  String user = null;
  String password = null;
  Properties properties = null;
  try {
    properties = new Properties();
    properties.put(Context.INITIAL_CONTEXT_FACTORY, icFactory);
    properties.put(Context.PROVIDER_URL, providerUrl);
    if (user != null) {
      properties.put(Context.SECURITY_PRINCIPAL, user);
      properties.put(Context.SECURITY_CREDENTIALS,
        password == null ? "" : password);
    }
    return new InitialContext(properties);
  }
```

```
    catch(Exception e) {
      System.out.println(
        "Unable to connect to JNDI server at " + providerUrl);
      throw e;
    }
  }
} // end class
```

Here is the output information provided by the client:

```
Adding new products...
Here are the products with name=Juice:
Carton of apple juice
Carton of cranberry juice
Here are the products that cost $1:
Toasted Sesame
Almond
Cranberry and Carrot
Glass of milk
Carton of apple juice
Carton of cranberry juice
Deleting products added..
```

Here is the output information provided by the client and the application server console:

```
Sat Jan 20 13:20:42 EST 2007:EntityCMPBean:setEntityContext() called.
Sat Jan 20 13:20:42 EST 2007:EntityCMPBean:ejbCreate()  [primary key=10000]
Sat Jan 20 13:20:42 EST 2007:EntityCMPBean:ejbPostCreate() [primary key=10000]
Sat Jan 20 13:20:42 EST 2007:EntityCMPBean:ejbStore() [primary key=10000]
Sat Jan 20 13:20:42 EST 2007:EntityCMPBean:setEntityContext() called.
Sat Jan 20 13:20:42 EST 2007:EntityCMPBean:ejbCreate()  [primary key=10001]
Sat Jan 20 13:20:42 EST 2007:EntityCMPBean:ejbPostCreate() [primary key=10001]
Sat Jan 20 13:20:42 EST 2007:EntityCMPBean:ejbStore() [primary key=10001]
Sat Jan 20 13:20:42 EST 2007:EntityCMPBean:setEntityContext() called.
Sat Jan 20 13:20:42 EST 2007:EntityCMPBean:ejbCreate()  [primary key=10002]
Sat Jan 20 13:20:42 EST 2007:EntityCMPBean:ejbPostCreate() [primary key=10002]
Sat Jan 20 13:20:42 EST 2007:EntityCMPBean:ejbStore() [primary key=10002]
Sat Jan 20 13:20:42 EST 2007:EntityCMPBean:setEntityContext() called.
Sat Jan 20 13:20:42 EST 2007:EntityCMPBean:ejbCreate()  [primary key=10003]
Sat Jan 20 13:20:42 EST 2007:EntityCMPBean:ejbPostCreate() [primary key=10003]
Sat Jan 20 13:20:42 EST 2007:EntityCMPBean:ejbStore() [primary key=10003]
Sat Jan 20 13:20:42 EST 2007:EntityCMPBean:setEntityContext() called.
Sat Jan 20 13:20:42 EST 2007:EntityCMPBean:ejbCreate()  [primary key=10004]
Sat Jan 20 13:20:42 EST 2007:EntityCMPBean:ejbPostCreate() [primary key=10004]
Sat Jan 20 13:20:42 EST 2007:EntityCMPBean:ejbStore() [primary key=10004]
Sat Jan 20 13:20:42 EST 2007:EntityCMPBean:setEntityContext() called.
Sat Jan 20 13:20:42 EST 2007:EntityCMPBean:ejbCreate()  [primary key=10005]
```

```
Sat Jan 20 13:20:42 EST 2007:EntityCMPBean:ejbPostCreate() [primary key=10005]
Sat Jan 20 13:20:42 EST 2007:EntityCMPBean:ejbStore() [primary key=10005]
Sat Jan 20 13:20:42 EST 2007:EntityCMPBean:setEntityContext() called.
Sat Jan 20 13:20:42 EST 2007:EntityCMPBean:setEntityContext() called.
Sat Jan 20 13:20:42 EST 2007:EntityCMPBean:setEntityContext() called.
Sat Jan 20 13:20:42 EST 2007:EntityCMPBean:ejbActivate() [primary key=10004]
Sat Jan 20 13:20:42 EST 2007:EntityCMPBean:ejbLoad() [primary key=10004]
Sat Jan 20 13:20:42 EST 2007:EntityCMPBean:ejbActivate() [primary key=10005]
Sat Jan 20 13:20:42 EST 2007:EntityCMPBean:ejbLoad() [primary key=10005]
Sat Jan 20 13:20:42 EST 2007:EntityCMPBean:ejbStore() [primary key=10004]
Sat Jan 20 13:20:42 EST 2007:EntityCMPBean:ejbStore() [primary key=10005]
Sat Jan 20 13:20:42 EST 2007:EntityCMPBean:ejbLoad() [primary key=10004]
Sat Jan 20 13:20:42 EST 2007:EntityCMPBean:ejbStore() [primary key=10004]
Sat Jan 20 13:20:42 EST 2007:EntityCMPBean:ejbLoad() [primary key=10005]
Sat Jan 20 13:20:42 EST 2007:EntityCMPBean:ejbStore() [primary key=10005]
Sat Jan 20 13:20:42 EST 2007:EntityCMPBean:setEntityContext() called.
Sat Jan 20 13:20:42 EST 2007:EntityCMPBean:setEntityContext() called.
Sat Jan 20 13:20:42 EST 2007:EntityCMPBean:setEntityContext() called.
Sat Jan 20 13:20:42 EST 2007:EntityCMPBean:setEntityContext() called.
Sat Jan 20 13:20:42 EST 2007:EntityCMPBean:setEntityContext() called.
Sat Jan 20 13:20:42 EST 2007:EntityCMPBean:setEntityContext() called.
Sat Jan 20 13:20:42 EST 2007:EntityCMPBean:ejbActivate() [primary key=10]
Sat Jan 20 13:20:42 EST 2007:EntityCMPBean:ejbLoad() [primary key=10]
Sat Jan 20 13:20:42 EST 2007:EntityCMPBean:ejbActivate() [primary key=40]
Sat Jan 20 13:20:42 EST 2007:EntityCMPBean:ejbLoad() [primary key=40]
Sat Jan 20 13:20:42 EST 2007:EntityCMPBean:ejbActivate() [primary key=50]
Sat Jan 20 13:20:42 EST 2007:EntityCMPBean:ejbLoad() [primary key=50]
Sat Jan 20 13:20:42 EST 2007:EntityCMPBean:ejbActivate() [primary key=10003]
Sat Jan 20 13:20:42 EST 2007:EntityCMPBean:ejbLoad() [primary key=10003]
Sat Jan 20 13:20:42 EST 2007:EntityCMPBean:ejbActivate() [primary key=10004]
Sat Jan 20 13:20:42 EST 2007:EntityCMPBean:ejbLoad() [primary key=10004]
Sat Jan 20 13:20:42 EST 2007:EntityCMPBean:ejbActivate() [primary key=10005]
Sat Jan 20 13:20:42 EST 2007:EntityCMPBean:ejbLoad() [primary key=10005]
Sat Jan 20 13:20:42 EST 2007:EntityCMPBean:ejbStore() [primary key=10]
Sat Jan 20 13:20:42 EST 2007:EntityCMPBean:ejbStore() [primary key=40]
Sat Jan 20 13:20:42 EST 2007:EntityCMPBean:ejbStore() [primary key=50]
Sat Jan 20 13:20:42 EST 2007:EntityCMPBean:ejbStore() [primary key=10003]
Sat Jan 20 13:20:42 EST 2007:EntityCMPBean:ejbStore() [primary key=10004]
Sat Jan 20 13:20:42 EST 2007:EntityCMPBean:ejbStore() [primary key=10005]
Sat Jan 20 13:20:42 EST 2007:EntityCMPBean:ejbLoad() [primary key=10]
Sat Jan 20 13:20:42 EST 2007:EntityCMPBean:ejbStore() [primary key=10]
Sat Jan 20 13:20:42 EST 2007:EntityCMPBean:ejbLoad() [primary key=40]
Sat Jan 20 13:20:42 EST 2007:EntityCMPBean:ejbStore() [primary key=40]
Sat Jan 20 13:20:42 EST 2007:EntityCMPBean:ejbLoad() [primary key=50]
Sat Jan 20 13:20:42 EST 2007:EntityCMPBean:ejbStore() [primary key=50]
```

```
Sat Jan 20 13:20:42 EST 2007:EntityCMPBean:ejbLoad() [primary key=10003]
Sat Jan 20 13:20:42 EST 2007:EntityCMPBean:ejbStore() [primary key=10003]
Sat Jan 20 13:20:42 EST 2007:EntityCMPBean:ejbLoad() [primary key=10004]
Sat Jan 20 13:20:42 EST 2007:EntityCMPBean:ejbStore() [primary key=10004]
Sat Jan 20 13:20:42 EST 2007:EntityCMPBean:ejbLoad() [primary key=10005]
Sat Jan 20 13:20:42 EST 2007:EntityCMPBean:ejbStore() [primary key=10005]
Sat Jan 20 13:20:42 EST 2007:EntityCMPBean:setEntityContext() called.
Sat Jan 20 13:20:42 EST 2007:EntityCMPBean:setEntityContext() called.
Sat Jan 20 13:20:42 EST 2007:EntityCMPBean:setEntityContext() called.
Sat Jan 20 13:20:42 EST 2007:EntityCMPBean:setEntityContext() called.
Sat Jan 20 13:20:42 EST 2007:EntityCMPBean:setEntityContext() called.
Sat Jan 20 13:20:42 EST 2007:EntityCMPBean:setEntityContext() called.
Sat Jan 20 13:20:42 EST 2007:EntityCMPBean:setEntityContext() called.
Sat Jan 20 13:20:42 EST 2007:EntityCMPBean:setEntityContext() called.
Sat Jan 20 13:20:42 EST 2007:EntityCMPBean:setEntityContext() called.
Sat Jan 20 13:20:42 EST 2007:EntityCMPBean:setEntityContext() called.
Sat Jan 20 13:20:42 EST 2007:EntityCMPBean:setEntityContext() called.
Sat Jan 20 13:20:42 EST 2007:EntityCMPBean:setEntityContext() called.
Sat Jan 20 13:20:42 EST 2007:EntityCMPBean:ejbActivate() [primary key=10]
Sat Jan 20 13:20:42 EST 2007:EntityCMPBean:ejbLoad() [primary key=10]
Sat Jan 20 13:20:42 EST 2007:EntityCMPBean:ejbActivate() [primary key=20]
Sat Jan 20 13:20:42 EST 2007:EntityCMPBean:ejbLoad() [primary key=20]
Sat Jan 20 13:20:42 EST 2007:EntityCMPBean:ejbActivate() [primary key=30]
Sat Jan 20 13:20:42 EST 2007:EntityCMPBean:ejbLoad() [primary key=30]
Sat Jan 20 13:20:42 EST 2007:EntityCMPBean:ejbActivate() [primary key=40]
Sat Jan 20 13:20:42 EST 2007:EntityCMPBean:ejbLoad() [primary key=40]
Sat Jan 20 13:20:42 EST 2007:EntityCMPBean:ejbActivate() [primary key=50]
Sat Jan 20 13:20:42 EST 2007:EntityCMPBean:ejbLoad() [primary key=50]
Sat Jan 20 13:20:42 EST 2007:EntityCMPBean:ejbActivate() [primary key=60]
Sat Jan 20 13:20:42 EST 2007:EntityCMPBean:ejbLoad() [primary key=60]
Sat Jan 20 13:20:42 EST 2007:EntityCMPBean:ejbActivate() [primary key=10000]
Sat Jan 20 13:20:42 EST 2007:EntityCMPBean:ejbLoad() [primary key=10000]
Sat Jan 20 13:20:42 EST 2007:EntityCMPBean:ejbActivate() [primary key=10001]
Sat Jan 20 13:20:42 EST 2007:EntityCMPBean:ejbLoad() [primary key=10001]
Sat Jan 20 13:20:42 EST 2007:EntityCMPBean:ejbActivate() [primary key=10002]
Sat Jan 20 13:20:42 EST 2007:EntityCMPBean:ejbLoad() [primary key=10002]
Sat Jan 20 13:20:42 EST 2007:EntityCMPBean:ejbActivate() [primary key=10003]
Sat Jan 20 13:20:42 EST 2007:EntityCMPBean:ejbLoad() [primary key=10003]
Sat Jan 20 13:20:42 EST 2007:EntityCMPBean:ejbActivate() [primary key=10004]
Sat Jan 20 13:20:42 EST 2007:EntityCMPBean:ejbLoad() [primary key=10004]
Sat Jan 20 13:20:42 EST 2007:EntityCMPBean:ejbActivate() [primary key=10005]
Sat Jan 20 13:20:42 EST 2007:EntityCMPBean:ejbLoad() [primary key=10005]
Sat Jan 20 13:20:42 EST 2007:EntityCMPBean:ejbStore() [primary key=10]
Sat Jan 20 13:20:42 EST 2007:EntityCMPBean:ejbStore() [primary key=20]
Sat Jan 20 13:20:42 EST 2007:EntityCMPBean:ejbStore() [primary key=30]
```

```
Sat Jan 20 13:20:42 EST 2007:EntityCMPBean:ejbStore() [primary key=40]
Sat Jan 20 13:20:42 EST 2007:EntityCMPBean:ejbStore() [primary key=50]
Sat Jan 20 13:20:42 EST 2007:EntityCMPBean:ejbStore() [primary key=60]
Sat Jan 20 13:20:42 EST 2007:EntityCMPBean:ejbStore() [primary key=10000]
Sat Jan 20 13:20:42 EST 2007:EntityCMPBean:ejbStore() [primary key=10001]
Sat Jan 20 13:20:42 EST 2007:EntityCMPBean:ejbStore() [primary key=10002]
Sat Jan 20 13:20:42 EST 2007:EntityCMPBean:ejbStore() [primary key=10003]
Sat Jan 20 13:20:42 EST 2007:EntityCMPBean:ejbStore() [primary key=10004]
Sat Jan 20 13:20:42 EST 2007:EntityCMPBean:ejbStore() [primary key=10005]
Sat Jan 20 13:20:42 EST 2007:EntityCMPBean:ejbLoad() [primary key=10]
Sat Jan 20 13:20:42 EST 2007:EntityCMPBean:ejbStore() [primary key=10]
Sat Jan 20 13:20:42 EST 2007:EntityCMPBean:ejbLoad() [primary key=20]
Sat Jan 20 13:20:42 EST 2007:EntityCMPBean:ejbStore() [primary key=20]
Sat Jan 20 13:20:42 EST 2007:EntityCMPBean:ejbLoad() [primary key=30]
Sat Jan 20 13:20:42 EST 2007:EntityCMPBean:ejbStore() [primary key=30]
Sat Jan 20 13:20:43 EST 2007:EntityCMPBean:ejbLoad() [primary key=40]
Sat Jan 20 13:20:43 EST 2007:EntityCMPBean:ejbStore() [primary key=40]
Sat Jan 20 13:20:43 EST 2007:EntityCMPBean:ejbLoad() [primary key=50]
Sat Jan 20 13:20:43 EST 2007:EntityCMPBean:ejbStore() [primary key=50]
Sat Jan 20 13:20:43 EST 2007:EntityCMPBean:ejbLoad() [primary key=60]
Sat Jan 20 13:20:43 EST 2007:EntityCMPBean:ejbStore() [primary key=60]
Sat Jan 20 13:20:43 EST 2007:EntityCMPBean:ejbLoad() [primary key=10000]
Sat Jan 20 13:20:43 EST 2007:EntityCMPBean:ejbStore() [primary key=10000]
Sat Jan 20 13:20:43 EST 2007:EntityCMPBean:ejbLoad() [primary key=10000]
Sat Jan 20 13:20:43 EST 2007:EntityCMPBean:ejbRemove() [primary key=10000]
Sat Jan 20 13:20:43 EST 2007:EntityCMPBean:ejbLoad() [primary key=10001]
Sat Jan 20 13:20:43 EST 2007:EntityCMPBean:ejbStore() [primary key=10001]
Sat Jan 20 13:20:43 EST 2007:EntityCMPBean:ejbLoad() [primary key=10001]
Sat Jan 20 13:20:43 EST 2007:EntityCMPBean:ejbRemove() [primary key=10001]
Sat Jan 20 13:20:43 EST 2007:EntityCMPBean:ejbLoad() [primary key=10002]
Sat Jan 20 13:20:43 EST 2007:EntityCMPBean:ejbStore() [primary key=10002]
Sat Jan 20 13:20:43 EST 2007:EntityCMPBean:ejbLoad() [primary key=10002]
Sat Jan 20 13:20:43 EST 2007:EntityCMPBean:ejbRemove() [primary key=10002]
Sat Jan 20 13:20:43 EST 2007:EntityCMPBean:ejbLoad() [primary key=10003]
Sat Jan 20 13:20:43 EST 2007:EntityCMPBean:ejbStore() [primary key=10003]
Sat Jan 20 13:20:43 EST 2007:EntityCMPBean:ejbLoad() [primary key=10003]
Sat Jan 20 13:20:43 EST 2007:EntityCMPBean:ejbRemove() [primary key=10003]
Sat Jan 20 13:20:43 EST 2007:EntityCMPBean:ejbLoad() [primary key=10004]
Sat Jan 20 13:20:43 EST 2007:EntityCMPBean:ejbStore() [primary key=10004]
Sat Jan 20 13:20:43 EST 2007:EntityCMPBean:ejbLoad() [primary key=10004]
Sat Jan 20 13:20:43 EST 2007:EntityCMPBean:ejbRemove() [primary key=10004]
Sat Jan 20 13:20:43 EST 2007:EntityCMPBean:ejbLoad() [primary key=10005]
Sat Jan 20 13:20:43 EST 2007:EntityCMPBean:ejbStore() [primary key=10005]
Sat Jan 20 13:20:43 EST 2007:EntityCMPBean:ejbLoad() [primary key=10005]
Sat Jan 20 13:20:43 EST 2007:EntityCMPBean:ejbRemove() [primary key=10005]
```

A Closer Look at Entity Beans (Prior to EJB 3.0)

This section provides a more in-depth look at the issues that need to be addressed when you're developing EJB component methods and configuration files.

Primary Keys

As mentioned, primary key classes can map to either a single field or multiple fields of an entity bean.

Single-field mappings are the simpler of the two cases. In this scenario, the *primkey-field* element in the deployment descriptor is used to specify which field specified by the container-managed element is to be used as the primary key. The *primkey-field* and the container-managed elements are required to be the same type.

In mapping to multiple fields, the fields of the primary key class must be a subset of the container-managed fields. The primary key class, its parameterless constructor, and all primary key class fields are required to be declared as public.

The Unknown Primary Key Class If the choice of the primary key field or fields is to be delayed until deployment time, the `findByPrimaryKey()` method, always a single-object finder, must be declared as *java.lang.Object*. The *prim-key-class* element in the deployment descriptor must also be *java.lang.Object*.

The isIdentical() Method Client applications can test to determine whether different object references are pointing to the same entity object by using the `isIdentitcal()` method.

The following example uses this method:

```
// Get Home references
CustomerRemoteHome custHome1 = (CustomerRemoteHome)
                            javax.rmi.PortableRemoteObject.narrow(
                            initialContext.lookup("java:ucny/um2z8/customers"),
                            CustomerRemoteHome.class);

CustomerRemoteHome custHome2 = (CustomerRemoteHome)
                            javax.rmi.PortableRemoteObject.narrow(
                            initialContext.lookup("java:ucny/um2z8/customers"),
                            CustomerRemoteHome.class);
// Get Remote references
Customer customer1 = custHome1.findByPrimaryKey(100);
Customer customer2 = custHome2.findByPrimaryKey(100);
if (customer1.isIdentical(customer2))
  System.out.println ("identical objects.");
else
  System.out.println ("non-identical objects");
```

Object Equality The EJB framework doesn't specify *object equality*—that is, the use of the `==` operator. Instead, the `isIdentical()` method should be used to support this functionality.

The equals() Method The `java.lang.Object equals()` method relies heavily on memory addresses. For this reason, the enterprise bean provider should either override this method or simply not use it. It is recommended that the `isIdentical()` method be used when possible.

The hashCode() Method The EJB framework doesn't specify the behavior of the `Object.hashCode()` method on object references pointing to entity objects. Instead, the `isIdentical()` method should be used to determine whether two entity object references refer to the same entity object. If the enterprise bean provider wants to use the `hashCode()` method, the following condition must be enforced:

```
if custHome1.equals(custHome2) {
   if (custHome1.hashCode() == custHome2.hashCode())
      System.out.println(this + " is implemented correctly");
   else
      System.out.println(this + " is NOT implemented correctly");
}
```

The EntityBean Class and Life Cycle Event Methods

Much like the primary key class, the bean class can be extended from any Java class. If the enterprise bean provider extends an *EntityBean* class from another Java class, the methods of the base class will be available to the client by defining the base class' methods in the stub for the *EntityBean*.

The Public Constructor To control the initial state of all objects, the enterprise bean provider creates a public constructor. This constructor enables the container to create stable instances of the *EntityBean* class. It is specified that this constructor should take no arguments. The container calls the public constructor to create a bean instance. The `ejbCreate()` and `ejbPostCreate()` methods are invoked to initialize that bean instance.

Accessor Methods When using CMP, the enterprise bean provider doesn't make direct read and write calls to persistent storage devices. Instead, relational data is accessed via `get` and `set` accessor methods. These accessor methods, as well as the persistent fields and relationships, are declared in the abstract persistence schema of the XML deployment descriptor file.

The *cmr-field-name* element in the abstract persistence schema corresponds to the name used for the `get` and `set` accessor methods used for the relationship. The *cmr-field-type* element is used only for collection-valued cmr-fields.

The ejbCreate( ) Method When you're using the `create()` and `remove()` methods, it is important to note the difference between session beans and entity beans. When these methods are used with session beans, bean objects are being created and destroyed. When these methods are used with entity beans, records in a database are being created and destroyed.

An entity bean can have zero or more `ejbCreate()` methods. However, the signature of each method must map to the entity bean home interface `create()` methods. When a client invokes a `create()` method to create an entity object, the container invokes the appropriate `ejbCreate()` method.

After the `ejbCreate()` method is completed for a CMP entity bean, the EJB container performs a database insert.

The ejbPostCreate( ) Method For each `ejbCreate()` method declared, a matching `ejbPostCreate()` method with a void return type should be declared. Immediately after the EJB container invokes the `ejbCreate()` method on an EntityBean instance, it will call the corresponding `ejbPostCreate()` method on that same instance. The `ejbPostCreate()` method can be used to refine the instance created by the `ejbCreate()` method before this instance becomes available to the client.

The `ejbCreate()` method can be used to initialize persistent data, whereas the `ejbPostCreate()` method might do initialization involving the entity's context. Context information isn't available while the `ejbCreate()` method is being invoked, but it is available when the `ejbPostCreate()` is being invoked.

After the `ejbPostCreate()` method is invoked, the instance can discover the primary key by calling the `getPrimaryKey()` method on its entity context object.

The ejbRemove( ) Method When a container invokes the `ejbRemove()` method on a bean instance or a client calls the corresponding `remove()` method in its remote home or remote interface, it not only removes the entity bean instance, but it also destroys physical data that is related to the bean instance.

Another way to destroy an entity object and its corresponding physical data is by use of the deployment descriptor's cascade-delete deployment descriptor element, contained in the *ejb-relationship-role* element.

The ejbFind( ) Method An `ejbFind()` method is defined for each of the `find()` methods in the home interface. This includes at least an `ejbFindByPrimaryKey()` method, which returns the primary key to the container.

The setEntityContext( ) Method For an entity bean instance to use its *EntityContext* information during the instance's lifetime, the bean instance must save the state of the *EntityContext* internally. The following code example accomplishes this:

```
EntityContext myContext;
public void setEntityContext(EntityContext ctx)
{
  myContext = ctx;
}
```

The unsetEntityContext( ) Method The container calls this method before terminating the entity bean instance:

```
public void unsetEntityContext()
{
  myContext = null;
}
```

The ejbLoad( ) Method When in ready state in the bean pool, an entity bean must keep its state synchronized with underlying data. During the call to the `ejbLoad()` method, the data is read from the database and stored in the instance variables. In addition, calling the `ejbStore()` method results in its saving the entity bean state to the database.

The container invokes the `ejbLoad()` method right after a bean is instantiated or when a transaction begins.

The ejbStore( ) Method When data is to be persisted to a permanent storage medium, the EJB container invokes the `ejbStore()` method.

The ejbPassivate( ) Method The EJB container invokes this method on the entity bean instance when the EJB container decides to return that instance to the bean pool. This method allows the entity bean instance to release any resources that shouldn't be held while in the bean pool. Therefore, if resources are created or acquired during the execution of the `ejbActivate()` method, they should be freed in the `ejbPassivate()` method.

The ejbActivate( ) Method The EJB container invokes this method when an entity bean instance is chosen from the bean pool and is assigned a specific *object identity (OID)*. This method lets the entity bean acquire or create any necessary resources while the bean is in the ready state.

Home Interfaces and create( ) Methods

Two types of home interfaces can be used. The first is the *remote* home interface, which implements the *EJBHome* interface. The second is the *local* home interface, which implements the *EJBLocalHome* interface. The remote interface creates entity beans that can be accessed from outside of the EJB container by clients. The local interface creates entity beans that are accessed inside the EJB container by clients. The performance gained by using local entity beans is significant because clients get access to entity bean objects by *reference* as opposed to getting access to them by *value*. This saves time that would have been spent communicating with RMI-IIOP.

The name of each `create()` method starts with the prefix *create*. Unique signatures for `create()` methods are made using different method names, as long as the prefix starts with *create*, as well as by overloading `create()` methods with different arguments.

When an entity bean is first created, the `ejbCreate()` method creates a primary key that uniquely identifies that entity object. For the life of the entity bean object, the entity bean is associated with this primary key. As a result, the entity bean object can be retrieved by providing this primary key object to the `findByPrimaryKey()` method.

A bean's home interface can declare zero or more `create()` methods. A `create()` method exists for each different way of creating an entity object. Each of these `create()` methods must have corresponding `ejbCreate()` and `ejbPostCreate()` methods in the bean class. These creation methods are linked at runtime; when a `create()` method is invoked on the home interface, the container delegates the invocation to the corresponding `ejbCreate()` and `ejbPostCreate()` methods on the bean class. Here's an example:

```
public interface ShoppingCartHome extends javax.ejb.EJBHome
{
  public ShoppingCart create(String firstName, String lastName,
     int idNumber) throws RemoteException, CreateException;

  public ShoppingCart create(Integer customerNumber, int idNumber)
  throws RemoteException, CreateException;
}
```

A client can create a new entity object with code similar to this:

```
ShoppingCartHome scartHome = ...
ShopingCart scart = scartHome.create("E", "Matias", 641224);
```

Home Interfaces and finder( ) Methods

As mentioned, when an entity bean is first created, the `ejbCreate()` method creates a primary key that uniquely identifies that entity object. For the life of the entity bean object, the entity bean is associated with this primary key. Therefore, the entity bean object can be retrieved by providing the `findByPrimaryKey()` method of the home interface with a unique primary key object. This primary key is defined by the developer and can be of any type as long as it is unique.

One or more `finder()` methods can be defined in the home interface. One method should exist for every way required to find an entity bean object or collection of entity bean objects. `finder()` methods start with a prefix *find*. The arguments of a `finder()` method are used to locate the desired entity objects. The return type of a `finder()` method is an instance of an entity bean or a collection of entity bean instances.

The following example shows the `findByPrimaryKey()` method:

```
public interface ItemHome extends javax.ejb.EJBLocalHome
{
  //...
  public Item findByPrimaryKey(String number)
  throws FinderException, RemoteException;
  public Item findItemByDescription(String description)
  throws FinderException, RemoteException;
  public Collection findAllItems()
  throws FinderException, RemoteException;
}
```

The following example demonstrates how a client uses the `findByPrimaryKey()` method:

```
ItemRemoteHome itemHome = (ItemRemoteHome)
                          javax.rmi.PortableRemoteObject.narrow(
                          initialContext.lookup("java:ucny/um2z8/items"),
                          ItemRemoteHome.class);
Item item = itemHome.findByPrimaryKey("27018301820A");
```

With BMP, the entity bean provider doesn't write the corresponding `ejbFind()` method for `finder()` methods. `finder()` methods are configured by the development tools provided by the EJB container and then created when the entity bean is deployed. With BMP, the bean provider does have to write the `ejbFind()` methods.

Home Interfaces and remove( ) Methods

The `remove()` method of *EJBHome* interface supports the destruction of entity bean objects using either a handle object or a primary key object.

The `remove()` method of an *EJBLocalHome* interface supports the destruction of entity bean objects using only the primary key object.

When a `remove()` method is invoked on an entity object, the container invokes the entity bean instance's `ejbRemove()` method. After the `ejbRemove()` method returns from its invocation, but before the `remove()` method acknowledges its completion to the client, the EJB container removes the entity object from all relationships in which it participates and then removes its persistent representation.

Home Interfaces and getEJBMetaData( ) Methods

The `getEJBMetaData()` method returns a reference to an object that implements the *EJBMetaData* interface. The *EJBMetaData* interface allows application assembly tools to discover metadata information about an entity bean. This metadata information allows for loose client/server binding.

Remote Interfaces

Remote interfaces extend the *EJBObject* interface. They specify what methods of an entity object, created by the enterprise bean provider while implementing the *EntityBean* interface, can be accessed by a client. The client can exist either inside or outside of the EJB container. If the client exists inside the EJB container, as in the case of a servlet application, the local interface should be used instead. Remote interfaces allow clients to

- Obtain the remote home interface for the entity object.
- Remove the entity object.
- Obtain the entity object's handle.
- Obtain the entity object's primary key.

Local Interfaces

Local interfaces extend the *EJBLocalObject* interface. These interfaces specify which methods of an entity object, created by the enterprise bean provider while implementing the *EntityBean* interface, can be accessed by a client that is local to the EJB container. The *EJBLocalObject* interface defines methods that allow the local client to

- Obtain the local home interface for the entity object.
- Remove the entity object.
- Obtain the entity object's primary key.

Local interfaces can be used as a replacement for JDBC—supporting fast calls to a database while the client runs inside the same JVM as the entity bean.

CERTIFICATION OBJECTIVE 7.05

State the Benefits and Costs of Container-Managed Persistence

One of the evolving aspects of the entity bean is its *persistence*. Before this persistence can be meaningful, you need to understand the details of developing entity beans. Here, you will learn about the uses of entity beans, the entity bean life cycle states, and the steps involved in developing entity beans. You will take a close look at developing entity beans and review how to manage persistence. You'll also learn the benefits and drawbacks of both CMP and BMP.

Managing Persistence (Prior to EJB 3.0)

Bean persistence can be managed in two ways: You can let the container manage the persistence of the bean via container-managed persistence (CMP), or you can use bean-managed persistence (BMP). This method requires that the developer implement the interaction code between the EJB and the persistence engine. This is a much more complicated task than opting for the first option. This mode should not be seen as a standard development model, but more as a means to get to and implement the lower level persistence mechanism. In other words, *use BMP only*

when the limits of CMP have been exceeded. It is not realistic to consider using this far more complicated model for every EJB that you need to build. The CMP model should be considered as the general persistence model for most of an application's development.

When the container handles the overhead necessary to support a bean in a manner that is satisfactory to the enterprise bean provider, it stands to reason that more of the enterprise bean provider's development efforts can be focused on actually writing business logic.

However, if the CMP engine provided with the Application Server at hand is not sufficient and there are no available resources to implement BMP and there is already so much commitment to the current Application Server that changing Application Servers is not an option, then integrating a third-party CMP engine with the current Application Server is another option that may be considered. Even though being able to plug-in a third-party CMP engine into an Application Server is not part of the Java EE specification, certain Application Servers will allow the integration of certain third-party CMP engines. This ability of should be considered when deciding which Application Server is to be used for a project.

Container-Managed Persistence

In CMP, entity bean data is maintained automatically by the container that uses the mechanism of its choosing. For example, a container implemented on top of a RDBMS may manage persistence by storing each bean's data as a row in a table. The container may also use the Java programming language serialization functionality for persistence. When a programmer chooses to utilize CMP beans, the programmer specifies which fields are to be retained.

A persistence manager is used to separate the management of bean relationships from the container. With this separation, a container has the responsibility for managing security, transactions, and so on, while the persistence manager is able to manage different databases via different containers. Using this architecture allows entity beans to become more portable across EJB vendors.

Entity beans are mapped to the database using a bean-persistence manager contract called the *abstract persistence schema*. The persistence manager is responsible for implementing and executing find methods based on EJB QL.

The persistence manager generates a mapping of CMP objects to a persistent data store's objects based on the information provided by the deployment descriptor, the bean's abstract persistence schema, and settings set by the deployer. Persistent data stores may include relational databases, flat files, Enterprise Resource Planning (ERP) systems, and so on.

A contract between the CMP entity bean and the persistence manager allows for defining complex and portable bean-to-bean, bean-to-dependent, and even dependent-to-dependent object relationships within an entity bean. For the persistence manager to be separated from the container, a contract between the bean and the persistence manager is defined.

When EJB is deployed, the persistence manager is used to generate an instantiated implementation of the EJB class and its dependent object classes. It does this based on its XML deployment descriptor and the bean class. The instantiated implementation will include the data access code that will read and write the state of the EJB to the database at runtime. When this happens, the container uses the subclasses generated by the persistence manager instead of the EJBs abstract classes defined by the bean provider. The persistence manager of the EJB container is also used to manage the persistence of the bean state or data. As previously mentioned, the enterprise bean provider in this scenario is able to concentrate on implementing business logic with Java code. A bean using CMP is simpler than one that uses BMP; however, it is also dependent on a container for database access.

Using CMP, the EJB container is responsible for saving the bean's state. Because the persistence process is container managed, the Java code used to retrieve and store data is independent of the data source. On the other hand, the container-managed fields do need to be specified in the deployment descriptor file so the EJB container's persistence manager can automatically handle the persistence process.

Some benefits and drawbacks of CMP are detailed next.

CMP Pros Benefits of container-managed persistence include the following:

- **Database-independence** The container, not the enterprise bean provider, maintains database access code to most popular databases.
- **Container-specific features** Features such as full-text search are available for use by the enterprise bean provider.

CMP Cons Drawbacks to container-managed persistence include the following:

- **Algorithms** Only container-supported algorithms persistence can be used.
- **Portability** Portability to other EJB containers may be lost.
- **Access** The developer has no access to the view and cannot modify the actual code.
- **Efficiency** Sometimes, the generated SQL is not the most efficient with respect to performance.

Bean-Managed Persistence

When using BMP, the bean is responsible for storing and retrieving persisted data. The entity bean interface provides methods for the container to notify an instance when it needs to store or retrieve data.

In BMP mode, the EJB must implement persistence. To do this, methods such as `ejbStore()` and `ejbLoad()` must be created and used, and communication with SQL databases must be implemented manually using JDBC.

BMP works well when data is being persisted to something other than a relational database, such as file system or a legacy enterprise application. When a bean manages its own persistence, it must also define its own JDBC calls. In this case, the entity bean is directly responsible for saving its own state. On the other hand, the container isn't required to make any database calls. Some benefits and drawbacks of BMP are detailed next.

BMP Pros Benefits of bean-managed persistence include the following:

- **Container independent** Entity bean code written for one EJB container should be easily portable to any other certified EJB container.
- **Standards based** The standard EJB and JDBC APIs can be used for data access calls.
- **Datatype access** The ability to access nonstandard datatypes and legacy applications is supported.
- **Maximum flexibility** Data validation logic of any complexity is supported.
- **Database specific features** The application is able to take advantage of nonstandard SQL features of different SQL servers.

BMP Cons Drawbacks to bean-managed persistence include the following:

- **Database specific** Because entity bean code is database specific, if access to multiple databases is required, the enterprise bean provider will have to account for this in its data access methods.
- **Knowledge of SQL** The enterprise bean provider must have knowledge of SQL.
- **Development time** These beans on average take much longer time to develop—as much as five times longer.

The Composition of a CMP Entity Bean

The many components of EJB entity beans must interact with each other and with their container. The following is a summary of the *javax.ejb* package entity bean components and the methods they use. The enterprise bean provider creates some of these components. The EJB container, building on what the enterprise bean provider created, generates the remainder of the required components.

Components Created by the Enterprise Bean Provider The enterprise bean provider is responsible for the creation of classes and interfaces. Business logic is placed in classes. Interfaces are used to define how the EJB container should build supporting objects as well as to define what business methods are to be visible to clients. The following sections describe varieties of these components in further detail.

The Primary Key Class The *primary key* is a unique identifier of an object. The primary key class doesn't necessarily have to relate directly to the primary key of a database table. Quite often, it isn't necessary to create a primary key class because you can use Java objects already defined in the entity bean class (for example, *strings* or *integers*) as primary keys. Note that Java's primitive datatypes (such as *int*) can be used only when wrapped by Java objects. A primary key class is defined within the deployment descriptor file. Although an entity bean class can define a unique class as its primary key, it is still possible for several entity beans to share the same primary key class.

The Remote Home Interface—Interface EJBHome The enterprise bean provider creates the remote home interface so that the EJB container can create an *EJBHome* object. The *EJBHome* object can be used to create, destroy, and find entity bean objects inside a home domain. When the EJB container implements the remote home interface, it enables clients to

- Create new entity objects within a home domain
- Find existing entity objects within a home domain
- Remove an entity object from a home domain
- Execute a home instance business method
- Obtain a handle to the home interface
- Get metadata information allowing for loose client/server binding and scripting

Zero or more `create()` methods, with the prefix name *create*, can be defined in the *EJBHome* interface. Each entity object should have one or more `create()` methods. Each method's argument should be used to initialize the state of the entity object.

One `finder()` method should be defined for each different way of finding an entity object or a collection of entity objects within a home domain. The name of a `finder()` method should begin with the prefix *find*. The arguments of the method are used to locate desired entity objects. The return type must be either the entity bean's remote interface or a type that is a collection of objects implementing the entity bean's remote interface.

You can also define `remove()` methods to remove entity objects that are qualified by either a handle or a primary key. Note that `home()` methods are static methods provided by the enterprise bean provider for global business logic that isn't specific to any bean instance; `home()` methods can have any name that doesn't begin with *create*, *find*, or *remove*.

The Home Handle—Interface HomeHandle A client can use the home handle after it loses the JNDI name of a remote home interface. If a home handle is returned as the result of a `getEJBHome()` method, the client will be required to narrow the results to get the remote home interface.

The Handle—Interface Handle The handle object created according to this interface is an abstraction on top of a network reference to an *EJBObject*. It can be used as a persistent reference to that object.

The Remote Interface—Interface EJBObject An entity bean's remote interface is used to define which methods in the abstract entity bean class will be visible to the client, the so-called *business methods*. In addition, other required methods of the remote interface allow the client to

- Obtain the handle of an entity object
- Obtain the home interface of an entity object
- Remove an entity object

The Local Home Interface—Interface EJBLocalHome The local home interface is similar to the remote home interface; however, it is faster because the local objects and the clients that call them are inside the same EJB container.

As a result, these local objects don't need to support the overhead associated with distributed applications.

The Local Interface—Interface EJBLocalObject Local interfaces are similar to the remote interface, having the benefit of speed like that of the local home interface when the bean and the client are executing within the same JVM instance.

The Remote Client A remote client accesses an entity bean via methods defined in both the remote interface and remote home interface of the entity bean. The EJB container provides remote Java objects that implement those interfaces. These objects are accessible from a client through the standard Java APIs for remote object invocation. A remote client of an entity object can be another enterprise bean deployed in either the same or in a different container. A client can also be an arbitrary Java program, such as an application, applet, or servlet. In addition, the remote client view of an entity bean can be mapped to non-Java client environments such as CORBA.

The Local Client A local client resides in the same JVM and EJB container as the entity bean it uses. In the interest of speed, a local client should access an entity bean through that entity bean's local and local home interfaces. Processing speed is gained because arguments of the methods of the local interface and local home interface are passed by reference instead of by value.

The EntityContext The EJB container provides entity bean instances with an EntityContext. This EntityContext gives an instance access to the references of any objects associated with the instance, including *EJBLocalObjects*, *EJBObjects*, and primary key objects. This access is provided by the `getEJBLocalObject()`, `getEJBObject()`, and `getPrimaryKey()` methods, respectively.

The instance is also given access to information returned by the following methods, which are inherited from the EJBContext object:

- **getEJBHome()** Returns the remote home interface of the entity bean.
- **getEJBLocalHome()** Returns the local home interface of the entity bean.
- **getCallerPrincipal()** Returns the *java.security.Principal,* identifying the invoker of the bean instance's *EJBObject*.
- **isCallerInRole()** Tests whether the caller of the entity bean instance has a particular role.

- **setRollbackOnly()** Marks the current transaction so that a rollback is the only outcome of that transaction.
- **getRollbackOnly()** Tests to see whether the current transaction of an instance has been marked for rollback.
- **getUserTransaction()** Entity bean instances must not call this method, which returns the *javax.transaction.UserTransaction* interface.

The XML Deployment Descriptor The *deployment descriptor* is used to declare entity bean persistent fields (cmp-fields) as well as field relationships (cmr-fields). It contains information about entity bean persistence and container-managed relationships in the form of XML elements. This information, known as the *abstract persistence schema,* includes the following:

- The ejb-name for each entity bean, which must be unique within an ejb-jar file.
- The abstract-schema-name for each entity bean, which must be unique within an ejb-jar file. This name can be used when specifying EJB QL queries.
- The ejb-relation set, containing a pair of ejb-relationship-role elements.
- The ejb-relationship-role, which describes a relationship role, including its name, its multiplicity within a relation, and its navigability. The name of the cmr-field is specified from the perspective of the relationship participant. Each relationship role refers to an entity bean via an ejb-name element contained in the relationship-role-source element.

Container-Created Objects

After all the required interfaces and abstract classes are designed and developed by the enterprise bean provider, they are introspected by EJB container. It examines the entity bean's deployment descriptor to create concrete objects that have the additional functionality required to integrate the business logic with the requirements of the EJB distributed object framework. These classes include the following:

- **Bean class** Extends the abstract entity bean class, which implements the *EntityBean* interface.
- **EJBHome class** Implements the remote home interface *EJBHome*. The EJB container makes these instances accessible to the clients through JNDI.

- **EJBObject class** Implements the remote interface *EJBObject*.
- **EJBLocalHome class** Implements the local home interface *EJBLocalHome*.
- **EJBLocalObject class** Implements the local interface *EJBLocalObject*.

Figure 7-8 illustrates EJB objects provided by beans and containers. Entity bean objects are considered to be *persistent objects*; their lifetime isn't limited by the lifetime of the JVM process in which the entity bean instance executes. To illustrate, a JVM crash might result in a rollback of a transaction, but it will neither destroy previously created entity objects nor invalidate references interfaces held by clients.

EJB Clients (Prior to EJB 3.0)

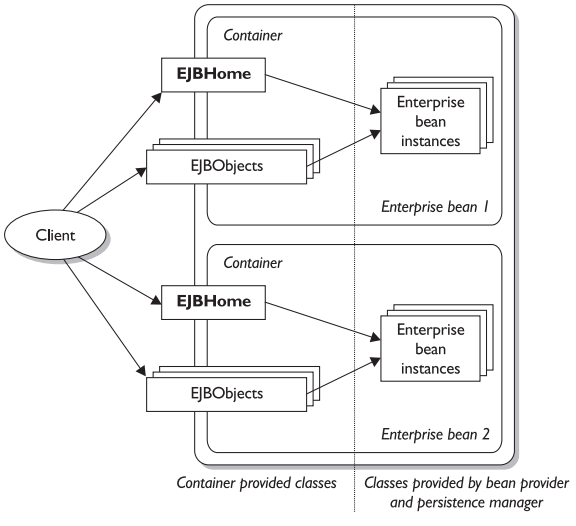
Two types of EJB clients exist: *remote* clients that exist outside of an EJB container and *local* clients that exist inside of an EJB container.

Multiple clients can access an entity object simultaneously while the EJB container synchronizes access to the entity objects via a *transaction manager*.

EJB containers make home interfaces available in a JNDI name space, therefore making these home interfaces available to clients. The home interfaces of entity beans allow clients to create, find, and remove entity objects within the enterprise bean's home domain. These interfaces also allow clients to execute static home business methods that aren't specific to a particular entity bean object.

FIGURE 7-8

EJBObjects provided by enterprise bean provider and EJB container



Remote Clients

Remote clients can be enterprise beans deployed in the same or different EJB containers, stand-alone Java applications or applets via Java APIs for remote object invocation, or non-Java clients such as CORBA clients.

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

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

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

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

The physical location of the EJB container is usually transparent to the client. A client locates an entity bean's home interface by using the JNDI, which enables applications to access multiple naming and directory services using a single interface. A client's JNDI name space can be configured to contain the home interfaces of enterprise beans located on multiple EJB containers on multiple machines on a network.

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

The remote home interface *ItemRemoteHome* for the *ItemMasterBean* entity bean can be located using the following code segment:

```
Context  initialContext = new InitialContext();
ItemRemoteHome itemHome = (ItemRemoteHome)
 javax.rmi.PortableRemoteObject.narrow(
initialContext.lookup("java:ucny/um2z8/items"), ItemRemoteHome.class);
```

Locating an entity bean's local home interface using JNDI is accomplished in a similar manner. It doesn't, however, involve the APIs for remote access. For example, if the *Item* entity bean provided a local home interface rather than a remote home interface, a local client might use the following code segment:

```
Context initialContext  = new InitialContext();
ItemLocalHome itemHome = (ItemLocalHome)
initialContext.lookup("java:ucny/um2z8/items");
```

Local Clients

Local clients access entity beans through local and local home interfaces with the arguments of the local and local home methods being passed by reference. The enterprise bean provider should be aware that argument objects shared between local clients and entity beans can be modified by either the local clients or the entity beans.

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

- Receive the reference as a result of a method call.
- Find the entity object using a `finder()` method defined in the entity bean's local home interface.

A local client that has a reference to an entity object's local interface can invoke business methods on the entity object through the local interface.

Entity Beans (EJB 3.0)

The CMP entity bean has been overhauled with EJB 3.0. Entity beans are now Plain Old Java Objects (POJOs) and component interfaces are no longer required. The bean class is now a concrete class and entity beans now support inheritance and polymorphism. Here is a summary of the major enhancements for EJB 3.0 entity beans:

- Java language annotations (including Object Relational Mapping)
- Support for inheritance and polymorphism
- Simpler lightweight persistence model for Create, Read, Update and Delete (CRUD) operations using Java Persistence Layer's EntityManager API
- Enhanced query capabilities

CERTIFICATION OBJECTIVE 7.06

State the Transactional Behavior in a Given Scenario for an Enterprise Bean Method with a Specified Transactional Deployment Descriptor

In this section, we review transactions, transaction management, and distributed transactions. Then we discuss the objective concerning transaction attribute settings as well as multiple transactions, the Java Transaction Service (JTS), the Java Transaction API (JTA), and the effect of the transaction attribute on entity and session bean methods.

Transactions and Transaction Management

A *transaction* is one or more tasks that execute as a single atomic operation or unit of work. If all tasks involved in a transaction do not proceed successfully, an inverse task or rollback procedure for all tasks is performed, setting all resources back to their original state. Transactions are characterized by the acronym *ACID,* which stands for Atomic, Consistent, Isolated, and Durable.

The EJB container provides the services and management functions required to support transaction demarcation, transactional resource management, synchronization, and transaction context propagation.

Since JDBC operates at the level of an individual database connection, it does not support transactions that span across multiple data sources. To compensate for this, the JTA provides access to the services offered by a transaction manager. If an EJB requires control of global transaction, it can get access to JTA via the container.

Distributed Transactions

Although the EJB framework can be used to implement a nontransactional system, the model was designed to support distributed transactions. EJB framework requires the use of a distributed transaction management system that supports two-phase commit protocols for flat transactions.

In addition to container-managed transactions, an EJB may participate in client-managed and bean-managed transactions.

The EJB architecture provides automatic support for distributed transactions in component-based applications. Such distributed transactions can automatically update data in multiple databases or even data distributed across multiple sites. The EJB model shifts the complexities of managing these transactions from the application developer to the container provider.

Transaction-Management Paradigms The Java Platform EE platform supports two transaction-management paradigms: *declarative transaction demarcation* and *programmatic transaction demarcation*.

Declarative transaction management refers to a nonprogrammatic demarcation of transaction boundaries, achieved by specifying within the deployment descriptor the transaction attributes for the various methods of the container-managed EJB component. This is a flexible approach that facilitates changes in the application's transactional characteristics without modifying any code. Container-managed transaction demarcation must be used by entity EJB components.

Transaction Attribute Settings

A transaction attribute supports declarative transaction demarcation and conveys to the container the intended transactional behavior of the associated EJB component's method. Six transaction attributes are possible for container-managed transaction demarcation:

- **`NotSupported` [or `@TransactionAttribute(NOT_SUPPORTED)` annotation in EJB 3.0]** The bean runs outside the context of a transaction. Existing transactions are suspended during method calls. The bean cannot be invoked within a transaction. An existing transaction is suspended until the method called in this bean completes.
- **`Required` [or `@TransactionAttribute(REQUIRED)` annotation in EJB 3.0]** Method calls require a transaction context. If a transaction already exists, the bean will use it; if a transaction does not exist, it will be created. The container starts a new transaction if no transaction exists.
- **`Supports` [or `@TransactionAttribute(SUPPORTS)` annotation in EJB 3.0]** Method calls use the current transaction context if one exists, but they don't create one if none exists. The container will not start a new transaction. If a transaction already exists, the bean will be included in that transaction. Note that with this attribute, the bean can run without a transaction.

- **`RequiresNew` [or `@TransactionAttribute(REQUIRES_NEW)` annotation in EJB 3.0]** Containers create new transactions before each method call on the bean and commit transactions before returning. A new transaction is always started when the bean method is called. If a transaction already exists, that transaction is suspended until the new transaction completes.
- **`Mandatory` [or `@TransactionAttribute(MANDATORY)` annotation in EJB 3.0]** Method calls require a transaction context. If one does not exist, an exception is thrown. An active transaction must already exist. If no transaction exists, the *javax.ejb.TransactionRequired* exception is thrown.
- **`Never` [or `@TransactionAttribute(NEVER)` annotation in EJB 3.0]** Method calls require that no transaction context be present. If one exists, an exception is thrown. The bean must never run with a transaction. If a transaction exists, the *java.rmi.RemoteException* exception is thrown.

Multiple Transactions

A container can manage multiple transactions in two different ways: The container could instantiate multiple instances of the bean, allowing the transaction management of the DBMS to handle any transaction processing issues. Conversely, the container could acquire an exclusive lock on the instance's state in the database, serializing access from multiple transactions to this instance.

Java Transaction Service

The JTS specifies the implementation of a transaction manager supporting the JTA. JTS also implements the Java mapping of the Object Management Group's (OMG) Object Transaction Service (OTS). The EJB specification suggests but does not require transactions based on the JTS API. JTS supports distributed transactions, which have the ability to span multiple databases on multiple systems coordinated by multiple transaction managers. By using JTS, an EJB container ensures that its transactions can span multiple EJB containers.

Java Transaction API

EJB applications communicate with a transaction service using the JTA, which provides a programming interface to start transactions, join existing transactions, commit transactions, and roll back transactions.

When a bean with bean-managed transactions is invoked, the container suspends any current transaction in the client's context. In its method implementation,

the bean will initiate the transaction through the JTA *UserTransaction* interface. In stateful beans, the container will associate the bean instance with the same transaction context across subsequent method calls until the transaction is explicitly completed by the bean. However, stateless beans aren't allowed to maintain transaction context across method calls. Each method invocation is required to complete any transaction it initiates.

Entity Bean Methods and Transaction Attributes

All developer-defined methods in the remote interface as well as all methods defined in the home interface (such as `create()`, `remove()`, and `finder()` methods) require transaction attributes. Note that entity beans have to use container-managed transactions (CMTs).

Session Bean Methods and Transaction Attributes

All developer-defined methods in the remote interface require transaction attributes. Transaction attributes are *not* needed for the methods in the home interface.

Methods in the remote interface run with the *NotSupported* attribute by default. Transaction attributes are also not needed for the methods in a session bean if you're using bean managed transactions (BMTs).

CERTIFICATION OBJECTIVE 7.07

Given a Requirement Specification Detailing Security and Flexibility Needs, Identify Architectures That Would Fulfill Those Requirements

Here, we provide a basic review of security and the EJB framework as an architecture, including containers and their functionality.

Security

To simplify the development process for the enterprise bean provider, the implementation of the security infrastructure is left to the EJB container provider and the task of defining security policies is left to the bean deployer. By avoiding putting hard-coded security policies inside bean code, EJB applications gain flexibility when configuring and reconfiguring security policies for complex

enterprise applications. Applications also gain portability across different EJB servers that may use different security mechanisms.

The EJB framework specifies flexibility with regard to security management, allowing it to be declarative (container-managed) or programmatic (bean-managed).

Container-Managed or Declarative Security

The security management that defines method permissions is usually declared in the enterprise bean's deployment descriptor or by using annotations (if using EJB 3.0). Container-managed security makes an enterprise bean more flexible, since it isn't tied to the security roles defined by a particular application.

A *security role* is a name given to a grouping of information resource access permissions that are defined for an application. Associating a principal with this security role grants the associated access permissions to that principal as long as the principal is "in" the role.

Here is an excerpt from a deployment descriptor *(ejb-jar.xml)* for an entity bean that is using container-managed security:

```
<assembly-descriptor>
...
  <security-role>
    <role-name>adm_role</role-name>
  </security-role>
  <method-permission>
    <description>only remote access</description>
    <role-name>adm_role</role-name>
    <method>
      <ejb-name>EntityBMP</ejb-name>
      <method-intf>Remote</method-intf>
      <method-name>withdraw</method-name>
    </method>
  </method-permission>
...
</assembly-descriptor>
```

The `<method-permission>` element identifies the only security role that is allowed to invoke the withdraw method on the remote interface. The `<method-permission>` element consists of an optional description, a list of security role names, and a list of method elements. The `<security-role>` element contains the definition of a security role used by the bean. The security roles used in the `<method-permission>` element must be defined in the `<security-role>` elements of the deployment descriptor, and the methods must be defined in the enterprise bean's interfaces.

You should also note that errors in bean code programming are less likely to be a factor in causing security holes when using container-managed security, because the container implements the security mechanism. These features make container-managed method access the preferred security method.

Bean-Managed or Procedural Security

However, programmatic (procedural) access control is sometimes necessary to satisfy fine-grained or application-specific conditions. Enterprise beans can programmatically manage their own security by using the `isCallerInRole()` and `getCallerPrincipal()` methods contained on the EJBs context object. The `isCallerInRole()` method tests whether the caller has a given security role, returning true if the caller has and false if not. The `getCallerPrincipal()` method returns the *java.security.Principal* that identifies the caller.

Here is an excerpt of code from a EJB that uses these methods in a bean-managed security situation:

```
...
  public void deposit(double amt) {
    if (amt >= 10000) {
      if (entityContext.isCallerInRole("admin")) {
        this.balance += amt;
      } else {
        log("REJECTED deposit(" + amt + ") by user "
          +entityContext.getCallerPrincipal().getName());
        throw new EJBException(
          "You do not have permission to deposit $10,000 or more");
      }
    } else {
      this.balance += amt;
    }
    log("deposit(" + amt + ") by user "
      +entityContext.getCallerPrincipal().getName()
      +" balance="+this.balance);
  }
...
```

The `deposit()` method here uses the `isCallerInRole()` method to determine whether the caller depositing more than $10,000 is in the "admin" role. If the caller is in this role, the operation is accepted and the balance is updated. If the caller is not in the "admin" role, the operation is rejected and an exception is thrown.

The enterprise bean developer is responsible for defining all the security role names that are used in the bean code. Each of these role names must be added to the deployment descriptor in the form of a `<security-role-ref>` element. Part of this element is the `<role-link>` element that associates the role name to a security role defined elsewhere in the descriptor file.

Security roles are defined with the element `<role-name>`. The following deployment descriptor fragment defines a role name *admin*, which is associated via a `<role-link>` element to role *adm_role*.

```
....
<enterprise-beans>
    ...
    <entity>
        <ejb-name>EntityBMP</ejb-name>
        <ejb-class>EntityBMPBean.class</ejb-class>
        ...
        <security-role-ref>
            <role-name>admin</role-name>
            <role-link>adm_role</role-link>
        </security-role-ref>
        ...
    </entity>
</enterprise-beans>
   .....
```

In this EJB deployment descriptor, the *EntityBMPBean* class uses the symbolic name *admin* to check permissions. In the assembly descriptor section of the deployment descriptor, the security role *adm_role* is defined as follows:

```
....
<assembly-descriptor>
    <security-role>
        <role-name>adm_role</role-name>
    </security-role>
</assembly-descriptor>
....
```

For completeness, here is an excerpt from the WebLogic deployment descriptor *<weblogic-ejb-jar.xml>* file that resolves the role to an actual principal:

```
<weblogic-ejb-jar>
....
   <security-role-assignment>
```

```
      <role-name>adm_role</role-name>
      <principal-name>system</principal-name>
    </security-role-assignment>
....
</weblogic-ejb-jar>
```

This use of the deployment descriptor to define a role name and associate it with a role link allows different enterprise beans to use different internal names to refer to the same cluster of permissions. For example, another bean can still refer to the *adm_role* internally using the string *adm* instead of *admin*. A `<security-role-ref>` is able to associate that bean's *adm* reference to the security role *adm_role*.

You should also note that a user or principal is allowed to belong to multiple roles simultaneously. In doing so, the user/principal will benefit from the union set of permissions that those roles grant.

Security Not Covered by the EJB Specification

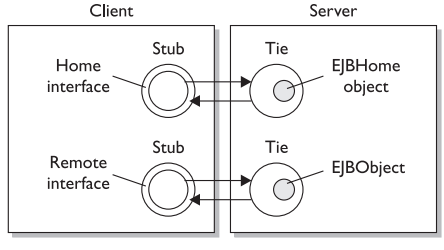
As opposed to access control, authentication and communication security are not specified in the EJB security guidelines. These aspects of security are left to the proprietary application server or the container.

EJB Framework (Prior to EJB 3.0)

The EJB components are declaratively customized. Customizable traits include transactional behavior, security features, life cycle, state management, and persistence. Figure 7-9 demonstrates how the home interface, remote interface, `EJBHome` object, and *`EJBObject`* of the EJB framework fit into the generic distributed programming model.

FIGURE 7-9

Distributed programming with EJB



Containers

The application server provides a container that supports services for components. A *container* is an entity that provides life cycle management, security, deployment, and runtime services to components. Each container type (including EJB, web, JSP, servlet, applet, and application client) provides component-specific services as well.

After a client invokes a server component, the container will automatically allocate a process thread and initiate the component. The container manages all resources on behalf of the component and interactions between the component and the external systems. A container provides EJB components with services such as the following:

- **Bean life cycle management and instance pooling** These services include creation, activation, passivity, and destruction. Individual EJBs do not need to explicitly manage process allocation, thread management, object activation, or object destruction. The EJB container automatically manages the object life cycle on behalf of the EJB.
- **State management** Individual EJBs do not need to explicitly save or restore the conversational object state between method calls. The EJB container automatically manages the object state on behalf of the EJB.
- **Bean transaction management** This service intercedes between client calls on the remote interface and the corresponding methods in a bean to enforce transaction and security constraints. It can provide notification at both the beginning and the ending of each transaction that involves a bean instance. Individual EJBs do not need to explicitly specify the transaction demarcation code to participate in distributed transactions. The EJB container can automatically manage the start, enrollment, commitment, and rollback of transactions on behalf of the EJB.
- **Security constraint enforcement** EJBs do not need to explicitly authenticate users or check authorization levels. The EJB container automatically performs all security checking on behalf of the EJB.
- **Distributed remote access** EJBs use technologies and protocols that are commonly used in distributed programming, such as RMI and IIOP.
- **Container-managed persistence** EJBs do not need to explicitly retrieve or store persistent object data from a database. The EJB container can automatically manage persistent data on behalf of the EJB. Entity beans can

either manage their own persistence or delegate those persistence services to their container. If persistence is delegated to the container, that container will also perform all data retrieval and storage operations automatically on behalf of the bean. (Note that the majority of the changes made between EJB 1.1 and EJB 2.0 are found in the definition of a new CMP component model. The new CMP model is extremely different from the previous CMP model because it introduces an entirely new entity, the persistence manager, and a completely new way of defining container-managed fields, as well as relationships with other beans and dependent objects.)

- **Generated remote stubs** The container will create remote stubs for wrappers such as RMI and CORBA.

Additional Functionality The EJB server provides an environment that supports the execution of applications developed using EJB architecture, managing and coordinating allocation of resources to the applications. The EJB server must provide one or more EJB containers, which provide homes for the EJBs. EJB containers manage the EJBs contained within them. For each EJB, the container is responsible for registering the object, providing a remote interface for the object, creating and destroying object instances, checking security for the object, managing the active state for the object, and coordinating distributed transactions. In addition, the container has the ability to manage all persistent data within the object.

Vendor-Specific Containers The exact environment for process and resource management, concurrency control, and thread pooling has not been defined in the EJB specification. So vendors can differentiate their products based on the simplicity or sophistication of the services provided by proprietary containers. A software vendor may choose to create a new application server to support EJB components specifically, or what is more likely, to adapt their existing servers.

Container Location Several EJB classes can be installed in a single EJB container. The physical implementation of an EJB container is not described in the EJB specification, so even though a particular class of EJB is assigned to a single EJB container, the container may not necessarily represent a physical location. The EJB container can be implemented as a physical unit, such as a single multithreaded process within a server, or it can be implemented as a logical unit that can be replicated and distributed across multiple systems and processes.

Benefits of Java Plaform EE and the EJB Framework as an Architecture

The use of the Java Platform EE and EJB framework as an architecture has the following primary benefits:

- EJB components are *server-side components* written entirely with the Java programming language; therefore, applications based on EJB components are not only platform independent but also middleware independent. They can run on any operating system and on any middleware that supports EJB.
- EJB components contain *business logic* only, giving developers freedom from maintaining system-level code that would be integrated with their business logic. The EJB server automatically manages system-level services such as transactions, security, life cycle, threading, and persistence for the EJB component.
- EJB architecture is inherently *transactional*, *distributed*, *portable*, *multi-tiered*, *scalable*, and *secure*.
- Java Platform EE architecture provides *authentication*, the means by which communicating entities prove to one another that they are acting on behalf of specific identities (for example, client to server and/or server to client).
- Java Platform EE architecture provides *authorization* (access control), the means by which interactions with resources are limited to collections of users or programs for the purpose of enforcing integrity, confidentiality, or availability constraints.
- Java Platform EE architecture provides *data integrity* (MAC, or message authentication check), which is the means used to prove that information could not have been modified by a third-party (for example, the recipient of data must be able to detect and discard messages that were modified after they were originally sent over an open network).
- Java Platform EE architecture provides *confidentiality* (data privacy), the means used to ensure that information is made available only to users who are authorized to access it.
- Java Platform EE architecture provides *nonrepudiation*, the means used to prove that a user performed some action such that the user cannot reasonably deny having done so.
- Java Platform EE architecture provides *auditing*, the means used to capture a tamper-resistant record of security-related events for the purpose of evaluating the effectiveness of security policies and mechanisms.

CERTIFICATION OBJECTIVE 7.08

Identify Costs and Benefits of Using an Intermediate Data Access Object Between an Entity Bean and the Data Resource

Here, we review data access objects for entity beans. This is covered in greater detail in Chapter 5.

Using Data Access Objects for Entity Beans

Sometimes if you decide that BMP is your best approach for entity beans, you must code the SQL into your entity beans. In a large development environment, it is sometimes a good idea to reduce the coupling of entity beans with the SQL necessary to select, insert, update, and delete data using EJB. To achieve reusability, developers create intermediate data access objects that contain implementation code to update and access the data source. The downside to the ease of reuse using a data access object is the additional layer and overhead of creation and garbage collection. Data access objects are typically developed in situations in which the developer is familiar with SQL and performance gains can be achieved over CMP, which sometimes does not provide the best performing SQL.

Why Use a Data Access Object?

The data access object (DAO) pattern separates the interface to a system resource from the underlying code used to access that resource. This allows the benefit of database vendor independence and the ability to represent XML data sources as objects.

Each enterprise bean that accesses a persistent data resource can have an associated DAO class, which defines an abstract API of operations on the resource. This abstract API will make no reference to the resource implementation. The DAO needs to know only how to load itself from a persistent store based on some identity information (such as a primary key) and how to store itself back to the persistent store. Therefore, an enterprise bean uses data it obtains from the DAO, defers the persistence responsibilities to the DAO, and can concentrate entirely on implementing business methods.

A DAO provides resource functionality for a particular resource, implemented for a particular persistence mechanism. For example, a class can be created to encapsulate the database resource access for both single as well as multiple rows. The benefits are independence achieved at a small cost (the development required to build the DAO) that ultimately may save development time.

For CMP entity beans, the EJB container automatically services all persistent storage access; therefore, applications using CMP entity beans do not need a DAO layer, since the application server transparently provides this functionality. However, DAOs may still be useful when an application has a combination of CMP for entity beans and BMP for session beans and/or servlets.

DAOs add a layer of objects between the data client and the data source that requires extra effort to be designed and implemented. However, the benefit realized by choosing this approach makes up for the added effort.

When a factory strategy is used, the hierarchy of concrete factories and concrete products produced by the factories requires additional design and implementation.

CERTIFICATION OBJECTIVE 7.09

State the Benefits of Bean Pooling in an EJB Container

Here, we review the concept of pooling resources. We explain how the EJB container uses this concept for EJBs.

Bean Pooling in the EJB Container

Given that the EJB container cannot handle an unlimited number of EJBs, the classic concept of *pooling* is used to share the resources among multiple users. When the EJB is deployed, the deployment descriptor specifies the number of instances to pool and reuse. The cost of creating and destroying an *EJBObject* can be high. The application server manages a pool of EJBs that can be used throughout the application. This pool allows the application server to handle more requests, since the server does not have to spend time creating and destroying *EJBObjects*. Note that stateful session beans cannot be a part of this bean pool.

Benefits of Bean Pooling in an EJB Container

Bean pooling is similar to *connection pooling*, a technique that was pioneered by the DBMS vendors to allow multiple clients to share a cached set of connections that provide access to a database resource.

Bean pooling is used in Java Platform EE for stateless session beans. To implement bean pooling, the application server, as part of a startup process, must create a pool manager object to control access to the stateless session beans. The client asks the pool manager to allocate a bean. If a bean is available in the pool, it is made available to the client immediately; otherwise, a bean is created. When a client no longer needs the bean, it returns the bean to the pool manager for reuse. This strategy allows beans to be quickly allocated to clients, avoiding the expense of setup and initialization.

CERTIFICATION OBJECTIVE 7.10

State the Benefits of Passivation in an EJB container

Here, we review the benefits of the passivation/activation technique that is employed by an EJB container.

Passivation/Activation

Passivation/activation is a technique that the EJB container can choose to temporarily serialize a bean and store it to the application server's file system or other persistent store is such a way to allow it to recreate the bean and its state at a later time. This technique allows the application to optimally manage resources.

Benefits of Bean Pooling in an EJB Container

The benefit of passivation is that it allows the EJB container to make the best possible use of server resources by passivating a bean to free up resources and then reactivating it when resources are available. Note that a session bean can be passivated only between transactions, and not within a transaction.

CERTIFICATION OBJECTIVE 7.11

Explain How the Enterprise JavaBeans Container Does Life Cycle Management and Has the Capability to Increase Scalability

Here, we review the life cycle of EJBs. We also describe their deployment, which is determined by the deployment descriptors settings.

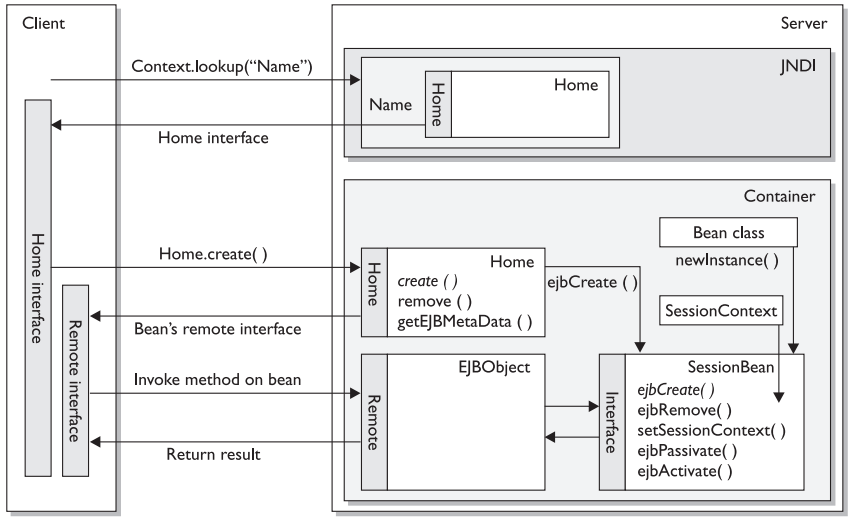
The Life Cycle of an EJB

Detailed documentation describing the life cycle of an EJB can be found in the EJB specifications on the Sun web site and is illustrated in Figure 7-10. The following list provides a general description of the life cycle states of an EJB session bean:

1. The client locates the bean's home reference using the JNDI services provided by the application server.
2. The JNDI service returns a home interface reference to the client.
3. The client uses the home interface reference to call the `home.create()` method. In response, the home object then creates an *EJBObject*. A new instance of the code in the bean class is also instantiated by the `newInstance()` method.
4. The new instance of the bean class, called a session bean, is allocated a session context.
5. The home object passes a reference to the *EJBObject* in the container to the client's remote interface.
6. The client's remote interface is now able to invoke methods on the *EJBObject* in the container. This *EJBObject* will pass these method calls to the session bean.
7. The session bean returns a result to the *EJBObject*, which in turn returns it to the client's remote interface.

FIGURE 7-10

Life cycle for an EJB session bean



How the EJB Container Manages Life Cycle and How This Allows for Increased Scalability

Here are the steps that the container takes to manage the life cycle of an entity bean:

1. The container populates the free pool with a working set of bean instances.
2. A client calls the `create()` method on a home object.
3. The home object obtains a bean instance from the free pool.
4. The home object forwards the `create()` arguments (if any) to the `ejbCreate()` method on the bean class.
5. The bean class inserts a row into the table in the database.
6. The bean class returns the primary key of the row to the home object.
7. The container creates an *EJBObject* for the bean class and sets its primary key.
8. The home object invokes the `ejbPostCreate()` method on the bean class to finish the initialization process now that the *EJBObject* can be referenced, because it now exists.
9. The home object returns the remote reference to the *EJBObject* back to the client.

10. The client can now invoke the business methods on the bean class (via the remote reference) that have been defined as available.
11. When the client is finished, the container moves the bean back to the free pool list after calling `ejbPassivate` (for an entity bean).

Note that a substantial overhead is incurred when instantiating bean instances. Scalability within the EJB container environment is increased by preinstantiating a pool of bean instances (bean pool) and allowing them to be quickly utilized by clients.

Deployment

When an EJB application is ready to be deployed to an EJB container, the desired beans and deployment information must be placed in a JAR file. The deployment information that is also placed in this JAR file is contained in an XML file called a *deployment descriptor*.

Deployment Descriptors As mentioned earlier, the deployment descriptor is an XML file containing elements that specify how to create and maintain EJB components and how to establish runtime service settings. The deployment descriptor contains settings that are not to be hard-coded inside EJB components. These settings tell the EJB container how to manage and control EJB components and can be set at application assembly time or at application deployment time.

Two basic types of elements are contained inside the deployment descriptor file:

- **Bean elements** These elements declare the internal structure and external dependencies of EJB components. The descriptor defines, among other things, the EJB class names, the JNDI namespace that represents the container, home interface names, and remote interface names.
- **Application assembly elements** These elements describe how EJB components are to be integrated into larger applications. Some of the application assembly elements describe environment values, security roles, and transaction attributes.

Packaging Hierarchies An important attribute of the EJB specification is that it not only provides the programming interfaces but also defines how the component/application has to be packaged. The deployment descriptor that has to go into the packaging is the standard way of customizing parameters of a specific installation.

EJB components can be packaged as individual EJBs, as a collection of EJBs, or as a complete application system. EJB components are distributed in a JAR file called an *ejb-jar* file. The ejb-jar file contains Java class files, as well as home and remote interfaces for EJBs. It also contains the XML deployment descriptor for the EJB.

Home and Remote Interfaces The client view is provided through the home interface and the remote interface. Classes constructed by the container when a bean is deployed, based on information provided by the bean, provide these interfaces. The home interface provides methods for creating a bean instance, while the remote interface provides the business logic methods for the component. By implementing these interfaces, the container can intercede in client operations of an EJB. This offers the client a simplified view of the component.

CERTIFICATION SUMMARY

If you have studied this chapter diligently, you should have an understanding of session and entity EJBs. You should also understand when it is appropriate to implement the different EJBs.

TWO-MINUTE DRILL

List the Required Classes/Interfaces That Must Be Provided for an Enterprise JavaBeans Component

- ❑ Prior to EJB 3.0, the required classes/interfaces that must be provided for an EJB component are the home *(EJBHome)* interface, the remote *(EJBObject)* interface, business logic (bean) class, context objects, and the XML deployment descriptor. For EJB 3.0, the only required class is an annotated bean class. The business interface can be generated by default and the XML deployment descriptor is now optional and largely unnecessary for simple EJBs.

Distinguish Between Session and Entity Beans

- ❑ A *session bean* is an EJB that is created by a client and usually exists only for the duration of a single client/server session.
- ❑ An *entity bean* is an object representation of persistent data maintained in a permanent data store such as a database. A primary key identifies each instance of an entity bean.

Recognize Appropriate Uses for Entity, Stateful Session, and Stateless Session Beans

- ❑ Use *stateful* session beans for functionality that requires data to be maintained across business logic method invocations.
- ❑ Use *stateless* session beans for functionality that does not require data to be maintained across business logic method invocations.

Distinguish Between Stateful and Stateless Session Beans

- ❑ Stateful session beans maintain data (state) across business logic method invocations.
- ❑ Stateless session beans do not maintain data (state) across business logic method invocations.
- ❑ Stateless session beans can utilize the bean-pooling feature of the EJB container.

State the Benefits and Costs of Container-Managed Persistence

- The benefits of CMP include database independence and container-specific features (such as full-text search). CMP has drawbacks, as only container-supported algorithms persistence can be used, and portability to other EJB containers may be lost.

State the Transactional Behavior in a Given Scenario for an Enterprise Bean Method with a Specified Transactional Deployment Descriptor

The following transactional behaviors can be identified for an enterprise bean method:

- In `NotSupported` [or `@TransactionAttribute(NOT_SUPPORTED)` annotation in EJB 3.0] transactional behavior, existing transactions are suspended during method calls. An existing transaction is suspended until the method called in this bean completes.
- In `Required` [or `@TransactionAttribute(REQUIRED)` annotation in EJB 3.0] transactional behavior, if an enterprise bean method already exists, it will be used. If one does not exist, it will be created.
- In `Supports` [or `@TransactionAttribute(SUPPORTS)` annotation in EJB 3.0] transactional behavior, the container will not start a new transaction, but if a transaction already exists, the bean will be included in that transaction.
- In `RequiresNew` [or `@TransactionAttribute(REQUIRES_NEW)` annotation in EJB 3.0] transactional behavior, a new transaction is always started when the bean method is called. If a transaction already exists, that transaction is suspended until the new transaction completes.
- In `Mandatory` [or `@TransactionAttribute(MANDATORY)` annotation in EJB 3.0] transactional behavior, if a transaction does not exist, an exception is thrown.
- In `Never` [or `@TransactionAttribute(NEVER)` annotation in EJB 3.0] transactional behavior, if a transaction exists, an exception is thrown.
- To encapsulate access to data, an application can use intermediate data access objects.
- The benefits of bean pooling in an EJB container include lowered cost, specific rates of pool reuse, and increased request handling by the application server.

Given a Requirement Specification Detailing Security and Flexibility Needs, Identify Architectures That Would Fulfill Those Requirements

The following is a list of some of the considerations when dealing with questions for the preceding objective:

- ❑ For EJB systems, deciding on Container-Managed or Declarative Security (flexibility) vs. Bean-Managed or Procedural Security (fine-grained approach)
- ❑ Which distributed object technology is most appropriate—RMI, CORBA, DCOM, or DCE
- ❑ Protocols supported in the deployed environment
- ❑ Ability for object serialization for transporting across a network

Identify Costs and Benefits of Using an Intermediate Data Access Object Between an Entity Bean and the Data Resource

To encapsulate access to data, an application can use intermediate data access objects. The use of separate objects to access data results in the following:

- ❑ Keeps entity bean code clear and simple
- ❑ Ensures easier migration to container-managed persistence for entity beans
- ❑ Allows for cross-database and cross-schema portability
- ❑ Provides a mechanism that supports tools from different vendors
- ❑ Not useful for CMP entity beans
- ❑ Adds an extra layer
- ❑ Needs more class hierarchy design when using a factory strategy

State the Benefits of Bean Pooling in an EJB Container

- ❑ The cost of creating and destroying an *EJBObject* can be expensive, so the concept of pooling is used to share the resources among multiple users.
- ❑ The external deployment descriptor specifies the number of instances to pool and reuse.
- ❑ This pool allows the application server to handle more requests, since the server does not have to spend time creating and destroy *EJBObjects*.

State the Benefits of Passivation in an EJB container

- Passivation allows the EJB container to make the best possible use of server resources by passivating a bean to free up resources and then reactivating it when resources are available.

Explain How the Enterprise JavaBeans Container Does Life Cycle Management and Has the Capability to Increase Scalability

- In the EJB container's life cycle management, while the container handles naming, management, transactional integrity, security, and persistence for the bean developer, the architect needs to determine the settings to tell the container how these concepts apply to a specific bean. This provides greater scalability.
- The mechanism for creating the deployment descriptors varies from application server to application server, but they all contain the same basic information.

SELF TEST

The following questions will help you measure your understanding of the material presented in this chapter. Read all the choices carefully because there might be more than one correct answer. Choose all correct answers for each question.

List the Required Classes/Interfaces That Must Be Provided for an Enterprise JavaBeans Component

1. Which of the following is not true about Enterprise JavaBeans (prior to EJB 3.0) objects?
   A. The home interface is responsible for locating or creating instances of the desired bean and returning remote references.
   B. The remote interface, or the *EJBObject* interface, typically provides method signatures for business methods.
   C. The bean implements either the *EntityBean* interface or the *SessionBean* interface but need not implement all the methods defined in the remote interface.
   D. The bean must implement one `ejbCreate()` method for each `create()` method in the home interface.
2. Which of the following is true about Enterprise JavaBeans (EJB 3.*x*) objects?
   A. The home interface is no longer required.
   B. The remote interface, or the *EJBObject* interface, typically provides method signatures for business methods.
   C. The bean class implements the *EJBInterface* class.
   D. The bean must be defined in the XML deployment descriptor.

Distinguish Between Session and Entity Beans

3. Which statement is not true when contrasting the use of entity beans and JDBC for database operations?
   A. Entity beans represent real data in a database.
   B. The bean managed entity bean functionally replaces the JDBC API.
   C. The container-managed entity bean automatically retrieves the data from the persistent storage (database).
   D. When using JDBC, you must explicitly handle the database transaction and the database connection pooling.

Recognize Appropriate Uses for Entity, Stateful Session, and Stateless Session Beans

4. Suppose that the business logic of an existing application is implemented using a set of CGI programs. Which Java technologies can be used to implement the CGI programs as a Java-based solution?
   A. JMAPI
   B. Screen scrapers
   C. Enterprise JavaBeans
   D. Servlets

Distinguish Between Stateful and Stateless Session Beans

5. Which of the following is not true about Enterprise JavaBeans (prior to EJB 3.0) session bean objects?
   A. A session bean can be defined without an `ejbCreate()` method.
   B. Stateful beans can contain multiple `ejbCreate()` methods as long as they match the home interface definition.
   C. The home interface of a stateless session bean must include a single `create()` method without any arguments.
   D. The stateless session bean class must contain exactly one `ejbCreate()` method without any arguments.

State the Benefits and Costs of Container-Managed Persistence

6. If you try to create an (prior to EJB 3.0) CMP-based entity bean for a table that does not have a primary key, which of the following statements is not true?
   A. You cannot create CMP entity beans without a database primary key.
   B. Duplicate records may be entered in the table.
   C. You can create CMP entity beans without primary keys, but the `findByPrimaryKey()` method will be unreliable.
7. Can you update the primary key field in a CMP entity bean (prior to EJB 3.0)?
   A. No; you cannot update the primary key field in a CMP entity bean.
   B. Yes; you can update the primary key field in a CMP entity bean by using accessor methods for the primary key cmp-fields in the component interface of the entity bean.
   C. Yes; you can update the primary key field in a CMP entity bean by calling `ejbStore()`.

State the Transactional Behavior in a Given Scenario for an Enterprise Bean Method with a Specified Transactional Attribute as Defined in the Deployment Descriptor

8. In an application with several stateless session Enterprise JavaBeans (prior to EJB 3.0), in terms of performance ramifications of storing the remote reference to a stateless session bean, which of the following statements is *least* accurate?

A. You can cache the stateless session bean reference using the `EJBObject.getHandle()`.
B. You can use the handle (SSB reference) when attempting to access the bean from here on.
C. The cost for a remote lookup on a stateless session bean is insignificant and generally does not justify using a handle (SSB reference) to access the bean.
D. The stateless session bean has no concurrency problems—that is, there is no shared data to be corrupted.

Given a Requirement Specification Detailing Security and Flexibility Needs, Identify Architectures That Would Fulfill Those Requirements

9. Which distributed object technology is most appropriate for systems that consist entirely of Java objects?

A. RMI
B. CORBA
C. DCOM
D. DCE

10. Which distributed object technology is most appropriate for systems that consist of objects written in different languages and that execute on different operating system platforms?

A. RMI
B. CORBA
C. DCOM
D. DCE

11. Which of the following are used by Java RMI?

A. Stubs
B. Skeletons
C. ORBs
D. IIOP

12. Which of the following is not a tier of a three-tier architecture?
- A. Client interface
- B. Business logic
- C. Security
- D. Data storage

13. Which of the following Java technologies implements transaction management?
- A. RMI
- B. JTS
- C. JMAPI
- D. JTA

14. Which of the following is not true when discussing application servers and web servers?
- A. A web server understands and supports only the HTTP protocol.
- B. An application server supports HTTP, TCP/IP, and many more protocols.
- C. A web server does not support caching, clustering, and load balancing.
- D. We can configure application servers to work as web servers.

15. Which statement is not true when discussing serialization in EJB?
- A. Serialization means that a machine A's object passed as part of a method call is flattened into a byte stream that can be sent over a network connection.
- B. All EJB methods arguments and return values must be serializable.
- C. Developers should make sure all objects passed as arguments implement *java.io.Serializable*.
- D. Serialization is not possible in EJB.

Explain How the Enterprise JavaBeans Container Does Life Cycle Management and Has the Capability to Increase Scalability

16. Which of the following are true about EJB components, containers, and application servers?
- A. Components run in containers.
- B. Containers are hosted by application servers.
- C. Containers run in components.
- D. Application servers run in containers.

17. Which objects would you find in an Enterprise JavaBeans (prior to EJB 3.0) directory service?
- A. An EJB home interface
- B. An EJB component

C. The EJB API
D. An EJB object interface

18. Containers and servers have the same function. What is the difference between an Enterprise JavaBeans container and an Enterprise JavaBeans server?
A. Containers run within servers.
B. Servers run within containers.
C. Only one server can run in a container.
D. Only one container can run in a server.

19. Which of the following Enterprise JavaBeans (prior to EJB 3.0) CMP entity bean methods are used by the container to alert the bean when its state is synchronized with the database?
A. `ejbLoad()`
B. `ejbStore()`
C. `ejbCreate()`
D. `ejbActivate()`

20. What happens if `remove()` is not invoked on a stateful Enterprise JavaBeans (prior to EJB 3.0) session bean?
A. Nothing happens; the bean will last forever.
B. The container will not honor any more requests for the bean.
C. An exception occurs in the session bean.
D. The bean is removed after the session time-out has been reached.

21. With respect to stateful Enterprise JavaBeans (prior to EJB 3.0) session beans, which of the following statements is *not* true?
A. Stateful session beans support instance pooling.
B. The life cycle of a stateful session bean is strictly connected with its client.
C. When the client removes the bean, it cannot be used by another client without being reinitialized.
D. A stateful session bean has three states: Does not exist, Method Ready, and Passivated.

SELF TEST ANSWERS

List the Required Classes/Interfaces That Must Be Provided for an Enterprise JavaBeans Component

1. ☑ **C** is correct for Enterprise JavaBeans (prior to EJB 3.0). The bean or enterprise bean is not the EJB object. It extends either the *EntityBean* interface or the *SessionBean* interface. It *must* implement all the methods defined in the remote interface.
   ☒ **A, B,** and **D** are incorrect because the *EJBObject* or remote object is a wrapper residing inside the container, between the client and the code. It performs setup and shutdown tasks pre– and post–bean call. The *EJBObject* is generated by the EJB server tools. The developer must write another interface, called the remote interface or the *EJBObject* interface, that extends the interface *EJBObject* and provides method signatures for all the business methods. The server automatically generates a Java class that implements the remote interface. The home interface is a factory object responsible for locating and creating instances of the bean. The developer must code for the *EJBHome* interface (that is, extend the interface *EJBHome*), and provide method signatures for all the desired `create()` and `find()` methods.
2. ☑ **A** is correct for Enterprise JavaBeans (EJB 3.*x*). The home interface is no longer required.
   ☒ **B, C,** and **D** are incorrect because the remote interface is no longer required, the bean class does not have to implement the methods of any interface, especially a nonexistent one *(EJBInterface)*, and the XML deployment descriptor entries for an EJB are now optional.

Distinguish Between Session and Entity Beans

3. ☑ **B** is correct. Entity beans represent the data in a data store. Entity beans do not obviate the JDBC API; they merely provide an alternative.
   ☒ **A, C,** and **D** are incorrect because in container-managed entity beans, when the bean is created, the container automatically retrieves the data from the persistent storage (for example, database) and assigns it to the bean's object variables for the user to manipulate or use it. The bean-managed entity bean for the class specifically has to obtain a database connection, retrieve the row/column values, and assign them to the objects in the `ejbLoad()`, which will be called by the container when it instantiates a bean object.

Recognize Appropriate Uses for Entity, Stateful Session, and Stateless Session Beans

4. ☑ **C** and **D** are correct. Both Enterprise JavaBeans and servlets may be used to upgrade CGI programs to Java-based solutions.
   ☒ **A** and **B** are incorrect. JMAPI and screen scrapers are not Java technology.

Distinguish Between Stateful and Stateless Session Beans

5. ☑ **A** is correct for Enterprise JavaBeans (prior to EJB 3.0). The Java Platform EE specification requires that the home interface of a Stateless session bean must include a single `create()` method without any arguments.
   ☒ **B, C,** and **D** are all true statements.

State the Benefits and Costs of Container-Managed Persistence

6. ☑ **A** is correct for Enterprise JavaBeans (prior to EJB 3.0). Yes, you can create CMP entity beans without primary keys.
   ☒ **B, C,** and **D** are incorrect because duplicate records may be entered in the table and the `findByPrimaryKey()` method may return varying rows.

7. ☑ **A** is correct for Enterprise JavaBeans (prior to EJB 3.0). You cannot change the primary key field of an entity bean.
   ☒ **B** and **C** are incorrect because, according to the EJB specification (prior to EJB 3.0), "Once the primary key for an entity bean has been set, the Bean Provider must not attempt to change it by use of set accessor methods on the primary key cmp-fields. The Bean provider should therefore not expose the set accessor methods for the primary key cmp-fields in the component interface of the entity bean." You can affect an update of a primary key field by removing (deleting) and then recreating (inserting) the bean.

State the Transactional Behavior in a Given Scenario for an Enterprise Bean Method with a Specified Transactional Deployment Descriptor

8. ☑ **C** is correct for Enterprise JavaBeans (prior to EJB 3.0). The cost for a remote lookup on a stateless session bean can be significant and can justify using a handle (SSB reference) to access the bean.
   ☒ **A, B,** and **D** are all true statements.

Given a Requirement Specification Detailing Security and Flexibility Needs, Identify Architectures That Would Fulfill Those Requirements

9. ☑ **A** is correct. RMI is the most appropriate distributed object technology for pure Java applications.
   ☒ **B, C,** and **D** are incorrect because they can work with non-Java objects.

10. ☑ **B** is correct. CORBA is the most appropriate object technology for systems that use objects written in different languages, and it supports a variety of operating system platforms.
    ☒ **A, C,** and **D** Each Works with specific platforms.

11. ☑ **A** and **B** are correct. RMI uses stubs and skeletons.
☒ **C** and **D** are incorrect because ORBs and IIOP are used with CORBA.

12. ☑ **C** is correct. Security is not a tier of a three-tiered architecture.
☒ **A, B,** and **D** are tiers of a three-tiered architecture.

13. ☑ **B** is correct. JTS provides an implementation of transaction management.
☒ **A, C,** and **D**. These do not implement transaction management. JTA defines an API for transaction management; it does not implement it.

14. ☑ **C** is correct. A web server understands and supports only the HTTP protocol, whereas an application server supports HTTP,TCP/IP, and many more protocols.
☒ **A, B,** and **D** are incorrect because web servers and application servers both support features such as caching, clustering, and load balancing. We can also configure an application server to work as web server.

15. ☑ **D** is correct. Serialization is possible.
☒ **A, B,** and **C** are incorrect because a good portion of EJB is the framework for underlying remote method invocation. To allow one JVM space A, the ability to invoke methods remotely on objects that are in JVM space B (objects running on another machine on the network), all arguments of each method call and their results must be serializable (that is, classes must implement *java.io.Serializable*).

Explain How the Enterprise JavaBeans Container Does Life Cycle Management and Has the Capability to Increase Scalability

16. ☑ **A** and **B** are correct because components run in containers that are hosted by application servers.
☒ **C** and **D** are not true. Containers do not run in components. Application servers do not run in containers.

17. ☑ **A** is correct for Enterprise JavaBeans (prior to EJB 3.0). *EJBHome* interfaces are placed in a directory service to facilitate access to an EJB component. The EJB home interface is used to obtain access to an *EJBObject* interface.
☒ **B, C,** and **D** are incorrect because EJB components are never accessed directly, but only through their *EJBHome* and *EJBObject* interfaces.

18. ☑ **A** is correct. Enterprise JavaBeans containers run within the context of servers.
☒ **B, C,** and **D** are incorrect. Servers do not run within containers. A server does not run in a container. Many containers can run in a server.

19. ☑ **A** and **B** are correct for Enterprise JavaBeans (prior to EJB 3.0). The container notifies the bean using the `ejbLoad()` and `ejbStore()` methods. The `ejbLoad()` method alerts the bean that its container-managed fields have just been populated with data from the database. This gives the bean an opportunity to do any postprocessing before the data can be used by the business methods. The `ejbStore()` method alerts the bean that its data is about to be written to the database. This give the bean an opportunity to do any preprocessing to the fields before they are written to the database.
☒ **C** and **D** have nothing to do with data persistence.

20. ☑ **D** is correct for Enterprise JavaBeans (prior to EJB 3.0). A stateful session bean would be put in an EXIST state until any of the following occurs:
Call `remove()` on the *EJBObjects*' stub from the client
Call `remove(handleToEJBObject)` on *EJBHome's* stub from the client
System exception in bean
Session time-out
Container failure
☒ **A, B,** and **C** are not true.

21. ☑ **A** is correct for Enterprise JavaBeans (prior to EJB 3.0). There is no clear indication that a stateful session bean is or is not pooled, while for the stateless session bean there is a specific paragraph that discusses the sequence for adding or removing a pooled bean instance. Performance reasons may motivate some containers to pool stateful session beans and avoid the overhead of recreating the entire object. In the black box, it may be pooled, hence the methods `ejbActivate()` and `ejbPassivate()` are included.
☒ **B, C,** and **D** are all true statements.