3

Enterprise Beans

Dale Green

ENTERPRISE beans are the J2EE components that implement Enterprise JavaBeans (EJB) technology. Enterprise beans run in the EJB container, a runtime environment within the J2EE server (see Figure 1–5, page 10). Although transparent to the application developer, the EJB container provides system-level services such as transactions to its enterprise beans. These services enable you to quickly build and deploy enterprise beans, which form the core of transactional J2EE applications.

In This Chapter

What Is an Enterprise Bean?

Written in the Java programming language, an *enterprise bean* is a server-side component that encapsulates the business logic of an application. The business logic is the code that fulfills the purpose of the application. In an inventory control application, for example, the enterprise beans might implement the business logic in methods called `checkInventoryLevel` and `orderProduct`. By invoking these methods, remote clients can access the inventory services provided by the application.

Benefits of Enterprise Beans

For several reasons, enterprise beans simplify the development of large, distributed applications. First, because the EJB container provides system-level services to enterprise beans, the bean developer can concentrate on solving business problems. The EJB container—not the bean developer—is responsible for system-level services such as transaction management and security authorization.

Second, because the beans—and not the clients—contain the application's business logic, the client developer can focus on the presentation of the client. The client developer does not have to code the routines that implement business rules or access databases. As a result, the clients are thinner, a benefit that is particularly important for clients that run on small devices.

Third, because enterprise beans are portable components, the application assembler can build new applications from existing beans. These applications can run on any compliant J2EE server.

When to Use Enterprise Beans

You should consider using enterprise beans if your application has any of the following requirements:

- The application must be scalable. To accommodate a growing number of users, you may need to distribute an application's components across multiple machines. Not only can the enterprise beans of an application run on different machines, but their location will remain transparent to the clients.
- Transactions are required to ensure data integrity. Enterprise beans support transactions, the mechanisms that manage the concurrent access of shared objects.
- The application will have a variety of clients. With just a few lines of code, remote clients can easily locate enterprise beans. These clients can be thin, various, and numerous.

Types of Enterprise Beans

Table 3–1 summarizes the three different types of enterprise beans. The following sections discuss each type in more detail.

Table 3–1 Summary of Enterprise Bean Types

Enterprise Bean Type	Purpose
Session	Performs a task for a client
Entity	Represents a business entity object that exists in persistent storage
Message-Driven	Acts as a listener for the Java Message Service API, processing messages asynchronously

What Is a Session Bean?

A *session bean* represents a single client inside the J2EE server. To access an application that is deployed on the server, the client invokes the session bean's methods. The session bean performs work for its client, shielding the client from complexity by executing business tasks inside the server.

As its name suggests, a session bean is similar to an interactive session. A session bean is not shared—it may have just one client, in the same way that an

interactive session may have just one user. Like an interactive session, a session bean is not persistent. (That is, its data is not saved to a database.) When the client terminates, its session bean appears to terminate and is no longer associated with the client.

For code samples, see Chapter 4.

State Management Modes

There are two types of session beans: stateful and stateless.

Stateful Session Beans

The state of an object consists of the values of its instance variables. In a stateful session bean, the instance variables represent the state of a unique client-bean session. Because the client interacts ("talks") with its bean, this state is often called the *conversational state*.

The state is retained for the duration of the client-bean session. If the client removes the bean or terminates, the session ends and the state disappears. This transient nature of the state is not a problem, however, because when the conversation between the client and the bean ends there is no need to retain the state.

Stateless Session Beans

A stateless session bean does not maintain a conversational state for a particular client. When a client invokes the method of a stateless bean, the bean's instance variables may contain a state, but only for the duration of the invocation. When the method is finished, the state is no longer retained. Except during method invocation, all instances of a stateless bean are equivalent, allowing the EJB container to assign an instance to any client.

Because stateless session beans can support multiple clients, they can offer better scalability for applications that require large numbers of clients. Typically, an application requires fewer stateless session beans than stateful session beans to support the same number of clients.

At times, the EJB container may write a stateful session bean to secondary storage. However, stateless session beans are never written to secondary storage. Therefore, stateless beans may offer better performance than stateful beans.

When to Use Session Beans

In general, you should use a session bean if the following circumstances hold:

- At any given time, only one client has access to the bean instance.
- The state of the bean is not persistent, existing only for a short period of time (perhaps a few hours).

Stateful session beans are appropriate if any of the following conditions are true:

- The bean's state represents the interaction between the bean and a specific client.
- The bean needs to hold information about the client across method invocations.
- The bean mediates between the client and the other components of the application, presenting a simplified view to the client.
- Behind the scenes, the bean manages the work flow of several enterprise beans. For an example, see the `AccountControllerEJB` session bean in Chapter 18.

To improve performance, you might choose a stateless session bean if it has any of these traits:

- The bean's state has no data for a specific client.
- In a single method invocation, the bean performs a generic task for all clients. For example, you might use a stateless session bean to send an e-mail that confirms an online order.
- The bean fetches from a database a set of read-only data that is often used by clients. Such a bean, for example, could retrieve the table rows that represent the products that are on sale this month.

What Is an Entity Bean?

An *entity bean* represents a business object in a persistent storage mechanism. Some examples of business objects are customers, orders, and products. In the J2EE SDK, the persistent storage mechanism is a relational database. Typically, each entity bean has an underlying table in a relational database, and each instance of the bean corresponds to a row in that table. For code examples of entity beans, please refer to chapters 5 and 6.

What Makes Entity Beans Different from Session Beans?

Entity beans differ from session beans in several ways. Entity beans are persistent, allow shared access, have primary keys, and may participate in relationships with other entity beans.

Persistence

Because the state of an entity bean is saved in a storage mechanism, it is persistent. *Persistence* means that the entity bean's state exists beyond the lifetime of the application or the J2EE server process. If you've worked with databases, you're familiar with persistent data. The data in a database is persistent because it still exists even after you shut down the database server or the applications it services.

There are two types of persistence for entity beans: bean-managed and container-managed. With *bean-managed persistence*, the entity bean code that you write contains the calls that access the database. If your bean has container-managed persistence, the EJB container automatically generates the necessary database access calls. The code that you write for the entity bean does not include these calls. For additional information, see the section Container-Managed Persistence (page 53).

Shared Access

Entity beans may be shared by multiple clients. Because the clients might want to change the same data, it's important that entity beans work within transactions. Typically, the EJB container provides transaction management. In this case, you specify the transaction attributes in the bean's deployment descriptor. You do not have to code the transaction boundaries in the bean—the container marks the boundaries for you. See Chapter 14 for more information.

Primary Key

Each entity bean has a unique object identifier. A customer entity bean, for example, might be identified by a customer number. The unique identifier, or *primary key*, enables the client to locate a particular entity bean. For more information see the section Primary Keys for Bean-Managed Persistence (page 113).

Relationships

Like a table in a relational database, an entity bean may be related to other entity beans. For example, in a college enrollment application, `StudentEJB` and `CourseEJB` would be related because students enroll in classes.

You implement relationships differently for entity beans with bean-managed persistence and those with container-managed persistence. With bean-managed persistence, the code that you write implements the relationships. But with container-managed persistence, the EJB container takes care of the relationships for you. For this reason, relationships in entity beans with container-managed persistence are often referred to as *container-managed relationships*.

Container-Managed Persistence

The term *container-managed persistence* means that the EJB container handles all database access required by the entity bean. The bean's code contains no database access (SQL) calls. As a result, the bean's code is not tied to a specific persistent storage mechanism (database). Because of this flexibility, even if you redeploy the same entity bean on different J2EE servers that use different databases, you won't need to modify or recompile the bean's code. In short, your entity beans are more portable.

In order to generate the data access calls, the container needs information that you provide in the entity bean's abstract schema.

Abstract Schema

Part of an entity bean's deployment descriptor, the *abstract schema* defines the bean's persistent fields and relationships. The term *abstract* distinguishes this schema from the physical schema of the underlying data store. In a relational database, for example, the physical schema is made up of structures such as tables and columns.

You specify the name of an abstract schema in the deployment descriptor. This name is referenced by queries written in the Enterprise JavaBeans Query Language ("EJB QL"). For an entity bean with container-managed persistence, you must define an EJB QL query for every finder method (except `findByPrimaryKey`). The EJB QL query determines the query that is executed by the EJB container when the finder method is invoked. To learn more about EJB QL, see Chapter 8.

You'll probably find it helpful to sketch the abstract schema before writing any code. Figure 3–1 represents a simple abstract schema that describes the relationships between three entity beans. These relationships are discussed further in the sections that follow.

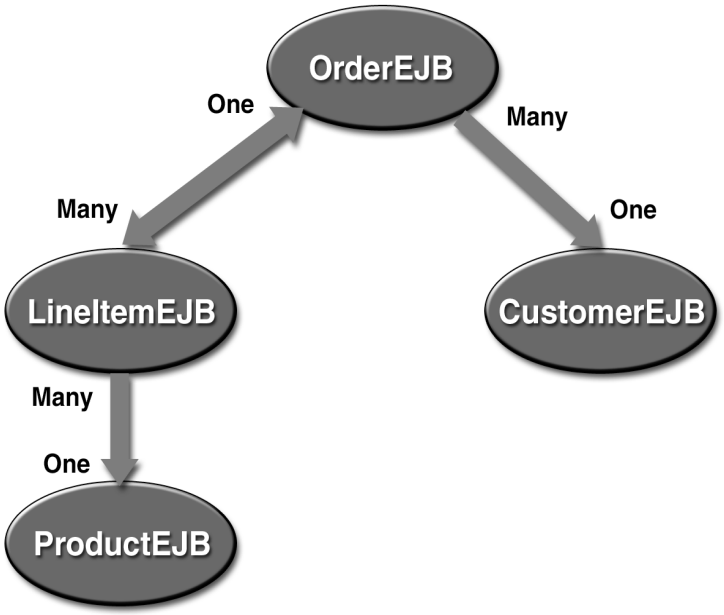

Figure 3–1 A High-Level View of an Abstract Schema

Persistent Fields

The persistent fields of an entity bean are stored in the underlying data store. Collectively, these fields constitute the state of the bean. At runtime, the EJB container automatically synchronizes this state with the database. During deployment, the container typically maps the entity bean to a database table and maps the persistent fields to the table's columns.

A `CustomerEJB` entity bean, for example, might have persistent fields such as `firstName`, `lastName`, `phone`, and `emailAddress`. In container-managed persistence, these fields are virtual. You declare them in the abstract schema, but you do not code them as instance variables in the entity bean class. Instead, the persistent fields are identified in the code by access methods (getters and setters).

Relationship Fields

A *relationship field* is like a foreign key in a database table—it identifies a related bean. Like a persistent field, a relationship field is virtual and is defined in the enterprise bean class with access methods. But unlike a persistent field, a relationship field does not represent the bean's state. Relationship fields are discussed further in Direction in Container-Managed Relationships (page 55).

Multiplicity in Container-Managed Relationships

There are four types of multiplicities:

One-to-one: Each entity bean instance is related to a single instance of another entity bean. For example, to model a physical warehouse in which each storage bin contains a single widget, `StorageBinEJB` and `WidgetEJB` would have a one-to-one relationship.

One-to-many: An entity bean instance may be related to multiple instances of the other entity bean. A sales order, for example, can have multiple line items. In the order application, `OrderEJB` would have a one-to-many relationship with `LineItemEJB`.

Many-to-one: Multiple instances of an entity bean may be related to a single instance of the other entity bean. This multiplicity is the opposite of a one-to-many relationship. In the example mentioned in the previous item, from the perspective of `LineItemEJB` the relationship to `OrderEJB` is many-to-one.

Many-to-many: The entity bean instances may be related to multiple instances of each other. For example, in college each course has many students, and every student may take several courses. Therefore, in an enrollment application, `CourseEJB` and `StudentEJB` would have a many-to-many relationship.

Direction in Container-Managed Relationships

The direction of a relationship may be either bidirectional or unidirectional. In a *bidirectional* relationship, each entity bean has a relationship field that refers to the other bean. Through the relationship field, an entity bean's code can access its related object. If an entity bean has a relative field, then we often say that it "knows" about its related object. For example, if `OrderEJB` knows what `LineItemEJB` instances it has and if `LineItemEJB` knows what `OrderEJB` it belongs to, then they have a bidirectional relationship.

In a *unidirectional* relationship, only one entity bean has a relationship field that refers to the other. For example, `LineItemEJB` would have a relationship field that identifies `ProductEJB`, but `ProductEJB` would not have a relationship field

for LineItemEJB. In other words, LineItemEJB knows about ProductEJB, but ProductEJB doesn't know which LineItemEJB instances refer to it.

EJB QL queries often navigate across relationships. The direction of a relationship determines whether a query can navigate from one bean to another. For example, a query can navigate from LineItemEJB to ProductEJB, but cannot navigate in the opposite direction. For OrderEJB and LineItemEJB, a query could navigate in both directions, since these two beans have a bidirectional relationship.

When to Use Entity Beans

You should probably use an entity bean under the following conditions:

- The bean represents a business entity, not a procedure. For example, CreditCardEJB would be an entity bean, but CreditCardVerifierEJB would be a session bean.
- The bean's state must be persistent. If the bean instance terminates or if the J2EE server is shut down, the bean's state still exists in persistent storage (a database).

What Is a Message-Driven Bean?

Note: This section contains text from *The Java Message Service Tutorial*. Because message-driven beans rely on Java Message Service (JMS) technology, to fully understand how these beans work you should consult the tutorial at this URL:

http://java.sun.com/products/jms/tutorial/index.html

A *message-driven bean* is an enterprise bean that allows J2EE applications to process messages asynchronously. It acts as a JMS message listener, which is similar to an event listener except that it receives messages instead of events. The messages may be sent by any J2EE component—an application client, another enterprise bean, or a Web component—or by a JMS application or system that does not use J2EE technology.

Message-driven beans currently process only JMS messages, but in the future they may be used to process other kinds of messages.

For a code sample, see Chapter 7.

What Makes Message-Driven Beans Different from Session and Entity Beans?

The most visible difference between message-driven beans and session and entity beans is that clients do not access message-driven beans through interfaces. Interfaces are described in the section Defining Client Access with Interfaces (page 58). Unlike a session or entity bean, a message-driven bean has only a bean class.

In several respects, a message-driven bean resembles a stateless session bean.

- A message-driven bean's instances retain no data or conversational state for a specific client.
- All instances of a message-driven bean are equivalent, allowing the EJB container to assign a message to any message-driven bean instance. The container can pool these instances to allow streams of messages to be processed concurrently.
- A single message-driven bean can process messages from multiple clients.

The instance variables of the message-driven bean instance can contain some state across the handling of client messages—for example, a JMS API connection, an open database connection, or an object reference to an enterprise bean object.

When a message arrives, the container calls the message-driven bean's `onMessage` method to process the message. The `onMessage` method normally casts the message to one of the five JMS message types and handles it in accordance with the application's business logic. The `onMessage` method may call helper methods, or it may invoke a session or entity bean to process the information in the message or to store it in a database.

A message may be delivered to a message-driven bean within a transaction context, so that all operations within the `onMessage` method are part of a single transaction. If message processing is rolled back, the message will be redelivered. For more information, see Chapter 7.

When to Use Message-Driven Beans

Session beans and entity beans allow you to send JMS messages and to receive them synchronously, but not asynchronously. To avoid tying up server resources, you may prefer not to use blocking synchronous receives in a server-side component. To receive messages asynchronously, use a message-driven bean.

Defining Client Access with Interfaces

Note: The material in this section applies only to session and entity beans, not to message-driven beans. Because they have a different programming model, message-driven beans do not have interfaces that define client access.

A client may access a session or an entity bean only through the methods defined in the bean's interfaces. These interfaces define the client's view of a bean. All other aspects of the bean—method implementations, deployment descriptor settings, abstract schemas, and database access calls—are hidden from the client.

Well-designed interfaces simplify the development and maintenance of J2EE applications. Not only do clean interfaces shield the clients from any complexities in the EJB tier, but they also allow the beans to change internally without affecting the clients. For example, even if you change your entity beans from bean-managed to container-managed persistence, you won't have to alter the client code. But if you were to change the method definitions in the interfaces, then you might have to modify the client code as well. Therefore, to isolate your clients from possible changes in the beans, it is important that you design the interfaces carefully.

When you design a J2EE application, one of the first decisions you make is the type of client access allowed by the enterprise beans: remote or local.

Remote Access

A remote client of an enterprise bean has the following traits:

- It may run on a different machine and a different Java virtual machine (JVM) than the enterprise bean it accesses. (It is not required to run on a different JVM.)
- It can be a Web component, a J2EE application client, or another enterprise bean.
- To a remote client, the location of the enterprise bean is transparent.

To create an enterprise bean with remote access, you must code a remote interface and a home interface. The *remote interface* defines the business methods that are specific to the bean. For example, the remote interface of a bean named `BankAccountEJB` might have business methods named `debit` and `credit`. The *home interface* defines the bean's life cycle methods—`create` and `remove`. For entity beans, the home interface also defines finder methods and home methods.

Finder methods are used to locate entity beans. Home methods are business methods that are invoked on all instances of an entity bean class. Figure 3–2 shows how the interfaces control the client's view of an enterprise bean.

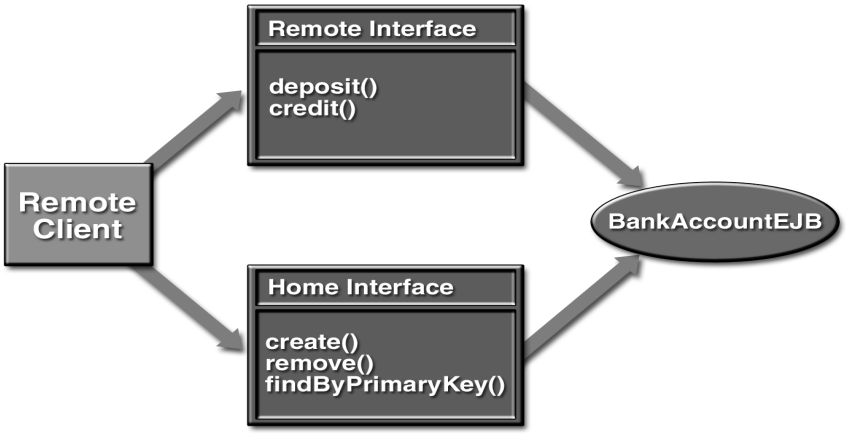

Figure 3–2 Interfaces for an Enterprise Bean With Remote Access

Local Access

A local client has these characteristics:

- It must run in the same JVM as the enterprise bean it accesses.
- It may be a Web component or another enterprise bean.
- To the local client, the location of the enterprise bean it accesses is not transparent.
- It is often an entity bean that has a container-managed relationship with another entity bean.

To build an enterprise bean that allows local access, you must code the local interface and the local home interface. The *local interface* defines the bean's business methods, and the *local home* interface defines its life cycle and finder methods.

Local Interfaces and Container-Managed Relationships

If an entity bean is the target of a container-managed relationship, then it must have local interfaces. The direction of the relationship determines whether or not

a bean is the target. In Figure 3–1, for example, `ProductEJB` is the target of a unidirectional relationship with `LineItemEJB`. Because `LineItemEJB` accesses `ProductEJB` locally, `ProductEJB` must have the local interfaces. `LineItemEJB` also needs local interfaces—not because of its relationship with `ProductEJB`—but because it is the target of a relationship with `OrderEJB`. And because the relationship between `LineItemEJB` and `OrderEJB` is bidirectional, both beans must have local interfaces.

Because they require local access, entity beans that participate in a container-managed relationship must reside in the same EJB JAR file. The primary benefit of this locality is increased performance—local calls are usually faster than remote calls.

Deciding on Remote or Local Access

The decision regarding whether to allow local or remote access depends on the following factors.

Container-managed relationships: If an entity bean is the target of a container-managed relationship, it must use local access.

Tight or loose coupling of related beans: Tightly coupled beans depend on one another. For example, a completed sales order must have one or more line items, which cannot exist without the order to which they belong. The `OrderEJB` and `LineItemEJB` entity beans that model this relationship are tightly coupled. Tightly coupled beans are good candidates for local access. Since they fit together as a logical unit, they probably call each other often and would benefit from the increased performance that is possible with local access.

Type of client: If an enterprise bean is accessed by J2EE application clients, then it should allow remote access. In a production environment, these clients almost always run on different machines than the J2EE server. If an enterprise bean's clients are Web components or other enterprise beans, then the type of access depends on how you want to distribute your components.

Component distribution: J2EE applications are scalable because their server-side components can be distributed across multiple machines. In a distributed application, for example, the Web components may run on a different server than the enterprise beans they access. In this distributed scenario, the enterprise beans should allow remote access.

If you aren't sure which type of access an enterprise bean should have, then choose remote access. This decision gives you more flexibility—in the future

you can distribute your components to accommodate growing demands on your application.

Although uncommon, it is possible for an enterprise bean to allow both remote and local access. Such a bean would require both remote and local interfaces.

Performance and Access

Because of factors such as network latency, remote calls may be slower than local calls. On the other hand, if you distribute components among different servers, you might improve the application's overall performance. Both of these statements are generalizations; actual performance can vary in different operational environments. Nevertheless, you should keep in mind how your application design might affect performance.

Method Parameters and Access

The type of access affects the parameters of the bean methods that are called by clients. The following topics apply not only to method parameters, but also to method return values.

Isolation

An argument in a remote call is passed by value; it is a copy of an object. But an argument in a local call is passed by reference, just like a normal method call in the Java programming language.

The parameters of remote calls are more isolated than those of local calls. With remote calls, the client and bean operate on different copies of a parameter object. If the client changes the value of the object, the value of the copy in the bean does not change. This layer of isolation can help protect the bean if the client accidentally modifies the data.

In a local call, both the client and the bean may modify the same object. In general, you should not rely on this side effect of local calls. Perhaps someday you will want to distribute your components, replacing the local calls with remote ones.

Granularity of Accessed Data

Because remote calls are likely to be slower than local calls, the parameters in remote methods should be relatively coarse-grained. Since a coarse-grained object contains more data than a fine-grained one, fewer access calls are required.

For example, suppose that a `CustomerEJB` entity bean is accessed remotely. This bean would have a single getter method that returns a `CustomerDetails` object, which encapsulates all of the customer's information. But if `CustomerEJB` is to be accessed locally, it could have a getter method for each instance variable: `getFirstName`, `getLastName`, `getPhoneNumber`, and so forth. Because local calls are fast, the multiple calls to these finer-grained getter methods would not significantly degrade performance.

The Contents of an Enterprise Bean

To develop an enterprise bean, you must provide the following files:

- **Deployment descriptor**: An XML file that specifies information about the bean such as its persistence type and transaction attributes. The `deploytool` utility creates the deployment descriptor when you step through the New Enterprise Bean wizard.
- **Enterprise bean class**: Implements the methods defined in the following interfaces.
- **Interfaces**: The remote and home interfaces are required for remote access. For local access, the local and local home interfaces are required. See the section Defining Client Access with Interfaces (page 58). (Please note that these interfaces are not used by message-driven beans.)
- **Helper classes**: Other classes needed by the enterprise bean class, such as exception and utility classes.

You package the files in the preceding list into an EJB JAR file, the module that stores the enterprise bean. An EJB JAR file is portable and may be used for different applications. To assemble a J2EE application, you package one or more modules—such as EJB JAR files—into an EAR file, the archive file that holds the application. When you deploy the EAR file that contains the bean's EJB JAR file, you also deploy the enterprise bean onto the J2EE server.

Naming Conventions for Enterprise Beans

Because enterprise beans are composed of multiple parts, it's useful to follow a naming convention for your applications. Table 3–2 summarizes the conventions for the example beans of this tutorial.

Table 3–2 Naming Conventions for Enterprise Beans

Item	Syntax	Example
Enterprise bean name (DD)	<*name*>EJB	AccountEJB
EJB JAR display name (DD)	<*name*>JAR	AccountJAR
Enterprise bean class	<*name*>Bean	AccountBean
Home interface	<*name*>Home	AccountHome
Remote interface	<*name*>	Account
Local home interface	Local<*name*>Home	LocalAccountHome
Local interface	Local<*name*>	LocalAccount
Abstract schema (DD)	<*name*>	Account

DD means that the item is an element in the bean's deployment descriptor.

The Life Cycles of Enterprise Beans

An enterprise bean goes through various stages during its lifetime, or life cycle. Each type of enterprise bean—session, entity, or message-driven—has a different life cycle.

The descriptions that follow refer to methods that are explained along with the code examples in the next two chapters. If you are new to enterprise beans, you should skip this section and try out the code examples first.

The Life Cycle of a Stateful Session Bean

Figure 3–3 illustrates the stages that a session bean passes through during its lifetime. The client initiates the life cycle by invoking the `create` method. The EJB container instantiates the bean and then invokes the `setSessionContext` and `ejbCreate` methods in the session bean. The bean is now ready to have its business methods invoked.

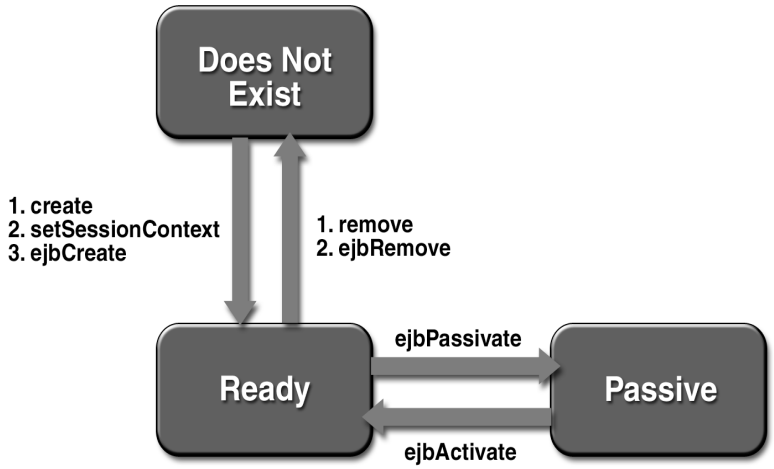

Figure 3–3 Life Cycle of a Stateful Session Bean

While in the ready stage, the EJB container may decide to deactivate, or *passivate*, the bean by moving it from memory to secondary storage. (Typically, the EJB container uses a least-recently-used algorithm to select a bean for passivation.) The EJB container invokes the bean's `ejbPassivate` method immediately before passivating it. If a client invokes a business method on the bean while it is in the passive stage, the EJB container activates the bean, moving it back to the ready stage, and then calls the bean's `ejbActivate` method.

At the end of the life cycle, the client invokes the `remove` method and the EJB container calls the bean's `ejbRemove` method. The bean's instance is ready for garbage collection.

Your code controls the invocation of only two life-cycle methods—the `create` and `remove` methods in the client. All other methods in Figure 3–3 are invoked by the EJB container. The `ejbCreate` method, for example, is inside the bean class, allowing you to perform certain operations right after the bean is instantiated. For instance, you may wish to connect to a database in the `ejbCreate` method. See Chapter 16 for more information.

The Life Cycle of a Stateless Session Bean

Because a stateless session bean is never passivated, its life cycle has just two stages: nonexistent and ready for the invocation of business methods. Figure 3–4 illustrates the stages of a stateless session bean.

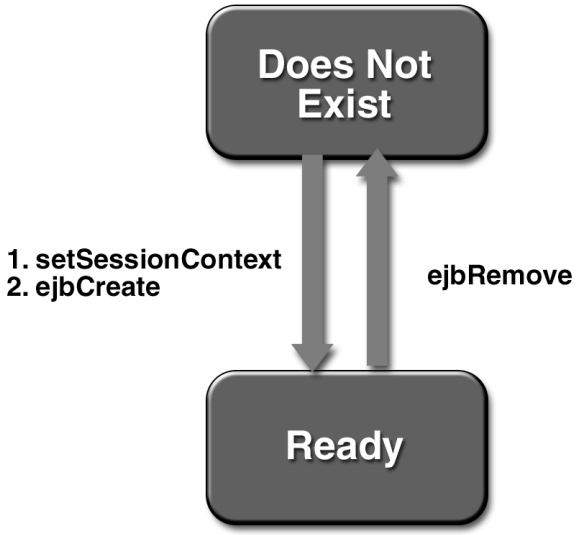

Figure 3–4 Life Cycle of a Stateless Session Bean

The Life Cycle of an Entity Bean

Figure 3–5 shows the stages that an entity bean passes through during its lifetime. After the EJB container creates the instance, it calls the `setEntityContext` method of the entity bean class. The `setEntityContext` method passes the entity context to the bean.

After instantiation, the entity bean moves to a pool of available instances. While in the pooled stage, the instance is not associated with any particular EJB object identity. All instances in the pool are identical. The EJB container assigns an identity to an instance when moving it to the ready stage.

There are two paths from the pooled stage to the ready stage. On the first path, the client invokes the `create` method, causing the EJB container to call the `ejbCreate` and `ejbPostCreate` methods. On the second path, the EJB container invokes the `ejbActivate` method. While in the ready stage, an entity bean's business methods may be invoked.

There are also two paths from the ready stage to the pooled stage. First, a client may invoke the `remove` method, which causes the EJB container to call the `ejbRemove` method. Second, the EJB container may invoke the `ejbPassivate` method.

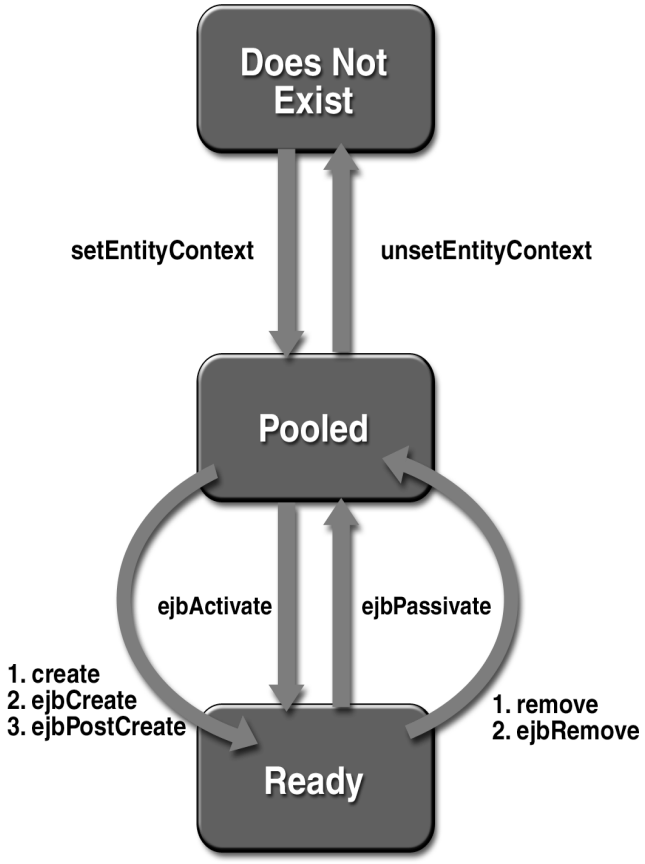

Figure 3–5 Life Cycle of an Entity Bean

At the end of the life cycle, the EJB container removes the instance from the pool and invokes the `unsetEntityContext` method.

In the pooled state, an instance is not associated with any particular EJB object identity. With bean-managed persistence, when the EJB container moves an instance from the pooled state to the ready state, it does not automatically set the primary key. Therefore, the `ejbCreate` and `ejbActivate` methods must assign a value to the primary key. If the primary key is incorrect, the `ejbLoad` and `ejbStore` methods cannot synchronize the instance variables with the database. In the section The SavingsAccountEJB Example (page 84), the `ejbCreate` method

assigns the primary key from one of the input parameters. The `ejbActivate` method sets the primary key (`id`) as follows:

```
id = (String)context.getPrimaryKey();
```

In the pooled state, the values of the instance variables are not needed. You can make these instance variables eligible for garbage collection by setting them to `null` in the `ejbPasssivate` method.

The Life Cycle of a Message-Driven Bean

Figure 3–6 illustrates the stages in the life cycle of a message-driven bean.

The EJB container usually creates a pool of message-driven bean instances. For each instance, the EJB container instantiates the bean and performs these tasks:

1. It calls the `setMessageDrivenContext` method to pass the context object to the instance.
2. It calls the instance's `ejbCreate` method.

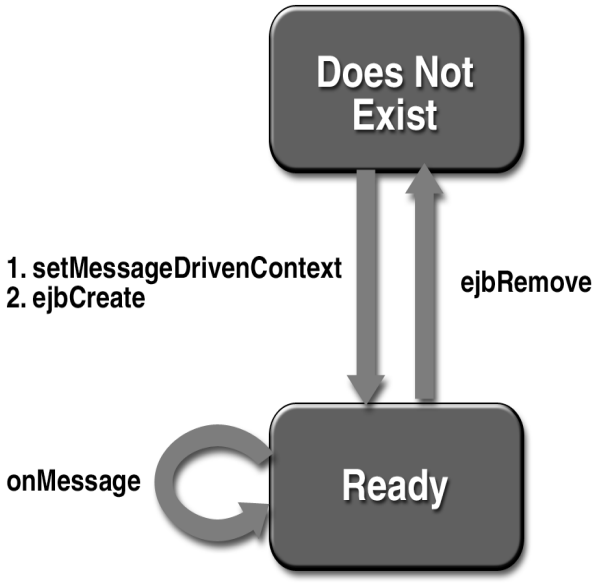

Figure 3–6 Life Cycle of a Message-Driven Bean

Like a stateless session bean, a message-driven bean is never passivated, and it has only two states: nonexistent and ready to receive messages.

At the end of the life cycle, the container calls the `ejbRemove` method. The bean's instance is then ready for garbage collection.