23

Enterprise Beans

ENTERPRISE beans are the J2EE components that implement Enterprise JavaBeans (EJB) technology. Enterprise beans run in the EJB container, a runtime environment within the Sun Java System Application Server Platform Edition 8 (see Figure 1–5, page 10). Although transparent to the application developer, the EJB container provides system-level services such as transactions and security to its enterprise beans. These services enable you to quickly build and deploy enterprise beans, which form the core of transactional J2EE applications.

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—and 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 provided that they use the standard APIs.

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 also their location will remain transparent to the clients.
- Transactions must 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 only 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 23–1 summarizes the three types of enterprise beans. The following sections discuss each type in more detail.

Table 23–1 Enterprise Bean Types

Enterprise Bean Type	Purpose
Session	Performs a task for a client; implements a web service
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 Application 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 can have only one client, in the same way that an interactive session can have only 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 25.

State Management Modes

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

Stateless Session Beans

A *stateless* session bean does not maintain a conversational state for the 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.

A stateless session bean can implement a web service, but other types of enterprise beans cannot.

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.

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 (perhaps a few hours).
- The bean implements a web service.

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 `AccountControllerBean` session bean in Chapter 36.

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 email 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 Application Server, 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 26 and 27.

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 can 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 Application 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 861).

Shared Access

Entity beans can 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 30 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 962).

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, `StudentBean` and `CourseBean` would be related because students enroll in classes.

You implement relationships differently for entity beans with bean-managed persistence than 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 if you use container-managed persistence than if they use bean-managed persistence.

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 29.

You'll probably find it helpful to sketch the abstract schema before writing any code. Figure 23–1 represents a simple abstract schema that describes the relationships between three entity beans. These relationships are discussed further in the sections that follow.

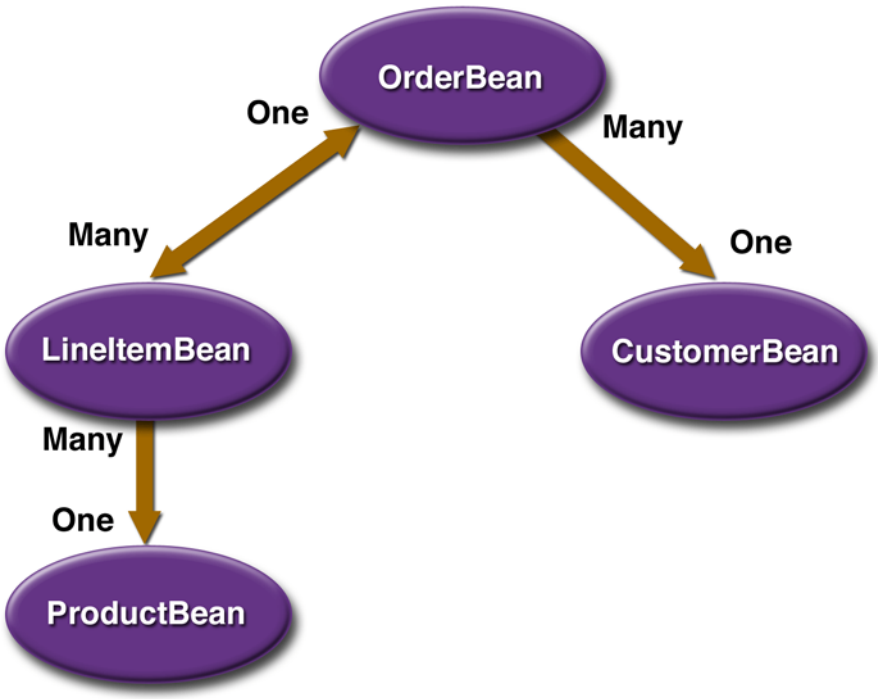

Figure 23–1 A High-Level View of an Abstract Schema

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 `CustomerBean` 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 via 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 863).

Multiplicity in Container-Managed Relationships

There are four types of multiplicities: one-to-one, one-to-many, many-to-one, and many-to-many.

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, `StorageBinBean` and `WidgetBean` would have a one-to-one relationship.

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

Many-to-one: Multiple instances of an entity bean can 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 just mentioned, from the perspective of `LineItemBean` the relationship to `OrderBean` is many-to-one.

Many-to-many: The entity bean instances can 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, `CourseBean` and `StudentBean` would have a many-to-many relationship.

Direction in Container-Managed Relationships

The direction of a relationship can 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 `OrderBean` knows what

LineItemBean instances it has and if LineItemBean knows what OrderBean 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, LineItemBean would have a relationship field that identifies ProductBean, but ProductBean would not have a relationship field for LineItemBean. In other words, LineItemBean knows about ProductBean, but ProductBean doesn't know which LineItemBean 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 LineItemBean to ProductBean but cannot navigate in the opposite direction. For OrderBean and LineItemBean, a query could navigate in both directions, because 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 and not a procedure. For example, CreditCardBean would be an entity bean, but CreditCardVerifierBean would be a session bean.
- The bean's state must be persistent. If the bean instance terminates or if the Application Server is shut down, the bean's state still exists in persistent storage (a database).

What Is a Message-Driven Bean?

A *message-driven bean* is an enterprise bean that allows J2EE applications to process messages asynchronously. It normally acts as a JMS message listener, which is similar to an event listener except that it receives JMS messages instead of events. The messages can 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 can process either JMS messages or other kinds of messages.

For a simple code sample, see Chapter 28. For more information about using message-driven beans, see Using the JMS API in a J2EE Application (page 1250) and Chapter 34.

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 866). 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.

Client components do not locate message-driven beans and invoke methods directly on them. Instead, a client accesses a message-driven bean through JMS by sending messages to the message destination for which the message-driven bean class is the `MessageListener`. You assign a message-driven bean's destination during deployment by using Application Server resources.

Message-driven beans have the following characteristics:

- They execute upon receipt of a single client message.
- They are invoked asynchronously.
- They are relatively short-lived.
- They do not represent directly shared data in the database, but they can access and update this data.
- They can be transaction-aware.
- They are stateless.

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 can call helper methods, or it can invoke a session or entity bean to process the information in the message or to store it in a database.

A message can be delivered to a message-driven bean within a transaction context, so 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 28.

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

The material in this section applies only to session and entity beans and 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 can 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, local, or web service.

Remote Clients

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

- It can 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, an application client, or another enterprise bean.
- To a remote client, the location of the enterprise bean is transparent.

To create an enterprise bean that has 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 `BankAccountBean` might have business methods named `deposit` 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 23–2 shows how the interfaces control the client's view of an enterprise bean.

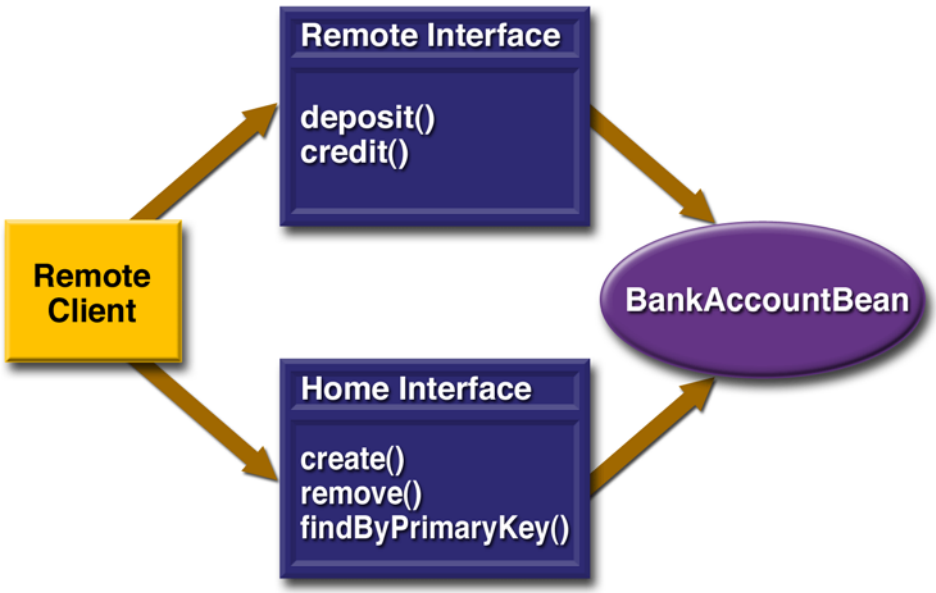

Figure 23–2 Interfaces for an Enterprise Bean with Remote Access

Local Clients

A local client has these characteristics:

- It must run in the same JVM as the enterprise bean it accesses.
- It can 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 23–1, for example, `ProductBean` is the target of a unidirectional relationship with `LineItemBean`. Because `LineItemBean` accesses `ProductBean` locally, `ProductBean` must have the local interfaces. `LineItemBean` also needs local interfaces, not because of its relationship with `ProductBean`, but because it is the target of a relationship with `OrderBean`. And because the relationship between `LineItemBean` and `OrderBean` 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

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 `OrderBean` and `LineItemBean` entity beans that model this relationship are tightly coupled. Tightly coupled beans are good candidates for local access. Because 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 application clients, then it should allow remote access. In a production environment, these clients almost always run on different machines than the Application Server does. 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 do the enterprise beans they access. In this distributed scenario, the enterprise beans should allow remote access.
- *Performance*: 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.

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 it is 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.

Web Service Clients

A web service client can access a J2EE application in two ways. First, the client can access a web service created with JAX-RPC. (For more information on JAX-RPC, see Chapter 8, Building Web Services with JAX-RPC, page 319.) Second, a web service client can invoke the business methods of a stateless session bean. Other types of enterprise beans cannot be accessed by web service clients.

Provided that it uses the correct protocols (SOAP, HTTP, WSDL), any web service client can access a stateless session bean, whether or not the client is written in the Java programming language. The client doesn't even "know" what technology implements the service—stateless session bean, JAX-RPC, or some other technology. In addition, enterprise beans and web components can be clients of web services. This flexibility enables you to integrate J2EE applications with web services.

A web service client accesses a stateless session bean through the bean's web service endpoint interface. Like a remote interface, a *web service endpoint interface* defines the business methods of the bean. In contrast to a remote interface, a web service endpoint interface is not accompanied by a home interface, which defines the bean's life-cycle methods. The only methods of the bean that may be invoked by a web service client are the business methods that are defined in the web service endpoint interface.

For a code sample, see A Web Service Example: HelloServiceBean (page 911).

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

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 can modify the same parameter 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.

As with remote clients, web service clients operate on different copies of parameters than does the bean that implements the web service.

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. A coarse-grained object contains more data than a fine-grained one, so fewer access calls are required. For the same reason, the parameters of the methods called by web service clients should also be coarse-grained.

For example, suppose that a `CustomerBean` 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 `CustomerBean` 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. For access by web service clients, the web service endpoint interface is required. See the section Defining Client Access with Interfaces (page 866). (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 can 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 Application Server. You can also deploy an EJB JAR that is not contained in an EAR file.

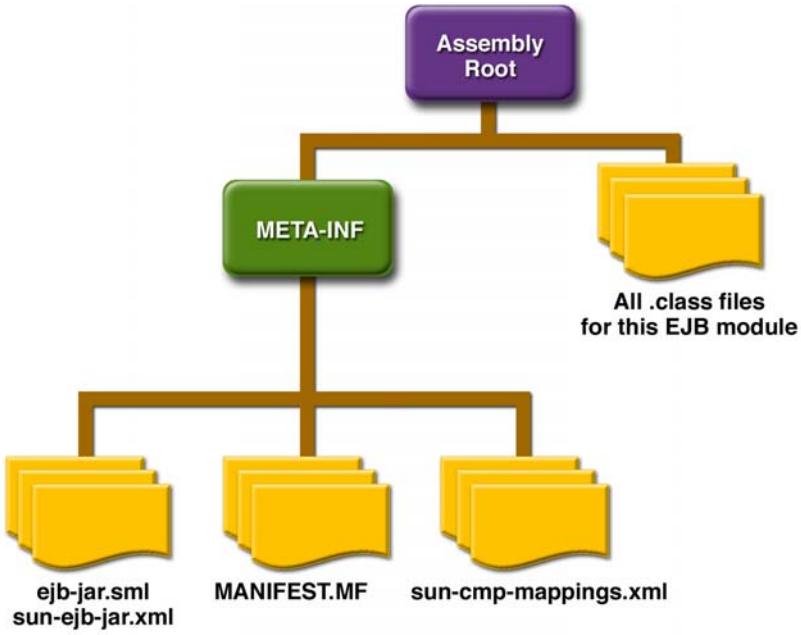

Figure 23–3 Structure of an Enterprise Bean JAR

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 23–2 summarizes the conventions for the example beans in this tutorial.

Table 23–2 Naming Conventions for Enterprise Beans

Item	Syntax	Example
Enterprise bean name (DD[a])	<*name*>Bean	AccountBean
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	<*name*>LocalHome	AccountLocalHome
Local interface	<*name*>Local	AccountLocal
Abstract schema (DD)	<*name*>	Account

a.*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 23–4 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.

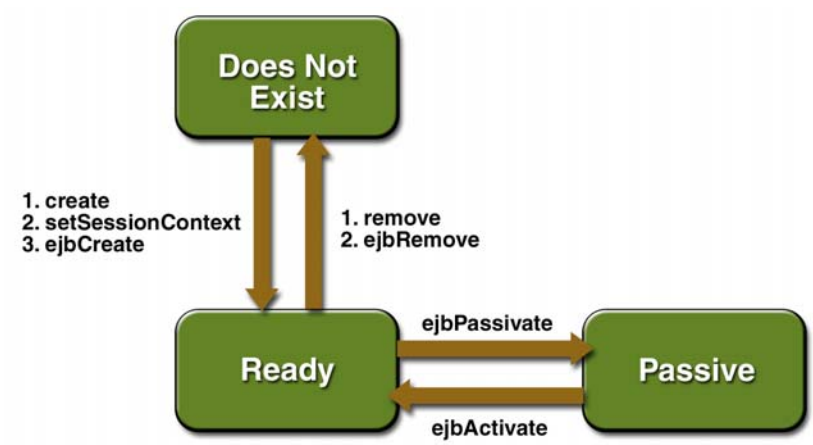

Figure 23–4 Life Cycle of a Stateful Session Bean

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, calls the bean's `ejbActivate` method, and then moves it to the ready stage.

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 23–4 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 example, you might wish to connect to a database in the `ejbCreate` method. See Chapter 31 for more information.

The Life Cycle of a Stateless Session Bean

Because a stateless session bean is never passivated, its life cycle has only two stages: nonexistent and ready for the invocation of business methods. Figure 23–5 illustrates the stages of a stateless session bean.

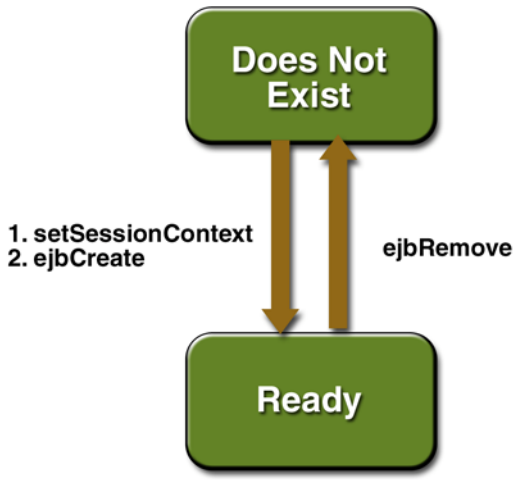

Figure 23–5 Life Cycle of a Stateless Session Bean

The Life Cycle of an Entity Bean

Figure 23–6 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 an entity bean is in the ready stage, an it's business methods can be invoked.

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

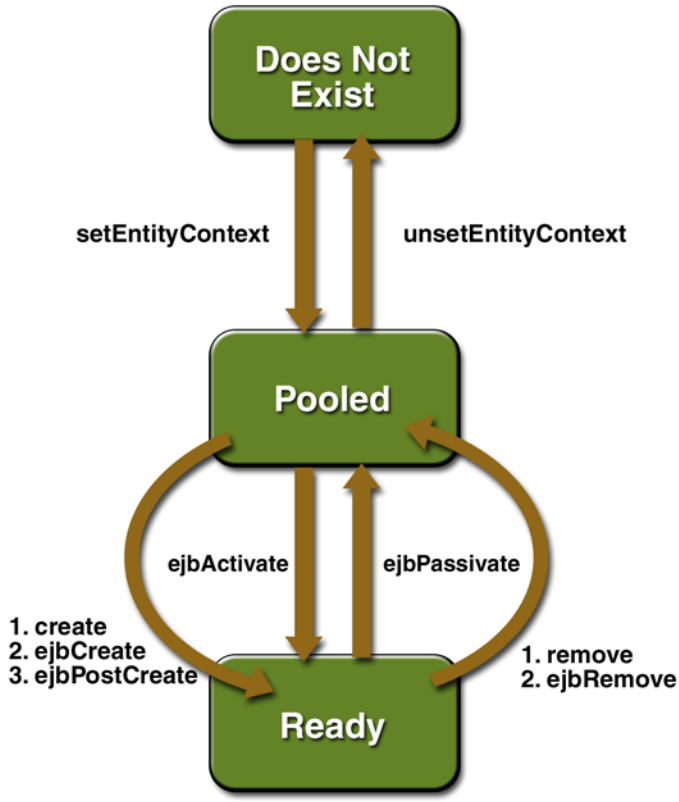

Figure 23–6 Life Cycle of an Entity Bean

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 SavingsAccountBean Example (page 931), 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 `ejbPassivate` method.

The Life Cycle of a Message-Driven Bean

Figure 23–7 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.

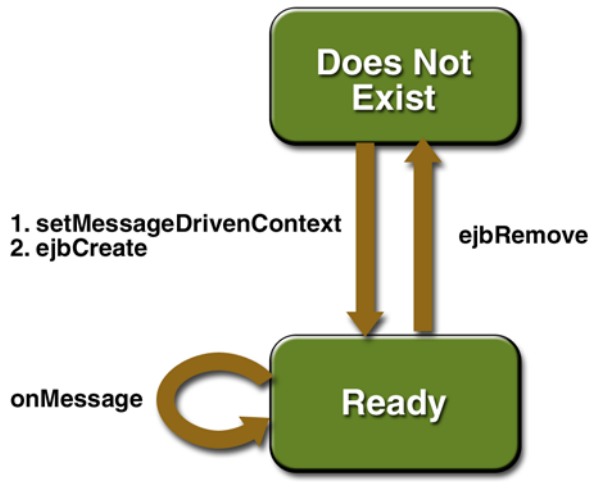

Figure 23–7 Life Cycle of a Message-Driven Bean

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.

Further Information

For further information on Enterprise JavaBeans technology, see the following:

- Enterprise JavaBeans 2.1 specification:
  `http://java.sun.com/products/ejb/docs.html`
- The Enterprise JavaBeans web site:
  `http://java.sun.com/products/ejb`

24

Getting Started with Enterprise Beans

THIS chapter shows how to develop, deploy, and run a simple J2EE application named `ConverterApp`. The purpose of `ConverterApp` is to calculate currency conversions between yen and eurodollars. `ConverterApp` consists of an enterprise bean, which performs the calculations, and two types of clients: an application client and a web client.

Here's an overview of the steps you'll follow in this chapter:

1. Create the J2EE application: `ConverterApp`.
2. Create the enterprise bean: `ConverterBean`.
3. Create the application client: `ConverterClient`.
4. Create the web client in `ConverterWAR`.
5. Deploy `ConverterApp` onto the server.
6. From a terminal window, run `ConverterClient`.
7. Using a browser, run the web client.

Before proceeding, make sure that you've done the following:

- Read Chapter 1.
- Become familiar with enterprise beans (see Chapter 23).
- Started the server (see Starting and Stopping the Application Server, page 27).
- Launched `deploytool` (see Starting the deploytool Utility, page 29)

Creating the J2EE Application

In this section, you'll create a J2EE application named `ConverterApp`, storing it in the file `ConverterApp.ear`.

1. In `deploytool`, select File→New→Application.
2. Click Browse.
3. In the file chooser, navigate to this directory:

   `<INSTALL>/j2eetutorial14/examples/ejb/converter/`
4. In the File Name field, enter `ConverterApp.ear`.
5. Click New Application.
6. Click OK.
7. Verify that the `ConverterApp.ear` file resides in the directory specified in step 3.

At this point, the application contains no J2EE components and cannot be deployed. In the sections that follow, when you run the `deploytool` wizards to create the components, `deploytool` will add the components to the `ConverterApp.ear` file.

Creating the Enterprise Bean

The enterprise bean in our example is a stateless session bean called `ConverterBean`. The source code for `ConverterBean` is in the `<INSTALL>/j2eetutorial14/examples/ejb/converter/src/` directory.

Creating `ConverterBean` requires these steps:

1. Coding the bean's interfaces and class (the source code is provided)
2. Compiling the source code with `asant`

3. With `deploytool`, packaging the bean into an EJB JAR file and inserting the EJB JAR file into the application's `ConverterApp.ear` file

Coding the Enterprise Bean

The enterprise bean in this example needs the following code:

- Remote interface
- Home interface
- Enterprise bean class

Coding the Remote Interface

A *remote interface* defines the business methods that a client can call. The business methods are implemented in the enterprise bean code. The source code for the `Converter` remote interface follows.

```
import javax.ejb.EJBObject;
import java.rmi.RemoteException;
import java.math.*;

public interface Converter extends EJBObject {
   public BigDecimal dollarToYen(BigDecimal dollars)
      throws RemoteException;
   public BigDecimal yenToEuro(BigDecimal yen)
      throws RemoteException;
}
```

Coding the Home Interface

A *home interface* defines the methods that allow a client to create, find, or remove an enterprise bean. The `ConverterHome` interface contains a single create method, which returns an object of the remote interface type. Here is the source code for the `ConverterHome` interface:

```
import java.rmi.RemoteException;
import javax.ejb.CreateException;
import javax.ejb.EJBHome;

public interface ConverterHome extends EJBHome {
  Converter create() throws RemoteException, CreateException;
}
```

Coding the Enterprise Bean Class

The enterprise bean class for this example is called ConverterBean. This class implements the two business methods (dollarToYen and yenToEuro) that the Converter remote interface defines. The source code for the ConverterBean class follows.

```
import java.rmi.RemoteException;
import javax.ejb.SessionBean;
import javax.ejb.SessionContext;
import java.math.*;

public class ConverterBean implements SessionBean {

  BigDecimal yenRate = new BigDecimal("121.6000");
  BigDecimal euroRate = new BigDecimal("0.0077");

  public BigDecimal dollarToYen(BigDecimal dollars) {
    BigDecimal result = dollars.multiply(yenRate);
    return result.setScale(2,BigDecimal.ROUND_UP);
  }

   public BigDecimal yenToEuro(BigDecimal yen) {
      BigDecimal result = yen.multiply(euroRate);
      return result.setScale(2,BigDecimal.ROUND_UP);
   }

   public ConverterBean() {}
   public void ejbCreate() {}
   public void ejbRemove() {}
   public void ejbActivate() {}
   public void ejbPassivate() {}
   public void setSessionContext(SessionContext sc) {}
}
```

Compiling the Source Files

Now you are ready to compile the remote interface (Converter.java), the home interface (ConverterHome.java), and the enterprise bean class (Converter-Bean.java).

1. In a terminal window, go to this directory:

   <*INSTALL*>/j2eetutorial14/examples/ejb/converter/

2. Type the following command:

```
asant build
```

This command compiles the source files for the enterprise bean and the application client, placing the class files in the `converter/build` subdirectory (not the `src` directory). The web client in this example requires no compilation. For more information about `asant`, see Building the Examples (page xxxvii).

Note: When compiling the code, the preceding `asant` task includes the `j2ee.jar` file in the classpath. This file resides in the `lib` directory of your Sun Java System Application Server Platform Edition 8 installation. If you plan to use other tools to compile the source code for J2EE components, make sure that the classpath includes the `j2ee.jar` file.

Packaging the Enterprise Bean

To package an enterprise bean, you run the Edit Enterprise Bean wizard of the `deploytool` utility. During this process, the wizard performs the following tasks:

- Creates the bean's deployment descriptor
- Packages the deployment descriptor and the bean's classes in an EJB JAR file
- Inserts the EJB JAR file into the `ConverterApp.ear` file

To start the Edit Enterprise Bean wizard, select File→New→Enterprise Bean. The wizard displays the following dialog boxes.

1. Introduction dialog box
   a. Read the explanatory text for an overview of the wizard's features.
   b. Click Next.
2. EJB JAR dialog box
   a. Select the button labeled Create New JAR Module in Application.
   b. In the combo box below this button, select `ConverterApp`.
   c. In the JAR Display Name field, enter `ConverterJAR`.
   d. Click Edit Contents.
   e. In the tree under Available Files, locate the `build/converter` subdirectory. (If the target directory is many levels down in the tree, you can simplify the tree view by entering all or part of the directory's path name in the Starting Directory field.)

f. In the Available Files tree select these classes: `Converter.class`, `ConverterBean.class`, and `ConverterHome.class`. (You can also drag and drop these class files to the Contents text area.)
g. Click Add.
h. Click OK.
i. Click Next.

3. General dialog box
   a. Under Bean Type, select the Stateless Session.
   b. In the Enterprise Bean Class combo box, select `converter.ConverterBean`.
   c. In the Enterprise Bean Name field, enter `ConverterBean`.
   d. In the Remote Home Interface combo box, select `converter.ConverterHome`.
   e. In the Remote Interface combo box, select `converter.Converter`.
   f. Click Next.
4. In the Expose as Web Service Endpoint dialog box, select No and click Next.
5. Click Finish.

Creating the Application Client

An application client is a program written in the Java programming language. At runtime, the client program executes in a different virtual machine than the Application Server. For detailed information on the `appclient` command-line tool, see the man page at `http://java.sun.com/j2ee/1.4/docs/relnotes/cliref/index.html`.

The application client in this example requires two JAR files. The first JAR file is for the J2EE component of the client. This JAR file contains the client's deployment descriptor and class files; it is created when you run the New Application Client wizard. Defined by the *J2EE Specification*, this JAR file is portable across all compliant application servers.

The second JAR file contains stub classes that are required by the client program at runtime. These stub classes enable the client to access the enterprise beans that are running in the Sun Java System Application Server. The JAR file for the stubs is created by `deploytool` when you deploy the application. Because this

JAR file is not covered by the J2EE specification, it is implementation-specific, intended only for the Application Server.

The application client source code is in the `ConverterClient.java` file, which is in this directory:

```
<INSTALL>/j2eetutorial14/examples/ejb/converter/src/
```

You compiled this code along with the enterprise bean code in the section Compiling the Source Files (page 882).

Coding the Application Client

The `ConverterClient.java` source code illustrates the basic tasks performed by the client of an enterprise bean:

- Locating the home interface
- Creating an enterprise bean instance
- Invoking a business method

Locating the Home Interface

The `ConverterHome` interface defines life-cycle methods such as `create` and `remove`. Before the `ConverterClient` can invoke the `create` method, it must locate and instantiate an object whose type is `ConverterHome`. This is a four-step process.

1. Create an initial naming context.

   ```
   Context initial = new InitialContext();
   ```

   The `Context` interface is part of the Java Naming and Directory Interface (JNDI). A *naming contex*t is a set of name-to-object bindings. A name that is bound within a context is the *JNDI name* of the object.

   An `InitialContext` object, which implements the `Context` interface, provides the starting point for the resolution of names. All naming operations are relative to a context.

2. Obtain the environment naming context of the application client.

   ```
   Context myEnv = (Context)initial.lookup("java:comp/env");
   ```

   The `java:comp/env` name is bound to the environment naming context of the `ConverterClient` component.

3. Retrieve the object bound to the name `ejb/SimpleConverter`.

   ```
   Object objref = myEnv.lookup("ejb/SimpleConverter");
   ```

   The `ejb/SimpleConverter` name is bound to an *enterprise bean reference*, a logical name for the home of an enterprise bean. In this case, the `ejb/SimpleConverter` name refers to the `ConverterHome` object. The names of enterprise beans should reside in the `java:comp/env/ejb` subcontext.

4. Narrow the reference to a `ConverterHome` object.

   ```
   ConverterHome home =
      (ConverterHome) PortableRemoteObject.narrow(objref,
      ConverterHome.class);
   ```

Creating an Enterprise Bean Instance

To create the bean instance, the client invokes the `create` method on the `ConverterHome` object. The `create` method returns an object whose type is `Converter`. The remote `Converter` interface defines the business methods of the bean that the client can call. When the client invokes the `create` method, the EJB container instantiates the bean and then invokes the `ConverterBean.ejbCreate` method. The client invokes the `create` method as follows:

```
Converter currencyConverter = home.create();
```

Invoking a Business Method

Calling a business method is easy: you simply invoke the method on the `Converter` object. The EJB container will invoke the corresponding method on the `ConverterBean` instance that is running on the server. The client invokes the `dollarToYen` business method in the following lines of code.

```
BigDecimal param = new BigDecimal ("100.00");
BigDecimal amount = currencyConverter.dollarToYen(param);
```

ConverterClient Source Code

The full source code for the `ConverterClient` program follows.

```
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.rmi.PortableRemoteObject;
import java.math.BigDecimal;

public class ConverterClient {

  public static void main(String[] args) {

    try {
      Context initial = new InitialContext();
      Context myEnv =
        (Context)initial.lookup("java:comp/env");
      Object objref = myEnv.lookup("ejb/SimpleConverter");

      ConverterHome home =
        (ConverterHome)PortableRemoteObject.narrow(objref,
                                  ConverterHome.class);

      Converter currencyConverter = home.create();

      BigDecimal param = new BigDecimal ("100.00");
      BigDecimal amount =
        currencyConverter.dollarToYen(param);
      System.out.println(amount);
      amount = currencyConverter.yenToEuro(param);
      System.out.println(amount);

      System.exit(0);

    } catch (Exception ex) {
      System.err.println("Caught an unexpected exception!");
      ex.printStackTrace();
    }
  }
}
```

Compiling the Application Client

The application client files are compiled at the same time as the enterprise bean files, as described in Compiling the Source Files (page 882).

Packaging the Application Client

To package an application client component, you run the New Application Client wizard of `deploytool`. During this process the wizard performs the following tasks.

- Creates the application client's deployment descriptor
- Puts the deployment descriptor and client files into a JAR file
- Adds the JAR file to the application's `ConverterApp.ear` file

To start the New Application Client wizard, select File→New→Application Client. The wizard displays the following dialog boxes.

1. Introduction dialog box
   a. Read the explanatory text for an overview of the wizard's features.
   b. Click Next.
2. JAR File Contents dialog box
   a. Select the button labeled Create New AppClient Module in Application.
   b. In the combo box below this button, select `ConverterApp`.
   c. In the AppClient Display Name field, enter `ConverterClient`.
   d. Click Edit Contents.
   e. In the tree under Available Files, locate this directory:
   `<INSTALL>/j2eetutorial14/examples/ejb/converter/build/`
   f. Select the `ConverterClient.class` file.
   g. Click Add.
   h. Click OK.
   i. Click Next.
3. General dialog box
   a. In the Main Class combo box, select `ConverterClient`.
   b. Click Next.
   c. Click Finish.

Specifying the Application Client's Enterprise Bean Reference

When it invokes the `lookup` method, the `ConverterClient` refers to the home of an enterprise bean:

```
Object objref = myEnv.lookup("ejb/SimpleConverter");
```

You specify this reference in `deploytool` as follows.

1. In the tree, select `ConverterClient`.
2. Select the EJB Ref's tab.
3. Click Add.
4. In the Coded Name field, enter `ejb/SimpleConverter`.
5. In the EJB Type field, select Session.
6. In the Interfaces field, select Remote.
7. In the Home Interface field enter, `converter.ConverterHome`.
8. In the Local/Remote Interface field, enter `converter.Converter`.
9. In the JNDI Name field, select `ConverterBean`.
10. Click OK.

Creating the Web Client

The web client is contained in the JSP page `<INSTALL>/j2eetutorial14/examples/ejb/converter/web/index.jsp`. A JSP page is a text-based document that contains JSP elements, which construct dynamic content, and static template data, which can be expressed in any text-based format such as HTML, WML, and XML.

Coding the Web Client

The statements (in bold in the following code) for locating the home interface, creating an enterprise bean instance, and invoking a business method are nearly identical to those of the application client. The parameter of the `lookup` method is the only difference; the motivation for using a different name is discussed in Mapping the Enterprise Bean References (page 893).

The classes needed by the client are declared using a JSP `page` directive (enclosed within the <%@ %> characters). Because locating the home interface and creating the enterprise bean are performed only once, this code appears in a JSP declaration (enclosed within the <%! %> characters) that contains the initialization method, `jspInit`, of the JSP page. The declaration is followed by standard HTML markup for creating a form that contains an input field. A scriptlet (enclosed within the <% %> characters) retrieves a parameter from the request and converts it to a `BigDecimal` object. Finally, JSP expressions (enclosed within <%= %> characters) invoke the enterprise bean's business methods and insert the result into the stream of data returned to the client.

```
<%@ page import="Converter,ConverterHome,javax.ejb.*,
javax.naming.*, javax.rmi.PortableRemoteObject,
java.rmi.RemoteException" %>
<%!
  private Converter converter = null;
  public void jspInit() {
     try {
        InitialContext ic = new InitialContext();
        Object objRef = ic.lookup("
           java:comp/env/ejb/TheConverter");
        ConverterHome home =
        (ConverterHome)PortableRemoteObject.narrow(
        objRef, ConverterHome.class);
        converter = home.create();
     } catch (RemoteException ex) {
        ...
     }
  }
  ...
%>
<html>
<head>
   <title>Converter</title>
</head>

<body bgcolor="white">
<h1><center>Converter</center></h1>
<hr>
<p>Enter an amount to convert:</p>
<form method="get">
<input type="text" name="amount" size="25">
<br>
<p>
<input type="submit" value="Submit">
<input type="reset" value="Reset">
```

```
</form>
<%
  String amount = request.getParameter("amount");
  if ( amount != null && amount.length() > 0 ) {
     BigDecimal d = new BigDecimal (amount);
%>
  <p><%= amount %> dollars are
     <%= converter.dollarToYen(d) %>  Yen.
  <p><%= amount %> Yen are
     <%= converter.yenToEuro(d) %>  Euro.
<%
   }
%>
</body>
</html>
```

Compiling the Web Client

The Application Server automatically compiles web clients that are JSP pages. If the web client were a servlet, you would have to compile it.

Packaging the Web Client

To package a web client, you run the New Web Component wizard of the `deploytool` utility. During this process the wizard performs the following tasks.

- Creates the web application deployment descriptor
- Adds the component files to a WAR file
- Adds the WAR file to the application's `ConverterApp.ear` file

To start the New Web Component wizard, select File→New→Web Component. The wizard displays the following dialog boxes.

1. Introduction dialog box
   a. Read the explanatory text for an overview of the wizard's features.
   b. Click Next.
2. WAR File dialog box
   a. Select the button labeled Create New WAR Module in Application.
   b. In the combo box below this button, select `ConverterApp`.
   c. In the WAR Name field, enter `ConverterWAR`.
   d. Click Edit Contents.

e. In the tree under Available Files, locate this directory:
<*INSTALL*>/j2eetutorial14/examples/ejb/converter/web/
f. Select `index.jsp`.
g. Click Add.
h. Click OK.
i. Click Next.

3. Choose Component Type dialog box
   a. Select the JSP Page button.
   b. Click Next.
4. Component General Properties dialog box
   a. In the JSP Filename combo box, select `index.jsp`.
   b. Click Finish.

Specifying the Web Client's Enterprise Bean Reference

When it invokes the `lookup` method, the web client refers to the home of an enterprise bean:

```
Object objRef = ic.lookup("java:comp/env/ejb/TheConverter");
```

You specify this reference as follows:

1. In the tree, select `ConverterWAR`.
2. Select the EJB Ref's tab.
3. Click Add.
4. In the Coded Name field, enter `ejb/TheConverter`.
5. In the EJB Type field, select Session.
6. In the Interfaces field, select Remote.
7. In the Home Interface field, enter `converter.ConverterHome`.
8. In the Local/Remote Interface field, enter `converter.Converter`.
9. In the JNDI Name field, select `ConverterBean`.
10. Click OK.

Mapping the Enterprise Bean References

Although the application client and the web client access the same enterprise bean, their code refers to the bean's home by different names. The application client refers to the bean's home as `ejb/SimpleConverter`, but the web client refers to it as `ejb/TheConverter`. These references are in the parameters of the `lookup` calls. For the `lookup` method to retrieve the home object, you must map the references in the code to the enterprise bean's JNDI name. Although this mapping adds a level of indirection, it decouples the clients from the beans, making it easier to assemble applications from J2EE components.

To map the enterprise bean references in the clients to the JNDI name of the bean, follow these steps.

1. In the tree, select `ConverterApp`.
2. Click the Sun-specific Settings button.
3. Select the JNDI Names in the View field.
4. In the Application table, note that the JNDI name for the enterprise bean is `ConverterBean`.
5. In the References table, enter `ConverterBean` in the JNDI Name column for each row.

Figure 24–1 shows what the JNDI Names tab should look like after you've performed the preceding steps.

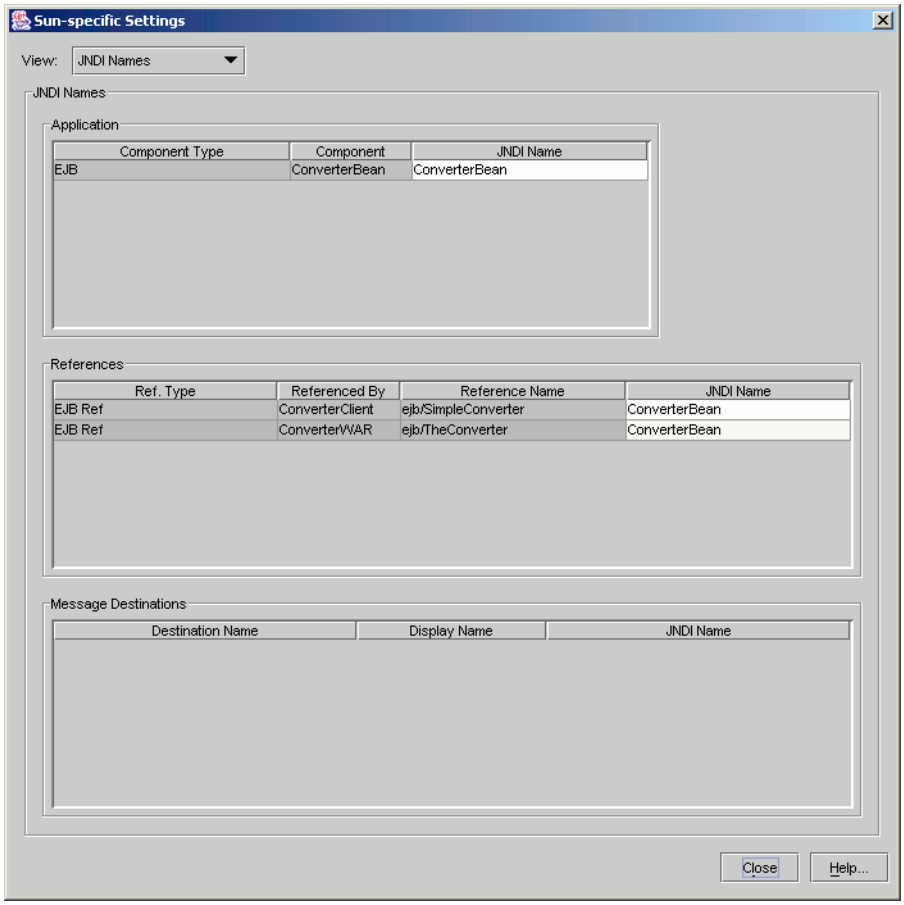

Figure 24–1 `ConverterApp` JNDI Names

Specifying the Web Client's Context Root

The context root identifies the web application. To set the context root, follow these steps:

1. In the tree, select `ConverterApp`.
2. Select the Web Context tab.
3. In the Context Root field, enter `/converter`.

For more information, see Setting the Context Root (page 92).

Deploying the J2EE Application

Now that the J2EE application contains the components, it is ready for deployment.

1. Select the `ConverterApp` application.
2. Select Tools→Deploy.
3. Under Connection Settings, enter the user name and password for the Application Server.
4. Tell `deploytool` to create a JAR file that contains the client stubs. (For more information on client JAR files, see the description under Creating the Application Client, page 884.)
   a. Select the Return Client JAR checkbox.
   b. In the field below the checkbox, enter `<INSTALL>/j2eetutorial14/examples/ejb/converter.`
5. Click OK.
6. In the Distribute Module dialog box, click Close when the deployment completes.
7. Verify the deployment.
   a. In the tree, expand the Servers node and select the host that is running the Application Server.
   b. In the Deployed Objects table, make sure that the `ConverterApp` is listed and its status is Running.
8. Verify that a stub client JAR named `ConverterAppClient.jar` resides in `<INSTALL>/j2eetutorial14/examples/ejb/converter.`

Running the Application Client

To run the application client, perform the following steps.

1. In a terminal window, go to this directory:

   `<INSTALL>/j2eetutorial14/examples/ejb/converter/`
2. Type the following command:

   ```
   appclient -client ConverterAppClient.jar
   ```

3. In the terminal window, the client displays these lines:

```
...
12160.00
0.77
...
```

Running the Web Client

To run the web client, point your browser at the following URL. Replace *<host>* with the name of the host running the Application Server. If your browser is running on the same host as the Application Server, you can replace *<host>* with `localhost`.

```
http://<host>:8080/converter
```

After entering `100` in the input field and clicking Submit, you should see the screen shown in Figure 24–2.

Figure 24–2 `ConverterApp` Web Client

Modifying the J2EE Application

The Application Server and `deploytool` support iterative development. Whenever you make a change to a J2EE application, you must redeploy the application.

Modifying a Class File

To modify a class file in an enterprise bean, you change the source code, recompile it, and redeploy the application. For example, if you want to change the exchange rate in the `dollarToYen` business method of the `ConverterBean` class, you would follow these steps.

1. Edit `ConverterBean.java`.
2. Recompile `ConverterBean.java`.
   a. In a terminal window, go to the `<INSTALL>/j2eetutorial14/examples/ejb/converter/` subdirectory.
   b. Type `asant build`.
3. In `deploytool`, select Tools→Update Module Files.
4. The Update Files dialog box appears. If the modified files are listed at the top of the dialog, click OK and go to step 6. If the files are listed at the bottom, they have not been found. Select one of those files and click Edit Search Paths.
5. In the Edit Search Paths dialog box, specify the directories where the Update Files dialog will search for modified files.
   a. In the Search Root field, enter the fully qualified name of the directory from which the search will start.
   b. In the Path Directory list, add a row for each directory that you want searched. Unless fully qualified, these directory names are relative to the Search Root field.
   c. Click OK.
6. Select Tools→Deploy. Make sure that the checkbox labeled Save Object Before Deploying is checked. If you do not want to deploy at this time, select Tools→Save to save the search paths specified in step 5.

To modify the contents of a WAR file, you follow the preceding steps. The Update Files operation checks to see whether any files have changed, including

HTML files and JSP pages. If you change the `index.jsp` file of `ConverterApp`, be sure to type `asant`. This task copies the `index.jsp` file from the `web` directory to the `build` directory.

Adding a File

To add a file to the EJB JAR or WAR of the application, perform these steps.

1. In `deploytool`, select the JAR or WAR in the tree.
2. Select the General tab.
3. Click Edit Contents.
4. In the tree of the Available Files field, locate the file and click Add.
5. Click OK.
6. From the main toolbar, select Tools→Update Module Files.
7. Select Tools→Deploy.

Modifying a Deployment Setting

To modify a deployment setting of `ConverterApp`, you edit the appropriate field in a tabbed pane and redeploy the application. For example, to change a JNDI name from `ATypo` to `ConverterBean`, you would follow these steps.

1. In `deploytool`, select `ConverterApp` in the tree.
2. Select the JNDI Names tab.
3. In the JNDI Name field, enter `MyConverter`.
4. From the main toolbar, select File→Save.
5. Select Tools→Update Module Files.
6. Select Tools→Deploy.

25

Session Bean Examples

SESSION beans are powerful because they extend the reach of your clients into remote servers yet are easy to build. In Chapter 24, you built a stateless session bean named `ConverterBean`. This chapter examines the source code of three more session beans:

- `CartBean`: a stateful session bean that is accessed by a remote client
- `HelloServiceBean`: a stateless session bean that implements a web service
- `TimerSessionBean`: a stateless session bean that sets a timer

The CartBean Example

The `CartBean` session bean represents a shopping cart in an online bookstore. The bean's client can add a book to the cart, remove a book, or retrieve the cart's contents. To construct `CartBean`, you need the following code:

- Session bean class (`CartBean`)
- Home interface (`CartHome`)
- Remote interface (`Cart`)

All session beans require a session bean class. All enterprise beans that permit remote access must have a home and a remote interface. To meet the needs of a specific application, an enterprise bean may also need some helper classes. The `CartBean` session bean uses two helper classes (`BookException` and `IdVerifier`) which are discussed in the section Helper Classes (page 906).

The source code for this example is in the *<INSTALL>*`/j2eetutorial14/examples/ejb/cart/` directory.

Session Bean Class

The session bean class for this example is called `CartBean`. Like any session bean, the `CartBean` class must meet these requirements:

- It implements the `SessionBean` interface.
- The class is defined as `public`.
- The class cannot be defined as `abstract` or `final`.
- It implements one or more `ejbCreate` methods.
- It implements the business methods.
- It contains a `public` constructor with no parameters.
- It must not define the `finalize` method.

The source code for the `CartBean` class follows.

```
import java.util.*;
import javax.ejb.*;

public class CartBean implements SessionBean {

  String customerName;
  String customerId;
  Vector contents;

  public void ejbCreate(String person)
    throws CreateException {

    if (person == null) {
      throw new CreateException("Null person not allowed.");
    }
    else {
      customerName = person;
    }
```

```
      customerId = "0";
      contents = new Vector();
   }

   public void ejbCreate(String person, String id)
      throws CreateException {

      if (person == null) {
         throw new CreateException("Null person not allowed.");
      }
      else {
         customerName = person;
      }

      IdVerifier idChecker = new IdVerifier();
      if (idChecker.validate(id)) {
         customerId = id;
      }
      else {
         throw new CreateException("Invalid id: "+ id);
      }

      contents = new Vector();
   }

   public void addBook(String title) {
      contents.addElement(title);
   }

   public void removeBook(String title) throws BookException {

      boolean result = contents.removeElement(title);
      if (result == false) {
         throw new BookException(title + "not in cart.");
      }
    }

    public Vector getContents() {
       return contents;
    }

    public CartBean() {}
    public void ejbRemove() {}
    public void ejbActivate() {}
    public void ejbPassivate() {}
    public void setSessionContext(SessionContext sc) {}

}
```

The SessionBean Interface

The SessionBean interface extends the EnterpriseBean interface, which in turn extends the Serializable interface. The SessionBean interface declares the ejbRemove, ejbActivate, ejbPassivate, and setSessionContext methods. The CartBean class doesn't use these methods, but it must implement them because they're declared in the SessionBean interface. Consequently, these methods are empty in the CartBean class. Later sections explain when you might use these methods.

The ejbCreate Methods

Because an enterprise bean runs inside an EJB container, a client cannot directly instantiate the bean. Only the EJB container can instantiate an enterprise bean. During instantiation, the example program performs the following steps.

1. The client invokes a create method on the home object:
   ```
   Cart shoppingCart = home.create("Duke DeEarl","123");
   ```
2. The EJB container instantiates the enterprise bean.
3. The EJB container invokes the appropriate ejbCreate method in CartBean:
   ```
   public void ejbCreate(String person, String id)
      throws CreateException {

      if (person == null) {
         throw new CreateException("Null person not allowed.");
      }
      else {
         customerName = person;
      }

      IdVerifier idChecker = new IdVerifier();
      if (idChecker.validate(id)) {
         customerId = id;
      }
      else {
         throw new CreateException("Invalid id: "+ id);
      }

      contents = new Vector();
   }
   ```

Typically, an `ejbCreate` method initializes the state of the enterprise bean. The preceding `ejbCreate` method, for example, initializes the `customerName` and `customerId` variables by using the arguments passed by the `create` method.

An enterprise bean must have one or more `ejbCreate` methods. The signatures of the methods must meet the following requirements:

- The access control modifier must be `public`.
- The return type must be `void`.
- If the bean allows remote access, the arguments must be legal types for the Java Remote Method Invocation (Java RMI) API.
- The modifier cannot be `static` or `final`.

The `throws` clause can include the `javax.ejb.CreateException` and other exceptions that are specific to your application. The `ejbCreate` method usually throws a `CreateException` if an input parameter is invalid.

Business Methods

The primary purpose of a session bean is to run business tasks for the client. The client invokes business methods on the remote object reference that is returned by the `create` method. From the client's perspective, the business methods appear to run locally, but they actually run remotely in the session bean. The following code snippet shows how the `CartClient` program invokes the business methods:

```
Cart shoppingCart = home.create("Duke DeEarl", "123");
...
shoppingCart.addBook("The Martian Chronicles");
shoppingCart.removeBook("Alice In Wonderland");
bookList = shoppingCart.getContents();
```

The `CartBean` class implements the business methods in the following code:

```
public void addBook(String title) {
   contents.addElement(title);
}

public void removeBook(String title) throws BookException {
   boolean result = contents.removeElement(title);
   if (result == false) {
      throw new BookException(title + "not in cart.");
   }
}
```

```
public Vector getContents() {
   return contents;
}
```

The signature of a business method must conform to these rules:

- The method name must not conflict with one defined by the EJB architecture. For example, you cannot call a business method `ejbCreate` or `ejbActivate`.
- The access control modifier must be `public`.
- If the bean allows remote access, the arguments and return types must be legal types for the Java RMI API.
- The modifier must not be `static` or `final`.

The `throws` clause can include exceptions that you define for your application. The `removeBook` method, for example, throws the `BookException` if the book is not in the cart.

To indicate a system-level problem, such as the inability to connect to a database, a business method should throw the `javax.ejb.EJBException`. When a business method throws an `EJBException`, the container wraps it in a `RemoteException`, which is caught by the client. The container will not wrap application exceptions such as `BookException`. Because `EJBException` is a subclass of `RuntimeException`, you do not need to include it in the `throws` clause of the business method.

Home Interface

A home interface extends the `javax.ejb.EJBHome` interface. For a session bean, the purpose of the home interface is to define the `create` methods that a remote client can invoke. The `CartClient` program, for example, invokes this `create` method:

```
Cart shoppingCart = home.create("Duke DeEarl", "123");
```

Every `create` method in the home interface corresponds to an `ejbCreate` method in the bean class. The signatures of the `ejbCreate` methods in the `CartBean` class follow:

```
public void ejbCreate(String person) throws CreateException
...
public void ejbCreate(String person, String id)
   throws CreateException
```

Compare the `ejbCreate` signatures with those of the `create` methods in the `CartHome` interface:

```
import java.io.Serializable;
import java.rmi.RemoteException;
import javax.ejb.CreateException;
import javax.ejb.EJBHome;

public interface CartHome extends EJBHome {
  Cart create(String person) throws
              RemoteException, CreateException;
  Cart create(String person, String id) throws
              RemoteException, CreateException;
}
```

The signatures of the `ejbCreate` and `create` methods are similar, but they differ in important ways. The rules for defining the signatures of the `create` methods of a home interface follow.

- The number and types of arguments in a `create` method must match those of its corresponding `ejbCreate` method.
- The arguments and return type of the `create` method must be valid RMI types.
- A `create` method returns the remote interface type of the enterprise bean. (But an `ejbCreate` method returns `void`.)
- The `throws` clause of the `create` method must include the `java.rmi.RemoteException` and the `javax.ejb.CreateException`.

Remote Interface

The remote interface, which extends `javax.ejb.EJBObject`, defines the business methods that a remote client can invoke. Here is the source code for the `Cart` remote interface:

```
import java.util.*;
import javax.ejb.EJBObject;
import java.rmi.RemoteException;

public interface Cart extends EJBObject {

   public void addBook(String title) throws RemoteException;
   public void removeBook(String title) throws
                  BookException, RemoteException;
   public Vector getContents() throws RemoteException;
}
```

The method definitions in a remote interface must follow these rules:

- Each method in the remote interface must match a method implemented in the enterprise bean class.
- The signatures of the methods in the remote interface must be identical to the signatures of the corresponding methods in the enterprise bean class.
- The arguments and return values must be valid RMI types.
- The `throws` clause must include the `java.rmi.RemoteException`.

Helper Classes

The `CartBean` session bean has two helper classes: `BookException` and `IdVerifier`. The `BookException` is thrown by the `removeBook` method, and the `IdVerifier` validates the `customerId` in one of the `ejbCreate` methods. Helper classes must reside in the EJB JAR file that contains the enterprise bean class.

Building the CartBean Example

Now you are ready to compile the remote interface (`Cart.java`), the home interface (`CartHome.java`), the enterprise bean class (`CartBean.java`), the client

class (`CartClient.java`), and the helper classes (`BookException.java` and `IdVerifier.java`).

1. In a terminal window, go to this directory:

   `<INSTALL>/j2eetutorial14/examples/ejb/cart/`

2. Type the following command:

   ```
   asant build
   ```

Creating the Application

In this section, you'll create a J2EE application named `CartApp`, storing it in the file `CartApp.ear`.

1. In `deploytool`, select File→New→Application.
2. Click Browse.
3. In the file chooser, navigate to `<INSTALL>/j2eetutorial14/examples/ejb/cart/`.
4. In the File Name field, enter `CartApp`.
5. Click New Application.
6. Click OK.
7. Verify that the `CartApp.ear` file resides in `<INSTALL>/j2eetutorial14/examples/ejb/cart/`.

Packaging the Enterprise Bean

1. In `deploytool`, select File→New→Enterprise Bean.
2. In the EJB JAR screen:
   a. Select Create New JAR Module in Application.
   b. In the Create New JAR Module in Application field, select `CartApp`.
   c. In the JAR Name field, enter `CartJAR`.
   d. Click Choose Module File.
   e. Click Edit Contents.
   f. Locate the `<INSTALL>/j2eetutorial14/examples/ejb/cart/build/` directory.
   g. Select `BookException.class`, `Cart.class`, `CartBean.class`, `CartHome.class`, and `IdVerifier.class`.

h. Click Add.
i. Click OK.
j. Click Next.

3. In the General screen:
   a. In the Enterprise Bean Class field, select `CartBean`.
   b. In the Enterprise Bean Name field, enter `CartBean`.
   c. In the Enterprise Bean Type field, select `Stateful Session`.
   d. In the Remote Home Interface field, select `CartHome`.
   e. In the Remote Interface field, select `Cart`.
   f. Click Next.
4. Click Finish.

Packaging the Application Client

To package an application client component, you run the New Application Client wizard of `deploytool`. During this process the wizard performs the following tasks.

- Creates the application client's deployment descriptor
- Puts the deployment descriptor and client files into a JAR file
- Adds the JAR file to the application's `CartApp.ear` file

To start the New Application Client wizard, select File→New→Application Client. The wizard displays the following dialog boxes.

1. Introduction dialog box
   a. Read the explanatory text for an overview of the wizard's features.
   b. Click Next.
2. JAR File Contents dialog box
   a. Select the button labeled Create New AppClient Module in Application.
   b. In the combo box below this button, select `CartApp`.
   c. In the AppClient Display Name field, enter `CartClient`.
   d. Click Edit Contents.
   e. In the tree under Available Files, locate the *<INSTALL>*`/j2eetutorial14/examples/ejb/cart/build` directory.
   f. Select `CartClient.class`.

g. Click Add.
h. Click OK.
i. Click Next.

3. General dialog box
   a. In the Main Class combo box, select `CartClient`.
   b. Click Next.
   c. Click Finish.

Specifying the Application Client's Enterprise Bean Reference

When it invokes the `lookup` method, the `CartClient` refers to the home of an enterprise bean:

```
Object objref =
  initial.lookup("java:comp/env/ejb/SimpleCart");
```

You specify this reference as follows.

1. In the tree, select `CartClient`.
2. Select the EJB Ref's tab.
3. Click Add.
4. In the Coded Name field, enter `ejb/SimpleCart`.
5. In the EJB Type field, select Session.
6. In the Interfaces field, select Remote.
7. In the Home Interface field, enter `CartHome`.
8. In the Local/Remote Interface field, enter `Cart`.
9. In the JNDI Name field, select `CartBean`.
10. Click OK.

Deploying the Enterprise Application

Now that the J2EE application contains the components, it is ready for deployment.

1. Select `CartApp`.

2. Select Tools→Deploy.
3. Under Connection Settings, enter the user name and password for the Sun Java System Application Server Platform Edition 8.
4. Under Application Client Stub Directory, check Return Client Jar.
5. In the field below the checkbox enter *<INSTALL>*/j2eetutorial14/examples/ejb/cart/.
6. Click OK.
7. In the Distribute Module dialog box, click Close when the deployment completes.
8. Verify the deployment.
   a. In the tree, expand the Servers node and select the host that is running the Application Server.
   b. In the Deployed Objects table, make sure that `CartApp` is listed and that its status is `Running`.
   c. Verify that `CartAppClient.jar` is in *<INSTALL>*/j2eetutorial14/examples/ejb/cart/.

Running the Application Client

To run the application client, perform the following steps.

1. In a terminal window, go to the *<INSTALL>*/j2eetutorial14/examples/ejb/cart/ directory.
2. Type the following command:
   ```
   appclient -client CartAppClient.jar
   ```
3. In the terminal window, the client displays these lines:
   ```
   The Martian Chronicles
   2001 A Space Odyssey
   The Left Hand of Darkness
   Caught a BookException: Alice in Wonderland not in cart.
   ```

A Web Service Example: HelloServiceBean

This example demonstrates a simple web service that generates a response based on information received from the client. `HelloServiceBean` is a stateless session bean that implements a single method, `sayHello`. This method matches the `sayHello` method invoked by the clients described in Static Stub Client (page 327). Later in this section, you'll test the `HelloServiceBean` by running one of these JAX-RPC clients.

Web Service Endpoint Interface

`HelloService` is the bean's web service endpoint interface. It provides the client's view of the web service, hiding the stateless session bean from the client. A web service endpoint interface must conform to the rules of a JAX-RPC service definition interface. For a summary of these rules, see Coding the Service Endpoint Interface and Implementation Class (page 322). Here is the source code for the `HelloService` interface:

```
package helloservice;
import java.rmi.RemoteException;
import java.rmi.Remote;

public interface HelloService extends Remote {

   public String sayHello(String name) throws RemoteException;
}
```

Stateless Session Bean Implementation Class

The `HelloServiceBean` class implements the `sayHello` method defined by the `HelloService` interface. The interface decouples the implementation class from the type of client access. For example, if you added remote and home interfaces to `HelloServiceBean`, the methods of the `HelloServiceBean` class could also

be accessed by remote clients. No changes to the HelloServiceBean class would be necessary. The source code for the HelloServiceBean class follows:

```
package helloservice;
import java.rmi.RemoteException;
import javax.ejb.SessionBean;
import javax.ejb.SessionContext;

public class HelloServiceBean implements SessionBean {

   public String sayHello(String name) {

      return "Hello "+ name + " from HelloServiceBean";
   }

   public HelloServiceBean() {}
   public void ejbCreate() {}
   public void ejbRemove() {}
   public void ejbActivate() {}
   public void ejbPassivate() {}
   public void setSessionContext(SessionContext sc) {}
}
```

Building HelloServiceBean

In a terminal window, go to the *<INSTALL>*/j2eetutorial14/examples/ejb/helloservice/ directory. To build HelloServiceBean, type the following command:

```
asant build-service
```

This command performs the following tasks:

- Compiles the bean's source code files
- Creates the MyHelloService.wsdl file by running the following wscompile command:

  ```
  wscompile -define -d build/output -nd build -classpath build
  -mapping build/mapping.xml config-interface.xml
  ```

The wscompile tool writes the MyHelloService.wsdl file to the *<INSTALL>*/j2eetutorial14/examples/ejb/helloservice/build/ subdirectory. For more information about the wscompile tool, see Chapter 8.

Use deploytool to package and deploy this example.

Creating the Application

In this section, you'll create a J2EE application named `HelloService`, storing it in the file `HelloService.ear`.

1. In `deploytool`, select File→New→Application.
2. Click Browse.
3. In the file chooser, navigate to *<INSTALL>*`/j2eetutorial14/examples/ejb/helloservice/`.
4. In the File Name field, enter `HelloServiceApp`.
5. Click New Application.
6. Click OK.
7. Verify that the `HelloServiceApp.ear` file resides in *<INSTALL>*`/j2eetutorial14/examples/ejb/helloservice/`.

Packaging the Enterprise Bean

Start the Edit Enterprise Bean wizard by selecting File→New→Enterprise Bean. The wizard displays the following dialog boxes.

1. Introduction dialog box
   a. Read the explanatory text for an overview of the wizard's features.
   b. Click Next.
2. EJB JAR dialog box
   a. Select the button labeled Create New JAR Module in Application.
   b. In the combo box below this button, select `HelloService`.
   c. In the JAR Display Name field, enter `HelloServiceJAR`.
   d. Click Edit Contents.
   e. In the tree under Available Files, locate the *<INSTALL>*`/j2eetutorial14/examples/ejb/helloservice/build/` directory.
   f. In the Available Files tree select the `helloservice` directory and `mapping.xml` and `MyHelloService.wsdl`.
   g. Click Add.
   h. Click OK.
   i. Click Next.

3. General dialog box
   a. In the Enterprise Bean Class combo box, select `helloservice.HelloServiceBean`.
   b. Under Enterprise Bean Type, select Stateless Session.
   c. In the Enterprise Bean Name field, enter `HelloServiceBean`.
   d. Click Next.
4. In the Configuration Options dialog box, click Next. The wizard will automatically select the Yes button for Expose Bean as Web Service Endpoint.
5. In the Choose Service dialog box:
   a. Select `META-INF/wsdl/MyHelloService.wsdl` in the WSDL File combo box.
   b. Select `mapping.xml` from the Mapping File combo box.
   c. Make sure that `MyHelloService` is in the Service Name and Service Display Name edit boxes.
6. In the Web Service Endpoint dialog box:
   a. Select `helloservice.HelloIF` in the Service Endpoint Interface combo box.
   b. In the WSDL Port section, set the Namespace to `urn:Foo`, and the Local Part to `HelloIFPort`.
   c. In the Sun-specific Settings section, set the Endpoint Address to `hello-ejb/hello`.
   d. Click Next.
7. Click Finish.
8. Select File→Save.

Deploying the Enterprise Application

Now that the J2EE application contains the enterprise bean, it is ready for deployment.

1. Select the `HelloService` application.
2. Select Tools→Deploy.
3. Under Connection Settings, enter the user name and password for the Application Server.
4. Click OK.

5. In the Distribute Module dialog box, click Close when the deployment completes.
6. Verify the deployment.
   a. In the tree, expand the Servers node and select the host that is running the Application Server.
   b. In the Deployed Objects table, make sure that `HelloService` is listed and that its status is `Running`.

Building the Web Service Client

In the next section, to test the web service implemented by `HelloServiceBean`, you will run the JAX-RPC client described in Chapter 8.

To verify that `HelloServiceBean` has been deployed, click on the target Application Server in the Servers tree in `deploytool`. In the Deployed Objects tree you should see `HelloServiceApp`.

To build the static stub client, perform these steps:

1. In a terminal go to the *<INSTALL>*`/j2eetutorial14/examples/jaxrpc/helloservice/` directory and type

   ```
   asant build
   ```

2. In a terminal go to the *<INSTALL>*`/j2eetutorial14/examples/jaxrpc/staticstub/` directory.
3. Open `config-wsdl.xml` in a text editor and change the line that reads

   ```
   <wsdl location="http://localhost:8080/hello-jaxrpc/hello?WSDL"
   ```

   to

   ```
   <wsdl location="http://localhost:8080/hello-ejb/hello?WSDL"
   ```

4. Type

   ```
   asant build
   ```

5. Edit the `build.properties` file and change the `endpoint.address` property to

   ```
   http://localhost:8080/hello-ejb/hello
   ```

For details about creating the JAX-RPC service and client, see these sections: Creating a Simple Web Service and Client with JAX-RPC (page 320) and Static Stub Client (page 327).

Running the Web Service Client

To run the client, go to the *<INSTALL>*/j2eetutorial14/examples/jaxrpc/staticstub/ directory and enter

```
asant run
```

The client should display the following line:

```
Hello Duke! (from HelloServiceBean)
```

Other Enterprise Bean Features

The topics that follow apply to session beans and entity beans.

Accessing Environment Entries

Stored in an enterprise bean's deployment descriptor, an *environment entry* is a name-value pair that allows you to customize the bean's business logic without changing its source code. An enterprise bean that calculates discounts, for example, might have an environment entry named `Discount Percent`. Before deploying the bean's application, you could run a development tool to assign `Discount Percent` a value of 0.05 in the bean's deployment descriptor. When you run the application, the bean fetches the 0.05 value from its environment.

In the following code example, the `applyDiscount` method uses environment entries to calculate a discount based on the purchase amount. First, the method locates the environment naming context by invoking `lookup` using the `java:comp/env` parameter. Then it calls `lookup` on the environment to get the values for the `Discount Level` and `Discount Percent` names. For example, if you assign a value of 0.05 to the `Discount Percent` entry, the code will assign 0.05 to the `discountPercent` variable. The `applyDiscount` method, which follows, is in the `CheckerBean` class. The source code for this example is in *<INSTALL>*/j2eetutorial14/examples/ejb/checker.

```
public double applyDiscount(double amount) {

  try {

     double discount;
```

```
      Context initial = new InitialContext();
         Context environment =
            (Context)initial.lookup("java:comp/env");

      Double discountLevel =
         (Double)environment.lookup("Discount Level");
            Double discountPercent =
               (Double)environment.lookup("Discount Percent");

      if (amount >= discountLevel.doubleValue()) {
         discount = discountPercent.doubleValue();
      }
      else {
         discount = 0.00;
      }

      return amount * (1.00 - discount);

    } catch (NamingException ex) {
      throw new EJBException("NamingException: "+
         ex.getMessage());
    }
  }
```

Comparing Enterprise Beans

A client can determine whether two stateful session beans are identical by invoking the `isIdentical` method:

```
bookCart = home.create("Bill Shakespeare");
videoCart = home.create("Lefty Lee");
...
if (bookCart.isIdentical(bookCart)) {
   // true ... }
if (bookCart.isIdentical(videoCart)) {
   // false ... }
```

Because stateless session beans have the same object identity, the `isIdentical` method always returns `true` when used to compare them.

To determine whether two entity beans are identical, the client can invoke the `isIdentical` method, or it can fetch and compare the beans's primary keys:

```
String key1 = (String)accta.getPrimaryKey();
String key2 = (String)acctb.getPrimaryKey();

if (key1.compareTo(key2) == 0)
   System.out.println("equal");
```

Passing an Enterprise Bean's Object Reference

Suppose that your enterprise bean needs to pass a reference to itself to another bean. You might want to pass the reference, for example, so that the second bean can call the first bean's methods. You can't pass the `this` reference because it points to the bean's instance, which is running in the EJB container. Only the container can directly invoke methods on the bean's instance. Clients access the instance indirectly by invoking methods on the object whose type is the bean's remote interface. It is the reference to this object (the bean's remote reference) that the first bean would pass to the second bean.

A session bean obtains its remote reference by calling the `getEJBObject` method of the `SessionContext` interface. An entity bean would call the `getEJBObject` method of the `EntityContext` interface. These interfaces provide beans with access to the instance contexts maintained by the EJB container. Typically, the bean saves the context in the `setSessionContext` method. The following code fragment shows how a session bean might use these methods.

```
public class WagonBean implements SessionBean {

   SessionContext context;
   ...
   public void setSessionContext(SessionContext sc) {
      this.context = sc;
   }
   ...
   public void passItOn(Basket basket) {
  ...
      basket.copyItems(context.getEJBObject());
   }
```

Using the Timer Service

Applications that model business work flows often rely on timed notifications. The timer service of the enterprise bean container enables you to schedule timed notifications for all types of enterprise beans except for stateful session beans. You can schedule a timed notification to occur at a specific time, after a duration of time, or at timed intervals. For example, you could set timers to go off at 10:30 AM on May 23, in 30 days, or every 12 hours.

When a timer expires (goes off), the container calls the `ejbTimeout` method of the bean's implementation class. The `ejbTimeout` method contains the business logic that handles the timed event. Because `ejbTimeout` is defined by the `javax.ejb.TimedObject` interface, the bean class must implement `TimedObject`.

There are four interfaces in the `javax.ejb` package that are related to timers:

- `TimedObject`
- `Timer`
- `TimerHandle`
- `TimerService`

Creating Timers

To create a timer, the bean invokes one of the `createTimer` methods of the `TimerService` interface. (For details on the method signatures, see the `TimerService` API documentation.) When the bean invokes `createTimer`, the timer service begins to count down the timer duration.

The bean described in The TimerSessionBean Example (page 921) creates a timer as follows:

```
TimerService timerService = context.getTimerService();
Timer timer = timerService.createTimer(intervalDuration,
  "created timer");
```

In the `TimerSessionBean` example, `createTimer` is invoked in a business method, which is called by a client. An entity bean can also create a timer in a business method. If you want to create a timer for each instance of an entity bean, you can code the `createTimer` call in the bean's `ejbCreate` method.

Timers are persistent. If the server is shut down (or even crashes), timers are saved and will become active again when the server is restarted. If a timer expires while the server is down, the container will call `ejbTimeout` when the server is restarted.

A timer for an entity bean is associated with the bean's identity—that is, with a particular instance of the bean. If an entity bean sets a timer in `ejbCreate`, for example, each bean instance will have its own timer. In contrast, stateless session and message-driven beans do not have unique timers for each instance.

The `Date` and `long` parameters of the `createTimer` methods represent time with the resolution of milliseconds. However, because the timer service is not intended for real-time applications, a callback to `ejbTimeout` might not occur with millisecond precision. The timer service is for business applications, which typically measure time in hours, days, or longer durations.

Canceling and Saving Timers

Timers can be canceled by the following events:

- When a single-event timer expires, the EJB container calls `ejbTimeout` and then cancels the timer.
- When an entity bean instance is removed, the container cancels the timers associated with the instance.
- When the bean invokes the `cancel` method of the `Timer` interface, the container cancels the timer.

If a method is invoked on a canceled timer, the container throws the `javax.ejb.NoSuchObjectLocalException`.

To save a `Timer` object for future reference, invoke its `getHandle` method and store the `TimerHandle` object in a database. (A `TimerHandle` object is serializable.) To reinstantiate the `Timer` object, retrieve the handle from the database and invoke `getTimer` on the handle. A `TimerHandle` object cannot be passed as an argument of a method defined in a remote or web service interface. In other words, remote clients and web service clients cannot access a bean's `TimerHandle` object. Local clients, however, do not have this restriction.

Getting Timer Information

In addition to defining the `cancel` and `getHandle` methods, the `Timer` interface defines methods for obtaining information about timers:

```
public long getTimeRemaining();
public java.util.Date getNextTimeout();
public java.io.Serializable getInfo();
```

The `getInfo` method returns the object that was the last parameter of the `createTimer` invocation. For example, in the `createTimer` code snippet of the preceding section, this information parameter is a `String` object with the value `created timer`.

To retrieve all of a bean's active timers, call the `getTimers` method of the `TimerService` interface. The `getTimers` method returns a collection of `Timer` objects.

Transactions and Timers

An enterprise bean usually creates a timer within a transaction. If this transaction is rolled back, the timer creation is also rolled back. Similarly, if a bean cancels a timer within a transaction that gets rolled back, the timer cancellation is rolled back. In this case, the timer's duration is reset as if the cancellation had never occurred.

In beans that use container-managed transactions, the `ejbTimeout` method usually has the `RequiresNew` transaction attribute to preserve transaction integrity. With this attribute, the EJB container begins the new transaction before calling `ejbTimeout`. If the transaction is rolled back, the container will try to call `ejbTimeout` at least one more time.

The TimerSessionBean Example

The source code for this example is in the *<INSTALL>*`/j2eetutorial14/examples/ejb/timersession/src/` directory.

`TimerSessionBean` is a stateless session bean that shows how to set a timer. The implementation class for `TimerSessionBean` is called `TimerSessionBean`. In the source code listing of `TimerSessionBean` that follows, note the `myCreateTimer` and `ejbTimeout` methods. Because it's a business method, `myCre-`

ateTimer is defined in the bean's remote interface (TimerSession) and can be invoked by the client. In this example, the client invokes myCreateTimer with an interval duration of 30,000 milliseconds. The myCreateTimer method fetches a TimerService object from the bean's SessionContext. Then it creates a new timer by invoking the createTimer method of TimerService. Now that the timer is set, the EJB container will invoke the ejbTimer method of TimerSessionBean when the timer expires—in about 30 seconds. Here's the source code for the TimerSessionBean class:

```
import javax.ejb.*;

public class TimerSessionBean implements SessionBean,
    TimedObject {

    private SessionContext context;

    public TimerHandle myCreateTimer(long intervalDuration) {

        System.out.println
            ("TimerSessionBean: start createTimer ");
        TimerService timerService =
            context.getTimerService();
        Timer timer =
            timerService.createTimer(intervalDuration,
             "created timer");
}

    public void ejbTimeout(Timer timer) {

        System.out.println("TimerSessionBean: ejbTimeout ");
    }

    public void setSessionContext(SessionContext sc) {
        System.out.println("TimerSessionBean:
            setSessionContext");
        context = sc;
     }

     public void ejbCreate() {
        System.out.println("TimerSessionBean: ejbCreate");
     }

     public TimerSessionBean() {}
     public void ejbRemove() {}
```

```
    public void ejbActivate() {}
    public void ejbPassivate() {}

}
```

Building TimerSessionBean

In a terminal window, go to the *<INSTALL>*/j2eetutorial14/examples/ejb/timersession/ directory. To build TimerSessionBean, type the following command:

```
asant build
```

Use deploytool to package and deploy this example.

Creating the Application

In this section, you'll create a J2EE application named TimerSessionApp, storing it in the file TimerSessionApp.ear.

1. In deploytool, select File→New→Application.
2. Click Browse.
3. In the file chooser, navigate to *<INSTALL>*/j2eetutorial14/examples/ejb/timersession/.
4. In the File Name field, enter TimerSessionApp.ear.
5. Click New Application.
6. Click OK.
7. Verify that the TimerSessonApp.ear file resides in *<INSTALL>*/j2eetutorial14/examples/ejb/timersession/.

Packaging the Enterprise Bean

Start the Edit Enterprise Bean wizard by selecting File→New→Enterprise JavaBean. The wizard displays the following dialog boxes.

1. In the Introduction dialog box:
   a. Read the explanatory text for an overview of the wizard's features.
   b. Click Next.

2. In the EJB JAR dialog box:
   a. Select the button labeled Create New JAR Module in Application.
   b. In the combo box below this button, select `TimerSessionApp`.
   c. In the JAR Display Name field, enter `TimerSessionJAR`.
   d. Click Edit Contents.
   e. In the tree under Available Files, locate the `<INSTALL>/j2eetutorial14/examples/ejb/timersession/build/` directory.
   f. Select these classes: `TimerSession.class`, `TimerSessionBean.class`, and `TimerSessionHome.class`.
   g. Click Add.
   h. Click OK.
   i. Click Next.
3. In the General dialog box:
   a. In the Enterprise Bean Class combo box, select `TimerSessionBean`.
   b. In the Enterprise Bean Name field, enter `TimerSessionBean`.
   c. Under Bean Type, select Stateless Session.
   d. In the Remote Interfaces section, select `TimerSessionHome` for the Remote Home Interface, and `TimerSession` for the Remote Interface.
   e. Click Next.
4. In the Expose as Web Service Endpoint dialog box:
   a. Select No for Expose Bean as Web Service Endpoint.
   b. Click Next.
5. Click Finish.

Compiling the Application Client

The application client files are compiled at the same time as the enterprise bean files.

Packaging the Application Client

To package an application client component, you run the New Application Client wizard of `deploytool`. During this process the wizard performs the following tasks.

- Creates the application client's deployment descriptor
- Puts the deployment descriptor and client files into a JAR file
- Adds the JAR file to the application's `TimerSessionApp.ear` file

To start the New Application Client wizard, select File→New→Application Client. The wizard displays the following dialog boxes.

1. Introduction dialog box
   a. Read the explanatory text for an overview of the wizard's features.
   b. Click Next.
2. JAR File Contents dialog box
   a. Select the button labeled Create New AppClient Module in Application.
   b. In the combo box below this button, select `TimerSessionApp`.
   c. In the AppClient Display Name field, enter `TimerSessionClient`.
   d. Click Edit Contents.
   e. In the tree under Available Files, locate the *<INSTALL>*`/j2eetutorial14/examples/ejb/timersession/build` directory.
   f. Select the `TimerSessionClient.class` file.
   g. Click Add.
   h. Click OK.
   i. Click Next.
3. General dialog box
   a. In the Main Class combo box, select `TimerSessionClient`.
   b. Click Next.
   c. Click Finish.

Specifying the Application Client's Enterprise Bean Reference

When it invokes the `lookup` method, the `TimerSessionClient` refers to the home of an enterprise bean:

```
Object objref =
  initial.lookup("java:comp/env/ejb/SimpleTimerSession");
```

You specify this reference as follows.

1. In the tree, select `TimerSessionClient`.
2. Select the EJB Ref's tab.
3. Click Add.
4. In the Coded Name field, enter `ejb/SimpleTimerSession`.
5. In the EJB Type field, select `Session`.
6. In the Interfaces field, select `Remote`.
7. In the Home Interface field, enter `TimerSessionHome`.
8. In the Local/Remote Interface field, enter `TimerSession`.
9. In the JNDI Name field, select `TimerSessionBean`.
10. Click OK.

Deploying the Enterprise Application

Now that the J2EE application contains the components, it is ready for deployment.

1. Select `TimerSessionApp`.
2. Select Tools→Deploy.
3. Under Connection Settings, enter the user name and password for the Application Server.
4. Under Application Client Stub Directory, check Return Client Jar.
5. In the field below the checkbox, enter *<INSTALL>*`/j2eetutorial14/examples/ejb/timersession/`.
6. Click OK.
7. In the Distribute Module dialog box, click Close when the deployment completes.

8. Verify the deployment.
   a. In the tree, expand the Servers node and select the host that is running the Application Server.
   b. In the Deployed Objects table, make sure that `TimerSessionApp` is listed and that its status is `Running`.
   c. Verify that `TimerSessionAppClient.jar` is in *<INSTALL>*`/j2eetutorial14/examples/ejb/timersession/`.

Running the Application Client

To run the application client, perform the following steps.

1. In a terminal window, go to the *<INSTALL>*`/j2eetutorial14/examples/ejb/timersession/` directory.
2. Type the following command:

   ```
   appclient -client TimerSessionAppClient.jar
   ```

3. In the terminal window, the client displays these lines:

   ```
   Creating a timer with an interval duration of 30000 ms.
   ```

The output from the timer is sent to the `server.log` file located in the *<J2EE_HOME>*`/domains/domain1/server/logs/` directory.

View the output in the Admin Console:

1. Open the Admin Console by opening a web browser window to `http://localhost:4848/asadmin/admingui`
2. Click the Logging tab.
3. Click Open Log Viewer.
4. At the top of the page, you'll see these four lines in the Message column:

   ```
   ejbTimeout
   start createTimer
   ejbCreate
   setSessionContext
   ```

Alternatively, you can look at the log file directly. After about 30 seconds, open `server.log` in a text editor and you will see the following lines:

```
TimerSessionBean: setSessionContext
TimerSessionBean: ejbCreate
TimerSessionBean: start createTimer
TimerSessionBean: ejbTimeout
```

Handling Exceptions

The exceptions thrown by enterprise beans fall into two categories: system and application.

A *system exception* indicates a problem with the services that support an application. Examples of these problems include the following: a database connection cannot be obtained, an SQL insert fails because the database is full, or a `lookup` method cannot find the desired object. If your enterprise bean encounters a system-level problem, it should throw a `javax.ejb.EJBException`. The container will wrap the `EJBException` in a `RemoteException`, which it passes back to the client. Because the `EJBException` is a subclass of the `RuntimeException`, you do not have to specify it in the `throws` clause of the method declaration. If a system exception is thrown, the EJB container might destroy the bean instance. Therefore, a system exception cannot be handled by the bean's client program; it requires intervention by a system administrator.

An *application exception* signals an error in the business logic of an enterprise bean. There are two types of application exceptions: customized and predefined. A customized exception is one that you've coded yourself, such as the `InsufficentBalanceException` thrown by the `debit` business method of the `SavingsAccountBean` example. The `javax.ejb` package includes several predefined exceptions that are designed to handle common problems. For example, an `ejbCreate` method should throw a `CreateException` to indicate an invalid input parameter. When an enterprise bean throws an application exception, the container does not wrap it in another exception. The client should be able to handle any application exception it receives.

If a system exception occurs within a transaction, the EJB container rolls back the transaction. However, if an application exception is thrown within a transaction, the container does not roll back the transaction.

Table 25–1 summarizes the exceptions of the `javax.ejb` package. All of these exceptions are application exceptions, except for the `NoSuchEntityException` and the `EJBException`, which are system exceptions.

Table 25–1 Exceptions

Method Name	Exception It Throws	Reason for Throwing
`ejbCreate`	`CreateException`	An input parameter is invalid.

Table 25–1 Exceptions (Continued)

Method Name	Exception It Throws	Reason for Throwing
`ejbFindByPrimaryKey` (and other finder methods that return a single object)	`ObjectNotFoundException` (subclass of `FinderException`)	The database row for the requested entity bean cannot be found.
`ejbRemove`	`RemoveException`	The entity bean's row cannot be deleted from the database.
`ejbLoad`	`NoSuchEntityException`	The database row to be loaded into the entity bean cannot be found.
`ejbStore`	`NoSuchEntityException`	The database row to be updated cannot be found.
(all methods)	`EJBException`	A system problem has been encountered.