Chapter 6. Enterprise JavaBeans

RMI and JavaBeans bring a standard distributed object framework and a component model, respectively, to the general Java environment. The Enterprise JavaBeans architecture builds on these foundations to provide a standard distributed component model for the enterprise Java environment.

So, you may ask, how are EJB components different from "regular" RMI and CORBA distributed objects or nondistributed JavaBeans components? Well, in a nutshell (so to speak), an EJB includes the capabilities of both and adds an entire set of enterprise services. An EJB component has the remote capabilities of an RMI or CORBA object, in the sense that it can be exported as a remote object using RMI or RMI/IIOP. An EJB component is also a JavaBeans component since it has properties that can be introspected and it uses JavaBeans conventions for defining accessor methods for its properties. The EJB architecture provides a framework in which a developer can easily take advantage of transaction processing, security, persistence, and resource-pooling facilities provided by an EJB container. These facilities don't come for free, of course. You need to understand how they work and what rules your EJB object needs to follow in order to take advantage of these services.

Enterprise JavaBeans are useful in any situation in which regular distributed objects are useful. They excel, however, in situations that take advantage of the component nature of EJB objects and the other services that EJB objects can provide with relative ease, such as transaction processing and persistence. A good example is an online banking application. A user sitting at home wants to connect to all of her financial accounts, no matter where and with whom they may live, and see them tied together into one convenient interface. The EJB component architecture allows the various financial institutions to export user accounts as different implementations of a common `Account` interface, just as we would do with other distributed object APIs. But since the `Account` objects are also EJBs, we can use the EJB server's transaction management capabilities to implement the `Accounts` as transactional components, which allows the client to perform a number of account operations within a single transaction and then either commit them all or roll them back. This can be a critical feature in financial applications, especially if you need to ensure that a supporting transfer can be executed before a withdrawal request is submitted. The transactional support in EJB ensures that if an error occurs during the transfer and an exception is raised, the entire transaction can be rolled back and the client-side application can inform you of the reason. And of course, an application such as this may also take advantage of any of the other EJB-related component services.

The EJB component model insulates applications and beans (for the most part) from the details of the component services included in the specification. A benefit of this separation is the ability to deploy the same enterprise bean under different conditions, as needed by specific applications. The parameters used to control a bean's transactional nature, persistence, resource pooling, and security management are specified in separate deployment descriptors , and not embedded in the bean implementation or the client application code. So, when a bean is deployed in a distributed application, the properties of the deployment environment (client load levels and database configuration, for example) can be accounted for and reflected in the settings of the bean's deployment options.

The EJB API is a standard extension to Java, available in the `javax.ejb` package and its subpackages. You must install this extension API explicitly to write code against the EJB interfaces. You can find the latest version of the API at http://java.sun.com/products/ejb. You should also note that EJB is just a specification for how distributed components should work within the Java environment. In order to actually create and use EJB objects, you need to install an EJB-enabled server. J2EE-compliant application servers provide this service and are bundled with the standard EJB API classes and interfaces.

6.1. What Version Is Covered Here?

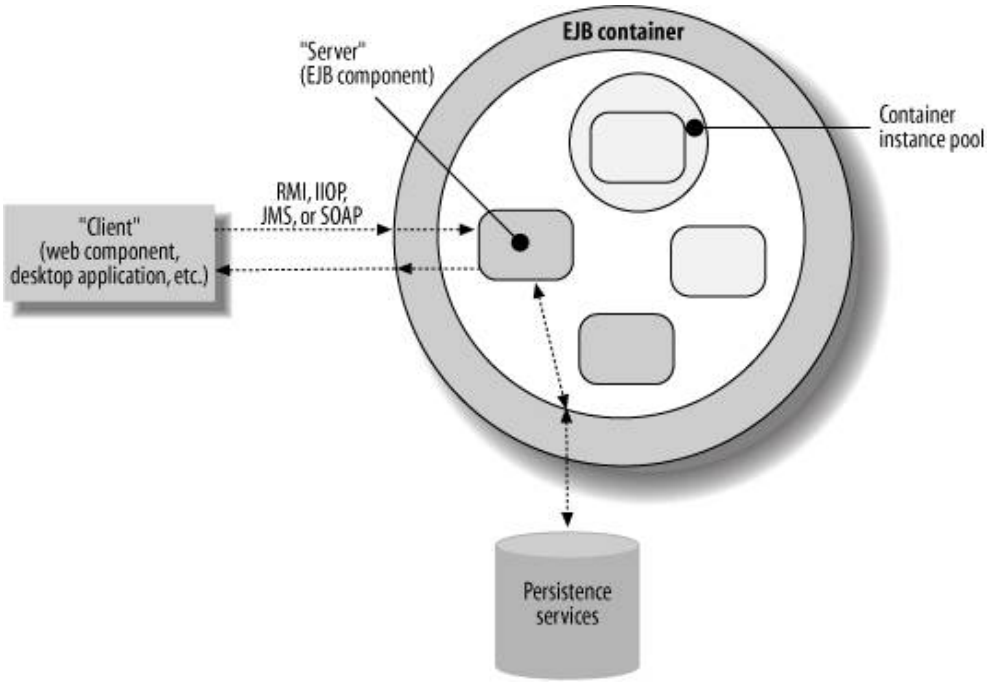
As discussed in the introduction to this book, the current versions of virtually all the prominent J2EE servers support the 1.4 version of the J2EE specification. This translates to supporting the EJB 2.1 specification, the version required by J2EE 1.4. At the time of this writing, however, draft versions of the EJB 3.0 specification have been distributed, and various vendors and open source projects have announced support for the new version. Since the EJB 3.0 specification is still in development and likely to change before it is finalized, we've restricted our coverage in the chapter to EJB 2.1. But at the end of the chapter, we do provide a short summary of the new programming model being planned for EJB 3.0.

The code examples in this chapter have been tested in the EJB container in JBoss 4.

6.2. EJB Component Model Overview

Chapter 13 discusses the two fundamental roles in the RMI environment: the client of the remote object and the object itself, which acts as a kind of server or service provider. These two roles exist in the EJB environment as well, but EJB adds a third role, called the container. Figure 6-1 shows a conceptual diagram of how the three EJB roles interact. The container is responsible for interceding between the client (any code that is invoking the EJB) and the EJB component itself. The container also provides all the extra services for an EJB object mentioned earlier: transaction processing, security, object persistence, and resource pooling. If you're familiar with CORBA, you can think of the EJB container as being roughly equivalent to the ORB in CORBA, with a few of the CORBA services thrown in as well. In EJB, however, the container is strictly a server-side entity. The client doesn't need its own container to use EJB objects, but an EJB object needs to have a container in order to be available for client use.

Figure 6-1. The basic roles in an EJB environment



6.2.1. The Enterprise JavaBeans Object

At the heart of an EJB component is the actual implementation object. The EJB implementation class is where the real value of the EJB liesthe business methods that clients want to invoke are implemented here.

The EJB object:

- Implements the business logic behind the operations in its client interfaces
- Interacts with the EJB container to make use of component services, like security and transaction management

6.2.1.1. Types of EJBs

The three fundamental types of Enterprise JavaBeans are session, entity, and message-driven. The key difference between these component models lies in how they are managed during their lifetime by the EJB container and what component services are available to them through the container. Before implementing an EJB component, it's important to understand these different types and decide which one is most appropriate for a particular situation.

Session bean

> Typically accessed by one client at a time and is nonpersistent. It lives for a specific period of time (a session) and then gets removed by the server.

Entity bean

> Represents a data entity in persistent storage (e.g., a database or filesystem). Entity EJBs that represent the same persistent entity can be accessed by multiple clients concurrently. The data that defines a particular entity can be persistent beyond a client session, beyond the life of a particular EJB component instance, and beyond the lifetime of the EJB server.

Message-driven beans

> Seen by clients as a JMS destination, but are managed by the EJB container similar to a session EJB. Clients send messages to the JMS destination, the messages are mapped to operations on the EJB component, and the results, if any, are mapped back into JMS messages that are accessible by the client through the JMS destination.

To illustrate the differences between session, entity, and message-driven beans , suppose you're building an online banking system using EJB components. An automated teller machine, which reports on account balances and executes deposits and withdrawals on specified accounts, could be implemented as a session bean. A single client uses the teller bean to perform services on bank accounts that are maintained in some separate persistent store (the bank's database). An EJB object that directly represents a bank account, however, should be an entity bean. Multiple clients can access the account to perform transactions, and the state of the account entity should be persistent across the lifetime of the online banking server. If we want to provide wireless access to the banking system from PDAs or mobile phones, we might have these devices communicate with the system using lightweight, asynchronous messaging, since their network connectivity is typically unstable and low in bandwidth. This may lead us to use message-driven EJBs, which can respond to these client messages but still take advantage of the lifecycle management capabilities of the EJB container.

6.2.1.2. EJB code artifacts

At a minimum, an EJB component consists of an implementation class that is deployed to the EJB container, plus a deployment descriptor that tells the EJB container how to manage the component. Depending on the specific type of EJB you are implementing, the EJB may also need to include one or more client interfaces and server-side support classes required for the EJB container to manage the component.

Specifically, a message-driven bean requires only an implementation class and a deployment descriptor. These beans are seen by clients as JMS destinations to which they send messages, so there's no need for client interfaces for these EJBs .

Session beans are created by clients using their home interface(s) and operated by clients using their client interfaces, so these interfaces (in addition to the implementation class and deployment descriptor) are required when implementing session EJBs.

Entity EJBs are the most complex to implement because they make the most use of the EJB container's services. In addition to the implementation class, home interface(s), client interface(s), and deployment descriptor, an entity bean may also require a primary key class. Entity beans also have a significant amount of additional information in their deployment descriptors, related to their persistence properties.

6.2.2. The EJB Client

An EJB client uses EJB objects to access data, perform tasks, and generally get things done. In the EJB environment, the first action a client performs is to find the home interface for the type of EJB object that it wants to use. This home interface is a kind of object factory, used to create new instances of the EJB type, look up existing instances (only when using entity EJB objects, discussed later), and delete EJB objects. This is a bit different from RMI, in which the client first has to get a direct handle to an existing RMI servant. In many RMI applications, this first RMI object is a kind of object factory that is used to create other RMI object references. So, in a sense, the use of home interfaces in EJB is just formalizing the role of factory objects in distributed component applications.

EJB home interfaces are located by clients using JNDI (described in Chapter 9), the same way that other J2EE resources like JDBC `DataSources` and JMS `ConnectionFactory` resources are accessed. An EJB server publishes the home interface for a particular EJB object under a particular name in a JNDI namespace. The EJB client needs to connect to the JNDI server and look up the EJB home interface under the appropriate name in order to start things off.

We'll see concrete examples of EJB clients a bit later in the chapter, but here is a summary of the fundamental steps an EJB client performs:

1. Gets a JNDI context from the EJB/J2EE server
2. Uses this context to look up a home interface for the bean you want to use
3. Uses this home interface to create (or find) a reference to an EJB
4. Calls methods on the bean

6.2.3. The EJB Container

Most readers need to be familiar with EJB containers only from the perspective of an EJB client or an EJB object. A J2EE application server provides an internal EJB container, along with other required aspects of the J2EE framework. EJB-enabled application servers, with their own EJB containers and deployment tools, are available from BEA, JBoss, IBM, Sun, and many others.

The EJB container supplies the value-added features that EJB provides over standard remote objects built using RMI or CORBA. The EJB container manages the lifecycle of your EJBs and all the details of transaction

processing, security, resource pooling, and data persistence. For components that require these services, the use of the EJB container reduces the burden on both client applications and EJB objects, allowing them to deal with just the business at hand.

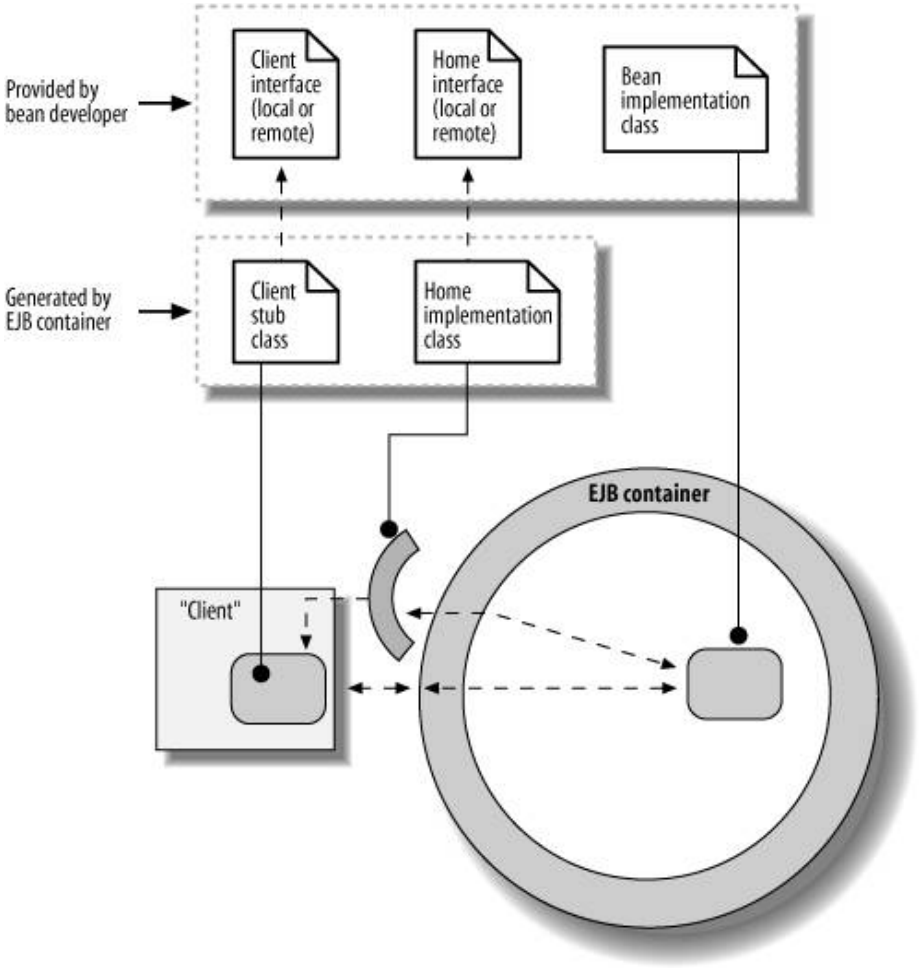
An EJB container is the heart of an EJB environment, in the same way that an ORB is the heart of a CORBA environment or a web server is the heart of a web environment. As depicted in Figure 6-1, all runtime interactions between the clients and the EJB objects themselves are mediated by the EJB container (although from the client's perspective, it seems as though they are interacting directly with the EJB component itself).

The EJB container takes all of the elements that make up your EJB (the implementation class, any client interfaces, etc.) and a deployment descriptor, and it uses these to generate the various support classes needed to manage and run the EJB component, as shown in Figure 6-2. You provide any needed home and client interfaces, and the container generates both the client and the server-side implementations for these interfaces. As the figure shows, EJBs can support both local and remote clients. When a remote client looks up a session or entity bean's remote home interface through JNDI, it receives an instance of a remote stub class. All methods invoked on this stub are remotely invoked, via RMI or IIOP, on the corresponding home implementation object on the EJB server. Similarly, if the client creates or finds any beans through the remote home stub, the client receives remote object stubs, and methods invoked on the stubs are passed through RMI or IIOP to corresponding implementation objects on the server. These remote objects are linked, through the EJB container, to a corresponding enterprise bean object, which is an instance of your bean implementation class. Local clients interact with EJBs in a simplified, more efficient way. Local home objects and bean interfaces are obtained the same way as remote ones, but the objects obtained are nonremote objects that interact locally (i.e., from within the same Java virtual machine) with their EJB implementations.

Note that Figure 6-2 demonstrates the situation for session and entity beans. Message-driven beans are accessed by clients using JMS messages sent to a JMS destination, and EJBs that are published as web services are invoked by clients using SOAP calls. So in both of these cases, clients do not need home or client interfaces. Details on JMS and SOAP clients can be found in Chapters 11 and 12, respectively.

It's important to remember that all client requests to create, look up, delete, or call methods on EJBs are mediated by the EJB container. It either handles them itself or passes the requests to corresponding methods on to the EJB object. Once the client obtains a reference to an interface for an EJB object, the container intercedes in all method calls to the bean to provide the bean with required services (transaction management, lifecycle management, and security) and to notify the bean of any events related to these services (for example, the bean needing to reload its persistent data or the bean instance being destroyed by the container).

Figure 6-2. Relationship of bean-provider classes and container-generated classes



When you deploy an EJB object within an EJB server, you use deployment descriptors to specify how the container should manage the bean during runtime, in terms of all the services the container provides. Deployment descriptors contain parameter settings for these various options. These settings can be customized for each deployment of an EJB object. You might purchase an EJB object from a vendor and deploy it on your EJB server with a particular set of container management options while someone else who purchased the same bean can deploy it with a different set of deployment options. We discuss the details of the runtime options available in deployment descriptors and how to use them later in this chapter when we talk about deploying EJB components.

6.3. EJB Tutorial

Now let's look at the generic steps that you take to implement an Enterprise JavaBeans component. In all of the examples that follow, we're showing you the "traditional" approach to writing EJBs: you create the client and home interfaces (where needed), the implementation classes, and the deployment descriptor by hand. A number of tools are available to simplify EJB development by creating these elements for you. Chapter 21 discusses XDoclet annotations that can be used to automatically generate interfaces and deployment descriptors for EJBs using special Javadoc comments in your code. At the end of this chapter we also give a preview of similar features being integrated into the EJB 3.0 specification. But even with current and upcoming development aids, it's important to understand the underlying EJB programming model so that you can apply the tools effectively.

As we discussed earlier, every type of EJB (session, entity, message-driven) must include a bean implementation

class. This class houses the bean's business logicit's the heart of the component. All the other trappings of the EJB either give clients ways to access the bean or help the container manage the bean.

If you are creating either an entity or a session bean, you also need to provide:

Home interfaces

> A home interface is used by clients to create or find EJB objects of a specific type. An EJB can have both local and remote home interfaces , though in most cases only one or the other is needed, based on the context in which the EJB is being deployed.

Client stub interfaces

> When a client creates or finds an EJB object through a home interface, the container provides the client with a reference to a stub that implements the client interface for the bean. The interface defines the methods the EJB object exports to clients. As with home interfaces, an EJB can have both local and remote bean interfaces.

The EJB object implementation must implement all the methods defined in its remote and local interfaces, provide methods that correspond to the methods on its home interface for creating or finding the bean, and also implement a set of required methods used by the EJB container to manage the bean.

Message-driven beans require only the bean implementation class. They don't have home interfaces or client interfaces since the client interface is implemented using JMS message passing (we'll give more details on creating and using message-driven beans later in the chapter).

To demonstrate the various components that make up an Enterprise JavaBeans object, we'll look at a simple example: a profile server. The profile server is a bean that provides profile information for named users. This profile information consists of name/value pairs that might represent preferences in an application, historical usage patterns, and the like. You might see a profile server running behind an online information service, personalizing the content and appearance of the site. After we've gone through this general example of writing a bean, we'll look more closely at the differences between implementing session and entity beans.

6.3.1. Client Interfaces

Although the bean implementation is central to the functions of the EJB component, it's typical to start putting together an EJB by defining its client interface(s). Client interfaces contain declarations of the methods that are available to clients.

6.3.1.1. Remote client interfaces

A remote client interface for our `ProfileManager` is shown in Example 6-1. A remote EJB client interface must extend the `javax.ejb.EJBObject` interface. `EJBObject` in turn extends the `java.rmi.Remote` interface, which makes the remote client interface an RMI remote interface as well.

Example 6-1. Remote interface for the profile manager bean

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

public interface ProfileManager extends EJBObject {
  public ProfileBean getProfile(String acctName)
  throws NoSuchPersonException, RemoteException;
}
```

The `ProfileManager` interface defines a single method, `getProfile( )`, that accepts an account name as its only argument. It returns a `ProfileBean` object containing the profile information for the person named. If the person's profile can't be found on the server, a `NoSuchPersonException` is thrown. This is an application-specific exception whose implementation isn't discussed in this chapter. Since the `ProfileManager` interface is an RMI remote interface, its methods must throw `RemoteException` in case some RMI communication problem occurs during a method call. Also, the arguments and return values for the methods have to be `Serializable`, or they need to be exportable RMI objects themselves. Our `getProfile( )` method returns a `ProfileBean` object, which we'll implement as a `Serializable` object, as shown below in Example 6-2. The `ProfileBean`, not shown here but included in the example source code download for the book, is a fairly straightforward Java bean: it simply has methods that allow you to get and set the profile entry values.

6.3.1.2. Local client interfaces

A local client interface for an EJB is defined in a similar way, but the rules and usage of the interface differ from those of its remote counterpart. As we've mentioned, local interfaces are used by clients that reside in the same JVM as the EJB itself, so they follow the same argument-passing and return-value rules as normal, nonremote Java classes: objects are passed by reference, and basic data types are passed by value. No remote operations are involved in the use of a local interface, so there's no need to ensure that method arguments and return values are `Remote` or `Serializable` object types. And methods in a local interface aren't required to throw `RemoteException`. Local interfaces extend the `EJBLocalObject` interface, which is a simplified, nonremote interface. A local interface for our profile server is shown in Example 6-2.

Example 6-2. Local interface for the profile server

```
import javax.ejb.*;

public interface ProfileManagerLocal extends EJBLocalObject {
  public ProfileBean getProfile(String acctName)
  throws NoSuchPersonException;
}
```

An EJB implementation class needs to provide implementations for all the methods exposed in its remote and local interfaces.

6.3.2. Home Interfaces

The client needs a way to create a reference to a profile server, so we have to provide a home interface for our bean. Since clients use home interfaces directly, and since clients can be either remote or local, an EJB can also have remote and local home interfaces.

Example 6-3 shows a remote home interface for our EJB. It provides a single `create( )` method that takes no arguments and returns the bean's remote interface type, `ProfileManager`.

Example 6-3. Remote home interface for the profile server bean

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

public interface ProfileManagerHome extends EJBHome {
  public ProfileManager create(  ) throws CreateException, RemoteException;
}
```

A remote home interface for an EJB object extends the `javax.ejb.EJBHome` interface. The remote home interface is also an RMI remote interface since `EJBHome` extends `java.rmi.Remote`. The home interface can contain

multiple `create( )` methods that take various initialization arguments to create the bean. A corresponding local home interface is shown in Example 6-4.

Example 6-4. Local home interface for the profile server bean

```
import javax.ejb.*;

public interface ProfileManagerLocalHome extends EJBLocalHome {
  public ProfileManagerLocal create(  ) throws CreateException;
}
```

For each `create( )` method on a home interface, the EJB object implementation must have a matching `ejbCreate( )` method that takes the same arguments. In either remote or local home interfaces, `create( )` methods are required to throw `javax.ejb.CreateException`, in case some error occurs during the EJB creation process. Create methods (and all other methods) on remote home interfaces must also throw `java.rmi.RemoteException` (or one of its parent exceptions) since the home interface is an RMI remote interface and some sort of network/communication error could occur between the client and the home implementation on the server. If the corresponding `ejbCreate( )` method on the bean implementation throws any other exceptions, the `create( )` method has to include these in its `throws` clause as well. As we'll see in our bean implementation in the next section, the bean's `ejbCreate( )` method doesn't throw any special exceptions, so we don't need to add any additional exceptions in the home interfaces.

Home interfaces for entity beans can also include finder methods, used to find previously created persistent entity beans. We'll discuss them when we talk about entity beans in detail.

6.3.3. The Bean Implementation

Now that we have a home interface that lets clients create EJB references and interfaces that describe what the EJB can do for the client, we need to actually implement the EJB object itself. Every bean implementation class can be divided into three sets of methods: business method implementations, container callback methods, and internal utility methods used by the bean class itself. A complete implementation class for the `ProfileManager` EJB can be found in the downloadable source code for the book; we'll just look at the highlights here in terms of these three categories of methods.

The business method implementations must match the business methods that are exposed in the EJB's client interfaces. For our `ProfileManagerBean`, we have only one business method in the client interfaces: `getProfile ( )`. Our implementation of this business method is simple enough:

```
public Profile getProfile(String name) throws NoSuchPersonException {
    // First validate the provided name, throw a NoSuchPersonException
    // if it's not valid
    if (!validateName(name)) {
      throw new NoSuchPersonException("No user found matching name \"" +
                                      name + "\"");
    }
    // If the name is valid, find/create the profile.  Here, we simply
    // create a new profile Java bean (e.g., no persistence is provided)
    Profile profile = new Profile(name);
    return profile;
}
```

We validate the name argument and create a new `Profile` JavaBean object for the named user. Note that this highly simplified implementation provides no persistence featuresevery time a username is given, a new, empty profile is created and returned. We'll see later in the tutorial how to integrate persistence into the bean implementation.

The `validateName( )` method is an internal utility method: it's private, and it's not included in any of the client interfaces, so it's not available for clients to invoke.

Every implementation class for any EJB object must implement the `javax.ejb.EnterpriseBean` interface. This is typically done indirectly, through the `SessionBean`, `EntityBean`, or `MessageDrivenBean` interface. These interfaces include the required callback/notification methods that the EJB container needs in order to effectively manage the lifecycle of the EJB at runtime. Our sample EJB implementation is a session bean, so the `ProfileManagerBean` class implements the `SessionBean` interface.

The implementation class must follow a few other rules. The class must be declared as `public` to allow the container to introspect the class when generating the classes that hook the bean to the container and to allow the container to invoke methods on the bean directly where necessary. The bean class doesn't implement the bean's remote or local interface. This may seem a bit strange at first, especially if you're familiar with remote object systems like RMI and CORBA, since the purpose of the bean is to provide a concrete implementation of the EJB's interfaces, and in these other contexts this is done through direct inheritance. But in an EJB context, the EJB container always mediates between a client method request and the actual call on the corresponding method on a bean instance. As long as the container is explicitly told the EJB's various interfaces and its implementation class (and we'll see how to do that with deployment descriptors in a later section), it has all the information it needs to make this connection. When the EJB server generates the classes that bridge the bean to the container, it also provides a class that implements the remote interface and acts as a proxy to the EJB class itself.

The EJB container callbacks actually make up the bulk of the implementation class. These methods serve as the hooks the EJB container uses to manage the bean as a component. They also implement some of the functionality provided to the client through the client home and stub interfaces. For example, since the home interfaces for this EJB include a no-argument `create( )` method, we must provide an equivalent `ejbCreate( )` method that will be invoked by the container on a bean instance when the `create( )` method is invoked by a client on a home interface. The `ejbCreate( )` method is a container callbackit gives the EJB implementation object a chance to initialize any resources it might need to operate properly. In our case, resources must be set up (because our implementation is so simple), but we still need to provide a method implementation for the container:

```
public void ejbCreate(  ) {
    System.out.println("ProfileManagerBean created.");
}
```

The other remaining methods on the implementation class, `ejbActivate( )`, `ejbRemove( )`, `ejbPassivate( )`, and `setSessionContext( )`, are also container callbacks, invoked by the container to manage the EJB instance during its lifetime.

All session and entity bean implementations have to provide the following container callback methods, which are used in the following ways:

public void ejbCreate(...)

> Called on a bean instance after a client invokes a corresponding `create( )` method on a bean's home interface. Create methods are required for session and message-driven beans and optional for entity beans. The arguments, if any, in the create method indicate the identity or starting state of the bean. Note that the return type of `ejbCreate( )` methods depends on the component model being used by the EJB. Specifics for each type of EJB model are given in the sections on session, entity, and message-driven beans.

public void ejbPassivate( )

> Called by the container just before a bean is to be serialized and stored in passive storage (e.g., disk,

database) on the server. This callback allows the bean to release any nonserializable resources (e.g., open files, network connections).

public void ejbActivate( )

Called by the container when the bean has been deserialized from passive storage on the server, and is becoming eligible to receive client method requests again. It allows the bean to reclaim any resources freed during passivation (e.g., file handles, network connections) or restore any other state not explicitly saved when the bean was serialized.

public void ejbRemove( )

Called by the container just before the bean is to be removed from the container and made eligible for garbage collection. Again, the exact semantics of this method depend on the component model of the EJB. "Removing" a session or a message-driven bean simply pulls the bean out of its active state before it is actually destroyed by the container. Removing an entity bean means to delete the bean's persistent state data from the underlying database.

In our example, the `ProfileManagerBean` doesn't need to perform any actions in these methods, so they are implemented as empty methods that simply print message to standard output, indicating that they have been called.

6.4. Deploying EJBs

As with web components (servlets and JavaServer Pages), EJB components in the J2EE environment are packaged into jar files, and the components are described and configured within an application server using deployment descriptors that are included in these jar files. These deployment descriptors are based on an XML DTD that is published as part of the EJB specification. An overview of J2EE deployment concepts can be found in Chapter 2. In this section, we discuss some EJB-specific deployment details that complement the material in Chapter 2.

Once you've written the home and remote interfaces and the implementation of your enterprise bean, you need to deploy your beans in an EJB container, which involves the following steps:

1. Specify the deployment information and options for your bean, in the form of an XML file called an EJB deployment descriptor. The information in this deployment descriptor includes the names of classes that serve as the client interfaces, the home interfaces, and implementation for your EJB as well as any transaction support options, access control settings, and so on. In addition to the basic deployment information, different types of EJBs (session, entity, message-passing) require different sets of additional metadata (e.g., data mappings for container-managed entity beans and session timeouts for session EJBs).

2. Provide any vendor-specific deployment information beyond the information in the standard J2EE deployment descriptor. Some EJB container implementations allow/require you to specify additional deployment parameters in a container-specific way, either in an additional configuration file for the EJB or through a server management console.

3. Generate the container-provided classes shown earlier in Figure 6-2.

4. Package your EJBs into an ejb-jar file.

As shown in Figure 6-2, the EJB container generates a set of classes that are needed to deploy your EJB object. It's up to the EJB container to provide a tool or tools for generating these classes. Some may be command-line tools that read a standard EJB deployment descriptor and generate the needed classes while others may be GUI tools that let you control the deployment options of your bean using a visual interface.

6.4.1. EJB Deployment Descriptors

All EJB deployment descriptors include a listing of the Java classes that serve as the home, client, and implementation classes for a particular EJB. They also include various metadata about the bean, such as a description, a display name for the bean (useful in application server management interfaces), and so on. Appendix A provides a complete reference for EJB deployment descriptors. We'll simply give a short overview here. It might be useful to refer back to the schema structure diagrams in "Enterprise JavaBeans (ejb-jar.xml)" in Appendix A as you read through this section.

The general structure for an EJB deployment descriptor is as follows:

```
<? xml version="1.0" encoding="UTF-8" ?>
<ejb-jar xmlns="http://java.sun.com/xml/ns/j2ee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
                         http://java.sun.com/xml/ns/j2ee/ejb-jar_2_1.xsd"    version="2.1">
  <!-- Metadata about the contents of the ejb-jar file ... -->
  <!-- Provide an enterprise-bean element for each EJB component
        contained in the jar -->
  <enterprise-beans>
    <!-- Describe a session bean -->
    <session>...</session>
    <!-- Describe an entity bean -->
    <entity>...</entity>
    <!-- Describe a message-driven bean -->
    <message-driven>...</message-driven>
  </enterprise-beans>
  <!-- Any additional metadata (inter-EJB relationships, assembly info,
       etc.) -->
  <relationships>...</relationships>
  <assembly-descriptor>...</assembly-descriptor>
  <ejb-client-jar>...</ejb-client-jar>
  ...
</ejb-jar>
```

The EJB descriptor starts with the usual XML preface. The top-level element in the deployment descriptor is an `<ejb-jar>` element, and it references the standard XML schema for EJB deployment descriptors, in this case referencing the schema from the java.sun.com server (see Chapter 7 for more details on XML schema and namespaces). The `<ejb-jar>` root element contains an `<enterprise-beans>` element that lists each EJB in the jar. Each entry within the `<enterprise-beans>` element contains the deployment information for a single EJB, including the various classes that make up the EJB's implementation, runtime management parameters, and so on.

A complete deployment descriptor that describes our `ProfileManager` EJB is shown in Example 6-5.

Example 6-5. Deployment descriptor for the profile server EJB

```
<? xml version="1.0" encoding="UTF-8" ?>

<ejb-jar xmlns="http://java.sun.com/xml/ns/j2ee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
                          http://java.sun.com/xml/ns/j2ee/ejb-jar_2_1.xsd"
         version="2.1">
    <!-- Description of contents -->
```

```
    <description>Introductory EJB Example:
                 Stateless ProfileManager Bean</description>
    <display-name>ProfileManager Bean Example (stateless)</display-name>
    <enterprise-beans>
        <!-- A stateless session profile server EJB -->
        <session>
            <display-name>ProfileManager Bean</display-name>
            <ejb-name>ProfileManagerBean</ejb-name>
            <!-- Remote home interface -->
            <home>
                com.oreilly.jent.ejb.stateless.ProfileManagerHome
            </home>
            <!-- Remote bean interface -->
            <remote>
                com.oreilly.jent.ejb.stateless.ProfileManager
            </remote>
            <!-- Local home interface -->
            <local-home>
                com.oreilly.jent.ejb.stateless.ProfileManagerLocalHome
            </local-home>
            <!-- Local bean interface -->
            <local>
                com.oreilly.jent.ejb.stateless.ProfileManagerLocal
            </local>
            <!-- Bean implementation class -->
            <ejb-class>
                com.oreilly.jent.ejb.stateless.ProfileManagerBean
            </ejb-class>
            <session-type>Stateless</session-type>
            <transaction-type>Container</transaction-type>
        </session>
    </enterprise-beans>
    <ejb-client-jar>ProfileManagerClient.jar</ejb-client-jar>
</ejb-jar>
```

We're using the `<description>` and `<display-name>` elements to provide some text descriptions of the contents within the top-level `<ejb-jar>` element. The information contained in these elements might be used by a management console on an application server or within an EJB-enabled IDE.

The `<enterprise-beans>` element contains one element for each EJB contained in an EJB jar. There are three possible subelements of `<enterprise-beans>`, one for each type of EJB: `<session>`, `<entity>`, and `<message-driven>`. Our `ProfileManager` EJB is a session bean, so we use the `<session>` element here. Within the `<session>` element, we provide a display name for the bean itself and an `<ejb-name>` element that provides a name that can be used to refer to this bean from other J2EE components. Then we specify all of the classes that make up our EJB: the remote home and client interfaces (`<home>` and `<remote>`), the local home and client interfaces (`<local-home>` and `<local>`), and the EJB implementation class itself (`<ejb-class>`).

At the end of the `<session>` element comes configuration information that is specific to session EJBs. In this case, we are using the `<session-type>` element to specify that this is a stateless session bean and the `<transaction-type>` element to indicate that we want transactions managed by the EJB container for this EJB. This information is used primarily by the EJB container to determine how the EJB should be managed at runtime. The information that is specified in this section of the deployment descriptor depends on the type of EJB being described (session, entity, message-driven). Examples of the kind of metadata required for the various types of EJBs are provided later in this chapter.

6.4.2. Security-Related Deployment Attributes

Security management is another important service provided by the EJB container for your components. Essentially, the EJB container needs to provide some means for mapping a client identity to a named user or role

defined on the EJB server itself. Then you, the bean provider, specify which users and/or roles can access each method on your bean. This approach to specifying security details keeps the access details out of the bean implementation code and allows the same bean to be deployed in different EJB containers, using different underlying security systems and access settings that suit the local context.

If you do need to resort to programmatic security checks in your EJB components, the `EJBContext` interface provides access to the user's identity and roles. Refer to "EJB Component Security" in Chapter 10 for further details on programmatic security in the EJB context.

6.4.2.1. Client security roles

The EJB specification allows you to specify security roles and associated access control levels in the EJB deployment descriptor. Suppose that we are deploying our EJB-based profile service and we want any client at all to be able to read profile entries, but we want only clients (individual users or client applications) identified as administrators to be able to set profile entries. To do this, we first have to define an appropriate security role in our EJB deployment descriptor. This is done in the `<assembly-descriptor>` element, which follows the `<enterprise-beans>` and `<relationships>` elements in the root `<ejb-jar>` element. Within the `<assembly-descriptor>`, you define a security role using the `<security-role>` element:

```
...
<assembly-descriptor>
    ...
    <security-role>
        <description>
          Profile administrators, allowed to update entries
        </description>
        <role-name>profile-admin</role-name>
    </security-role>
    ...
</assembly-descriptor>
...
```

In order for these security roles to actually mean anything, they need to be assigned to concrete principals and/or groups defined within the security realms of whatever J2EE application server you are using to deploy your EJBs. The specific details of how this is done depends on the particular vendor you are using. In BEA WebLogic, you would use the WebLogic administration interface to create new realms that refer to existing LDAP, RDBMS, native OS, and/or custom authentication services and then map your security roles to principals defined in these realms. Other application servers will provide other schemes for authenticating users and tying their identities into the EJB context.

6.4.2.2. Method permissions

So far we've declared some application-specific security roles, but we haven't given these roles any rights to invoke methods on our EJBs. To do this, we have to add one or more `<method-permission>` elements to the `<assembly-descriptor>`. A `<method-permission>` element gives a particular role rights to access one or more methods on an EJB.

Suppose we defined a `Profile` EJB, with an implementation class called `BMPProfileBean`, representing persistent data about a profile. This is an entity EJB version of the serializable `Profile` bean that we used in our stateless session example earlier (this EJB will be shown in detail in later sections). We might have a `setEntry ( )` method on this EJB that's used to alter entry values on the `Profile`, and we might want to give only the `profile-admin` role the rights to execute the `setEntry( )` method on our `Profile` bean. If so, we would use a `<method-permission>` element such as:

```
...
<assembly-descriptor>
```

```
    ...
    <method-permission>
        <description>
          Only allow profile administrators to set entries on profiles
        </description>
        <role-name>profile-admin</role-name>
        <method>
            <ejb-name>BMPProfileBean</ejb-name>
            <method-name>setEntry</method-name>
        </method>
    </method-permission>
    ...
</assembly-descriptor>
...
```

This allows any user who has the `profile-admin` role to invoke the `setEntry( )` method on the `Profile`, while all other client methods on the bean are accessible to everyone. The `<role-name>` indicates which role should have access to this method, and the `method` element(s) indicates the EJB and method that this role can access. Here, we have a single method specified: any method named `setEntry` on the `BMPProfileBean` EJB defined in this deployment descriptor. If `setEntry( )` is overloaded on our bean, we can be more specific by including the method parameters of the single method we want made accessible:

```
...
<assembly-descriptor>
    ...
    <method-permission>
        <description>
          Only allow profile administrators to set entries on profiles
        </description>
        <role-name>profile-admin</role-name>
        <method>
            <ejb-name>BMPProfileBean</ejb-name>
            <method-name>setEntry</method-name>
            <method-params>
                <method-param>java.lang.String</method-param>
                <method-param>java.lang.String</method-param>
            </method-params>
        </method>
    </method-permission>
    ...
</assembly-descriptor>
...
```

You can also give access to all methods on an EJB by using an asterisk (`*`) as the value of the `<method-name>` element. By default, all methods on all EJBs are accessible to everyone. Any methods not specified in a `<method-permission>` element in the deployment descriptor maintain this global accessibility.

6.4.2.3. Propagating identities

It's also possible in EJB to specify how identities are propagated from your EJB to other EJBs and resources, by indicating which role your EJB will assume when its methods are executed. The identity used to execute the EJB methods will be passed forward when the EJB makes calls to other EJBs or accesses other J2EE resources. This identity will be used to determine access rights to these resources in the same way that the caller's credentials are used to control access to the EJB itself and so on down the call chain.

To specify the runtime identity used to execute your EJB methods, use the `<security-identity>` element in the `<session>`, `<entity>`, and `<message-driven>` sections of the deployment descriptor. The `<security-identity>` element allows you to specify either that the methods on the bean will be invoked using the identity

of the calling client or that a specific named security role will be used to invoke the bean methods.

The first option simply passes on the identity of the calling client when an EJB method invokes other EJBs or resources. To specify this option, include the following in the EJB's deployment descriptor section (a session bean is used as the example here):

```
...
<enterprise-beans>
    ...
    <session>
        ...
        <security-identity>
            <use-caller-identity />
        </security-identity>
        ...
    </session>
    ...
</enterprise-beans>
...
```

This scheme is most appropriate when the client operates in a context in which it can be authenticated against a shared authority (e.g., a servlet is the user entry point, and the user is challenged with an HTTP authentication prompt), and the user's identity can be propagated to the EJB container. Note that the `<use-caller-identity>` option can't be used with message-driven beans since there is no path for a user's credentials to be propagated to the EJB container via message destinations. A message-driven bean has to assume its own role when handling messages.

The other alternative is to specify a role under which all invocations of your EJB will be executed. For this, use the `<run-as>` option in the `<security-identity>`:

```
...
<enterprise-beans>
    ...
    <session>
        ...
        <security-identity>
            <run-as>
                <role-name>profile-admin</role-name>
            </run-as>
        </security-identity>
        ...
    </session>
    ...
</enterprise-beans>
...
```

Here, we're specifying that all method invocations on our EJB will run under the `profile-admin` role and will assume that role's access rights when invoking other EJBs and resources.

6.4.3. Packaging EJBs

An ejb-jar file is the standard packaging format for Enterprise JavaBeans. It is a normal Java archive (jar) file, created using the standard jar utility, but it contains specific files that provide all the information needed for an EJB container to deploy the beans that are contained in the jar file. An ejb-jar file can contain one or more EJBs.

An ejb-jar file contains two types of files:

- The class files for each bean, including their home and client interfaces, and the bean implementations. An ejb-jar file can also include container-generated classes (concrete implementations of home and client interfaces, for example).

- Deployment descriptor files, as described in the previous sections. At a minimum, an ejb-jar file must include a standard ejb-jar.xml file. Different EJB container vendors may also require additional deployment descriptor files that allow you to control vendor-specific aspects of how your EJBs will be managed at runtime.

An ejb-jar file is a compact, efficient way to package a set of EJBs for deployment in an EJB container; it contains all the information an EJB container requires to manage your EJBs.

The compiled Java classes that make up your EJBs are included in the ejb-jar file in package-specific directories, just as in a regular jar file. The deployment descriptor that describes and configures the EJBs must be included in the ejb-jar file as META-INF/ejb-jar.xml.

Some EJB container vendors include a utility to facilitate the creation of ejb-jar files from your bean classes. Ant also includes a task, `ejbjar`, that can be used to create ejb-jar files from your Ant buildfiles (see Chapter 17 for more details). It's a simple matter, however, to create an ejb-jar using the jar utility provided with the JDK. Assuming that you have created a valid ejb-jar.xml deployment descriptor file, such as the one shown earlier, simply compile all of your relevant classes into a given directory, create a META-INF directory in this same area, and put the ejb-jar.xml file in it, then use jar to create your ejb-jar file:

```
% jar cvf ProfileManager.jar com/oreilly/jent/ejb/*.class META-INF
added manifest
adding: com/oreilly/jent/ejb/ProfileManager.class(in = 344) (out= 225)(deflated 34%)
adding: com/oreilly/jent/ejb/ProfileManagerBean.class(in = 1254) (out= 598)(deflated 52%)
adding: com/oreilly/jent/ejb/ProfileManagerHome.class(in = 317) (out= 208)(deflated 34%)
adding: com/oreilly/jent/ejb/ProfileManagerLocal.class(in = 327) (out= 214)(deflated 34%)
adding: com/oreilly/jent/ejb/ProfileManagerLocalHome.class(in = 305) (out= 198)
(deflated 35%)
ignoring entry META-INF/
adding: META-INF/ejb-jar.xml(in = 0) (out= 0)(stored 0%)
```

This command creates an ejb-jar file named ProfileManager.jar in the current directory. Note that this ejb-jar file doesn't contain any container-generated classes; these need to be created using tools provided by the EJB vendor, as described in the next section.

6.4.3.1. Generating the container classes

So far we've seen the basic elements that make up an EJB (home interface(s), client interface(s), and a bean implementation class), how the configuration parameters for one or more beans are specified in an XML deployment descriptor, and how all of these can be packaged into a standard ejb-jar file. In order for an EJB container to deploy your EJBs, it still needs to generate the container-specific classes depicted in Figure 6-2. How this is done depends on the EJB container you are using. If you are using BEA's WebLogic application server, for example, you can use the appc tool that is provided with the server to generate the runtime support classes needed by WebLogic. If we've packaged our EJB classes and deployment descriptor into an ejb-jar file named ProfileManager.jar, we can use the appc tool like so (for a Unix environment):

```
> java -classpath $WL_HOME/lib/weblogic.jar:$CLASSPATH weblogic.appc -output
wl-ProfileManager.jar ProfileManager.jar
```

If the EJB(s) contained in the specified ejb-jar file check out (the interfaces, implementation classes, and deployment descriptor meet the EJB specification requirements), this will generate a new ejb-jar file, wl-ProfileManager.jar, that contains all the classes needed to run your EJB(s) within the WebLogic server. This ejb-

jar file can be directly deployed to the server.

Each EJB container will have its own scheme for generating support classes for your EJBs. Consult your EJB vendor's documentation for specific details.

6.4.3.2. Delivering the package

Finally, after writing, configuring, and packaging one or more EJB components, you need to actually deliver the EJBs to the server running the EJB container. Again, this procedure is not specified by the EJB or J2EE specificationseach server implementation is free to do this in whatever way is most appropriate for it. In some cases, EJB components are deployed by simply placing the ejb-jar file in a specific directory on the server. In other cases, a server management tool (web-based or otherwise) needs to be used to upload the ejb-jar file to the server to deploy it.

6.5. Using Enterprise JavaBeans

So far, we've seen the high-level details on implementing and deploying an EJB. Now let's look at how you use an enterprise bean as a client.

It's important to note that a "client" for a given EJB can be many things. In a fairly typical scenario, a client may be a Java servlet or JSP, running in the same or different J2EE server as the EJBs being used. A client can also be a standalone GUI client (in J2EE parlance, an "application client") that directly connects to a local or remote EJB container and makes requests of beans. A client can also be another EJB, invoking an EJB on the same or different EJB server in order to satisfy a client request. Regardless of which of these scenarios is the case, the following details apply.

6.5.1. Local Versus Remote Clients

A local client (i.e., a client that resides in the same Java virtual machine as the EJB container) can interact with EJBs using either the local or remote interfaces for the EJB. If the EJB has a local home and client interface, this provides an efficient way for the client to make requests, since their method calls don't need to be marshaled into a remote method call and transmitted over an RMI or IIOP connection to the EJB container. If the EJB doesn't have a local client and home interface, or if the local interfaces don't expose the functionality that the client needs (remember that local and remote interfaces for the same EJB don't need to be equivalent to each other), a local client must use its remote home and client interface. Local EJB interfaces were introduced in EJB 2.0, so if you are using EJB 1.1, your EJBs can have only remote interfaces.

Remote clients (i.e., clients that run in a different Java virtual machine than the EJB container) must use the EJB's remote client and home interfaces . Remote interfaces are RMI client stubs (see Chapter 13 for full details on Java RMI), and the underlying remote method protocol being used can be RMI/JRMP (the "native" RMI protocol) or RMI/IIOP (the CORBA protocol). In order for a client to maintain portability across different EJB implementations, it's important that remote references obtained from the EJB container be cast to their interface types using the `narrow( )` method on `javax.rmi.PortableRemoteObject`. This ensures that RMI/IIOP remote references will be safely cast to their expected type.

6.5.2. Finding Home Interfaces Through JNDI

Once an enterprise bean is deployed within an EJB container, the home interface(s) for the bean have been exported under a particular name using JNDI according to the settings in the deployment descriptor. As a client, you need to know how to connect to the JNDI context of the EJB/J2EE server and know the name for the bean home interface you're interested in. Full details on using JNDI and the options for creating `InitialContexts` can be found in Chapter 9, but in general you need to set connection properties required for the JNDI provider you're

connecting to and then create an `InitialContext` object that points to the root of the naming system:

```
Hashtable props = new Hashtable(  );
// Specify the necessary properties
...
// Try looking up the context
javax.naming.Context ctx = null;
try {
  ctx = new javax.naming.InitialContext(props);
}
catch (NamingException ne) {
  System.out.println("Failed to connect to JNDI service for EJB server");
}
```

If you're running within a J2EE environment (e.g., the EJB client is actually a web component running within the same J2EE server), you typically won't need to specify any connection properties. The J2EE server is configured so that the default JNDI connection properties point to its internal JNDI service, and you can use the no-argument form of the `InitialContext` constructor:

```
ctx = new javax.naming.InitialContext(  );
```

Now that we have a JNDI naming context from the EJB server, we can look up the home interface for the bean we're interested in. Assuming that we deployed our bean and specified to the EJB container that its JNDI name should be `ProfileManagerHome`, we can obtain a reference to a home interface to the bean with code such as the following:

```
ProfileManagerHome pHome = null;
try {
  Object pRef = ctx.lookup("ProfileManagerHome");
  pHome =
    (ProfileManagerHome)PortableRemoteObject.narrow(pRef,
                                            ProfileManagerHome.class);
}
catch (NamingException ne) {
  System.out.println("Failed to lookup home for ProfileManager bean.");
}
```

Notice that we're using the `javax.rmi.PortableRemoteObject.narrow( )` method to safely cast the remote reference obtained from the EJB container to the expected home interface type. This allows our client to work safely with EJB servers that provide RMI/IIOP remote references to their beans.

6.5.3. Creating and Finding Beans

As we saw in the previous sections, home interfaces for an EJB contain methods that allow a client to create new beans or find existing beans (for entity beans). Continuing our example client, assuming we're using our `ProfileManager` bean and its corresponding home interface shown in Example 6-3, we can create a reference to a `ProfileManager` bean as follows:

```
ProfileManager pServer = null;
try {
  pServer = pHome.create(  );
}
catch (RemoteException re) {
  System.out.println("Remote exception while creating
 ProfileManager: " +
                     re.getMessage(  ));
```

```
}
catch (CreateException ce) {
  System.out.println("Error occurred during creation:
                     " + ce.getMessage(  ));
}
```

We're using the `create( )` method defined on our `ProfileManagerHome` interface. Since this is a remote home interface, we need to catch both `RemoteException` and `CreateException` in our client code, in case some error occurs while communicating with the remote EJB container or during the create process itself. Assuming all goes well, we can now use our `ProfileManager` bean, invoking its business methods as needed by our client.

In the sections that follow, we'll examine client usage of each type of EJB (session, entity, and message-driven) in more detail, after we discuss the implementation and deployment details of each EJB component type.

6.5.4. EJB Handles and Home Handles

Every bean's remote client interface (if it has one) extends the `EJBObject` interface. Among other things, this interface allows the client to obtain a serializable handle on the remote enterprise bean. This handle is a persistent reference to the bean that can be serialized and then stored in local storage on the client side or emailed as an attachment to other users, for example. Later, a client can deserialize the handle object and continue interacting with the bean it references. The handle contains all of the information needed to reestablish a remote reference to the enterprise bean it represents. Since this is useful only for beans that are still valid when the handle is reconstituted, it is usually applicable only to entity beans, since they are persistent. Handles to session beans can also be used this way, as long as the handle is serialized and deserialized during the lifetime of the session bean (e.g., the EJB container doesn't shut down in the meantime or the session bean is not removed from memory by the container).

You can get the handle for a bean using the `getHandle( )` method on a remote bean object reference:

```
ProfileManager profileServ = ...;
Handle pHandle = pServ.getHandle(  );
```

The `getHandle( )` method returns a `javax.ejb.Handle` object. Typically, the `Handle` implementation for a particular EJB is provided by the EJB container and is generated automatically from the EJB remote client interface and bean implementation class that you provide. Every `Handle` implementation is a `Serializable` object, which allows it to be transmitted to remote clients and to be stored in serialized format, if needed:

```
ObjectOutputStream oout = ...;
oout.writeObject(pHandle);
```

Later, you can read the object back from its serialized state and obtain a reference to the same remote EJB, using the `getEJBObject( )` method on the handle:

```
ObjectInputStream oin = ...;
Handle pHandleIn = (Handle)oin.readObject(  );
EJBObject pRef = pHandleIn.getEJBObject(  );
ProfileManager pIn =
  (ProfileManager)PortableRemoteObject.narrow(pRef, ProfileManager.class);
ProfileBean prof = pIn.getProfile("JohnSmith");
```

When we call `getEJBObject( )` on the `Handle`, the `Handle` communicates with the source EJB container and attempts to construct a remote reference to the bean that the handle came from originally. As with other situations in which we receive a remote reference from the EJB container, we use the `PortableRemoteObject.narrow( )`

method to safely cast the remote EJB interface to the expected type, and then we can use the EJB.

It's also possible to get a handle for a remote home EJB interface. The same basic sequence of events can be followed for obtaining and serializing a home handle: obtain the handle by calling the `getHomeHandle( )` method on a home stub instance, use the appropriate serialization technique (instantiate an `ObjectOutputStream`, store the handle in a binary field in a database, and so on) to store the handle:

```
ProfileManagerHome pHome = ...
HomeHandle pHomeHandle = pHome.getHomeHandle(  );
ObjectOutputStream oout = ...;
oout.writeObject(pHomeHandle);
```

We can reconstitute the serialized `HomeHandle` later and use its `getEJBHome( )` method to get a new remote home reference for the same EJB:

```
ObjectInputStream oin = ...;
HomeHandle pHomeHandleIn = (HomeHandle)oin.readObject(  );
EJBHome pHomeRef = pHomeHandleIn.getEJBHome(  );
ProfileManagerHome profileHomeIn =
  (ProfileManagerHome)PortableRemoteObject.narrow(
                          pHomeRef, ProfileHome.class);
ProfileManager pServer = profileHomeIn.create(  );
```

The EJB specification doesn't dictate the usable lifetime of a home handle. It may become invalid after the source EJB container restarts, it may expire after some timeout period, or it may remain valid for an extended period of time. Consult your EJB vendor documentation for specific details.

6.6. Session Bean Specifics

Now that we've seen all the basic elements of implementing, deploying, and using an EJB, let's look a bit at the specifics of implementing session beans.

Beans that simply provide a set of well-defined functionality to clients are best-suited to be session beans. A simplified view of session beans is to think of them as regular remotely callable objects (like RMI objects) that are managed by an EJB container and therefore have access to all the basic EJB services (such as lifecycle management and security). Another way to decide whether an EJB should be a session bean is to remember what a session bean isn't: session beans are not meant to represent persistent data entities. A client acquires a reference to a session bean and asks it to perform functional services by calling methods on the bean. If the chunk of functionality you want to package for clients fits this description, the session bean model is probably the right one.

A session bean gets its name from the fact that it doesn't live beyond the lifetime of its serverin other words, it is not persistent. If an EJB client has a reference to a session bean and the host EJB server restarts, that session bean reference is no longer valid. You can reacquire a session bean of the same type from the same server after the restart, but it's not guaranteed to be in the same state as the bean you had before the server restart.

In terms of implementing session beans, the following sections cover specific requirements for session beans in addition to the general EJB requirements spelled out earlier.

6.6.1. Session Bean Implementation Classes

Session bean implementation classes must follow these rules:

- Session bean classes must implement the `javax.ejb.SessionBean` interface.
- The implementation class also must implement the `setSessionContext( )` method specified in the `SessionBean` interface. The container calls this method on the session bean just after the bean has been created, passing in a `javax.ejb.SessionContext` object that represents the runtime context for the bean. This session context is valid for the life of the bean.

6.6.2. Stateless Versus Stateful Session Beans

Session beans can be stateful or stateless . A stateless session bean doesn't maintain client state across method calls. If a client makes a series of method calls or transactions with the stateless bean, the bean should be in the same effective state at the start of each method call or transaction. Our `ProfileManagerBean` is implemented as a stateless session beanthe key signal of this is the use of the `Stateless` value for the `session-type` parameter in its deployment descriptor in Example 6-5. The stateless session model makes sense for the `ProfileManager`: the component doesn't need to keep any client-specific state to perform its function. It just locates a profile for a named user and returns it, and then it's out of the picture.

Since stateless beans don't have any client state, they are entirely interchangeableany instance of a stateless bean of a given type can be used to respond to a client request. And that's exactly how an EJB container manages stateless beansit keeps a pool of precreated beans on the server, and when a client request comes in, the container picks an instance at random, effectively, and invokes the required method on it. Stateless beans also don't need to be passivated since they have no state that needs to be restored when they're reactivated. The container simply destroys any stateless session beans it feels are no longer needed.

A stateful session bean, on the other hand, does maintain client-specific state that can be accessed and changed directly by a client. So a client that creates a stateful session bean owns it for the life of the bean, and no other clients ever get access to it. The bean stays active in the container until the client removes it or until the client's reference to the bean expires (e.g., the client dies and the lease on the remote reference expires), whichever comes first. Again, the container won't associate that particular bean instance with any other clientthe client that created it owns it for its lifetime.

To illustrate the difference between stateless and stateful session beans, let's take our stateless `ProfileManagerBean` and convert it to a stateful session bean. The `ProfileManagerBean` is stateless because all it does is accept requests for user profiles and return the profiles directly to the client as `Profile` objects, which are `Serializable` JavaBean objects. The client then interacts with the `Profile` directly, and the `Profile` holds the conversational state for the client, in the form of the values of the profile entries.

We can provide the same functionality by merging the stateless session `ProfileManager` EJB and the `Profile` Java bean into a single stateful session `Profile` EJB. Instead of creating a reference to a `ProfileManager` and calling its `getProfile( )` business method to get a Java bean, the clients create a stateful `Profile` EJB directly using its home interface.

Example 6-6 shows the remote interface for the `Profile` EJB (we won't implement a local interface for this example). It's similar to the interface for the `Profile` Java bean we used in the stateless `ProfileManager` example. It has `setEntry( )` and `getEntry( )` methods that access entries using their names. The `Profile` EJB also has accessors for the name of its user.

Example 6-6. Remote client interface for the Profile EJB

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

public interface Profile extends EJBObject {
```

```
  // Get the name of the user associated with this profile
  public String getName(  ) throws RemoteException;
  // Look up an entry by name
  public String getEntry(String name) throws RemoteException;
  // Set an entry value
  public void setEntry(String name, String value) throws RemoteException;
}
```

We'll need a home interface that clients can use to create these profiles. It's shown in Example 6-7we provide only a single `create( )` method, taking the name of the user who owns the profile. Notice that the `create( )` method throws the exceptions required by the EJB standard (`RemoteException`, because this is a remote home interface, and `CreateException`), but it also throws an application-specific exception, `NoSuchPersonException`. We defined this exception for this application (it indicates that the named user could not be found on the server). We can add any application-specific exceptions that we want to `create( )` methods on home interfaces, as long as we also include them in the signature of the corresponding `ejbCreate( )` methods on the bean implementation class.

Example 6-7. Remote home interface for the Profile EJB

```
import java.rmi.RemoteException;

import javax.ejb.CreateException;
import javax.ejb.EJBHome;

public interface ProfileHome extends EJBHome {
  // Create a profile for a named person. Throws an exception if the person's
  // profile can't be found.
  public Profile create(String name)
    throws RemoteException, CreateException, NoSuchPersonException;
}
```

Creating a stateful session implementation of this `Profile` EJB is much the same as creating the implementation of the `ProfileManager` EJB. We need to provide all the same container callbacks because they are both session EJBs, we need to implement `ejbCreate( )` methods for any `create( )` methods on the home interface(s), and we need to implement the business methods exposed in the client interface(s). We won't show all the container callbacks here because they're much the same as the earlier `ProfileManager` example, and the full details can be found in the source code bundle for the book.[*] Our home interface in Example 6-7 has a single create method, so we need a corresponding `ejbCreate( )` method:

```
// Create method with name of profile owner
public void ejbCreate(String name) throws NoSuchPersonException {
  mName = name;
  System.out.println("Profile EJB created for " + mName + ".");
}
```

The method implementation simply stores the username into an instance variable on the bean and prints a diagnostic message.

We also need to provide implementations of the business methods exposed in the client interface in Example 6-6:

```
  // Get the name associated with this profile
  public String getName(  ) {
```

```
    return mName;
  }

  // Get the value for a particular property in the profile
  public String getEntry(String key) {
    return mEntries.getProperty(key);
  }

  // Change a value in the profile
  public void setEntry(String key, String value) {
    mEntries.put(key, value);
  }
```

Deploying this bean is very much like deploying our `ProfileManager` example. We need to put its details into the deployment descriptor as a `<session>` element within the `<enterprise-beans>` element of the ejb-jar.xml file. The key difference here is that we need to specify a value of `Stateful` for the `<session-type>` element, to instruct the container that it should manage this EJB using the stateful session bean model (e.g., clients own the beans they create, no other client gets access to an instance created by another client, and so on). We then package the ejb-jar file and perform the rest of the deployment steps as required by our particular EJB container.

Clients use the `Profile` bean much the same way as the `ProfileManager`. They first look up its home interface using a JNDI call to the EJB container, then create a `Profile` EJB using the home stub's `create( )` method. This gives them a client stub that they can use to set and get profile entries. An example client use scenario is shown here:

```
// Get the Profile bean's home interface
ProfileHome pHome = (ProfileHome) context.lookup("ejb/ProfileHome");
// Create a profile for a person
System.out.println("Creating profile for " + name);
Profile profile = pHome.create(name);
// Get/set some entries in the profile
System.out.println("Setting profile entries for " + name);
profile.setEntry("favoriteColor", "blue");
profile.setEntry("language", "German");
System.out.println("Getting profile entries for " + name);
System.out.println("\tFavorite color: " +
                   profile.getEntry("favoriteColor"));
System.out.println("\tLanguage: " + profile.getEntry("language"));
```

This example assumes that we've deployed the `Profile` bean's home interface under the name *ejb/ProfileHome* in the EJB container's JNDI context.

6.6.3. Container Management of Session Beans

Since stateless beans can be used by any client, the container pools stateless beans and doles them out to clients as needed. If new stateless beans are required, the container creates them; when they aren't (e.g., the rate of client requests decreases), they are simply destroyed. To allow the container to fill its pool, any stateless session bean must provide a single `ejbCreate( )` method with no arguments. The container can call this whenever it decides another bean is needed in its ready pool.

Stateful beans are tied to the clients that create them, and they do maintain conversational state with their clients, so the container necessarily manages them differently. Stateful beans are created on demand when the container receives client requests for them. The container may still keep a pool of preconstructed instances of a stateful bean, but they aren't ready to accept method requests until a client calls one of the `create( )` methods on the home interface, and the container subsequently calls the corresponding `ejbCreate( )` method on one of the beans. At this point, the bean is "owned" by the client, and it's not removed until the client explicitly removes it

by calling `remove( )` on the client interface for the bean or until the client's reference to the bean goes away (the client quits or the bean reference goes out of scope on the client).

Stateful session beans can also be passivated and activated by the EJB container. The container may decide to do this in order to free up runtime resources on the server, for example. Passivating a session bean involves serializing its state and saving it to disk or some other storage area. When the container decides to passivate a stateful session bean, it invokes its `ejbPassivate( )` method. The bean can do any cleanup of resources that it deems appropriate before it is serialized (close I/O streams or database connections, release file handles, and so forth). When the container decides to reactivate a passivated session bean, it first deserializes the bean, then invokes the bean's `ejbActivate( )` method to give the bean a chance to restore any resources that it may need during runtime.

Stateless session beans aren't passivated or activated by the container since they aren't designed to maintain client state data. Instead, when the container decides a stateless session bean instance is no longer needed, it simply removes it entirely, and all of the stateless session bean's internal state is lost.

6.6.4. Optional Transaction Support

Stateful session beans can optionally receive notification of transaction boundaries from the EJB container. The container can notify the bean when a new client transaction is beginning and when the client transaction has either been completed or rolled back. For more information on EJB transaction management, see "Message-Driven Beans" later in this chapter.

Since session beans don't typically represent persistent shared data and stateful session beans can be accessed only by a single client at a time, user transaction boundaries may not be important in general to session beans. If, however, the session bean is managing database data for the user, it may want to know about the beginning and ending of user transactions, so that it can cache data at the start and commit its database updates at the end. For this reason, the EJB specification allows stateful session beans to optionally implement the `javax.ejb.SessionSynchronization` interface. By implementing this interface, the session bean indicates that it wants the container to notify it about the beginning and end of transactions.

In this case, the bean must implement the three methods that are declared on the `SessionSynchronization` interface: `afterBegin( )`, `beforeCompletion( )`, and `afterCompletion( )`. The container calls the bean's `afterBegin( )` method just after a new transaction begins. This lets the bean allocate any resources it might need during the transaction and cache database data, for example. Just before the transaction completes, the container calls the bean's `beforeCompletion( )` method. In this method, the bean can release any resources or cached data it may have initialized during the transaction. The `afterCompletion( )` method is called just after the transaction has completed. The container passes in a `boolean` value that is `true` if the transaction was committed and `false` if the transaction was rolled back. The bean can use this notification to deal with rollbacks, for example, allowing the bean to undo any changes made during the transaction.

6.7. Entity Beans

An entity bean represents data that is stored in a database or some other persistent storage. Entity beans are persistent across client sessions and the lifetime of the server. No matter when or where you get a reference to an entity bean with a given identity, the bean should reflect the current state of the persistent data it represents. Multiple clients can access entity beans with the same identity at the same time. The EJB container manages these concurrent transactions for the underlying entity, ensuring that client transactions are properly isolated from each other, consistent, and persistent.

Returning to our running example of a profile service, we started with a session EJB, `ProfileManager`, that simply created profiles (represented as Java beans) for the client. Then we changed the design into a single

`Profile` EJB, implemented as a stateful session bean. The major drawback in our stateful session `Profile` is that its profile data isn't persistent. A profile is created by a client and updated through client method calls, but once the `Profile` reference is given up by the client, or if the server crashes or shuts down for some reason, the accumulated profile data is lost. What we really want is a bean whose state is stored in a relational database or some other persistent storage that can be reloaded at a later time when the user reenters a profiled application. An entity EJB provides this functionality, and we can make our profiles persistent by implementing the `Profile` bean as an entity EJB. We'll use an entity bean implementation of the `Profile` EJB to demonstrate the features of entity EJBs.

At a basic level, entity beans are implemented similarly to session beans. You need to provide a local and/or remote home interface, a local and/or remote client interface, and a bean implementation. An entity bean, however, requires some additional methods in its home interface and bean implementation, to support the management of its persistent state and to allow clients to look up an entity in persistent storage. Entity beans must also provide a class that serves as its primary key, a unique identifier for entities.

The persistent data associated with an entity bean can be managed in two ways by the EJB containerusing bean-managed persistence (BMP) or container-managed persistence (CMP). With CMP, you leave the database calls to the container, and instead you tell the container (in the EJB's deployment descriptor) about bean properties and references that need to be persisted. The deployment tools provided with the EJB server are responsible for generating the corresponding database calls in the classes it uses to implement and deploy your bean. With BMP, you provide the actual database calls for managing your bean's persistent storage as part of your bean implementation.

If you can rely on the EJB container to handle your entity bean's persistence, this can be a huge benefit since it saves you from having to add (and debug) JDBC code to your beans and potentially makes your bean more portable across different persistent storage schemes (database vendors, schema variations, and even other persistence mechanisms like object databases). But even with the expanded CMP support provided in EJB 2.0 and 2.1, the automated persistence support in EJB is limited, and at times you'll need to manage persistence directly in your bean implementation. We discuss the pros and cons of each of these scenarios a bit later in this section.

6.7.1. Finder Methods

Since entity beans are persistent and can be accessed by multiple clients, clients have to be able to find them as well as create them. To this end, an entity bean's home interface(s) can provide finder methods , named with a `find`*`XXX`*`( )` scheme, that a client can invoke to look up preexisting entities in persistent storage and have them returned in the form of an EJB reference. For each finder method, the bean implementation has to have a corresponding `ejbFind`*`XXX`*`( )` method that takes the same arguments. The `find`*`XXX`*`( )` methods on the home interface can have any name, as long as the method name begins with `find`. A person bean, for example, might define a `findBySSN( )` method that lets you look up citizens of the U.S. using their Social Security number. The EJB implementation class must then have an `ejbFindBySSN( )` method that implements this lookup function.

The entity version of our `Profile` bean has two finder methods defined in its remote home interface:

```
public Profile findByPrimaryKey(String name)
    throws RemoteException, FinderException;

public Collection findByEntryValue(String key, String value)
    throws RemoteException, FinderException;
```

The first one finds a `Profile` based on the name of the profile owner, which in our case is the unique identifier for a `Profile`. The second finder finds any `Profile`s that have a particular value for a named entry.

A client can use the `find`*`XXX`*`( )` methods on the home interface to determine if an entity (or entities) with a given

identity already exists in persistent storage. From the client's point of view, if a `find`*`XXX`*`( )` method finds an appropriate entity or entities, entity bean instances are initialized and associated with these persistent entities, and references to the matching beans are returned to the client. If the identified entity is not found in persistent storage, a `javax.ejb.FinderException` is thrown. All `find`*`XXX`*`( )` methods on the bean's home interface must declare that they can throw `FinderException` and, if it's a remote home interface, `RemoteException` (or one of the parents of `Remote Exception`).

Each `find`*`XXX`*`( )` method on the home interface must return either an instance of the bean's remote interface or a collection of these objects. If a finder method can potentially return multiple beans, it has to have a return type that is either an `Enumeration` or a `Collection`. In our profile bean example, the `findByPrimaryKey( )` method can return only a single profile as a result, so it should be declared as returning an instance of the profile bean type. The `findByEntryValue()` method, on the other hand, can match multiple profiles, so it returns a `Collection`.

Every entity bean is required to have a `findByPrimaryKey( )` method defined on each of its home interfaces. This finder method takes a single argument (an instance of the primary key for the entity bean) and returns the entity corresponding to that primary key. By definition, a primary key uniquely identifies a single entity, so this finder method should return a single bean instance. We discuss primary keys shortly in "Entity Bean Implementations."

6.7.2. Select Methods

In addition to finder methods, entity beans that make use of container-managed persistence (discussed later) can have select methods as well. Select methods are utility methods that are accessible only from the bean implementation class itself; they aren't visible to the client. One of the principles behind CMP is to encapsulate the persistence details and not expose them in the EJB implementation. However, it is sometimes necessary to perform lookups against the underlying persistent store from business methods in your EJB implementation. Select methods provide this lookup functionality without forcing you to put database details into your bean code. You can define any number of `ejbSelect`*`XXX`*`( )` methods as abstract methods on your bean implementation class. The EJB container is responsible for providing implementations for these methods based on the persistence logic you provide for these methods in your deployment descriptors.

6.7.3. Entity Bean Implementations

Entity bean implementations must include all the basic container callbacks mentioned earlier: `ejbActivate( )`, `ejbPassivate( )`, `ejbRemove( )`, and an `ejbCreate( )` method for each `create( )` method provided in the EJB's home interface(s). Entity beans must also provide several persistence-related callbacks as well as some additional callbacks related to the lifecycle model applied to entity beans. Before we discuss these, we should understand how primary keys play a role with entity beans.

6.7.3.1. Primary keys

Every type of entity bean must have an associated primary key class. The primary key serves as a unique identifier for the persistent entities represented by the entity bean. The primary key for a person's records in a database, for example, might be a first and last name, a Social Security number (for U.S. citizens), or some other identification number. The primary key used for entity beans is public, in that clients can see the primary key for an entity bean, and the primary key is used directly by the bean implementation to load or update its state from persistent storage.

A primary key can be either a custom class that we write as part of the EJB implementation, or it can be a basic Java object type, like a `String`, `Float`, or `Integer`. The name of the user uniquely identifies our `Profile` bean, for example. A custom primary key class for an entity bean implementation of our `Profile` might look something like the following:

```
public class ProfilePK implements java.io.Serializable {
```

```
  public String mName;
  public ProfilePK(  ) {
    mName = null;
  }
  public ProfilePK(String name) {
    mName = name;
  }
  public boolean equals(Object other) {
    if (other instanceof ProfilePK &&
        this.mName.equals(((ProfilePK)other).mName)) {
      return true;
    }
    else {
      return false;
    }
  }
  public int hashCode(  ) {
    return mName.hashCode(  );
  }
}
```

However, since the unique identifier for a `Profile` is simply a single `String`, we can just use `java.lang.String` as the primary key for our entity `Profile` bean and avoid having to write the primary key class in the first place.

A primary key class needs to satisfy the requirements of being a `Value` type in RMI-IIOP, because it may need to be transmitted to remote clients. In practical terms, this means that the primary key class has to be `Serializable` and must not implement the `java.rmi.Remote` interface. A primary key class also has to have a valid implementation of the `equals( )` and `hashCode( )` methods so that the container can manage primary keys internally using collections, for example. If your entity bean uses container-managed persistence, the primary key class must obey the following additional rules:

- It must be a public class.
- It must have a default constructor (one with no arguments) that is public.
- All its data members must be public.
- All of the names of the data members on the class must match the names of container-managed data members on the entity bean.

We've satisfied all of these requirements with our `ProfilePK` primary key class (note that the only data member, `mName`, has to have a matching `mName` data member on the entity implementation of the `Profile` bean).

6.7.3.2. Entity bean container callbacks

Instead of the single `setSessionContext( )` callback method required on session EJBs, an entity EJB implementation is required to have the following two context-related callbacks:

public void setEntityContext(EntityContext ctx)

> The container calls this method after a new instance of the bean has been constructed, but before any of its `ejbCreate( )` methods are called. The bean is responsible for storing the context object. This method takes the place of the corresponding `setSessionContext( )` method on session beans.

public void unsetEntityContext(EntityContext ctx)

The container calls this method before the entity bean is destroyed to disassociate the bean from its identity, represented by its primary key.

When an EJB container is managing an entity EJB, it requires several additional container callbacks related to persistence operations:

public *primaryKeyType* ejbFindByPrimaryKey(*primaryKeyType*) throws FinderException

This is the only required finder method on an entity bean. Both the argument and the return type must be the bean's primary key type. The container is responsible for converting the returned primary key into a valid EJB reference for the client.

public void ejbPostCreate( )

If needed, an entity bean can optionally supply an `ejbPostCreate( )` method for each `ejbCreate( )` method it provides, taking the same arguments. The container calls the `ejbPostCreate( )` method after the bean's `ejbCreate( )` method has been called and after the container has initialized the transaction context for the bean.

public void ejbLoad( )

This method is called by the container to cause the bean instance to load its state from persistent storage. The container can call this bean method anytime after the bean has been created to do an initial load from persistent storage or to refresh the bean's state from the database (e.g., after a business method on the bean has completed or after the container has detected an update to the same persistent entity by another bean instance).

public void ejbStore( )

This method is called by the container to cause the bean to write its current runtime state to persistent storage. This method can be called anytime after a bean is created.

In addition to these entity-specific methods on bean implementations, the semantics of some of the other standard methods are slightly different for entity beans . Each `ejbCreate( )` method, for example, is actually a request to create a new entity in persistent storage. The implementations of the create methods should not only assign any state data passed in as arguments but also create a record in persistent storage for the new entity bean described by the method arguments. In addition to this semantic difference, the signatures of `ejbCreate( )` methods on entity beans can be different from session beans. For an entity bean that manages its own persistence (a bean-managed entity bean), the `ejbCreate( )` methods return the primary key type for the bean. For a container-managed entity bean, the `ejbCreate( )` methods return `void`, the same as for session beans.

An entity bean can be passivated by its container, but the meaning of being passivated is slightly different as well. A container passivates an entity bean (calling its `ejbPassivate( )` method in the process) when it wants to disassociate the bean from the persistent data entity it has been representing. After being passivated, the bean may be put into the container's "wait" pool, to be associated with another client-requested entity at a later time, or it may be removed from the server altogether.

A client calling the `remove( )` method on an entity bean reference or on the home interface for a bean is also

interpreted differently: it's a request to remove the entity from persistent storage entirely. The entity bean implementation should therefore remove its state from the persistent storage in its `ejbRemove( )` implementation.

6.7.4. Deployment Options for Entity Beans

When deploying entity EJBs, include an `<entity>` element in the `<enterprise-beans>` element within the deployment descriptor. A sample `<entity>` element for our `Profile` EJB is shown here:

```
...
<enterprise-beans>
    ...
    <entity>
        <display-name>BMP Profile Bean</display-name>
        <ejb-name>BMPProfileBean</ejb-name>
        <home>com.oreilly.jent.ejb.beanManaged.ProfileHome</home>
        <remote>com.oreilly.jent.ejb.Profile</remote>
        <ejb-class>com.oreilly.jent.ejb.beanManaged.ProfileBean</ejb-class>
        <persistence-type>Bean</persistence-type>
        <prim-key-class>java.lang.String</prim-key-class>
        <reentrant>false</reentrant>
        <resource-ref>
            <res-ref-name>jdbc/ProfileDB</res-ref-name>
            <res-type>javax.sql.DataSource</res-type>
            <res-auth>Container</res-auth>
        </resource-ref>
    </entity>
    ...
</enterprise-beans>
...
```

The `<entity>` element contains much of the same basic required information as a session element (`<ejb-name>`, `<home>`, `<remote>`, `<local-home>`, `<local>`, `<ejb-class>`), but it also contains several elements specific to entity beans. Three of these are required: `<persistence-type>`, `<prim-key-class>`, and `<reentrant>`.

The `<persistence-type>` element specifies how the persistence for your entity bean is to be managed. The allowed values for this element are `Bean` or `Container`, indicating the two fundamental ways that entity persistence can be handled. Our `ProfileBean` is written to use bean-managed persistence, so we set the persistence type to `Bean`.

The `<prim-key-class>` element specifies which class is being used as the primary key for this bean. We chose to use `String`s rather than our custom `ProfilePK` class, so we specify `java.lang.String` as our primary key type.

The `<reentrant>` element indicates whether an EJB can be accessed by multiple clients concurrently. If an EJB is marked reentrant, it's eligible for concurrent access; otherwise, it's run in a single-threaded manneronly one client request is processed at a time by any given EJB instance within the container. In our example, we haven't written our `ProfileBean` to be threadsafe (a quick look at the `setEntry( )` implementation in the source code will confirm that), so we've set the `<reentrant>` element to `false`. The `<reentrant>` property has other implications related to loopback calls (method implementations on the bean that result in calls back to the same bean instance). Refer to O'Reilly's Enterprise JavaBeans or the EJB specification for full details on loopback calls.

6.7.5. The Entity Context

The EJB container provides context information to an entity bean in the form of an `EntityContext` object. The container sets this object using the bean's `setEntityContext( )` method and removes it when the bean is being

removed by calling the bean's `unsetEntityContext( )` method. Like `SessionContext`, `EntityContext` provides the bean with access to its corresponding remotely exported object through the `getEJBObject( )` method. The `EntityContext` also gives an entity bean access to its primary key through `getPrimaryKey( )`. The declared return type of this method is `Object`, but the object returned is of the bean's primary key type. Note that the data accessed through the `EntityContext` might be changed by the EJB container during the bean's lifetime, as explained in the next section. For this reason, you shouldn't store the EJB remote object reference or primary key in data variables in the bean object since they might not be valid for the entire lifetime of the bean. Our entity `ProfileBean`, for example, stores the `EntityContext` reference in an instance variable, where it can access the context data as needed during its lifetime.

6.7.6. Lifecycle of an Entity Bean

Before the first client asks for an entity bean by calling a `create( )` or `find`*`XXX`*`( )` method on its home interface, an EJB container might decide to create a pool of entity beans to handle client requests for beans. This potentially reduces the amount of time it takes for a client to receive an entity bean remote reference after it makes a request for an entity bean. To add a bean to its pool, the container creates an instance of your bean implementation class and sets its context using the `setEntityContext( )` method. At this point, the entity bean hasn't been associated with a particular data entity, so it doesn't have an identity and therefore isn't eligible for handling client business method calls on bean references.

When a client calls a `create( )` method on the bean's home interface, the container picks a bean out of this wait pool and calls the corresponding `ejbCreate( )` method on the bean. If the `ejbCreate( )` method is successful, it returns one or more primary key objects to the container. For each primary key, the container picks an entity bean out of its pool to be assigned to the entity represented by the key. Next, the container assigns the bean's identity by setting the properties in its `EntityContext` object (e.g., its primary key and remote object values). If the bean has an `ejbPostCreate( )` method, that gets called after the bean's entity identity has been set. The `ejbCreate( )` method should create the entity in persistent storage, if the bean is managing its own persistence.

Alternatively, the client might call a `find`*`XXX`*`( )` method on the home interface. The container picks one of the pooled entity beans and calls the corresponding `ejbFind`*`XXX`*`( )` method on it. If the finder method finds one or more matching entities in persistent storage, the container uses entity beans in the wait pool to represent these entities. It picks entity beans out of the pool and calls their `ejbActivate( )` methods. Before calling `ejbActivate( )`, the container sets the bean's context by assigning the corresponding primary key and remote object reference in its context.

After an entity bean has been activated (either by being created through one of its `ejbCreate( )` methods or by being found and having its `ejbActivate( )` method called), it is associated with a specific entity in persistent storage and with a specific client stub. At any point after this, the container can call the bean's `ejbLoad( )` or `ejbStore( )` method to force the bean to read or write its state from or to persistent storage. The bean's business methods can also be invoked by clients when it is in this state.

At some point, the container may decide to put the bean back into its internal pool. This might happen after all remote references to the bean have been released or after a certain period of inactivity with the bean. The container might also do this as a reaction to client-loading issues (e.g., time-sharing pooled beans between client requests). When the container wants to remove the association between the entity bean instance and the client stub but doesn't want the object's state removed from persistent store, it calls the bean's `ejbPassivate( )` method. The bean can release any resources it allocated while in the active state, but it doesn't have to update persistent storage for the entity it represents, as this was done the last time the container invoked the bean's `ejbStore( )` method.

The bean can also lose its association with an entity when the client decides to remove the entity. The client does this either by calling a `remove( )` method on the bean's home interface or calling the `remove( )` method directly on an EJB object. When one of these things happens, the container calls the bean's `ejbRemove( )` method, and the bean deletes the data in persistent storage pertaining to the entity it represents. After the `ejbRemove( )`

method completes, the container puts the bean back into its internal pool.

6.7.7. Bean-Managed Persistence

Bean-managed persistence (BMP) is one option for dealing with the persistence of the state of entity EJBs. In BMP, all of the persistence management is done directly in the EJB implementation class, using custom JDBC calls or whatever API is appropriate for the persistent storage being used. The persistence management of an entity bean is handled in the `ejbCreate( )`, `ejbRemove( )`, `ejbLoad( )`, and `ejbStore( )` methods on the bean implementation. The persistent store is also accessed directly in the `ejbFind`*`XXX`*`( )` methods on the implementation of BMP beans.

Let's look at a BMP entity implementation of our `Profile` EJB. We've already seen some details on the home and client interfaces for this entity bean in earlier sections. The purpose of the bean is the same as our stateful session version: it represents a profile for a named application user, maintaining a list of name/value pairs for various attributes and options. The difference is that this `Profile` EJB represents a profile entity that exists as a set of persistent data (stored in a database, in this case). The full source code for the implementation class can be found in the file com/oreilly/jent/ejb/beanManaged/ProfileBean.java in the code bundle for the book; we'll cover some highlights in this tutorial.

The structure of the entity `Profile` implementation is similar to the stateful session bean version. All of the EJB-required methods, including `ejbActivate( )`, `ejbPassivate( )`, `ejbCreate( )`, and `ejbRemove( )`, are present. The `ejbActivate()` and `ejbPassivate( )` methods are called as the container moves the bean in and out of the entity bean "wait" pool, so it's a chance to free up any resources, such as file handles, before the bean is passivated and reclaim them if needed when it is activated again.

The `ejbCreate( )` methods on the `ProfileBean` create a new profile entity in the database, so our `ejbCreate ( )` method (associated with the `create( )` method in our home interface) is a bit more complicated than in our session beans because now it needs to make all the JDBC calls to create a profile entity in the underlying database:

```
public String ejbCreate(String name)
      throws DuplicateProfileException, CreateException {
      Connection conn = null;
      Statement s = null;
      try {
          conn = getConnection();
          s = conn.createStatement();
          s.executeUpdate("insert into profile (name) values ('"
                          + name + "')");
      }
      catch (SQLException se) {
          System.out.println("Error creating profile, assuming duplicate.");
          try {
              StringWriter strw = new StringWriter();
              PrintWriter prntw = new PrintWriter(strw);
              se.printStackTrace(prntw);
              throw new DuplicateProfileException(
                      "SQL error creating profile for " +
                      name + ": " + se.toString() +
                      "\n" + strw.toString());
          }
          catch (Exception e) {}
      }
      catch (NamingException ne) {
          System.out.println("Error accessing DataSource");
          throw new CreateException("Error accessing DataSource.");
      }
      finally {
          try { s.close(); } catch (Exception e) {}
```

```
            try { conn.close(); } catch (Exception e) {}
        }
        System.out.println("ProfileBean created for " + name + ".");
        return name;
    }
```

Notice that the `ejbCreate( )` method is using an internal utility method on our implementation, `getConnection ( )`, to get a JDBC connection to the database. This method looks for a `DataSource` stored in the JNDI context of the J2EE server and uses it to create a connection to the database. For details on configuring and using `DataSources`, see Chapter 8.

Another difference in this `ejbCreate( )` method is that it returns the EJB's primary key type (a `String`, in this case). The returned primary key value must represent the identity of the actual entity that was created by the `ejbCreate( )` method. In our case, we just return the name of the profile owner, since that's our unique identifier for profiles. The EJB container is responsible for intercepting the primary key object returned by the `ejbCreate ( )` method, converting it to a reference to a live `Profile` EJB, and returning a `Profile` stub to the client that originally called the `create( )` method on the `ProfileHome` interface.

An entity bean can also optionally provide an `ejbPostCreate( )` method for each `create( )` method on its home interface(s). The container calls this method after the `ejbCreate( )` method returns and after the container initializes the bean's transaction context. The `ejbPostCreate( )` method has to match the arguments to its corresponding `create( )` and `ejbCreate( )` methods. In our case, nothing needs to be done in the `ejbPostCreate( )` method, so we simply log that it's been called and return.

The `ejbRemove( )` method on our `ProfileBean` deletes all the records for this profile entity from the database:

```
    public void ejbRemove() {
        // Get this profile's name
        String key = (String)mContext.getPrimaryKey();
        Connection conn = null;
        Statement s = null;
        try {
            conn = getConnection();
            s = conn.createStatement();
            // Clear out any profile entries
            s.executeUpdate("delete from profile_entry where name = '"
                            + key + "'");
            // Clear out the profile itself
            s.executeUpdate("delete from profile where name = '" + key + "'");

            System.out.println("ProfileBean removed.");
        }
        catch (SQLException se) {
            System.out.println("Error removing profile for " + key);
            se.printStackTrace();
        }
        catch (NamingException ne) {
            System.out.println("Error accessing DataSource");
            ne.printStackTrace();
        }
        finally {
            try { s.close(); } catch (Exception e) {}
            try { conn.close(); } catch (Exception e) {}
        }
    }
```

In terms of the actual business methods of the `Profile` EJB, a `Properties` object holds the profile entries for the user, and the `getEntry( )` and `setEntry( )` remote method implementations access this `Properties` object for

the client. The name is found in the primary key object, and the primary key is stored for us in the `EntityContext` that the container gives us through the `setEntityContext( )` method. The `getName( )` method on the `ProfileBean` implementation shows how we retrieve the username for the profile using the `getPrimaryKey( )` method on the `EntityContext`:

```
public String getName(  ) {
    return (String)mContext.getPrimaryKey(  );
}
```

We've removed the `setName( )` method from the entity version of our `Profile` EJB since we don't want to allow the client to change the name of an existing, active entity bean. The `Profile` client interface for this bean, not shown here, is similar to the `Profile` interface in Example 6-6, with the `setName( )` method removed. Since the `Profile` is now a persistent entity bean and the name is the primary key, or identifying attribute, of the bean, the name of the bean should be set only when the bean is created. While the entity bean is active, it is associated with a profile entity for a specific user, and the client should read only the name associated with the profile.

Two additional persistence-related container callbacks are needed on entity bean implementations, `ejbLoad( )` and `ejbStore( )`. Since this bean is using BMP, our `ejbLoad( )` and `ejbStore( )` methods do all the JDBC calls directly, just like the other persistence callbacks in this version of the `Profile` EJB:

```
  public void ejbLoad() {
      try {
          String key = (String)mContext.getPrimaryKey();
          loadFromDB(key);
      }
      catch (Exception e) {
          System.out.println("Failed to load ProfileBean: ");
          e.printStackTrace();
          throw new EJBException("ejbLoad failed: ", e);
      }
      System.out.println("ProfileBean load finished.");
  }

  public void ejbStore() {
      String key = (String)mContext.getPrimaryKey();
      Connection conn = null;
      Statement s = null;
      try {
          conn = getConnection();
          s = conn.createStatement();
          // Clear out old profile entries; replace with current ones
          s.executeUpdate(
             "delete from profile_entry where name = '" + key + "'");
          Enumeration pKeys = mEntries.propertyNames();
          while (pKeys.hasMoreElements()) {
              String pKey = (String)pKeys.nextElement();
              String pValue = mEntries.getProperty(pKey);
              s.executeUpdate(
                "insert into profile_entry (name, key, value) values "
                + "('" + key + "', '" + pKey + "', '" + pValue + "')");
          }
      }
      catch (Exception e) {
          System.out.println("Failed to store ProfileBean: ");
          e.printStackTrace();
          throw new EJBException("ejbStore failed: ", e);
      }
      finally {
          try { s.close(); } catch (Exception e) {}
          try { conn.close(); } catch (Exception e) {}
```

```
        }
        System.out.println("ProfileBean store finished.");
    }
```

An `ejbFind`*`XXX`*`( )` method in our entity `ProfileBean` corresponds to each `find`*`XXX`*`( )` method in `ProfileHome`. The `ejbFindByPrimaryKey( )` method simply takes the primary key passed in as an argument and attempts to load the data for the entity from the database.

```
public String ejbFindByPrimaryKey(String key) throws FinderException {
    loadFromDB(key);
    return key;
}
```

If successful, it returns the primary key to the container, where it is converted to a remote `Profile` object to be returned to the client. It's not necessary to actually load all the profile data here in the finder method; we need only to verify that the specified entity exists in the database and either return the primary key to signal success or throw an exception. Since we already have the internal `loadFromDB( )` method used in `ejbLoad( )`, it is a simple matter to reuse it in this finder method. If the performance hit for loading the profile data twice is too great, we would have to rewrite the finder method to simply check the `PROFILE` table for a record matching the name in the primary key.

The `ejbFindByEntryValue( )` method takes key and value `String` arguments and attempts to find any and all profile entities with a matching key/value pair in the `PROFILE_ENTRY` table. Each name that has such a record is converted to a primary key object and returned to the container in a `Collection`.

```
  public Collection ejbFindByEntryValue(String key, String value)
      throws FinderException {
      LinkedList userList = new LinkedList();
      // Get a new connection from the EJB server
      Connection conn = null;
      Statement s = null;
      try {
          conn = getConnection();
          s = conn.createStatement();
          // Issue a query for matching profile entries, grabbing
          // just the name
          s.executeQuery("select distinct(name) from profile_entry where " +
                         " key = '" + key + "' and value = '"
                         + value + "'");
          // Convert the results in primary keys and return an enumeration
          ResultSet results = s.getResultSet();
          while (results.next()) {
              String name = results.getString(1);
              userList.add(name);
          }
      }
      catch (SQLException se) {
          // Failed to do database lookup
          throw new FinderException();
      }
      catch (NamingException ne) {
          // Failed to access DataSource
          throw new FinderException();
      }
      finally {
          try { s.close(); } catch (Exception e) {}
          try { conn.close(); } catch (Exception e) {}
      }
      return userList;
```

```
}
```

The container converts each primary key object in the returned `Collection` into a remote `Profile` object and returns the set to the client. If we encounter a database problem along the way, we throw a `FinderException`.

6.7.8. Container-Managed Persistence

In our first entity-based `Profile`, the persistent state of the profile entity is managed by the bean itself. Custom JDBC code in the `ProfileBean` implementation loads, stores, and removes the entity's database entries. The EJB container calls the appropriate callback methods on your entity bean, but your bean implementation is responsible for connecting to the database and making all of the necessary queries and updates to reflect the lifecycle of the data entity.

Container-managed persistence(CMP) allows you to define properties on your entity bean implementation that represent the state of the entity and tell the EJB container how to map these properties to persistent storage. With CMP, the container is responsible for loading, updating, and removing the entity data from persistent storage, based on the mapping you provide. The container invokes the callbacks on your bean implementation to notify it when it has performed these tasks for you, allowing your bean to do any follow-up work that may be required. The container also implements all the finder methods required by the bean's home interface.

If you want to take advantage of CMP, you have to indicate this to the EJB container when you deploy the bean, you need to provide a bean implementation that assumes that the container is performing all persistence management, and you need to supply the container with a data mapping at deployment time.

Implementing CMP-based entity beans involves describing the persistence properties of your bean rather than implementing them directly:

1. In your bean implementation class, declare abstract accessor methods for persistent properties that represent the state of the bean.

2. Declare finder and select methods in the home interface(s) for the EJB.

3. Map the persistent state properties to persistent storage in the EJB deployment descriptor.

4. Provide the container with the logic needed to implement the finder and select methods.

The EJB container is responsible for generating the relevant code to load, store, and update the state of the EJB to and from persistent storage and to implement the finder and select methods defined on your home interface(s). The various persistence-related container callback methods on your implementation class (`ejbLoad( )`, `ejbStore( )`, `ejbCreate( )`, and `ejbRemove( )`) are still used as callbacks by the EJB container, but in the case of CMP, they're invoked by the container just before and after it performs the corresponding persistence operation.

The complete source code for a CMP implementation of our `Profile` bean can be found in the source bundle for the book, in the file com/oreilly/jent/ejb/containerManaged/ProfileBean.java. We discuss the key features here.

In comparison to our earlier BMP-based `ProfileBean`, this bean is somewhat simpler since the `ejbRemove( )`, `ejbLoad( )`, and `ejbStore( )` methods don't need to perform database calls.

The container handles the loading and storing of the bean's data and the removal of any entities from the database, so we don't need to do anything about these operations in our bean implementation. In our `Profile` bean case, all we do in our `ejbLoad( )` and `ejbStore()` methods is map the persistence data managed by the container into our own internal data structures. This is not typical for CMP beans; we'll discuss the reasons that we need to do this in the next section when we discuss abstract persistence mappings. Here is the code for

`ejbLoad( )` and `ejbStore( )`:

```
public void ejbLoad(  ) {
    try {
      transferToProps(  );
    }
    catch (IOException e) {
      System.out.println("Failed to load ProfileBean: ");
      e.printStackTrace(  );
      throw new EJBException("ejbLoad failed: ", e);
    }
}

public void ejbStore(  ) {
    try {
      transferToBytes(  );
    }
    catch (IOException e) {
      System.out.println("Failed to store ProfileBean: ");
      e.printStackTrace(  );
      throw new EJBException("ejbStore failed: ", e);
    }
}
```

The important thing to note here is that these callbacks aren't doing any database operationsthe container is doing the database calls, and we're just getting ready for this (`ejbStore( )`) or cleaning up afterward (`ejbLoad( )`).

The same is true of the `ejbRemove( )` methodit's called just before the container is about to remove this entity's data from the database. In our case, there's nothing to do, so we provide an empty method:

```
public void ejbRemove(  ) {
    System.out.println("ProfileBean removed.");
}
```

6.7.8.1. Mapping container-managed fields: Abstract persistence schema

In the CMP model, the persistent state of your entity EJBs is specified in terms of an abstract persistence schema . The abstract schema consists of a set of named EJBs (the "tables" of the schema) and the persistent properties declared on your entity EJBs (the "columns" of the schema). You describe the components of this abstract schema in your EJB deployment descriptor and provide the container with a mapping from this abstract schema to a real persistence store (such as a relational database schema or an object database). The container takes all of this information and uses it to generate persistence management code for your EJB.

You indicate abstract persistence fields on your bean by providing abstract accessors (JavaBeans-style set and get methods) on the EJB implementation class. The EJB container provides concrete implementations for these abstract accessor methods in its generated classes. In our `ProfileBean` implementation class, we have two sets of accessors for the two abstract persistent fields needed by our bean: `get/setName( )` for the `name` property and `get/setEntriesBytes( )` for the `entriesBytes` property:

```
abstract public String getName(  );
abstract public void setName(String name);

abstract public byte[] getEntriesBytes(  );
abstract public void setEntriesBytes(byte[] entries);
```

The use of a byte array for our profile entries may seem like an odd implementation decision. It has to do with the

limitations of automatic CMP persistence. We'll return to this later, but for now just take it on faith that our profile entries need to be maintained as a byte array.

Note that, since we need to define persistent field accessors as abstract in our implementation class, the class itself must be declared as abstract as well:

```
abstract public class ProfileBean implements EntityBean {
```

Next, we need to describe our abstract persistence schema in the EJB's deployment descriptor. This essentially amounts to specifying an abstract schema name for each EJB we are creating (thus defining the "tables") and listing the persistent properties of each EJB (defining the "columns"). A partial listing of the ejb-jar.xml file for our CMP `Profile` bean is shown in Example 6-8.

Example 6-8. Section of ejb-jar.xml file for CMP profile EJB

```
<?xml version="1.0" encoding="UTF-8"?>
<ejb-jar xmlns="http://java.sun.com/xml/ns/j2ee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
                             http://java.sun.com/xml/ns/j2ee/ejb-jar_2_1.xsd"
         version="2.1">
    <enterprise-beans>
        <!-- A Profile EJB using container-managed persistence -->
        <entity>
            <ejb-name>CMPProfileBean</ejb-name>
            <home>com.oreilly.jent.ejb.containerManaged.ProfileHome</home>
            <remote>com.oreilly.jent.ejb.containerManaged.Profile</remote>
            <local-home>
                com.oreilly.jent.ejb.containerManaged.ProfileLocalHome
            </local-home>
            <local>
                com.oreilly.jent.ejb.containerManaged.ProfileLocal
            </local>
            <ejb-class>
                com.oreilly.jent.ejb.containerManaged.ProfileBean
            </ejb-class>
            <persistence-type>Container</persistence-type>
            <prim-key-class>java.lang.String</prim-key-class>
            <reentrant>false</reentrant>
            <cmp-version>2.x</cmp-version>
            <abstract-schema-name>ProfileBean</abstract-schema-name>

            <!-- Indicate which fields need to be managed persistently -->
            <cmp-field>
                <description>Name of the profile owner</description>
                <field-name>name</field-name>
            </cmp-field>
            <cmp-field>
               <description>Binary data containing profile entries.</description>
                <field-name>entriesBytes</field-name>
            </cmp-field>

            <!-- Since our primary key is simple (one-column), we need to specify
                 to the container which field maps to the primary key -->
            <primkey-field>name</primkey-field>
        </entity>
        ...
    </enterprise-beans>
    ...
</ejb-jar>
```

Before listing our persistent fields, we provide an `<abstract-schema-name>` element, which labels our EJB with an abstract name within the abstract persistence schema we're defining. Think of this abstract name as a sort of virtual table name; it's a way to refer to it in the EJB query language statements that define finder and select methods.

Following this, we list the `<cmp-field>` elements that describe our EJB's persistent properties. Our `Profile` bean has two persistent properties: `name` and `entriesBytes`, represented by the corresponding accessor methods in the bean implementation. So we have two `<cmp-field>` elements that serve to "tag" these bean properties as ones that need to be treated as persistent state.

In addition, the container needs to know how the bean's primary key is mapped. The primary key has to be mapped to one or more of the `<cmp-field>` elements listed in the deployment descriptor. If it maps to a single `<cmp-field>`, the name of the field has to be provided in the deployment descriptor as a `<primkey-field>` element. If the primary key is a complex key that maps to multiple `<cmp-field>` elements on the EJB, no `<primkey-field>` is provided and instead the data members on the primary key class must have variable names that match the corresponding persistent fields in the EJB implementation class. The container automatically makes the mapping from primary key fields to persistent bean fields. In our case, the primary key is simply the bean's `name` property, so we've included a `<primkey-field>` element in our deployment descriptor to that effect.

All we've done so far in the ejb-jar.xml file is provide the basic deployment information for our EJB and list the fields in the bean implementation that need to be persisted. But the container doesn't know how to persist these fields yet. The EJB specification does not define a standard way to provide this mapping to physical data storage. One reason for not including this in the specification is that it would require assumptions about the type of persistence being used by the container. EJB containers can (ostensibly) use any kind of persistent storage to maintain the state of entity beans: relational databases, object databases, XML databases, even flat files. Undoubtedly, virtually all of the major J2EE vendors provide relational persistence support in their CMP implementations, but the EJB specification team did not want to build this assumption into the specification, so they left the mapping of abstract persistent fields to physical storage up to the container vendors.

If we're using BEA's WebLogic application server, for example, we would specify the CMP deployment parameters using some additional deployment files that are specific to WebLogic. All EJBs (session, entity, or message-driven) deployed in WebLogic need to have an additional deployment file named weblogic-ejb-jar.xml, where things such as the JNDI binding for the EJB home interface can be specified. For container-managed entity beans using relational persistence, an additional deployment file, weblogic-cmp-rdbms-jar.xml, is used to specify persistence management information for the bean.

Each EJB-enabled server has its own scheme, but the required information is roughly the same. Regardless of how you specify the CMP field mapping, the EJB container reads the information and generates the necessary code to load and store the bean's persistent data to and from the underlying persistent storage system. It's your responsibility to ensure that the underlying physical persistence schema (whether it's relational, object-oriented, or hierarchical) is compatible with the deployment information provided to the container.

6.7.8.2. Handling complex data structures

Now we can finally explain why we implemented our CMP `Profile` EJB using a byte array for the entry data.

Each EJB container is limited to some degree in the way that data on your bean implementation can be mapped into persistent data fields. There is no standard format defined for the data mapping a particular container supports, so it's possible a particular EJB provider won't support whatever complicated mapping you require for your bean. For the most part, however, you can expect EJB providers to limit the format to a single persistent data field being mapped to a single data member on your bean implementation. If your bean's data structures are too complicated for you to provide an explicit mapping to persistent data fields, you have to decide how to deal with this.

In our entity `Profile` example, the profile entries need to be a list of name/value pairs that represent the parameters in the profile. We don't know at deployment time how many entries there will be, so we can't enumerate a mapping to particular database fields. Ideally, we really want each entry in the profile to be stored in a separate entry table in our database, along with the name of the owner of the entry, which is exactly how we implemented our bean-managed implementation. But since we're using CMP, we can't count on the EJB container supporting this type of persistence mapping (and, in fact, chances are that it won't).

One option is to give up on CMP and manage it yourself in the bean implementation using BMP. Another is to make each entry in the profile its own EJB and store the entries as a list of `Entry` beans on the profile bean. In this case, you could use EJB relationships, described in a later section, to tell the EJB container how these two entity EJBs are related to each other (e.g., when a given profile EJB is loaded, all of its corresponding entry EJBs should also be found and loaded). This makes the implementation pretty complex, though, since we now have two entity EJBs required to manage the data associated with profiles, and the amount of deployment information we need to provide gets much more complicated. This could also turn out to be expensive in terms of interactions with the container and memory usage. Each entry in the profile would need to be managed separately by the container, with all of the relevant lifecycle notifications and the like. CMP is supposed to make things simpler and faster, not the reverse, so we would probably not take this path in a real-world situation.

Another option, and the one we used in our CMP `Profile` EJB, is to convert the data structure into something that can be persisted into a single database table. Since the EJB specification gives us well-defined container callbacks to mark persistence operations, we can use these to do the conversion of our internal data structures to and from simplified persistent properties. In our `Profile` EJB, we defined the profile entries as a persistent byte array property. In our deployment descriptors for the bean, we mapped this persistent property to the binary `PROFILE_BIN.ENTRIES_BYTES` column in the database.

This gives us a persistent binary field for our EJB, but we still have to serialize the `Properties` object into the byte array when it needs to be stored persistently and deserialize the byte array into the `Properties` object when the persistent data is loaded. As we mentioned before, when using CMP, the `ejbLoad( )` and `ejbStore( )` methods are used by the container as callbacks. The `ejbStore( )` method is called just before the container performs an automatic store of the bean's persistent fields to the database, and `ejbLoad( )` is called just after the container has loaded the persistent fields from the database. So we can use the `ejbStore( )` method on our bean to copy the `Properties` object on our `ProfileBean` to the `mEntriesBytes` persistent data member. If you look at the `ejbStore( )` method, you'll see that it simply calls our `transferToBytes( )` method:

```
// Transfer the list of entries from our Properties member to the byte
// array
private void transferToBytes(  ) throws IOException {
  // Serialize the Properties into a byte array using an ObjectOutputStream
  if (mEntries != null && !mEntries.isEmpty(  )) {
    ByteArrayOutputStream byteOut = new ByteArrayOutputStream(  );
    ObjectOutputStream objOut = new ObjectOutputStream(byteOut);
    objOut.writeObject(mEntries);
    setEntriesBytes(byteOut.toByteArray(  ));
  }
  else {
    setEntriesBytes(null);
  }
}
```

This method uses I/O streams to serialize the `mEntries Properties` object into the `mEntriesBytes` byte array. After the container calls our `ejbStore( )` method, it can write the `mEnTRiesBytes` data member on our bean to a raw data field in the database (e.g., a `LONG BINARY` field in a SQL database). On the reading end, the container will load the bytes stored in the database column into the `mEntriesBytes` byte array and call our `ejbLoad( )` method. We can use the `ejbLoad( )` method to convert the bytes loaded by the container into a `Properties` object; our `ejbLoad( )` method simply calls `TRansferToProps( )`:

```
// Convert the serialized byte array into our Properties entry list
private void transferToProps(  ) throws IOException {
  // Take the raw byte array and deserialize it
  // back into a Properties object using an ObjectInputStream
  try {
    if (getEntriesBytes(  ) != null) {
      ByteArrayInputStream byteIn =
        new ByteArrayInputStream(getEntriesBytes(  ));
      ObjectInputStream objIn = new ObjectInputStream(byteIn);
      mEntries = (Properties)objIn.readObject(  );
    }
    // If no entries in database, set properties
    // to a new, empty collection
    else {
      mEntries = new Properties(  );
    }
  }
  catch (ClassNotFoundException cnfe) {
    System.out.println(
                "Properties class not found during de-serialization");
  }
}
```

This workaround is a good example of how the `ejbLoad( )` and `ejbStore( )` methods can be useful for CMP entity beans. It may also seem like an ideal solution for our `Profile` example since it allows us to deploy our entity `Profile` with container-managed persistence and still have a variable-sized list of entries on a profile. But this solution comes with a price tag: it makes our database records unusable for other, non-Java applications. The data stored in the `PROFILE_BIN.ENTRIES_ BYTES` column is a serialized Java object, so there's no way, for example, to check on a user's profile entries using a simple SQL query or to use a standard RDBMS reporting tool to show categories of users sorted by profile entries. As we'll see in the next section, the limitations of CMP also make it more difficult to implement certain complex finder methods.

6.7.8.3. Finder and select methods

When using container-managed persistence, the EJB container also generates all of the `ejbFind`*`XXX`*`( )` methods required for the finder methods on the home interface(s). It can automatically generate an `ejbFindByPrimaryKey ( )` method, based on the data mapping and primary key information you provide at deployment time (although the EJB specifications are unclear as to whether the container is required to do this). But for any other `ejbFind`*`XXX`*`( )` methods, you need to provide the container with the logic for the methods. The container can't infer the semantics of what you're trying to do based solely on method arguments. The same is true of select methods.

As an example, suppose we want to add a finder method for our `Profile` EJB, called `findEmptyProfiles( )` that finds all the profiles in the persistent store that have no entries in them. We would add the finder method to the home interface(s) for the `Profile` bean:

```
public Collection findEmptyProfiles(  )
                               throws RemoteException, FinderException;
```

The finder method returns a `Collection` because it could potentially find more than one empty profile in the database. Now we need to tell the EJB container how to implement this method.

The persistence logic for both finder and select methods is provided using EJB QL, a standard query language defined as part of the EJB specification. EJB QL is similar in syntax to SQL, except that queries are defined using the abstract schema elements defined in the deployment descriptors for your EJBs.

Query logic for finder and select methods is included in the ejb-jar.xml deployment descriptor, using `<query>` elements within the corresponding `<entity>` section. So to provide the query logic for the `findEmptyProfiles( )` finder method on our `Profile` EJB, we would add the following stanza to our ejb-jar.xml file, within the `<entity>` element for our `Profile` bean:

```
...
<enterprise-beans>
    ...
    <entity>
        ...
        <query>
            <query-method>
                <method-name>findEmptyProfiles</method-name>
                <method-params></method-params>
            </query-method>
            <ejb-ql>
                <![CDATA[SELECT OBJECT(p) FROM ProfileBean AS p
                        WHERE p.entriesBytes IS NULL]]>
            </ejb-ql>
        </query>
        ...
    </entity>
    ...
</enterprise-beans>
...
```

The full syntax for EJB QL is provided in Appendix C, but it should be fairly obvious in our example what we're doing. The `findEmptyProfiles( )` method takes no arguments, so we specify the `<query-method>` (using `findEmptyProfiles` as its `<method-name>`) and an empty `<method-params>` element. We want this finder to return all `Profiles` that have no entries in the database, so we specify an EJB QL statement that selects all `ProfileBeans` (using the abstract "table" name we defined earlier for our `Profile` EJB) whose `entriesBytes` persistent fields are null.

Select methods are defined using the same `<query>` element in the deployment descriptor. We simply specify the corresponding `ejbSelect`*`XXX`*`( )` method name in the `<method-name>` element.

This persistence logic, in combination with the abstract schema information provided earlier in the deployment descriptor, is used by the container to generate the concrete implementations of the finder and select methods for our bean. The EJB QL might be "compiled" into SQL for a relational database, Object Query Language (OQL) for an object-oriented database, or some other native persistence query language.

You may have noticed in our CMP bean implementation (if you peruse the source code) that we've removed the `ejbFindByEntryValue( )` finder method that we had in our BMP version. Because of the way we decided to implement the persistence of the entries of the `Profile` bean (using a single serialized byte array stored in a single binary database column), there isn't any way to implement the logic of `findByEntryValue( )`. From the perspective of EJB QL, the `entriesBytes` "column" is simply an array of bytes, with no way to query for the data stored in the `Properties` object that it came from. Again, we could fix this problem by changing our persistence mappingfor example, we could create a new `Entry` EJB and have the `Profile` bean maintain a set of references to `Entry` beans. But as mentioned in the previous section, this would impose some significant resource needs on our profile service, since each entry in each profile would now be a full-blown EJB object with all of the requisite container management.

6.7.8.4. EJB relationships

In keeping with the idea of an abstract persistence schema, EJB relationships can be thought of as abstract foreign key constraints between our EJB "tables." Suppose, for example, that in addition to our `Profile` bean, we also defined a `Person` entity bean that provided access to more general information about a person (name, address,

etc.). Within the `Profile` bean implementation, it might be useful to access the `Person` bean corresponding to the owner of a particular `Profile`. In the EJB 2.1 CMP model, we can do this using EJB relationships.

Since EJB relationships are a way to extend the abstract persistence schema of a set of CMP EJBs, they can be established only between EJBs that are using CMP. Session beans, message-driven beans, and BMP entity beans can't participate in EJB relationships. EJB relationships are accessible only from within the bean implementationsthey aren't exposed directly to clients. EJB relationships can be one-to-one or one-to-many. In our case, we want a single `Person` to be related to a single `Profile`, so the relationship will be one-to-one. If we plan to change our `Profile` bean so that it uses a new `EnTRy` EJB to represent its entries and we want to establish a persistence relationship between the two, we would make the relationship one-to-many.

It's important to note that EJB relationships are created using the local interfaces for EJBs. If an EJB doesn't have a local interface, it can have one-way relationships to other beans, but other EJBs can't have relationships with it.

The process for establishing an EJB relationship is similar to defining persistent fields; you define the accessor method(s) on either end of the relationship, specify the relationship in the EJB deployment descriptor, and then provide the container with the information on how the relationship is mapped to the underlying persistent storage system. In our case, we want to define a one-to-one relationship between our `Profile` bean and a new `Person` bean. Let's suppose that the `Person` bean has a very simple local client interface:

```
import javax.ejb.*;

public interface PersonLocal extends EJBLocalObject {
  // Access the person's first and last name
  public String getFirstName(  );
  public String getLastName(  );
}
```

We won't show all of the details of the implementation of this `Person` EJB; we'll just highlight the details relevant to the EJB relationship.

The first step in creating the relationship is defining abstract accessor methods on the implementations of the beans involved. On our `ProfileBean` implementation class, we would add the following accessors:

```
// Get local person related to this profile
abstract public PersonLocal getPersonLocal(  );
abstract public void setPersonLocal(PersonLocal person);
```

If we wanted the relationship to be bidirectional, we would also define accessor methods on the implementation class for our `Person` bean to read and write the `Profile` associated with the `Person`:

```
// Get/set local Profile using CMP relationships
abstract public ProfileLocal getProfileLocal(  );
abstract public void setProfileLocal(ProfileLocal profile);
```

Notice that these accessors need to be defined in terms of the local interfaces of the beans involved.

Now we need to specify the relationship between these two beans in the EJB deployment descriptor. Assuming that both beans are defined in the same ejb-jar.xml file, the relationship is defined like so:

```
...
<ejb-jar ...>
    <enterprise-beans>
        <!-- A Profile EJB using container-managed persistence -->
```

```
        <entity>
            <ejb-name>CMPProfileBean</ejb-name>
            ...
        </entity>
        <!-- A Person EJB using container-managed persistence -->
        <entity>
            <ejb-name>CMPPersonBean</ejb-name>
            ...
        </entity>
    </enterprise-beans>
    <!-- Establish EJB relationships between Person and Profile -->
    <relationships>
        <ejb-relation>
            <ejb-relation-name>Person-Profile</ejb-relation-name>
            <!-- Relation from person to profile -->
            <ejb-relationship-role>
                <ejb-relationship-role-name>
                    Person-has-Profile
                </ejb-relationship-role-name>
                <!-- One profile per person -->
                <multiplicity>One</multiplicity>
                <relationship-role-source>
                    <ejb-name>CMPPersonBean</ejb-name>
                </relationship-role-source>
                <cmr-field>
                    <cmr-field-name>profileLocal</cmr-field-name>
                </cmr-field>
            </ejb-relationship-role>
            <!-- Relation from profile to person -->
            <ejb-relationship-role>
                <ejb-relationship-role-name>
                    Profile-belongs-to-Person
                </ejb-relationship-role-name>
                <!-- One person per profile -->
                <multiplicity>One</multiplicity>
                <relationship-role-source>
                    <ejb-name>CMPProfileBean</ejb-name>
                </relationship-role-source>
                <cmr-field>
                    <cmr-field-name>personLocal</cmr-field-name>
                </cmr-field>
            </ejb-relationship-role>
        </ejb-relation>
    </relationships>
    ...
</ejb-jar>
```

We've added a `<relationships>` element to our ejb-jar.xml file, after the `<enterprise-beans>` section that describes the EJBs themselves. Each `<ejb-relation>` element in this section defines a relationship between two EJBs. Here we have a single `<ejb-relation>` that defines the relationship between the `Person` and `Profile` beans described earlier in the deployment descriptor. Each "side" of the relationship is described using an `<ejb-relationship-role>` element. The `<ejb-relationship-role>` element includes a name to assign to the role, the multiplicity of the role ("one" or "many"), the source EJB of the role (using the `<ejb-name>` associated with the EJB), and a `<cmr-field>` element that specifies the property on the bean that the role is associated with. In our case, we use the `profileLocal` and `personLocal` properties for the `<cmr-field>` elements to correspond to the accessor methods we defined earlier on our beans.

The last thing we need to do is provide the EJB container with the information specifying how this relationship is represented in persistent storage. Again, a format for specifying this information isn't defined in the EJB specification. You need to determine what form of persistence your J2EE server supports and how it requires the persistence mapping to be defined. The details of providing this mapping will be server-specific: JBoss will do it

one way, WebLogic will do it another, and so on. Consult your application server's documentation for full details.

Since we defined both read and write accessors for either end of this EJB relationship, we can alter the persistent relationship between a `Person` and his `Profile` by simply using the `setPersonLocal( )` or `setProfileLocal ( )` methods on the `Profile` or `Person` implementations, respectively. The data in persistent storage will be adjusted accordingly by the EJB container to reflect the new relationship. In the case of RDBMS persistent storage, the foreign key column(s) point to the appropriate row in the target table.