Oracle8*i*

Enterprise JavaBeans Developer's Guide and Reference

Release 3 (8.1.7)

July 2000

Part No. A83725-01

ORACLE®

Enterprise JavaBeans Developer's Guide and Reference, Release 3 (8.1.7)

Part No. A83725-01

Release 3 (8.1.7)

Primary Authors: Sheryl Maring

Contributors: Tim Smith, Ellen Barnes, Matthieu Devin, Steve Harris, Hal Hildebrand, Susan Kraft, Thomas Kurian, Wendy Liau, Angie Long, Sastry Malladi, John O'Duinn, Jeff Schafer, Aniruddha Thakur, Iyad Elayyan, Cheuk Chau

Contents

3   Configuring IIOP Applications

4 Entity Beans

5 JNDI Connections and Session IIOP Service

6   IIOP Security

7   Transaction Handling

A   XML Deployment Descriptors

B   Example Code: EJB

C   Abbreviations and Acronyms

x

Send Us Your Comments

Enterprise JavaBeans Developer's Guide and Reference, Release 3 (8.1.7)

Part No. A83725-01

Oracle Corporation welcomes your comments and suggestions on the quality and usefulness of this publication. Your input is an important part of the information used for revision.

- Did you find any errors?
- Is the information clearly presented?
- Do you need more information? If so, where?
- Are the examples correct? Do you need more examples?
- What features did you like most about this manual?

If you find any errors or have any other suggestions for improvement, please indicate the chapter, section, and page number (if available). You can send comments to us in the following ways:

- Electronic mail — jpgcomnt@us.oracle.com
- FAX - 650-506-7225.   Attn: Java Platform Group, Information Development Manager
- Postal service:
  Oracle Corporation
  Information Development Manager
  500 Oracle Parkway, Mailstop 4op978
  Redwood Shores, CA 94065
  USA

Please indicate if you would like a reply.

If you have problems with the software, please contact your local Oracle World Wide Support Center.

Preface

This guide gets you started building Enterprise JavaBeans for Oracle8*i*. It includes many code examples to help you develop your application.

Who Should Read This Guide?

Anyone developing server-side Enterprise JavaBeans for Oracle8*i* will benefit from reading this guide. Written especially for programmers, it will also be of value to architects, systems analysts, project managers, and others interested in network-centric database applications. To use this guide effectively, you must have a working knowledge of Java and Oracle8*i*.

Prerequisite Reading

Before consulting this Guide, you should read the following:

- *Oracle8i Java Developer's Guide* gives you the technical background information necessary to understand Java in the database server. As well as a comprehensive discussion of the advantages of the JServer implementation for enterprise application development, it explains the fundamentals of the JServer Java virtual machine and gives a technical overview of the tools that JServer provides.

- Sun Microsystems's EJB 1.1 specification as a supplement to this Guide. This guide assumes that you already have a base understanding of the EJB 1.1 specification details.

See "Your Comments Are Welcome" on page iv for more sources of information on Enterprise JavaBeans.

Suggested Reading

Books

- *Core Java* by Cornell & Horstmann, second edition, Volume II (Prentice-Hall, 1997) has good presentations of several Java concepts relevant to EJB. For example, this book documents the Remote Method Invocation (RMI) interface.

- The *Developer's Guide to Understanding Enterprise JavaBeans,* an overview of EJBs, is available at `http://www.Nova-Labs.com`.

Online Sources

There are many useful online sources of information about Java. For example, you can view or download guides and tutorials from the Sun Microsystems home page on the Web:

```
http://www.sun.com
```

The current 1.1 EJB specification is available at:

```
http://java.sun.com/products/ejb/docs.html
```

Another popular Java Web site is:

```
http://www.gamelan.com
```

For Java API documentation, see:

```
http://www.javasoft.com
```

A white paper by Anne Thomas of the Patricia Seybold group (paper sponsored by Sun Microsystems) is available at:

```
http://java.sun.com/products/ejb/white_paper.html
```

Related Publications

Occasionally, this guide refers you to the following Oracle publications for more information:

Oracle8i Application Developer's Guide - Fundamentals

Oracle8i Java Developer's Guide

Oracle8i JDBC Developer's Guide and Reference

Oracle8i SQL Reference

How This Guide Is Organized

This guide consists of the following:

Chapter 1, "Overview", presents a brief overview of the EJB development model from an Oracle8*i* perspective.

Chapter 2, "Enterprise JavaBeans", discusses EJB development for the Oracle8*i* server. Although this chapter is not a tutorial on EJB, it contains some of the basic EJB concepts included in the Sun Microsystems specification. The examples focus on a session bean implementation.

Chapter 3, "Configuring IIOP Applications", describes the configuration required to execute an EJB within the database.

Chapter 4, "Entity Beans", describes how to implement an entity bean. This details both a container-managed persistent and a bean-managed persistent model for entity beans.

Chapter 5, "JNDI Connections and Session IIOP Service", covers session management, the session IIOP service and the JNDI namespace. This chapter contains examples and scenarios for accessing EJBs deployed within the server using JNDI and the session IIOP service.

Chapter 6, "IIOP Security", discusses security options for authentication.

Chapter 7, "Transaction Handling", documents the JTA transaction interfaces that you use when developing EJBs.

Appendix A, "XML Deployment Descriptors" describes both the EJB and Oracle-specific deployment descriptors. This appendix contains the full details and semantics for all elements contained within both deployment descriptors.

Appendix B, "Example Code: EJB", includes examples of EJB applications.

Appendix C, "Abbreviations and Acronyms", supplies a convenient list of acronyms.

Notational Conventions

This guide follows these conventions:

| *Italic* | Italic font denotes terms being defined for the first time, words being emphasized, error messages, and book titles. |
| Courier | Courier font denotes Java program names, file names, path names, and Internet addresses. |

Java code examples follow these conventions:

{ }	Braces enclose a block of statements.
//	A double slash begins a single-line comment, which extends to the end of a line.
/* */	A slash-asterisk and an asterisk-slash delimit a multi-line comment, which can span multiple lines.
...	An ellipsis shows that statements or clauses irrelevant to the discussion were left out.
lower case	Lower case is used for keywords and for one-word names of variables, methods, and packages.
UPPER CASE	Upper case is used for names of constants (static final variables) and for names of supplied classes that map to built-in SQL datatypes.
Mixed Case	Mixed case is used for names of classes and interfaces and for multi-word names of variables, methods, and packages. The names of classes and interfaces begin with an upper-case letter. In all multi-word names, the second and succeeding words also begin with an upper-case letter.

Your Comments Are Welcome

We appreciate your comments and suggestions. In fact, your opinions are the most important feedback we receive. We encourage you to use the Reader's Comment Form at the front of this book. You can also send comments to the following address:

Documentation Manager, Oracle8*i* Java Products Group
Oracle Corporation
500 Oracle Parkway
Redwood Shores, CA 94065
USA
email: jpgcomnt@us.oracle.com

1

Overview

This chapter gives you a general picture of distributed object development in the Oracle8*i* JServer. This overview focuses on the aspects of Enterprise JavaBeans (EJB) development particular to JServer, giving a brief general description of the EJB standard development model.

This chapter covers the following topics:

- About Enterprise JavaBeans
- Implementing an EJB
- Basic Concepts
- Types of EJBs

About Enterprise JavaBeans

Oracle8*i* JServer complies with the EJB 1.1 specification and offers a highly scalable and high-performance execution environment for EJBs.

Enterprise JavaBeans (EJB) is an architecture for developing distributed applications. Additionally, EJB applications are developed entirely in Java. It is not necessary for developers to learn a new language, such as IDL for CORBA applications.

EJB is an architecture for transactional, component-based distributed computing. The specification for EJBs lays out not just the format of a bean itself, but also a set of services that must be provided by the container in which the bean runs. This makes EJBs a powerful development methodology for distributed application development. Neither the bean developer nor the client application programmer needs to be concerned with service details such as transaction support, security, remote object access, and many other complicated and error-prone issues. The EJB server and container furnishes these features for you.

EJB architecture makes server-side development much easier for the Java application programmer. Because the implementation details are hidden from the developer, and because services such as transaction support and security are provided in an easy-to-use manner, you can develop EJBs relatively quickly. Furthermore, EJBs offer portability. A bean developed on one EJB server should run on other EJB servers that meet the EJB specification.

EJB Development Roles

The EJB specification describes enterprise bean development in five roles:

- The *EJB developer* writes the code that implements an individual EJB. This code is the business logic of the application, usually involving database access.

  The EJB developer is a Java applications programmer familiar with both SQL and with database access using SQLJ or JDBC.

- The *EJB deployer* installs and publishes the EJB. This involves interaction with the EJB developer so that the transactional nature of the EJB is understood. The EJB deployer writes the *deployment descriptor files* that specify the properties of each bean to be deployed.

  The EJB deployer must be familiar with the runtime environment of the EJBs, including database-specific matters such as network ports, database roles required, and other schema-specific requirements. For the Oracle8*i* server, the

EJB deployer is responsible for publishing the EJB home interfaces in a database and communicating this information to the client-side application developer.

- The *EJB server vendor* implements the framework in which the EJB containers run. For Oracle, the Oracle8*i* data server is the framework that supports the EJB containers.

- The *EJB container vendor* supplies the services that support the EJB at runtime. For example, when a client expects the bean to handle transaction support automatically, the container framework together with the data resource supports this.

- The *application developer* writes the client-side code that calls methods on server EJBs.

The roles of the EJB server and EJB container developers are not clearly distinguished. There is, for example, no standardized API between the container and the server. For this reason, the same vendor is likely to perform initial implementations for EJB servers and containers. This is the case for Oracle8*i*.

Oracle8*i* EJB Implementation Features

The Oracle8*i* EJB implementation is able to leverage the Oracle database server and offers the following features:

- a simple-to-use way of locating and activating beans, using a JNDI interface to an underlying OMG CosNaming service

- a session name space that uses the database as a name server, with its performance advantages, such as fast access to indexed tables

- secure socket layer (SSL) connections for added security

- standard Oracle database authentication and multi-layer access control to objects

- an implementation of the Java Transaction Architecture (JTA) for transaction demarcation

- a UserTransaction interface for bean-managed transactions

- tools that assist you in deploying your EJB application

RMI over IIOP

EJB specifies Java Remote Method Invocation (RMI) as the transport protocol. EJBs are based conceptually on the Java Remote Method Invocation (RMI) model. For

example, remote object access and parameter passing for EJBs follow the RMI specification.

The EJB specification does not prescribe that the transport mechanism has to be pure RMI. The Oracle8*i* EJB server uses RMI over IIOP for its transport protocol. Because the CORBA Internet Inter-ORB Protocol (IIOP) is the transport protocol for CORBA and for a future version of RMI, Oracle8*i* effectively enables direct object-oriented access to an array of open systems.

IIOP Transport Protocol

Oracle8*i* offers a Java interpreter for the IIOP protocol. Oracle embeds a pure Java ORB of a major CORBA vendor (VisiBroker for Java version 3.4 by Inprise) and repackaging the Visigenic Java IIOP interpreter to run in the database. Because Oracle8*i* is a highly scalable server, only the essential components of the interpreter are necessary—namely, a set of Java classes that do the following:

- decode the IIOP protocol

- find or activate the relevant Java object

- invoke the method the IIOP message specifies

- write the IIOP reply back to the client

Oracle8*i* does not use the ORB scheduling facilities. The Oracle multi-threaded server performs the dispatching, enabling the server to process IIOP messages efficiently and in a highly scalable manner.

On top of this infrastructure, Oracle8*i* implements the EJB programming model.

JNDI

EJB developers follow the EJB specification and use JNDI for access. Each bean is published automatically during the deployment process. JNDI provides access to the published bean through a CosNaming layer.

Figure 1–1 shows how applications access remote objects published in the database using JNDI.

Figure 1–1   Remote Object Access
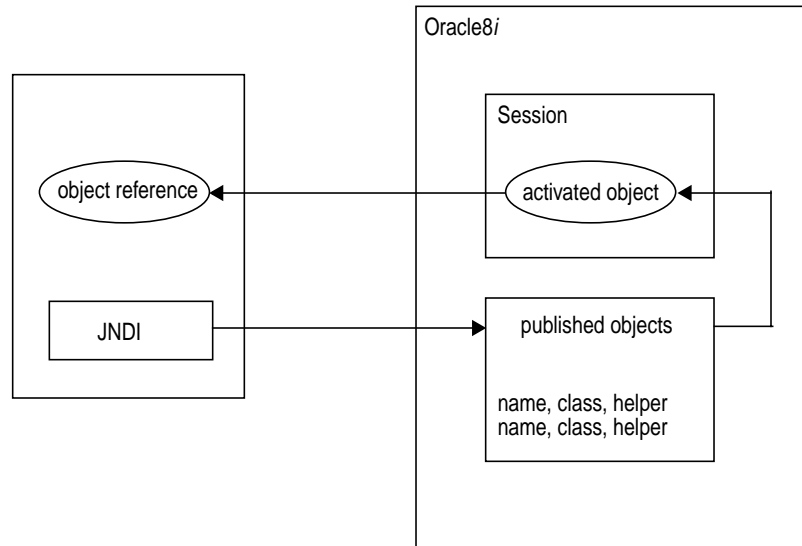
Stateful and Stateless Session Beans

The EJB specification calls for two types of session bean: stateless and stateful beans.

- Stateless beans—which do not share state or identity between method invocations—find use mainly in middle tier application servers that provide a pool of beans to process frequent and brief requests, such as those involved in an OLTP application.

- Stateful beans are useful for longer-duration sessions, in which it is necessary to maintain state, such as instance variable values or transactional state, between method invocations.

Because the Oracle8*i* ORB and Java VM run under the multi-threaded server (MTS), the distinction between stateless and stateful session beans is not important for JServer. Thus, EJB activates only stateful session beans on demand in a new session. Stateful beans can offer the same performance as stateless beans, while preserving the "conversational state".

Implementing an EJB

There are four major components that you must create to develop a complete EJB:

- the *home interface*
- the *remote interface*
- the *implementation* of the remote interface—the actual bean class
- a *deployment descriptor* for each EJB

Component	Description
The home interface	Specifies the interface to an object that the container itself implements: the *home object*. The home interface contains the create() methods that specify how a bean is created. The home interface, with the home object serves as a factory object for EJBs.
The remote interface	Specifies the methods that you implement in the bean. These methods perform the business logic of the bean. The bean must also implement additional service methods that the EJB container calls at various times in the life cycle of a bean. See Basic Concepts on page 1-7 for more information about these service methods.
The bean implementation	Contains the Java code that implements the remote interface and the required container methods.
The deployment descriptor	Specifies attributes of the bean for deployment and loading into the database. For example, the deployment descriptor declares the transactional properties of the bean. At deployment time, the EJB deployer, together with the application developer, can decide whether the container should manage transaction support or have the client do it.

The client application itself does not access the bean directly. Rather, the container generates a server-side object known as the *EJBObject* that serves as a server-side proxy for the bean. The EJBObject receives the messages from the client, and thus the container can interpose its own processing before the messages are sent to the bean implementation.

Figure 1–2 on page 1-7 illustrates the interaction among these components.

Figure 1–2   Basic EJB Component Relationships
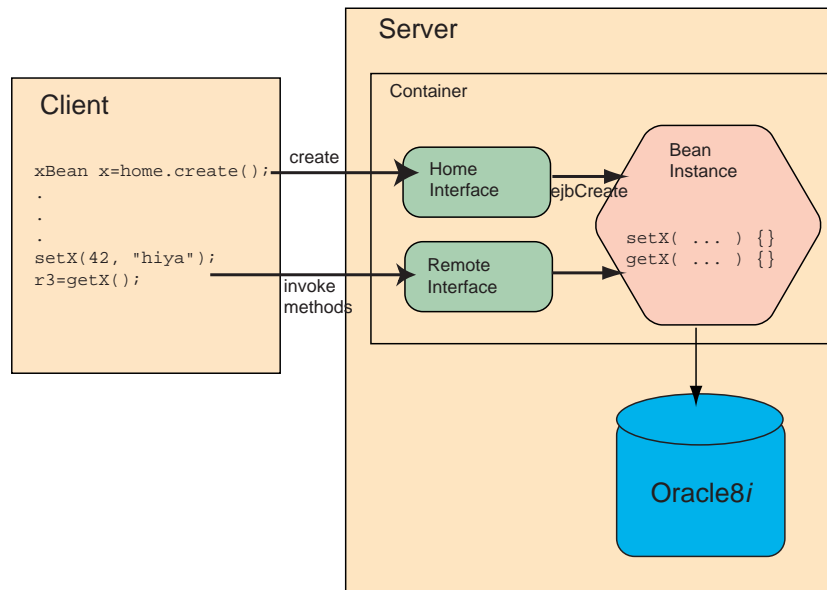
Basic Concepts

Before going into details about implementing EJBs, some basic concepts must be clarified. First of all, recall that a bean runs in a container. The container, which is part of the EJB server, provides a number of services to the bean. These include transaction services, synchronization services, and security.

To provide these services, the bean container must be able to intercept calls to bean methods. For example, a client application calls a bean method that has a transaction attribute that requires the bean to create a new transaction context. The bean container must be able to interpose code to start a new transaction before the method call, and to commit the transaction, if possible, when the method completes, and before any values are returned to the client.

For this reason and others, a client application does not call the remote bean methods directly. Instead, the client invokes the bean method through a two-step process, mediated by the ORB and by the container:

1. The client invokes the bean method off of the remote interface object.

    a. The client actually calls a local proxy stub for the remote method.

    b. The stub marshals any parameter data, and then calls a remote skeleton on the server.

2. The skeleton unmarshals the data, and upcalls to the bean container.

This step is required because of the remote nature of the call. Note that this step is completely transparent both to the client application developer as well as to the bean developer. It is a detail that you do not need to know about to write your application code, either on the client or the server. Nevertheless, it is useful to know what is going on, especially when it comes to understanding what happens during bean deployment.

3. The bean container receives the skeleton upcall, then interposes whatever services are required by the context. These can include:

- authenticating the client, on the first method call

- performing transaction management

- calling synchronization methods in the bean itself

- identity checks and switch

4. The container delegates the method call to the bean.

5. The bean method executes.

6. When it returns, the thread of control returns to the bean container, which interposes whatever services are required by the context.

For example, if the method is running in a transaction context, the bean container performs a commit operation, if possible, depending on the transaction attributes in the bean descriptor.

7. The bean container calls the skeleton, which marshals return data and returns it to the client stub.

These steps are completely invisible to client-side and server-side application developers. One of the major advantages of the EJB development model is that it hides the complexity of transaction and identity management from developers.

Types of EJBs

There are two types of EJBs: *session beans* and *entity beans.* An easy way to think of the difference is that a session bean implements one or more business tasks, while an entity bean is a complex business entity. A session bean might contain methods that query and update data in a relational table; an entity bean represents business data directly or indirectly through another persistent bean.

Session beans are often used to implement services. For example, an application developer might implement one or several session beans that retrieve and update inventory data in a database. You can use session beans to replace stored procedures in the database server, thereby achieving the scalability inherent in the Oracle8*i* Java server.

Entity beans are often used to facilitate business services that involve data and computations on that data. For example, an application developer might implement an entity bean to retrieve and perform computation on items within a purchase order. Your entity bean can manage multiple dependent persistent objects in performing its necessary tasks.

Persistence

Session beans are not inherently *persistent.* Persistence can refer either to a characteristic of the bean—entity beans are persistent, session beans are not inherently persistent—or it can refer to data that a bean might save, so that the data can be retrieved in a future instantiation. Persistent data is saved in the database.

Therefore, a session bean can save its state in an Oracle8*i* database, but it does not directly represent business data. Entity beans persist the business data either automatically—in a container-managed persistent entity bean—or by way of methods that use JDBC or SQLJ and are coded into the bean—in a bean-managed persistent entity bean.

Implementing the synchronization interface can make data storage and retrieval automatic for session beans.

Session Beans

Created by a client, a session bean is usually specific to that client. In Oracle8*i* more than one client can share a session bean.

Session beans are transient in that they do not survive a server crash or a network failure. If, after a crash, you instantiate a bean that had previously existed, the state of the previous instance is not restored. State can only be restored to entity beans.

Stateful Session Beans

A stateful session bean maintains its state between method calls. For example, a single instance of a session bean might open a JDBC database connection and use the connection to retrieve some initial data from the database. For example, a shopping-cart application bean could load a customer profile from the database as soon as it's activated, then that profile would be available for any method in the bean to use.

A typical stateful session EJB is a relatively coarse-grained object. A single bean almost always contains more than one method, and the methods provide a unified, logical service. For example, the session EJB that implements the server side of a shopping cart on-line application would have methods to return a list of objects that are available for purchase, put items in the customer's cart, place an order, change a customer's profile, and so on.

The state that a session bean maintains is called the "conversational state" of the bean, as the bean is maintaining a connection with a single client, similar to a telephone conversation.

Keep in mind that the state of a bean is still transient data, with respect to the bean itself. If the connection from the client to the bean is broken, the state can be lost. This depends on whether the client is unable to reconnect before timeout.

Entity Beans

Entity beans are persistent in that they do survive a server crash or a network failure. When an entity bean is re-instantiated, the state of previous instances is automatically restored. For more information on entity beans, see "Definition of an Entity Bean" on page 4-2.

2

Enterprise JavaBeans

This chapter describes the development and deployment of Enterprise JavaBeans in the Oracle8*i* server environment. Although it is not a complete tutorial on EJB and the EJB architecture, this chapter supplies you with enough information to start developing EJB applications.

This chapter covers the following topics:

- Invoking Enterprise JavaBeans
- Creating Enterprise JavaBeans
- Implementing the Bean
- Developing Your Client Application
- Deploying an EJB
- Programming Restrictions
- Debugging Techniques

Invoking Enterprise JavaBeans

An Enterprise JavaBean has two client interfaces: a remote interface and a home interface. The remote interface specifies the methods that the object's clients can invoke; the home interface defines how clients can create the object, which returns a reference to the object. The client uses both of these interfaces when invoking a method on a bean.

The events that occur when a client invokes a method within a bean are explained in the following diagram and steps:

Figure 2–1   Sequence of events in a session bean lifecycle
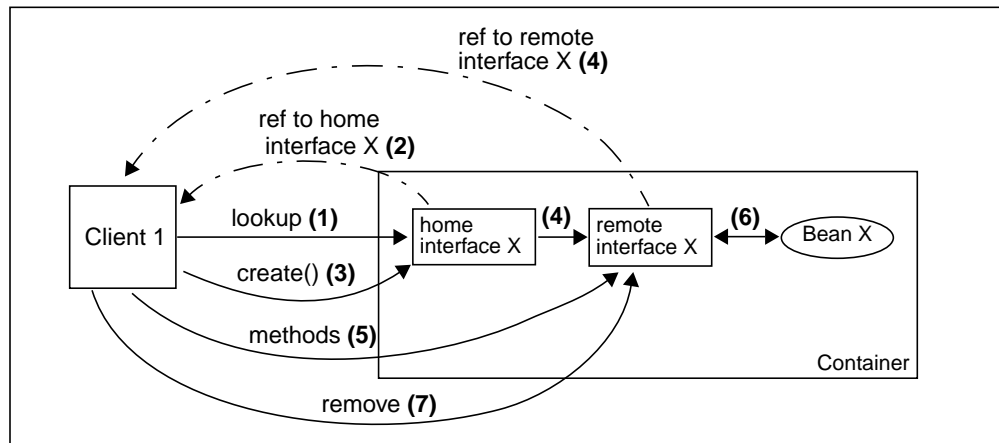
The numbers in the figure correspond to the following numbered steps:

1. Client 1 looks up home interface of bean X.

   During the JNDI lookup, the database creates a session for the server-side of the request.

2. Reference to home interface X is returned to client 1.

3. Client 1 invokes `create` on home interface X.

   The bean instance is created within the session that was established on the JNDI lookup.

4. Home interface X instantiates remote interface X. The container instantiates the bean for this client and is destroyed only when the client invokes the remove.

   The object reference of remote interface X is returned to client 1.

5. Client 1 uses remote interface X to invoke methods on bean instance X.

6. Remote interface X delegates call to a bean.

7. Client 1 invokes `remove` on remote interface X when it is done with the bean instance. This destroys the remote interface and the bean instance.

Creating Enterprise JavaBeans

To create an EJB, you must perform the following steps:

1. Create a remote interface for the bean. The remote interface declares the methods that a client can invoke. It must extend `javax.ejb.EJBObject`.

2. Create a home interface for the bean. The home interface must extend `javax.ejb.EJBHome`. In addition, it defines the `create` method for your bean.

3. Implement the bean. This includes the following:

   a. The implementation for the methods declared in your remote interface.

   b. The methods defined in either the `javax.ejb.SessionBean` or `javax.ejb.EntityBean` interfaces. For the differences between these types of beans, see "Definition of an Entity Bean" on page 4-2.

   c. The `ejbCreate` method with parameters matching those of the `create` method defined of the home interface.

4. Create the bean deployment descriptor. The deployment descriptor specifies properties for the bean. See Chapter 4, "Creating Deployment Files".

5. Create an ejb-jar file containing the bean, the remote and home interfaces, and the deployment descriptor. The ejb-jar file must define all beans within your application. Refer to Chapter 4, "Creating Deployment Files" for more details.

Requirements for Remote and Home Interface Implementation

Requirement	Description
RMI conformance	Because the `javax.ejb.EJBObject` and `javax.ejb.EJBHome` interfaces extend `java.rmi.Remote`, they must be compliant with the Remote Method Invocation (RMI) specification. This means that their methods can only use the data types allowed in RMI, and that methods in both interfaces must throw the `java.rmi.RemoteException` exception.
	You can get the RMI specifications from the JavaSoft site, **http://www.javasoft.com**.
Naming conventions	The interface names, method names, and constants defined within these interfaces cannot start with an underbar (_) or contain a dollar sign ($). In addition, the application and bean names can include the slash sign (/).

Creating the Remote Interface

The remote interface of a bean provides an interface for the methods that the client will invoke. That is, the remote interface defines the methods that you implement for remote access.

1. The bean's remote interface must extend the `javax.ejb.EJBObject` interface, which has the following definition:

```
public interface javax.ejb.EJBObject extends java.rmi.Remote {
   public abstract EJBHome getEJBHome()
         throws java.rmi.RemoteException  // returns reference to home
                                          // interface for this bean
   public abstract Handle getHandle()
         throws java.rmi.RemoteException // returns serializeable handle
   public abstract Object getPrimaryKey()
         throws java.rmi.RemoteException // returns key to an entity bean
   public abstract boolean isIdentical(EJBObject obj)
         throws java.rmi.RemoteException
   public abstract void remove()
         throws java.rmi.RemoteException, RemoveException //remove EJB object
}
```

You do not need to implement the methods in the `EJBObject` interface; these methods are implemented for you by the container.

Function	Description
getEJBHome()	Retrieves the object reference for the home interface associated with this particular bean. Note that you cannot typecast the returned object to the home interface type. See "Using the getEJBHome Method" on page 2-13 for more information.
getHandle()	A serializable Java representation of the EJB object reference can be obtained using the getHandle method of the remote interface. The handle can be serialized and used to re-establish a connection to the same object. With session beans, the connection is re-established as long as the bean instance is still active; with entity beans, the connection is re-established irregardless of whether the bean is active or not.
	You use the getEJBObject method within the Handle class to retrieve the bean instance.
getPrimaryKey()	The getPrimaryKey method retrieves the primary key associated with the EJB EntityBean.
isIdentical()	Tests that the object calling this method and the object in the argument are identical (as far as the container is concerned). This identifies that both objects are the same for all purposes.
remove()	Deactivates the EJB bean. This, in turn, destroys the session bean instance (if stateful).

2. The signature for each method in the remote interface must match the signature in the bean implementation.

3. The remote interface must be declared as public.

4. You do not declare public variables in the remote interface. Only the public methods are declared.

5. Any exception can be thrown to the client, as long as it is serializable. Runtime exceptions are transferred back to the client as a remote runtime exception.

Example

The following code sample shows a remote interface called **Employee**, which declares the getEmployee method, which will be implemented in the bean.

```
package employee;

import employee.EmpRecord;
import javax.ejb.EJBObject;
import java.rmi.RemoteException;

public interface Employee extends EJBObject  {
  public EmpRecord getEmployee (int empNumber)
       throws java.sql.SQLException, EmpException, RemoteException;
}
```

Creating the Home Interface

The home interface should define the appropriate `create` method for your bean. The home interface specifies one or more `create` methods. For each `create` method, a corresponding `ejbCreate` method must be defined in the remote interface. All of the `create` methods return the bean type; all of the `ejbCreate` methods return void.

The client invokes the `create` method declared within the home interface. The container turns around and calls the `ejbCreate` method, with the appropriate parameter signature, within your bean implementation. The parameter arguments can be used to initialize the state of a new EJB object.

1. The home interface must extend the `javax.ejb.EJBHome` interface which has the following definition:

    ```
    public interface javax.ejb.EJBHome extends java.rmi.Remote {
        public abstract EJBMetaData getEJBMetaData();
        public abstract void remove(Handle handle);
        public abstract void remove(Object primaryKey);
    }
    ```

    The methods in the `EJBHome` interface are implemented by the container. A client can remove an EJB object using the `remove` methods defined in either of its home or remote interfaces.

2. A bean's home interface can also be used to retrieve metadata information about the bean through the `javax.ejb.EJBMetaData` interface or remove the bean instance, given a handle.

3. All `create` methods must throw the following exceptions:

    ■   `javax.ejb.CreateException`

    ■   either `java.rmi.RemoteException` or `javax.ejb.EJBException`

> **Note:** The `deployejb` tool publishes a reference to the home
> object in the database. See the *Oracle8i Java Tools Reference* for a full
> description of `deployejb`.

Example

The following code sample shows a home interface called **EmployeeHome**. The
create method contains no arguments.

```
package employee;

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

public interface EmployeeHome extends EJBHome {
  public Employee create()
       throws CreateException, RemoteException;
}
```

Creating the Exception Class

Some methods in the Employee class can throw the `EmpException` exception. For
an exception to be transported from the object to the client, you need to define a
class for the exception.

The following code defines an exception class and is found in **EmpException.java**.

```
package employee;

public class EmpException extends RemoteException
{
  public EmpException(String msg)
  {
   super(msg);
  }
}
```

Implementing the Bean

The bean contains the business logic for your bean. It implements the following
methods:

- The bean methods declared in the remote interface.

The bean in the example application consists of one class, `EmployeeBean`, that retrieves an employee's information.

- The methods declared in the `SessionBean` or `EntityBean` interface.

- The `ejbCreate` methods that corresponds to the `create` methods declared in the home interface. The container invokes the `ejbCreate` method when the client invokes the corresponding `create` method.

Interface Implemented

Your bean implements the methods within either the `SessionBean` or `Entity-Bean` interface. This example implements the `SessionBean` interface. Basically, a session bean is used for process oriented beans—those beans that perform tasks to achieve an end. Entity beans are complex remote objects that are organized around persistent data. See "Definition of an Entity Bean" on page 4-2 for more information on the differences between the two types of beans.

The session bean implements the `javax.ejb.SessionBean` interface, which has the following definition:

```
public interface javax.ejb.SessionBean extends javax.ejb.EnterpriseBean {
   public abstract void ejbActivate();
   public abstract void ejbPassivate();
   public abstract void ejbRemove();
   public abstract void setSessionContext(SessionContext ctx);
}
```

At a minimum, an EJB must implement the following methods, as specified in the `javax.ejb.SessionBean` interface:

`ejbActivate()`	Implement this as a null method, because it is never called in this release of the EJB server.
`ejbPassivate()`	Implement this as a null method, because it is never called in this release of the server.
`ejbRemove()`	A container invokes this method before it ends the life of the session object. This method performs any required clean-up, for example closing external resources such as file handles.

setSessionContext (SessionContext ctx)	Associate's a bean's instance with context information. The container calls this method after the bean creation. The enterprise bean can store the reference to the context object in an instance variable, for use in transaction management. Beans that manage their own transactions can use the session context to get the transaction context.

Using setSessionContext

This method is used by a session bean instance to retain a reference to its context. Session beans have session contexts that the container maintains and makes available to the beans. The bean may use the methods in the session context to make callback requests to the container.

The container invokes setSessionContext method, after it first instantiates the bean, to enable the bean to retrieve the session context. The container will never call this method from within a transaction context. If the bean does not save the session context at this point, the bean will never gain access to the session context.

When the container calls this method, it passes the reference of the SessionContext object to the bean. The bean can then store the reference for later use. The following example shows the bean saving the session context in the **sessctx** variable.

```
import javax.ejb.*;
import oracle.oas.ejb.*;

public class myBean implements SessionBean {
    SessionContext sessctx;

    void setSessionContext(SessionContext ctx) {
        sessctx = ctx;    // session context is stored in
                          // instance variable
    }
    // other methods in the bean
}
```

The javax.ejb.SessionContext interface has the following definition:

```
public interface SessionContext extends javax.ejb.EJBContext {
    public abstract EJBObject getEJBObject();
}
```

And the javax.ejb.EJBContext interface has the following definition:

```
public interface EJBContext {
    public abstract Properties      getEnvironment();
    public abstract UserTransaction getUserTransaction();
    public abstract boolean         getRollbackOnly();
    public abstract void            setRollbackOnly();
    public abstract boolean         isCallerInRole(Identity);
                                                    // not supported
    public abstract Identity        getCallerIdentity(); // not supported
    public abstract EJBHome         getEJBHome();
}
```

A bean needs the session context when it wants to perform the operations listed in Table 2–1.

Table 2–1   SessionContext operations

Method	Description
getEnvironment()	Get the values of properties for the bean.
getUserTransaction()	Get a transaction context, which allows you to demarcate transactions programmatically. This is only valid for beans that have been designated transactional.
setRollbackOnly()	Set the current transaction so that it cannot be committed.
getRollbackOnly()	Check whether the current transaction has been marked for rollback only.
getEJBHome()	Get the object reference to the bean's corresponding EJBHome (home interface).

Bean Implementation Example

The following code implements methods of a session bean called **EmployeeBean**. The **SessionBean** interface methods are implemented along with the public methods declared in the remote interface.

The JDBC code opens a default connection, which is the standard way that JDBC code that runs on the Oracle8*i* server opens a server-side connection. A JDBC prepared statement is used to prepare the query, which has a WHERE clause. Then the setInt method is used to associate the empNumber input parameter for the getEmployee method with the'?' placeholder in the prepared statement query. This is identical to the JDBC code that you would write in a client application.

```
package employeeServer;
```

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

public class EmployeeBean implements SessionBean {
  SessionContext ctx;

  //implement the bean method, getEmployee
  public EmpRecord getEmployee (int empNumber)
       throws SQLException, RemoteException {

    //create a new employee record
    EmpRecord empRec = new EmpRecord();

   //establish a connection to the database using JDBC
    Connection conn =
      new oracle.jdbc.driver.OracleDriver().defaultConnection();

    //retrieve the employee's information from the database
    PreparedStatement ps =
      conn.prepareStatement("select ename, sal from emp where empno = ?");
    ps.setInt(1, empNumber);
    ResultSet rset = ps.executeQuery();
    if (!rset.next())
        throw new RemoteException("no employee with ID " + empNumber);
    empRec.ename = rset.getString(1);
    empRec.sal = rset.getFloat(2);
    empRec.empno = empNumber;
    ps.close();
    return empRec;
  }

  //implement the SessionBean methods: ejbCreate, ejbActivate,
  // ejbPassivate, ejbRemove and setSessionContext

  //implement ejbCreate, which is called by the container when
  //the Home create is invoked by the client.
  public void ejbCreate() throws CreateException, RemoteException {
    //you can do any initialization for the bean at this point.
    //this particular example does not require any initialization or
    //environment variable retrieval.
  }

  //ejbActivate and ejbPassivate are never called in this release.
```

```
//Both methods should be declared, but be null methods.
public void ejbActivate() {
}
public void ejbPassivate() {
}

//implement anything that needs to be done before the
//bean is destroyed. this would include closing any open
//resources. however, for this example, no open resources need
//to be closed. thus, the method is empty.
public void ejbRemove() {
}

//retreive the session context
public void setSessionContext(SessionContext ctx) {
  this.ctx = ctx;
}
}
```

Developing Your Client Application

All EJB clients perform the following to instantiate a bean, invoke its methods, and destroy the bean:

1. Look up the bean home interface, which is published in the Oracle8*i* database as part of the bean deployment process. Use the Java Naming and Directory Interface (JNDI) to look up the home interface.

2. Create instances of the bean in the server through the home interface. Invoking the `create` method on the home interface causes a new bean to be instantiated. This returns a bean reference to the bean's remote interface.

3. Invoke the methods defined in the remote interface. The container forwards the requests to the instantiated bean.

4. After the bean is no longer needed, invoke the `remove` method to destroy the bean.

These steps are completely illustrated by example in Figure 2–1.

As a quick example, suppose that `EmployeeHome` is a reference that you have obtained to the home interface of a bean called `Employee`. The `Employee` home interface must have at least one `create` method that lets you instantiate the bean. You create a new instance of the bean on the remote server by coding:

```
Context ic = new InitialContext(env);
```

```
EmployeeHome home =
      (EmployeeHome) ic.lookup(serviceURL + objectName);  // lookup the bean
Employee testBean = home.create();   // create a bean instance
```

Then, you would invoke `Employee` methods using the usual syntax

```
testBean.getEmployee(empNumber);
```

Using the getEJBHome Method

When you use the `getEJBHome` method to retrieve the home interface given an object reference, you cannot cast the returned object to the home interface's type. Instead, the returned object is of type `org.omg.CORBA.Object`. Once received, the object is cast to the correct home interface type through the `Helper.narrow` method. The following shows the Hello example retrieve Hello's home interface using JNDI, creating the remote interface, and then later retrieving the home interface again using the getEJBHome interface. Notice that the HelloHomeHelper.narrow method is used to correctly typecast the home interface:

```
HelloHome hello_home = (HelloHome)ic.lookup (serviceURL + objectName);
Hello hello = hello_home.create ();
System.out.println (hello.helloWorld ());

org.omg.CORBA.Object newHome = (org.omg.CORBA.Object) hello.getEJBHome();
HelloHome newHello = HelloHomeHelper.narrow(newHome);
```

Parameter Passing

When you implement an EJB or write the client code that calls EJB methods, you have to be aware of the parameter-passing conventions used with EJBs.

A parameter that you pass to a bean method—or a return value from a bean method—can be any Java type that is serializable. Java primitive types, such as `int`, `double`, are serializable. Any non-remote object that implements the `java.io.Serializable` interface can be passed. A non-remote object passed as a parameter to a bean or returned from a bean is passed by *value*, not by reference. So, for example, if you call a bean method as follows:

```
public class theNumber {
  int x;
}
...
bean.method1(theNumber);
```

then `method1()` in the bean receives a copy of `theNumber`. If the bean changes the value of `theNumber` object on the server, this change is not reflected back to the client, because of pass-by-value semantics.

If the non-remote object is complex—such as a class containing several fields—only the non-static and non-transient fields are copied.

When passing a remote object as a parameter, the stub for the remote object is passed. A remote object passed as a parameter must extend remote interfaces.

The next section demonstrates parameter passing to a bean and remote objects as return values.

A Parameter Object

The `EmployeeBean getEmployee` method returns an `EmpRecord` object, so this object must be defined somewhere in the application. In this example, an `EmpRecord` class is included in the same package as the EJB interfaces.

The class is declared as **public**, and must implement the `java.io.Serializable` interface so that it can be passed back to the client by value, as a serialized remote object. The declaration is as follows:

```
package employee;

public class EmpRecord implements java.io.Serializable {
  public String ename;
  public int empno;
  public double sal;
}
```

> **Note:**   The `java.io.Serializable` interface specifies no methods, it just indicates that the class is serializable. Therefore, there is no need to implement extra methods in the `EmpRecord` class.

The Client Code

This section shows the client code that you can use to send messages to the example bean described above, and get and print results from it. This client code demonstrates how a client:

- locates a remote object such as the bean home interface
- authenticates itself to the server

- activates an instance of the bean

- invokes a method on the bean

Locating Remote Objects

The first step with any remote object implementation, whether it's pure RMI, or EJBs, or CORBA, is to find out how to locate a remote object. To get a remote object reference you must know:

- the name of the object

- where the name server is located

With EJBs, the initial object name is the name of an EJB home interface, and you locate it using the Java Naming and Directory Interface (JNDI). The EJB specification requires that EJB implementations expose a JNDI interface as the means of locating a remote bean.

About JNDI

JNDI is an interface to a naming and directory service. For example, JNDI can serve as an interface to a file system that you can use to look up directories and the files they contain. Or, JNDI can be used as an interface to a naming or directory service, for example a directory protocol such as LDAP.

This section briefly describes JNDI. The EJB specification requires the use of JNDI for locating remote objects by name.

This section of the manual describes only those parts of JNDI that you need to know to write EJB applications for Oracle8*i*. To obtain the complete JNDI API (and SPI) specifications, see `http://www.javasoft.com/products/jndi`.

Sun Microsystems supplies JNDI in the `javax.naming` package, so you must import these classes in your client code:

```
import javax.naming.*;
```

For the Oracle8*i* EJB server, JNDI serves as an interface (SPI driver) to the OMG *CosNaming* service. But you do not have to know all about CosNaming, or even all about JNDI, to write and deploy EJBs for the Oracle8*i* server. To start, all you must know is how to use the JNDI methods used to access permanently-stored home interface objects and how to set up the environment for the JNDI `Context` object.

The remainder of this JNDI section describes the data structures and methods of the `javax.naming` package that you will need to access EJB objects.

Getting the Initial Context

You use JNDI to retrieve a `Context` object. The first `Context` object that you receive is bound to the root naming context of the Oracle8*i* publishing context. EJB home interfaces are published in the database, and are arranged in a manner similar to a file system hierarchy. See *Oracle8i Java Tools Reference* for more details about the `publish` tool.

You get the root naming context by creating a new JNDI `InitialContext`, as follows:

```
Context initialContext = new InitialContext(environment);
```

The `environment` parameter is a Java hashtable. Table 2–2 contains the six properties that you can set in the hashtable that are passed to the `javax.naming.Context.`

Table 2–2   Context Properties

Property	Purpose
`javax.naming.Context.` `URL_PKG_PREFIXES`	The environment property that specifies the list of package prefixes to use when loading in URL context factories. You must use the value `"oracle.aurora.jndi"` for this property.

`javax.naming.Context.`
`SECURITY_AUTHENTICATION`
	The type of security for the database connection. The possible values are:
	
	■　`oracle.aurora.sess_iiop.ServiceCtx.`
	　　`NON_SSL_LOGIN`
	
	■　`oracle.aurora.sess_iiop.ServiceCtx.`
	　　`SSL_CREDENTIAL`
	
	■　`oracle.aurora.sess_iiop.ServiceCtx.`
	　　`SSL_LOGIN`
	
	■　`oracle.aurora.sess_iiop.ServiceCtx.`
	　　`SSL_CLIENT_AUTH`

`javax.naming.Context.` `SECURITY_PRINCIPAL`	The Oracle8*i* username, for example "SCOTT".
`javax.naming.Context.` `SECURITY_CREDENTIALS`	The password for the username, for example "TIGER".
`oracle.aurora.sess_iiop.` `ServiceCtx.SECURITY_ROLE`	An optional property that establishes a database role for the connection. For example, use the string "CLERK" to connect with the CLERK role.

Table 2–2 Context Properties

Property	Purpose
`oracle.aurora.sess_iiop.` `ServiceCtx.SSL_VERSION`	The client-side SSL version number.

See Chapter 5, "JNDI Connections and Session IIOP Service", for more information about JNDI and connecting to an Oracle8*i* instance.

Getting the Home Interface Object

Once you have the initial references context, you can invoke its methods to get a reference to an EJB home interface. To do this, you must know the published full pathname of the object, the host system where the object is located, the IIOP port for the listener on that system, and the database system identifier (SID). When you obtain this information—for example, from the EJB deployer—construct a URL using the following syntax:

```
<service_name>://<hostname>:<iiop_listener_port>:<SID>/<published_obj_name>
```

For example, to get a reference to the home interface for a bean that has been published as `/test/myEmployee`, on the system whose TCP/IP hostname is `myHost`, the listener IIOP port is 2481, and the system identifier (SID) is `ORCL`, construct the URL as follows:

```
sess_iiop://myHost:2481:ORCL/test/myEmployee
```

The listener port for IIOP requests is configured in the *listener.ora* file. The default for Oracle8*i* is 2481. See the *Net8 Administrator's Guide* for more information about IIOP configuration information. See also Chapter 5, "JNDI Connections and Session IIOP Service" for more information about IIOP connections.

You get the home interface using the `lookup` method on the initial context, passing the URL as the parameter. For example, if the home interface's published name is `/test/myEmployee`, you would code:

```
...
String ejbURL = "sess_iiop://localhost:2481:ORCL/test/myEmployee";
Hashtable env = new Hashtable();
env.put(javax.naming.Context.URL_PKG_PREFIXES, "oracle.aurora.jndi");
// Tell sess_iiop who the user is
env.put(Context.SECURITY_PRINCIPAL, "SCOTT");
// Tell sess_iiop what the password is
env.put(Context.SECURITY_CREDENTIALS, "TIGER");
```

```
// Tell sess_iiop to use non-SSL login authentication
env.put(Context.SECURITY_AUTHENTICATION, ServiceCtx.NON_SSL_LOGIN);
// Lookup the URL
EmployeeHome home = null;
Context ic = new InitialContext(env);
home = (EmployeeHome) ic.lookup(ejbURL);
...
```

Invoking EJB Methods

Once you have the home interface for the bean, you can invoke one of the bean's `create` methods to instantiate a bean. See Chapter 5, "JNDI Connections and Session IIOP Service" for information about granting execution rights. For example:

```
Employee testBean = home.create();
```

Then you can invoke the EJB's methods in the normal way:

```
int empNumber = 7499;
EmpRecord empRec = testBean.getEmployee(empNumber);
```

Here is the complete code for the client application:

```
import employee.Employee;
import employee.EmployeeHome;
import employee.EmpRecord;

import oracle.aurora.jndi.sess_iiop.ServiceCtx;

import javax.naming.Context;
import javax.naming.InitialContext;
import java.util.Hashtable;


public class Client {

  public static void main (String [] args) throws Exception {

    String serviceURL = "sess_iiop://localhost:2481:ORCL";
    String objectName = "/test/myEmployee";
    int empNumber = 7499;    // ALLEN
    Hashtable env = new Hashtable();

    env.put(Context.URL_PKG_PREFIXES, "oracle.aurora.jndi");
    env.put(Context.SECURITY_PRINCIPAL, "scott");
    env.put(Context.SECURITY_CREDENTIALS, "tiger");
```

```
        env.put(Context.SECURITY_AUTHENTICATION, ServiceCtx.NON_SSL_LOGIN);

        Context ic = new InitialContext(env);

        EmployeeHome home =
          (EmployeeHome) ic.lookup(serviceURL + objectName);  // lookup the bean
        Employee testBean = home.create();    // create a bean instance
        EmpRecord empRec =
                testBean.getEmployee(empNumber); // get the data and print it
        System.out.println("Employee name is " + empRec.ename);
        System.out.println("Employee sal is  " + empRec.sal);
  }
}
```

Deploying an EJB

The EJB deployment process consists of the following steps:

1. Get the beans from the EJB developer. In the typical case, you compile the beans and put the beans and their accompanying classes, including the home and remote interfaces and any classes dependent on the bean into a JAR file—one JAR file for each bean.

2. Develop the EJB deployment descriptor for each bean.

3. Create the Oracle-specific deployment descriptor.

4. Run the `deployejb` tool, which:

   a. reads the deployment descriptor and the bean JAR file

   b. maps the logical names defined in the EJB deployment descriptor to existing JNDI names and database tables

   c. loads the bean classes into the Oracle8*i* database

   d. publishes the bean home interface

5. Make sure that the application developer has the information necessary about the bean remote interface and the name of the published beans.

> **Note:** The deployment process is the same for *i*AS as it is for
> Oracle8*i* JServer. You simply have to provide the correct server for
> the deployment. You can choose to either deploy within the
> middle-tier or the database backend. If deploying to the
> middle-tier, you must have either an *i*Cache or Oracle8*i* JVM
> installed in this tier. Then, pass the URL for the installed *i*Cache or
> JVM to the command-line tools.

Deployment Steps

The format used to package EJBs is defined by the EJB specification. This section
describes the steps that the EJB developer and the EJB deployer take to compile,
package, and deploy an EJB. Oracle8*i* supplies a deployment tool, deployejb, that
automatically performs most of the steps necessary to deploy an EJB. The
deployejb tool deploys only one bean at a time. This tool is described in *Oracle8i
Java Tools Reference.*

To deploy an EJB, follow these four steps:

1. Compile the code for the bean. This includes:

    ■   the home interface

    ■   the remote interface

    ■   the bean implementation

    ■   all Java source files dependent on the bean implementation class (this
        dependency is normally taken care of by the Java compiler)

        Use the standard client-side Java compiler to compile the bean source files.
        A bean typically consists of one or more Java source files and might have
        associated resource files.

        Oracle8*i* supports the Sun Microsystems Java Developer's Kit compiler
        versions 1.1.6 or 1.2. Alternatively, you might be able to use another
        JCK-tested Java compiler to create EJBs to run in the Oracle8*i* server.

2. Write the XML deployment descriptor for the EJB. See Programming
    Restrictions on page 2-28 for specific information about creating deployment
    descriptors.

3. Write the Oracle deployment mapping file.

4. Create a JAR file containing the interface and implementation class files—the
    home interface, the remote interface, and the bean implementation—for the

bean. If you have other dependent classes and resource files, it is better to create a separate JAR file for these. The `deployejb` tool uses this JAR file as an input file.

5. Call the `deployejb` tool (see *Oracle8i Java Tools Reference* for information on `deployejb`) to load and publish the JAR'd bean.

Write the Deployment Descriptor

With EJB 1.1, the deployment descriptor is now defined using XML. Sun Microsystems's provides the DTD file, which describes the required entries for defining the bean and application. The deployment descriptor was designed to contain logical names—that is, names that do not necessarily match the true name of the object loaded in JServer. These logical names are mapped to existing names through a companion deployment file—the Oracle deployment mapping file. The Oracle deployment mapping file can map the logical bean name to an existing JNDI name and map any container-managed entity bean fields to existing database columns.

Alternatively, if you use the actual JNDI and database column names within the XML deployment file, the Oracle deployment map file will be created for you automatically by `deployejb`.

Note: Since this chapter discusses session beans, the only fields discussed here will pertain to session beans. For a full description of the XML elements, see either Appendix A, "XML Deployment Descriptors" or the "Deployment Descriptor" chapter within the EJB 1.1 specification, located on `http://www.javasoft.com`.

The following example shows the sections necessary for the Employee example. This example uses logical names within the XML deployment descriptor that map to JNDI names within the Oracle deployment map file.

Example 2–1   XML Deployment Descriptor for Employee Bean

```
<?xml version="1.0"?>
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems Inc.//DTD Enterprise JavaBeans 1.1
   //EN" "ejb-jar.dtd">
<ejb-jar>
   <enterprise-beans>
      <session>
         <description>Session Bean Employee Example</description>
```

```
                        <ejb-name>Employee</ejb-name>
                        <home>employee.EmployeeHome</home>
                        <remote>employee.Employee</remote>
                        <ejb-class>employee.EmployeeBean</ejb-class>
                        <session-type>Stateful</session-type>
                        <transaction-type>Container</transaction-type>
                    </session>
                </enterprise-beans>
                <assembly-descriptor>
                    <security-role>
                        <description>Public</description>
                        <role-name>PUBLIC</role-name>
                    </security-role>
                    <method-permission>
                        <description>public methods</description>
                        <role-name>PUBLIC</role-name>
                        <method>
                            <ejb-name>EmployeeBean</ejb-name>
                            <method-name>*</method-name>
                        </method>
                    </method-permission>
                    <container-transaction>
                        <description>no description</description>
                        <method>
                            <ejb-name>EmployeeBean</ejb-name>
                            <method-name>*</method-name>
                        </method>
                        <trans-attribute>Supports</trans-attribute>
                    </container-transaction>
                </assembly-descriptor>
            </ejb-jar>
```

The following sections describes each of the pieces of the XML deployment descriptor:

XML Version Number

```
<?xml version="1.0"?>
```

DTD Filename

```
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems Inc.//DTD Enterprise JavaBeans 1.1
//EN" "ejb-jar.dtd">
```

JAR file

The first element to be declared is the `<ejb-jar>` element. Within this element, you define two sections: the `<enterprise-beans>` section and the `<assembly-descriptor>` section. The `<enterprise-beans>` section defines the bean. The `<assembly-descriptor>` section defines the application's security and transaction attributes.

```
<ejb-jar>                           //Start of JAR file descriptor
 <enterprise-beans>                 //EJB Descriptor section
 ...                                //Bean definition
 </enterprise-beans>
 <assembly-descriptor>              //Application Descriptor section
 ...                                //Transaction and security definition
 </assembly-descriptor>
</ejb-jar>
```

Enterprise JavaBeans Element

The beans are described within the `<enterprise-beans>` element. This element contains information such as the type of bean, the home interface name, the remote interface name, and the bean class name.

The following segment shows the following:

- The bean is a session bean, denoted by the `<session>` element.

- The logical name for this bean is `Employee`, defined within the `<ejb-name>` element.

- The home, remote, and bean class names are within the `employee` package and are `EmployeeHome`, `Employee`, and `EmployeeBean` respectively.

- All session beans within JServer are stateful.

- The transaction is managed by the container, not by the bean.

```
<enterprise-beans>
   <session>
      <description>Session Bean Employee Example</description>
      <ejb-name>Employee</ejb-name>
      <home>employee.EmployeeHome</home>
      <remote>employee.Employee</remote>
      <ejb-class>employee.EmployeeBean</ejb-class>
      <session-type>Stateful</session-type>
      <transaction-type>Container</transaction-type>
   </session>
</enterprise-beans>
```

Assembly Descriptor Element

The `<assembly-descriptor>` element describes the security and transaction attributes for the application.

- For security, you must define the roles used within this application in the `<security-role>` element. These roles are assigned to certain methods within the bean within the `<method-permission>` element. In this example, the Employee bean allows all methods to be accessed within PUBLIC.

- For transactions, you define the type of transaction support necessary for each method within the `<container-transaction>` element. In this example, all methods require the Supports transactional attribute.

```
<assembly-descriptor>
   <security-role>
      <description>Public</description>
      <role-name>PUBLIC</role-name>
   </security-role>
   <method-permission>
      <description>public methods</description>
      <role-name>PUBLIC</role-name>
      <method>
         <ejb-name>EmployeeBean</ejb-name>
         <method-name>*</method-name>
      </method>
   </method-permission>
   <container-transaction>
      <description>no description</description>
      <method>
         <ejb-name>EmployeeBean</ejb-name>
         <method-name>*</method-name>
      </method>
      <trans-attribute>Supports</trans-attribute>
   </container-transaction>
</assembly-descriptor>
```

Create the Oracle Deployment Map File

If you had declared the true JNDI name within `<ejb-name>`, you do not have to create an Oracle deployment map file. This example used a logical name, `Employee`, within the XML deployment file. In this example, the JNDI name for `Employee` is `/test/EmployeeBean`.

In the Oracle deployment map file, you have the following structure:

1. Headers for XML version and DTD file.

2. EJB logical name mapped to JNDI name.

   In this example, the logical name defined within `<ejb-name>`—`Employee`—is mapped to `"/test/EmployeeBean"` within the `<jndi-name>` element.

3. The identity that this bean will run under.

   a. The `<run-as>` element defines the identity that the bean runs under within the `<mode>` element. The possible values can be the following:

   ■ CLIENT_IDENTITY—The bean runs as the client.

   ■ SPECIFIED_IDENTITY—The bean runs as the specified identity. This identity is defined in a `<security-role>` element. See Appendix A, "XML Deployment Descriptors" for details.

   ■ SYSTEM_IDENTITY—The bean runs as the DBA or ROOT for the server. For an Oracle8*i* server, the bean would run as SYS.

   b. The `<method>` element under the `<run-as>` element defines the methods that run under the `<mode>` identity.

   In this example, all methods within the `EmployeeBean` run as the client.

Example 2–2   Oracle Deployment Map File for Employee

```
<?xml version="1.0"?>
<!DOCTYPE oracle-descriptor PUBLIC "-//Oracle Corporation.//DTD Oracle 1.1//EN"
"oracle-ejb-jar.dtd">
<oracle-descriptor>
  <mappings>
    <ejb-mapping>
      <ejb-name>Employee</ejb-name>
      <jndi-name>/test/EmployeeBean</jndi-name>
    </ejb-mapping>
  </mappings>
  <run-as>
    <description>no description</description>
    <mode>CLIENT_IDENTITY</mode>
    <method>
      <ejb-name>EmployeeBean</ejb-name>
      <method-name>*</method-name>
     </method>
  </run-as>
</oracle-descriptor>
```

Create a JAR File

The `deployejb` command-line tool creates a JAR file to use on the client side to access the bean.

```
deployejb -user scott -password tiger -service sess_iiop://dbserver:2481:orcl \
  -descriptor employee.xml -oracledescriptor oracle_employee.xml \
  -temp /tmp/ejb -generated empClient.jar employee.jar
```

One of the tasks that the `deployejb` tool performs is to create a JAR file with the required stubs and skeletons, which is used by the client at execution time. Unless this JAR file name is specified in the `<ejb-client-jar>` element in the XML deployment descriptor, the default name for this JAR file is `server_generated.jar`. This JAR file must be included in the `CLASSPATH` for client execution.

> **Note:**   If your application uses any circular method references between beans, you will have to load the referred bean first using `loadjava`, then invoke `deployejb` for the JAR file. A circular method reference is when any method in bean A references a method in bean B and any method within bean B references any method in bean A. Even though they are different methods, the loading resolution fails when it cannot find the referenced bean already loaded within the database. If you load bean B using `loadjava`, the load resolution succeeds.

Publish the Home Interface

A bean provider must make the bean's home interface available for JNDI lookup so that clients can find and activate the bean. However, when you create the JAR file with the `deployejb` command-line tool, this tool publishes the bean in the JNDI namespace under the `<jndi-name>` element name that you specify in the Oracle deployment mapping file.

Dropping an EJB

Drop an EJB from the database by following these steps:

- Run the `dropjava` tool to delete the classes from the database. Provide the original bean JAR file that contains the class files for the bean.

- Use the session shell tool to remove the bean home interface name from the published object name space.

See *Oracle8i Java Tools Reference* for documentation of the `dropjava` and session shell tools.

Run the Example

To run this example, execute the client class using the client-side JVM. For this example, you must set the CLASSPATH for the `java` command to include:

- the standard Java library archive (`classes.zip`)
- any class files the client ORB uses, such as those in VisiBroker for Java `vbjapp.jar` and `vbjorb.jar`
- the Oracle8*i*-supplied JAR files: `mts.jar` and `aurora_client.jar`

If you are using JDBC, include one of the following JAR files:

- `classes111.zip` for JDBC 1.1 support
- `classes12.zip` for JDBC 1.2 support

If you are using SSL, include one of the following JAR files:

- `javax-ssl-1_1.jar` and `jssl-1_1.jar` for SSL 1.1 support
- `javax-ssl-1_2.jar` and `jssl-1_2.jar` for SSL 1.2 support

You can locate these libraries in the `lib` and `jlib` directories under the Oracle home location in your installation.

The following invocation of the JDK `java` command runs this example.

> **Note:** The UNIX shell variable $ORACLE_HOME might be represented as %ORACLE_HOME% on Windows NT. The JDK_HOME is the installation location of the Java Development Kit (JDK).

```
% java -classpath .:$(ORACLE_HOME)/lib/aurora_client.jar
  :$(ORACLE_HOME)/lib/mts.jar
  :$(ORACLE_HOME)/jdbc/lib/classes111.zip:
  $(ORACLE_HOME)/lib/vbjorb.jar:
  $(ORACLE_HOME)/lib/vbjapp.jar:$(JDK_HOME)/lib/classes.zip
    Client
    sess_iiop://localhost:2481:ORCL
    /test/myEmployee
    scott tiger
```

Programming Restrictions

The EJB 1.1 specification contains the following programming restrictions, which you must follow when implementing the methods of an EJB class:

- An EJB should not start new threads nor attempt to terminate the running thread. In the current release if an EJB starts a new thread no exception is thrown, but the application behavior becomes unpredictable due to interactions with local thread objects in the ORB.

- An EJB is not allowed to use thread synchronization primitives.

- An EJB can only use the `javax.Transaction.UserTransaction` interface to demarcate transactions.

- An EJB is not allowed to change its `java.security.Identity`. Any attempt to do so results in the `java.security.SecurityException` being thrown.

- An EJB is not allowed to use JDBC commit and rollback methods nor to issue direct SQL commit or rollback commands using SQLJ or JDBC.

The 1.1 EJB specification states that "an EJB is not allowed to use read/write static fields. Using read-only static fields is allowed. Therefore, all static fields must be declared as final." This is *not* a restriction for Oracle8*i*.

Debugging Techniques

Until Java IDEs and JVMs support remote debugging, you can adopt several techniques for debugging your CORBA client and server code.

1. Use JDeveloper for debugging any Java applications. JDeveloper has provided a user interface that utilizes JServer's debugging facilities. You can successfully debug an object loaded into the database by using JDeveloper's debugger. See the JDeveloper documentation for instructions.

2. Use a prepublished `DebugAgent` object for debugging objects executing on a server. See "Using a Debug Agent for Debugging Server Applications" on page 2-29 for more information.

3. Perform stand-alone ORB debugging using one machine and ORB tracing.

   Debug by placing both the client and server in a single address space in a single process. Use of an IDE for client or server debugging is optional, though highly desirable.

4. Use Oracle8*i* trace files.

In the client, the output of `System.out.println()` goes to the screen.
However, in the Oracle8*i* ORB, all messages are directed to the server trace files.
The directory for trace files is a parameter specified in the database initialization
file. Assuming a default install of the product into a directory symbolically
named $ORACLE_HOME, then the trace file would appear as

```
${ORACLE_HOME}/admin/<SID>/bdump/ORCL_s000x_xxx.trc
```

where ORCL is the SID, and x_xxx represents a process ID number. Do not
delete trace files after the Oracle instance has been started, or no output is
written to a trace file. If you do delete trace files, stop and then restart
the server.

5. Use a single Oracle MTS server.

    For debugging only, set the MTS_SERVERS parameter in your INIT*SID*.ORA
    file to MTS_SERVERS = 1, and set the MTS_MAX_SERVERS to 1. Having
    multiple MTS servers active means that a trace file is opened for each server
    process, and, thus, the messages get spread out over several trace files, as
    objects get activated in more than one session.

6. Use the printback example to redirect System.out. This example is available in
    the demo directory, `demo/examples/corba/basic/printback`.

Using a Debug Agent for Debugging Server Applications

The procedure for setting up your debugging environment is discussed fully in the
Oracle8i Java Developer's Guide. However, it discusses starting the debug agent
using a DBMS_JAVA procedures. Within a CORBA application, you can start, stop,
and restart the debug agent using the `oracle.aurora.debug.DebugAgent` class
methods. These methods perform exactly as their DBMS_JAVA counterparts
perform.

```
public void start( java.lang.String host, int port, long timeout_seconds)
                throws DebugAgentError
public void stop() throws DebugAgentError
public void restart(long timeout) throws DebugAgentError
```

Example 2–3   Starting a DebugAgent on the Server

The following example shows how to debug an object that exists on the server. First,
you need to start a debug proxy through the `debugproxy` command-line tool. This
example informs the `debugproxy` to start up the `jdb` debugger when contacted by
the debug agent.

Once you execute this command, start your client, which will lookup the intended object to be debugged, lookup the DebugAgent that is prepublished as "/etc/debugagent", and start up the DebugAgent.

Once the DebugAgent starts, the debugproxy starts up the jdb debugger and allows you to set your breakpoints. Since you have a specified amount of time before the DebugAgent times out, the first thing you should do is suspend all threads. Then, set all of your breakpoints before resuming. This suspends the timeout until you are ready to execute.

proxy window on tstHost

```
% debugproxy -port 2286 start jdb -password
. (wait until a debug agent starts up and
.  contact this proxy... when it does, jdb
.  starts up automatically and you can set
.  breakpoints and debug the object, as follows:)
> suspend
> load SCOTT:Bank
> stop in Bank:updateAccount
> resume
> ...
```

client code

```
main( ... )
{
 //retrieve the object that you want to debug
 Bank b = (Bank)ic.lookup(sessURL + "/test/Bank");
 //lookup debugagent from JNDI
 DebugAgent dbagt = (DebugAgent)ic.lookup(svcURL + "/etc/debugagent");
 //start the debug agent and give the proxy host, port, and a timeout
 dbagt.start("tstHost",2286,30);
 ...
 //execute methods within Bank)
 ...
 //stop the agent when you want to
 dbagt.stop();
 //restart the agent when you want to
 dbagt.restart(30);
}
```

3

Configuring IIOP Applications

Configuring IIOP-based applications, whether EJB or CORBA applications, involves configuring the appropriate listener and MTS server for session-based IIOP communications. The process for configuring IIOP-based applications involves both database and network configuration. These elements are discussed in the sections below:

- Overview
- Oracle8i Typical or Minimal Installation
- Oracle8i Custom Installation
- Manual Install and Configuration
- Advanced Configuration Options

Overview

Clients access EJB and CORBA applications in the database over an Internet Inter-Orb Protocol (IIOP) connection. IIOP is an implementation of General Inter-Orb Protocol (GIOP) over TCP/IP. All IIOP connections for CORBA or EJB clients that communicate with the database must have IIOP configured on the database and within the Net8 listener, unless you have the following scenario:

- the listener that you want to connect to is on the same node as the database

- the database that you want to use has configured a generic dispatcher

- you use the dynamic registration tool described in "Dynamic Listener Endpoint Registration" on page 3-15 to enable the listener to manage IIOP requests

Otherwise, you must configure the database and the listener, as follows:

Entity	Description	Configuration Tool
Database	To support an IIOP connection, you must configure the database for GIOP in MTS mode.	Configure the database MTS dispatchers for IIOP through the Database Configuration Assistant. This tool is started under the covers by the Typical and Custom Oracle8*i* install.
Net8 Listener	To support an IIOP connection, you must configure the Net8 listener to accept an IIOP connection over defined ports 2481 or 2482.	Configure the Net8 listener for IIOP through the Net8 Assistant.

The database supports incoming requests through a presentation. The presentation protocol is responsible for making sure data is represented in a format the application and session layers can accommodate. Both the listener and the dispatcher accept incoming network requests based upon the presentation that is configured. For IIOP, you configure a GIOP presentation.

> **Note:** For security concerns, you must decide if your IIOP connection will be Security Socket Layer (SSL) enabled.
>
> - See "Using the Secure Socket Layer" on page 6-3 for information on SSL.
>
> - See "Configuring SSL for EJB and CORBA" on page 3-17 for information on how to configure SSL.

The configuration for an IIOP connection can be handled in one of three ways:

- Oracle8i Typical or Minimal Installation—If you choose the Typical or Minimal Oracle8*i* installation, you receive configuration for session-based, non-SSL IIOP connections for both the database and the listener.

- Oracle8i Custom Installation—If you choose the JServer option within a "Custom" Oracle8*i* installation, you receive configuration for session-based, non-SSL IIOP connections for the database. You have to invoke the Net8 Assistant to configure IIOP for the listener.

- Manual Install and Configuration—If you install JServer by invoking the `initjvm.sql` script, you must manually configure your IIOP connection. All configuration is done manually either by invoking the Database Configuration and Net8 Assistants directly or by editing the various initialization parameter files.

Oracle8*i* Typical or Minimal Installation

During a Typical installation of the server, JServer is installed and configured for you. You automatically receive configuration for an MTS database with session-based IIOP connections through the listener using non-SSL TCP/IP.

After the typical install is complete, the following line is added to your database initialization file:

```
mts_dispatchers="(protocol=tcp)(presentation=oracle.aurora.server.SGiopServer)"
```

If, instead, you installed the Advanced Security Option and you want the SSL-based TCP/IP connection, edit your database initialization file to remove the hash mark (#) from the following line:

```
mts_dispatchers="(protocol=tcps)(presentation=oracle.aurora.server.SGiopServer)"
```

> **Note:** The (`protocol=tcps`) attribute identifies the connection as SSL-enabled.

In addition, the listener is configured for IIOP. The following is placed within your `listener.ora` file:

```
listener=
        (description_list =
             (description=
                  (address=(...)
                  (protocol_stack=
                       (presentation=GIOP)
                       (session=RAW)
                  )
             )
        )
```

After configuration, the client directs its request to a URL that includes the host and port, which identifies the listener, and either the SID or database service name, which identifies the database. The following shows the syntax for this request:

```
session_iiop://<hostname>/:<portnumber>/:<SID | service_name>
```

Oracle8*i* Custom Installation

If, within a Custom install, you choose the JServer option (as shown in Figure 3–1), the Database Configuration Assistant configures an MTS database for session-based IIOP connections, using non-SSL TCP/IP.

> **Note:** If you choose the Typical or Minimal options within the custom choices, your configuration is the same as defined in "Oracle8i Typical or Minimal Installation" on page 3-3.

Figure 3–1   Choosing the JServer option

This places the following line within your database initialization file:

```
mts_dispatchers="(protocol=tcp)(presentation=oracle.aurora.server.SGiopServer)"
```

If, instead, you installed the Advanced Security Option and you want the SSL-based TCP/IP connection, edit your database initialization file to remove the hash mark (#) from the following line:

```
mts_dispatchers="(protocol=tcps)(presentation=oracle.aurora.server.SGiopServer)"
```

> **Note:** The (`protocol=tcps`) attribute identifies the connection as SSL-enabled.

After the installation is complete, you must bring up the Net8 Assistant to configure the listener for IIOP connections.

Net8 Assistant

The Net8 Assistant can modify any of the listener settings. The following is a brief description of the task you must do to configure the listener through the Net8 Assistant. For a fuller explanation, see the *Net8 Administrator's Guide.*

1. Start Net8 Assistant

   - On UNIX, run `netasst` at `$ORACLE_HOME/bin`.

   - On Windows NT, choose Start > Programs > Oracle - *HOME_NAME* > Network Administration > Net8 Assistant.

2. In the navigator pane, expand Local > Listeners.

   This brings you to the listener location panel, as shown in Figure 3–2.

Figure 3–2   IIOP Listening Port Configuration
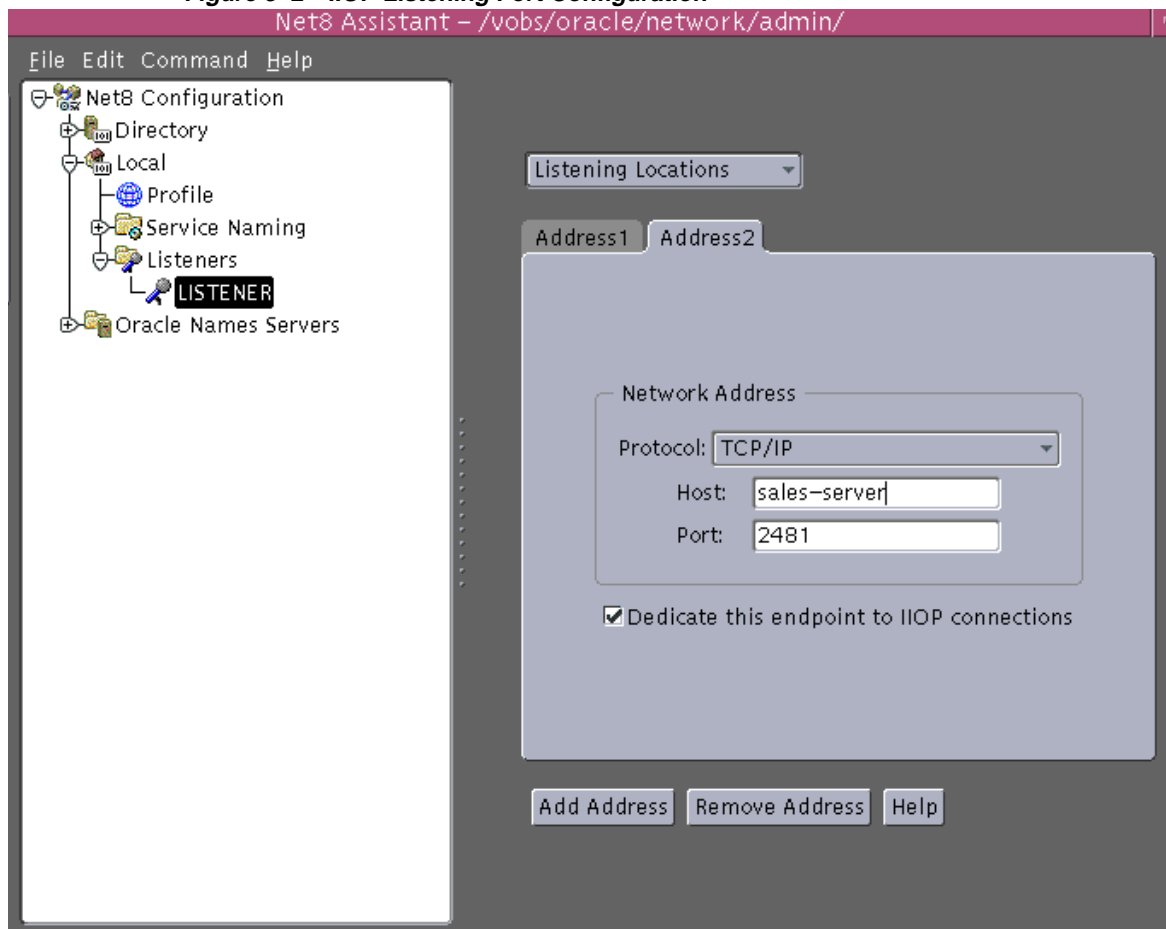
3.   Select a listener.

4.   From the list in the right pane, select Listening Locations.

5.   Select the TCP/IP or TCP/IP with SSL protocol from the Protocol list.

6.   Enter the host name of the database in the Host field.

7.   Enter port 2481 in the Port field if the chosen protocol is TCP/IP, or enter port 2482 if the chosen protocol is TCP/IP with SSL.

8.   Click "Dedicate this endpoint to IIOP connections".

9.   Choose File > Save Network Configuration.

This places the following within your `listener.ora` file:

```
listener=
    (description_list =
      (description=
          (address=(protocol=tcp)(host=sales-server)(port=2481))))
          (protocol_stack=
              (presentation=GIOP)
              (session=RAW)
          )
      )
    )
```

After configuration, the client directs its request to a URL that includes the host and port, which identifies the listener, and either the SID or database service name, which identifies the database. The following shows the syntax for this request:

```
session_iiop://<hostname>/:<portnumber>/:<SID | service_name>
```

Manual Install and Configuration

If you did not install JServer through either the Typical or Custom install options, you can add JServer to an existing database with the `initjvm.sql` script. See the *Oracle8i Java Developer's Guide* for more information on this script.

Once you have installed JServer, you can configure your IIOP connections either through the tools—Database Configuration and Net8 Assistants—or by manually editing the initialization files.

Configuring Through Tools

1. Configure the database for IIOP through the Database Configuration Assistant. To start up the Database Configuration Assistant, do the following:

   - On UNIX, run `dbassist` at `$ORACLE_HOME/bin`.

   - On Windows NT, choose Start > Programs > Oracle - *HOME_NAME* > Database Administration > Database Configuration Assistant.

   After starting up the Database Configuration Assistant, choose the JServer option. For information on what this does to your initialization files, see "Oracle8i Custom Installation" on page 3-4

2. Configure the listener for IIOP through the Net8 Assistant. These steps are described in "Net8 Assistant" on page 3-6.

Configuring Through Editing Initialization Files

The presentation layer within the database identifies the type of connection your client is using to access the database. To identify the GIOP presentation, you use `oracle.aurora.server.SGiopServer`, which is the configuration for session-based IIOP connections. EJB and CORBA applications can activate objects within multiple sessions and are not limited to objects within the single session that the client initiated. These connections identify both a session and the standard IIOP semantics.

To configure an IIOP connection, you specify the GIOP presentation in the following initialization files:

1. Configure the IIOP connection in the database initialization file—You configure the PRESENTATION attribute of the MTS_DISPATCHERS parameter.

    This section describes only the PRESENTATION attribute for the MTS_DISPATCHERS parameter. For a full description of MTS configuration, see the *Net8 Administrator's Guide*.

2. Configure the Net8 listener for IIOP connections.

Both steps are described in more detail below.

1. Configure the IIOP Connection in the Database Initialization File

To configure an IIOP connection within the database, you can manually edit the database initialization file.

The following is the syntax for the MTS_DISPATCHERS parameter:

```
mts_dispatchers="(protocol=tcp | tcps)
        (presentation=oracle.aurora.server.SGiopServer)"
```

The attributes for MTS_DISPATCHER are described below:

Attribute	Description
PROTOCOL (PRO or PROT)	Specifies the TCP/IP or TCP/IP with SSL protocol, which the dispatcher will generate a listening end point for.
	Valid values: TCP (for TCP/IP) or TCPS (for TCP/IP with SSL)

Attribute	Description
PRESENTATION (PRE or PRES)	Enables support for GIOP. Supply the following value for a GIOP presentation:
	■ `oracle.aurora.server.SGiopServer` for session-based GIOP connections. This presentation is valid for TCP/IP and TCP/IP with SSL.

> **Note:** If you configure several MTS_DISPATCHERS within your database initialization file, each MTS definition must follow each other. You should not define any other configuration parameters between the MTS_DISPATCHER definitions.

For example, to configure MTS for session-based IIOP connections through the listener using non-SSL TCP/IP, add the following within your database initialization file:

```
mts_dispatchers="(protocol=tcp)(presentation=oracle.aurora.server.SGiopServer)"
```

2. Configure a Listener for the Incoming Connection

Each listener is configured to listen on a well-known port number, and the client communicates with the listener using this port number. To support CORBA and EJB, the listener must be configured to listen for IIOP clients on either ports 2481 or 2482.

You can either use the Net8 Assistant to configure your listener or manually configure the listener within the `listener.ora` file. Oracle recommends that you use the Net8 Assistant. See "Net8 Assistant" on page 3-6 for information on the Net8 Assistant.

To configure the listener manually, you must modify the listener's DESCRIPTION parameter within the `listener.ora` file.

Modify the LISTENER.ORA DESCRIPTION Parameter You must configure the listener with a GIOP listening address. The following example configures a GIOP presentation for non-SSL TCP/IP with port number 2481. You use port 2481 for non-SSL and port 2482 for SSL.

For GIOP, the PROTOCOL_STACK parameter is added to the DESCRIPTION when configuring an IIOP connection to `sales-server`:

```
listener=
  (description_list=
   (description=
     (address=(protocol=tcp)(host=sales-server)(port=2481))
     (protocol_stack=
         (presentation=giop)
         (session=raw))))
```

The following table gives the definition for each of the GIOP parameters:

Attribute	Description
PROTOCOL_STACK	Identifies the presentation and session layer information for a connection.
(PRESENTATION=GIOP)	Identifies a presentation of GIOP for IIOP clients. GIOP supports `oracle.aurora.server.SGiopServer` using TCP/IP.
(SESSION=RAW)	Identifies the session layer. There is no specific session-layer for IIOP clients.
(ADDRESS=...)	Specifies a listening address that uses TCP/IP on either port 2481 for non-SSL or port 2482 for SSL. If non-SSL, the protocol should be TCP; for SSL, the protocol should be defined as TCPS.

Advanced Configuration Options

- Database Listeners and Dispatchers

- Dynamic Listener Endpoint Registration

- Direct Dispatcher Connection

- Configuring SSL for EJB and CORBA

Database Listeners and Dispatchers

Figure 3–3 shows the interaction between the listener and the dispatchers, and also illustrates how an Oracle8*i* ORB session is activated.

Figure 3–3   Listener/Dispatcher Interaction
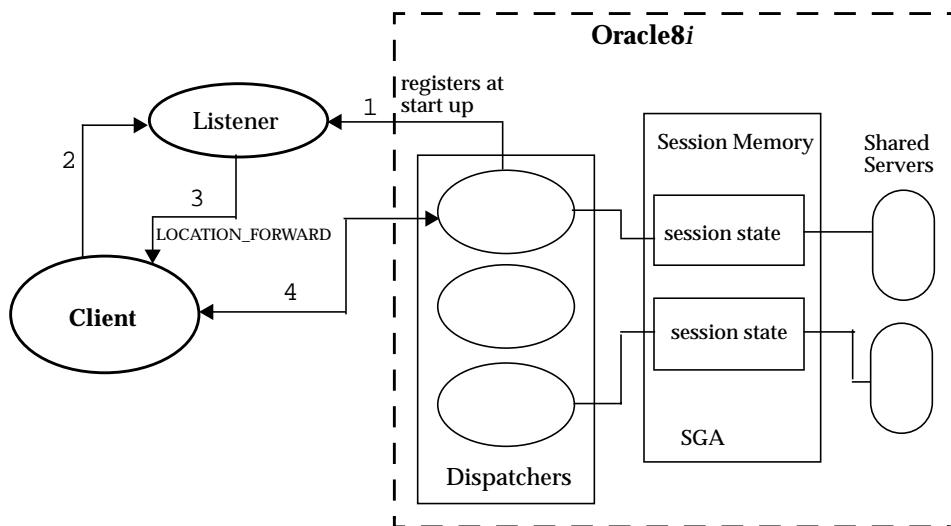
1. Upon database startup, the dispatcher registers itself with the listener.

2. The client invokes a method, giving the listener's URL address as the destination.

3. The listener sends back a LOCATION_FORWARD response to the client's ORB layer informing it of the dispatcher's address. This redirects the request to the appropriate dispatcher.

> **Note:**   The client is unaware of the redirection logic, which is performed by the ORB runtime layer that supports the client.

4. The underlying ORB runtime layer resends the initial request to the dispatcher. All future method invocations are directed to the dispatcher. The listener is no longer a part of the communication.

The incoming request is examined by the shared server services to see if the request is for an existing session. If so, the request is forwarded to the indicated session. If not, the service creates a new database session for the request and activates the ORB in the session. This session is very similar to the database sessions created for incoming Net8 connections. In the session, the ORB reads the incoming IIOP messages, authenticates the client, finds and activates the corresponding server-side

objects, and sends IIOP messages as needed to reply to the connected client. Subsequent messages from the client are directed to the existing session.

When you configure a listener, you need to configure separate ports as listening endpoints for both Net8 and IIOP connections. Similarly, if you want any endpoint to use the secure socket layer (SSL), you will also need a separate endpoint for an SSL-enabled IIOP endpoint. See "Using the Secure Socket Layer" on page 6-3 for more information about connecting using IIOP and SSL.

Handling Incoming Requests

The administrator for your database configures an MTS server with a dispatcher that is GIOP enabled. In addition, the administrator configures a listener that this dispatcher registers with upon database startup.

When the database does start up, all dispatchers register with all listeners configured within the same database initialization file. However, when an IIOP client invokes a request, the listener will only redirect the request to a GIOP dispatcher or hand off to a generic dispatcher.

These are discussed fully in the following sections:

- Redirect to GIOP Dispatcher
- Hand Off to Generic Dispatcher

Redirect to GIOP Dispatcher The listener recognizes the IIOP protocol and redirects the request to a registered GIOP dispatcher.

Figure 3–4 IIOP Listener Redirect to GIOP Dispatcher
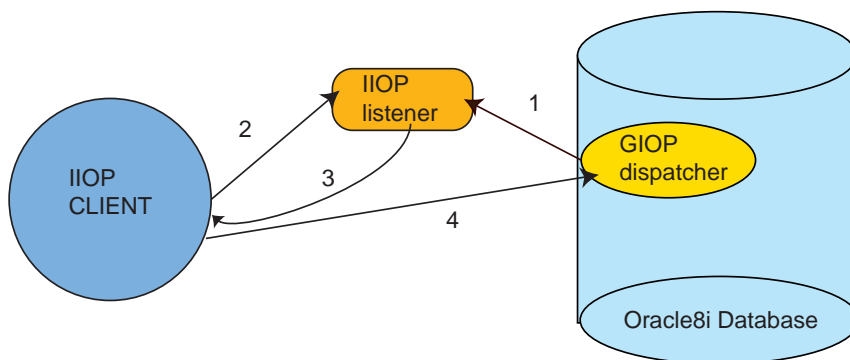
1. GIOP dispatcher registers itself with the listener.

2. IIOP client—an EJB or CORBA client—invokes a method, giving the address of the listener. For redirection to occur, the listener must be statically configured to receive IIOP requests.

3. The listener sends back a response to the client informing it of the GIOP dispatcher's address.

   The listener redirects if a GIOP dispatcher is configured. If no GIOP dispatcher is configured, the listener can hand off the request to a generic dispatcher. See "Hand Off to Generic Dispatcher" on page 3-14 for more information.

4. The underlying ORB runtime layer on the client resends its initial request to the GIOP dispatcher. All future method invocations are directed to the dispatcher. The listener is no longer a part of the communication.

Hand Off to Generic Dispatcher If there is no GIOP dispatcher statically configured, but there is a generic dispatcher configured, the listener can hand off the request to this dispatcher. The only restrictions are that the listener and dispatcher must exist on the same node in order for the hand off to occur and that the listener must be either statically or dynamically configured to receive an IIOP request.

For hand off to occur, the listener forfeits the socket to the dispatcher. Thus, this can only occur within a single node.

Figure 3–5 shows the dispatcher and listener combination in a hand off environment.

Figure 3–5   Hand Off to Dispatcher
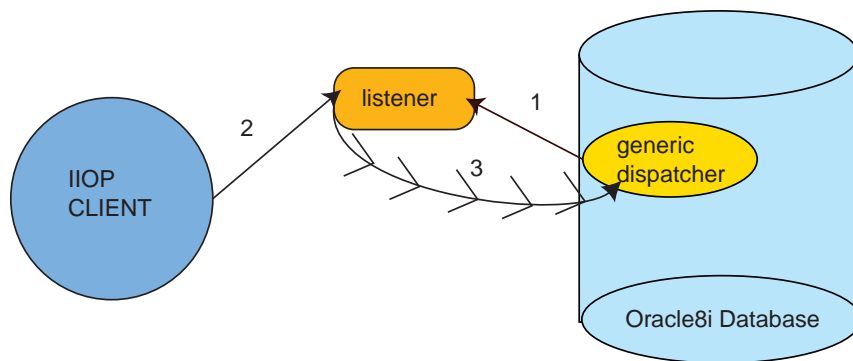
1. When the database starts, the generic dispatcher registers itself with the configured listener.

   > **Note:** The listener must be configured to receive IIOP requests. You can either statically configure the listener through the Net8 configuration, or you can dynamically configure the listener through a dynamic registration tool, regep. See "Dynamic Listener Endpoint Registration" on page 3-15 for more information.

2. The client sends a request to the listener.

3. The listener hands off the request to the generic dispatcher. The listener negotiates with the generic dispatcher on a separate channel. On this channel, the socket is handed off to the dispatcher through the operating system mechanisms.

   The client communicates directly with the dispatcher from this point on. The client is never made aware that the socket was handed off.

Dynamic Listener Endpoint Registration

As discussed in "Hand Off to Generic Dispatcher" on page 3-14, a listener will hand off the socket to an existing generic dispatcher. In order for a hand off to occur for an IIOP incoming request, the listener must have an IIOP endpoint registered. You can register any listening endpoint through either of the following:

- static configuration—configured by the Net8 configuration tool
- dynamic configuration—registered by the dynamic registration tool, regep

The dynamic registration tool, regep, adds any type of listening endpoint to your listener. This includes an IIOP listening endpoint. The following describes how to use the dynamic registration tool for an IIOP listening endpoint.

The restrictions for this scenario are as follows:

- Both the listener and generic dispatcher always exist on the same node.
- A GIOP configured dispatcher cannot exist.

> **Note:**   If a GIOP configured dispatcher does exist, the listener will redirect the request, rather than hand off the request, to the configured dispatcher.

The advantage for dynamically registering a listener endpoint is that you do not need to restart your database for this listener to be IIOP enabled. The listening endpoint is active immediately.

For full details on the regep tool, see the *Oracle8i Java Tools Reference Guide.*

Example 3–1   Dynamically Registering a LIstener at Port 2241

The following line dynamically registers a listener on the SUNDB host on endpoint port number 2241. This tool logs on to the SUNDB host.

```
regep –pres oracle.aurora.server.SGiopServer –host sundb –port 2241
```

Direct Dispatcher Connection

If you want your client to go to a dispatcher directly, bypassing the listener, you direct your client to the dispatcher's port number. Do one of the following to discover the dispatcher's port number:

- Configure a port number for the dispatcher by adding the ADDRESS parameter that includes a port number.

- Discover the port assigned to the dispatcher by invoking lsnrctl service.

If you choose to configure the port number, the following shows the syntax:

```
mts_dispatchers="(address=(protocol=tcp | tcps)
                 (host=<server_host>)(port=<port>))
                 (presentation=oracle.aurora.server.SGiopServer)"
```

The attributes are described below:

Attribute	Description
ADDRESS (ADD or ADDR)	Specifies the network address that the dispatchers will listen on. The network address may include either the TCP/IP (TCP) or the TCP/IP with SSL (TCPS) protocol, the host name of the server, and a GIOP listening port, which may be any port you choose that is not already in use.

Attribute	Description
PRESENTATION (PRE or PRES)	Enables support for GIOP. Supply the following value for a GIOP presentation: ■ `oracle.aurora.server.SGiopServer` for session-based GIOP connections. This presentation is valid for TCP/IP and TCP/IP with SSL.

The client supplies the port number on its URL, as follows:

```
session_iiop://<hostname>/:<portnumber>
```

Notice that the URL excludes a SID or service name. The dispatcher does not need the SID instance or service name because it is a directed request.

Configuring SSL for EJB and CORBA

Oracle8*i* also supports the use of authentication data such as certificates and private keys required for use by SSL in combination with GIOP. To configure your transport to be SSL-enabled with GIOP, do the following:

1. Enable the MTS_DISPATCHERS to be SSL-enabled.

2. Specify the SSL wallet to be used when configuring both the listener and database.

3. Configure the listener to accept SSL.

The following sections detail how to accomplish these three steps.

Enable the MTS_DISPATCHERS for SSL

You must edit the database initialization file to add an SSL-enabled dispatcher. Uncomment the MTS_DISPATCHERS parameter in the database initialization file that defines the TCPS port. During installation, the Database Configuration Assistant always includes a commented out line for SSL TCP/IP. This line is as follows:

```
mts_dispatchers="(protocol=tcps)(presentation=oracle.aurora.server.SGiopServer)"
```

Configure the Wallet Location through Net8 Assistant

Modify the listener to accept SSL requests on port 2482.

1. Start Net8 Assistant.

- On UNIX, run `netasst` at `$ORACLE_HOME/bin`.

- On Windows NT, choose Start > Programs > Oracle - *HOME_NAME* > Network Administration > Net8 Assistant.

2. In the navigator pane, expand Local > Profile.

3. From the pull-down list, select Oracle Advanced Security > SSL.

   This brings you to the listening port panel, as shown in Figure 3–6.

Figure 3–6   IIOP listening port configuration
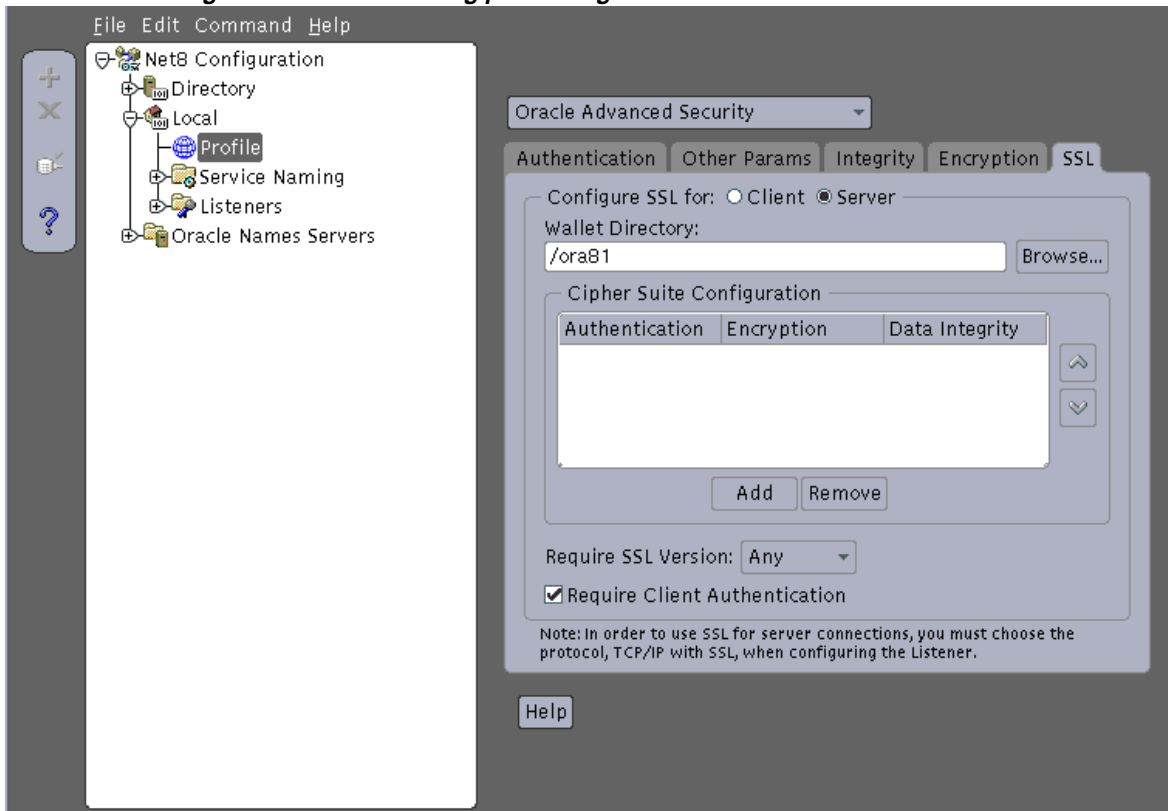
4. On the "Configure SSL for:" line, select the "Server" radio button.

5. Under "Wallet Directory", enter the location for the wallet.

6. If you desire a certain SSL version, choose the appropriate version on the SSL version pulldown list.

7. If you want the client to authenticate itself by providing certificates, select the "Require Client Authentication" checkbox.

8. Choose File > Save Network Configuration.

These steps will add wallet and SSL configuration information into both the listener and database configuration files. The SSL wallet location must be specified in both the listener and database configuration files. Both entities must locate the wallet for certificate handshake capabilities.

The `listener.ora` file:

```
ssl_client_authentication=false
ssl_version=undetermined
```

The default is for the database to authenticate the client. If you want the listener to authenticate the client, change the `ssl_client_authentication` parameter to true.

The database's `sqlnet.ora` file:

```
ssl_client_authentication=true
ssl_version=0
sqlnet.crypto_seed=<seed_info>
```

If you did not request client authentication, the `ssl_client_authentication` parameter will be false. The default value is for client authentication to be true. In addition, you can specify a specific SSL version number, such as 3.0, in the `ssl_version` parameter. The `ssl_version` value of 0 means that the version is undetermined and will be agreed upon during handshake.

Within both the listener's `listener.ora` and database's `sqlnet.ora` files, the wallet location is specified:

```
oss.source.my_wallet=
        (source=
            (method=file)
            (method_data=
              (directory=wallet_location)))
```

The *Oracle Advanced Security Administrator's Guide* describes how to set up the SSL wallet with the appropriate certificates.

Configure an SSL-Enabled Listener through Net8 Assistant

1. Back in the navigator pane, expand Local > Listener.

This brings you to the listener location panel, as shown in Figure 3–2.

Figure 3–7   IIOP listening port configuration
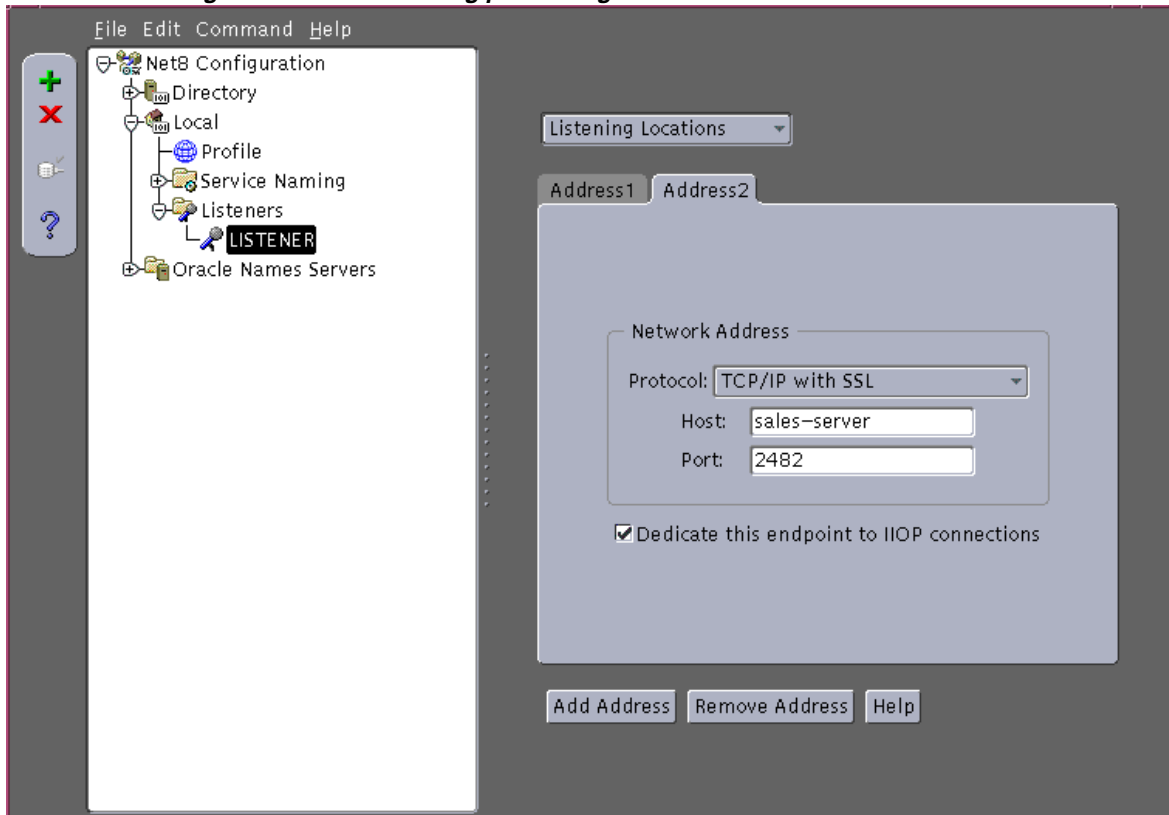
2. Select a listener.

3. From the list in the right pane, select Listening Locations. If none of the current listening addresses are feasible for your SSL listening address, you can add a new address by clicking on the "Add Address" button.

4. Select the TCP/IP with SSL protocol from the Protocol list.

5. Enter the host name of the database in the Host field.

6. Enter port 2482 in the Port field.

7. Click "Dedicate this endpoint to IIOP connections".

8. Choose File > Save Network Configuration.

This places the following in the `listener.ora` file: it modifies the listener to specify TCPS —instead of TCP—as the protocol with port number 2482. The following shows an example of an SSL-enabled listener on the `sales-server` host.

```
listener=
  (description_list=
   (description=
      (address=(protocol=tcps)(host=sales-server)(port=2482))))
      (protocol_stack=
        (presentation=giop)
        (session=raw)))
```

4

Entity Beans

This chapter discusses what an entity bean is, how to create one, and how it is different from a session bean.

- Definition of an Entity Bean

- Difference Between Session and Entity Beans

- Implementing Callback Methods

- Creating Entity Beans

- Difference Between Bean-Managed and Container-Managed Beans

- Accessing EJB References and JDBC DataSources

Definition of an Entity Bean

An entity bean is a remote object that manages persistent data, performs complex business logic, potentially uses several dependent Java objects, and can be uniquely identified by a primary key. Entity beans are normally coarse-grained persistent objects, in that they utilize persistent data stored within several fine-grained persistent Java objects.

> **Note:** Fine-grained persistent Java objects typically manage persistent data that has a one-to-one mapping between the data and a table column. Coarse-grained persistent Java objects use or manage persistent data stored in several fine-grained persistent objects.

Managing Persistent Data

An entity bean manages its data persistentcy through callback methods, which are defined in the `javax.ejb.EntityBean` interface. When you implement the `EntityBean` interface in your bean class, you develop each of the callback functions as designated by the type of persistence that you choose: bean-managed persistence or container-managed persistence. The container invokes the callback functions at designated times. That is, the contract between the container and the entity bean designates the order that the callback methods are invoked and who manages the bean's persistent data.

Uniquely Identified by a Primary Key

Each entity bean has a persistent identity associated with it. That is, the entity bean contains a unique identity that can be retrieved if you have the primary key. Given the primary key, a client can retrieve the entity bean. If the bean is not available, the container instantiates the bean and repopulates the persistent data for you.

The type for the unique key is defined by the bean provider.

Performing Complex Logic Involving Dependent Objects

When designing your EJB application, you need to keep in mind the aspects of each type of object.

- Session beans are typically used for performing simple tasks for a remote client.

- Entity beans are typically used for performing complex tasks that involve coarse-grained persistence for remote clients.

- Java objects, persistent or otherwise, are used for simple tasks for local clients.

Enterprise JavaBeans are remote objects and are used for interacting with clients over a network. Remote objects have a higher overhead for verifying security and transaction information. Thus, when you design your application, you may have an entity or session bean interacting with the client, but also have the Enterprise JavaBean invoke other dependent Java objects to perform tasks or manage persistent data.

Entity beans are normally used to manage complex, coarse-grained persistent data for a remote client. Be careful to separate the difference between an entity bean and a persistent object. Your entity bean should be more than just a persistent object; it should manage and return complex data to justify using a remote object for managing data.

You can have an entity bean that calls one or more dependent objects within the application. The entity bean is a remote object and thus its primary function is interacting with the client over the network. You should not have an entity bean invoking another entity bean within the same node on the network. If you need to design multiple objects within your application, design your application so that the entity bean facilitates the communication and data management between the client and other Java objects.

Figure 4–1 demonstrates how the client interacts with either a session or an entity bean, which then manages the application for the client with other Java objects within the application. The Java objects that make up the backend of the application can be persistent objects. The figure also shows how both the entity bean and a persistent Java object can be persistent and store data within the database.

Figure 4–1   Relationship of Enterprise JavaBeans to Java objects
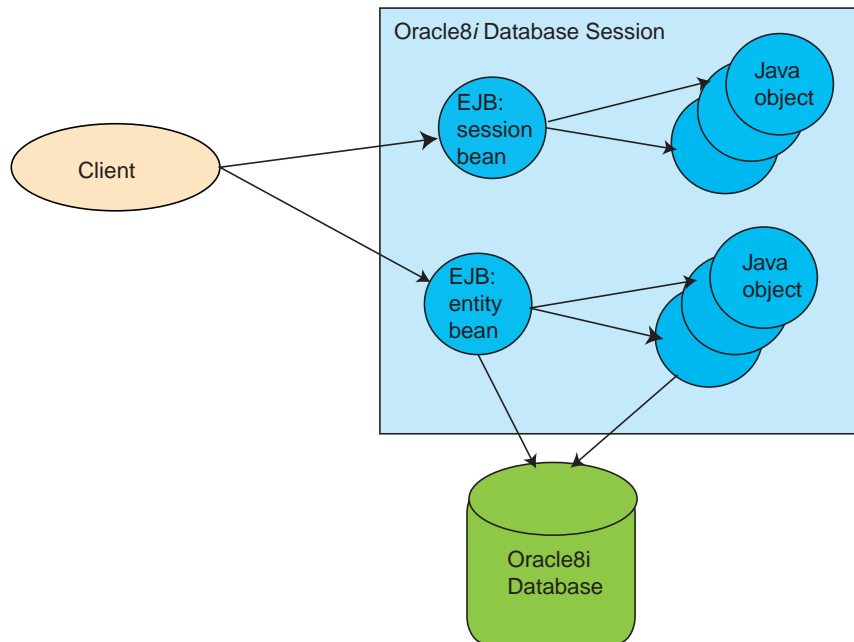
For example, if you are managing a shopping cart for an online bookstore, you would have the following requirements:

- identify the customer for the shopping cart
- add items to the customer's shopping cart
- calculate the price for all items taking into account any discounts, sale items, and shipping costs
- ship items to the customer

In this scenario, the entity bean could do the following:

- retrieve the customer information from a dependent persistent object
- retrieve item information from an item dependent persistent object
- record the number of items for each item in the order
- retrieve discount and shipping costs
- coordinate the shipment to the designated customer address

Thus, this entity bean not only retrieves persistent information from other objects, but would also maintain its own persistent data and perform complex calculations.

Difference Between Session and Entity Beans

The major differences between session and entity beans is that entity beans involve a framework for persistent data management, a persistent identity, and complex business logic. The interface requirements on entity beans provides callback functions that the container calls when persistent data should be managed or when a bean should be retrieved based upon its identity.

With an entity bean, the interfaces have been designed so that each callback method is called at the appropriate time. For example, right before the transaction is committed, the `ejbStore` method is always invoked. This enables the entity bean to save all of its persistent data before the transaction is completed. Each of these callback methods are discussed further in "Implementing Callback Methods" on page 4-5.

The following table illustrates the different interfaces for session and entity beans. Notice that the difference between the two types of EJBs exists within the bean class and the primary key. All of the persistent data management is done within the bean class methods.

	Entity Bean	Session Bean
Remote interface	Extends `javax.ejb.EJBObject`	Extends `javax.ejb.EJBObject`
Home interface	Extends `javax.ejb.EJBHome`	Extends `javax.ejb.EJBHome`
Bean class	Extends `javax.ejb.EntityBean`	Extends `javax.ejb.SessionBean`
Primary key	Used to identify and retrieve specific bean instances	Not used for session beans

Implementing Callback Methods

An entity bean is a remote object that manages its data persistently through callback methods, which are defined in the `javax.ejb.EntityBean` interface. When you implement the `EntityBean` interface in your bean class, you develop each of the callback functions as designated by the type of persistence that you choose: bean-managed persistence or container-managed persistence. The container invokes the callback functions at designated times, to manage the bean and its persistent

data. That is, the contract between the container and the entity bean involves in what order the callback methods are invoked and who manages the bean's persistent data.

Your bean class implements the methods of the EntityBean interface. The javax.ejb.EntityBean interface has the following definition:

```
public interface javax.ejb.EntityBean extends javax.ejb.EnterpriseBean {
    public abstract void ejbActivate();
    public abstract void ejbLoad();
    public abstract void ejbPassivate();
    public abstract void ejbRemove();
    public abstract void ejbStore();
    public abstract void setEntityContext(EntityContext ctx);
    public abstract voic unsetEntityContext();
}
```

The container expects these methods to have the following functionality:

- ejbCreate — You must implement an ejbCreate method corresponding to one create method declared in the home interface. When the client invokes the create method, the container first invokes the constructor to instantiate the object, then it invokes the corresponding ejbCreate method. The ejbCreate method performs the following:

  - creates any persistent storage for its data, such as database rows

  - intializes a unique primary key and returns it

- ejbPostCreate — The container invokes this method after the environment is set. For each ejbCreate method, an ejbPostCreate method must exist with the same arguments. This method can be used to initialize parameters within or from the entity context.

- ejbRemove — The container invokes this method before it ends the life of the session object. This method may perform any required clean-up, for example closing external resources such as file handles.

- `ejbStore` The container invokes this method right before a transaction commits. It saves the persistent data to an outside resource, such as a database.

- `ejbLoad` The container invokes this method within a transaction when the data should be reinitialized from the database. This normally occurs after the transaction begins.

- `setEntityContext` Associates the bean instance with context information. The container calls this method after the bean creation. The enterprise bean can store the reference to the context object in an instance variable, for use in transaction management. Beans that manage their own transactions can use the session context to get the transaction context.

  You can also allocate any resources that will exist for the lifetime of the bean within this method. You should release these resources in `unsetEntityContext`.

- `unsetEntityContext` Unset the associated entity context and release any resources allocated in `setEntityContext`.

- `ejbActivate` Implement this as a null method, because it is never called in this release of the server.

- `ejbPassivate` Implement this as a null method, because it is never called in this release of the server.

Using ejbCreate and ejbPostCreate

An entity bean is similar to a session bean in that certain callback methods, such as `ejbCreate`, are invoked at specified times. Entity beans use callback functions for managing its persistent data, primary key, and context information. The following diagram shows what methods are called when an entity bean is created.

Figure 4–2   Creating the Entity Bean

```
Client                        Entity Bean
─────────────────────────────────────────────────────

                             ⎧ <Bean> constructor
                             ⎪ ejbCreate(...)
create ───────────────────→  ⎨   primary key constructor
                             ⎪ ejbSetEntityContext()
                             ⎩ ejbPostCreate(...){
```
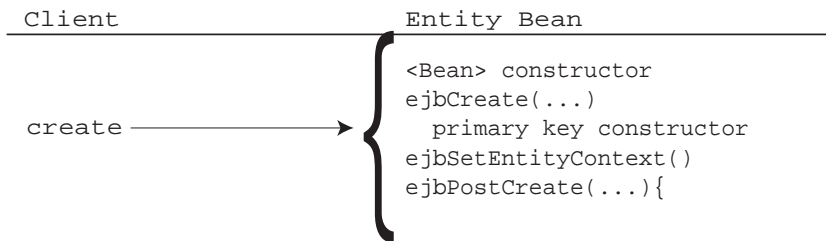
Using setEntityContext

This method is used by an entity bean instance to retain a reference to its context. Entity beans have contexts that the container maintains and makes available to the beans. The bean may use the methods in the entity context to retrieve information about the bean, such as security, and transactional role. Refer to the Enterprise Java-Beans 1.1 specification for the full range of information that you can retrieve about the bean from the context.

The container invokes setEntityContext method, after it first instantiates the bean, to enable the bean to retrieve the context. The container will never call this method from within a transaction context. If the bean does not save the context at this point, the bean will never gain access to the context.

> **Note:**   You can also use the setEntityContext and
> unsetEntityContext methods to allocate and destroy any
> resources that will exist for the lifetime of the instance.

When the container calls this method, it passes the reference of the EntityContext object to the bean. The bean can then store the reference for later use. The following example shows the bean saving the context in the **this.ctx** variable.

```
public void setEntityContext(EntityContext ctx)
{
  this.ctx = ctx;
  Properties props = ctx.getEnvironment();
}
```

Using ejbRemove

When the client invokes the remove method, the container invokes the following methods.

Figure 4–3   Removing the Entity Bean
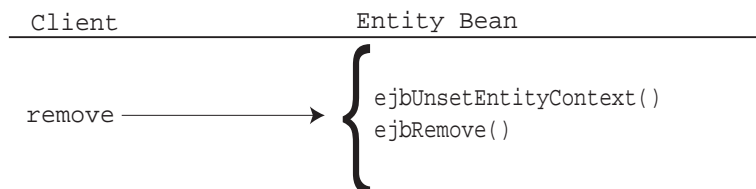
Using ejbStore and ejbLoad

In addition, the `ejbStore` and `ejbLoad` methods are called for managing your persistent data. These are the most important callback methods—for bean-managed persistent beans.

- The `ejbStore` method is called by the container whenever a transaction is about to end. Its purpose is to save the persistent data to an outside resource, such as a database.

- The `ejbLoad` method is called by the container whenever a transaction has begun or when an entity bean is instantiated. Its purpose is to restore any persistent data that exists for this particular bean instance.

Creating Entity Beans

The steps for creating an entity bean are the same as for a session bean. The difference is contained in the methods and data within the bean class. There are two types of entity beans: bean-managed persistent and container-managed persistent. This section discusses a bean-managed persistent bean. The "Container-Managed Persistence" on page 4-28 gives an example of a container-managed persistent bean.

To create an entity bean, you perform the following steps:

1. Create a remote interface for the bean. The remote interface declares the methods that a client can invoke. It must extend `javax.ejb.EJBObject`.

2. Create a home interface for the bean. The home interface must extend `javax.ejb.EJBHome`. It defines the `create` and finder methods, including `findByPrimaryKey`, for your bean.

3. Define the primary key for the bean. The primary key identifies each entity bean instance. The primary key must either be a well-known class, such as `java.lang.String`, or be defined within its own class.

4. Implement the bean. This includes the following:

    a. The implementation for the methods declared in your remote interface.

    b. An empty constructor for the bean.

    c. The methods defined in the `javax.ejb.EntityBean` interface.

    d. The methods that match the methods declared in your home interface. This includes the following:

        * The `ejbCreate` and `ejbPostCreate` methods with parameters matching those of the `create` method defined of the home interface.

        * An `ejbFindByPrimary` key method which corresponds to the `findByPrimaryKey` method of the home interface.

        * Any other finder methods that were defined in the home interface.

5. If the persistent data is saved to or restored from a database, you must ensure that the correct tables exist for the bean.

6. Create the bean deployment descriptor. The deployment descriptor specifies properties for the bean through XML properties. See "Deploying an EJB" on page 2-19 for more details.

7. Create an ejb-jar file containing the bean, the remote and home interfaces, and the deployment descriptor. The ejb-jar file must define all beans within your application. Refer to "Create a JAR File" on page 2-26 for more details.

Home Interface

Similar to session beans, the entity bean's home interface must contain a `create` method, which the client invokes to create the bean instance. Each `create` method can have a different signature.

For an entity bean, you must develop a `findByPrimaryKey` method. Because of the persistent data associated with the instance, each entity bean instance is uniquely identified by a primary key. The type for the unique key is defined by the developer. For example, the customer bean's primary key is the customer number.

The purchase order's primary key is a purchase order number. The primary key can be anything—as long as it is unique.

When the entity bean is first created, the `ejbCreate` method creates a primary key to identify the bean. A unique primary key is created and initialized within the `ejbCreate` method in the bean class. From this time onward, this bean is associated with this primary key. Thus, you can retrieve the bean by providing the primary key object to the `findByPrimaryKey` method.

Optionally, you can develop other finder methods to find the bean. These methods are named `find<name>`.

> **Note:** The return type for all finder methods within the home interface must be either the entity bean's remote interface or an `Enumeration` of objects that implement the entity bean's remote interface. Returning a `Collection` is not supported.
>
> The return type for all finder methods implemented within the bean class returns the primary key or an `Enumeration` of primary keys. The container retrieves the appropriate entity bean remote interface for each primary key returned on any `ejbFind<name>` method.

Example 4–1   Purchase Order Home Interface

To demonstrate an entity bean, we are creating a bean that manages a purchase order. The entity bean contains a list of items ordered by the customer.

The home interface extends `javax.ejb.EJBHome` and defines the `create` and `findByPrimaryKey` methods.

```
package purchase;

import javax.ejb.*;
import java.rmi.RemoteException;
import java.sql.SQLException;

public interface PurchaseOrderHome extends EJBHome
{
  // Create a new PO
  public PurchaseOrder create() throws CreateException, RemoteException;

  // Find an existing one
  public PurchaseOrder findByPrimaryKey (String POnumber)
```

```
                throws FinderException, RemoteException;
    }
```

Remote Interface

The entity bean remote interface is the interface that the customer sees and invokes methods upon. It extends javax.ejb.EJBObject and defines the business logic methods. For our purchase order entity bean, the remote interface contains methods for adding items to the purchase order, for retrieving a list of all items within the purchase order, and computing the full price for the purchase order.

```java
package purchase;

import javax.ejb.EJBObject;
import java.rmi.RemoteException;
import java.sql.SQLException;
import java.util.Vector;

public interface PurchaseOrder extends EJBObject  {
    // Price the purchase order
    public float price() throws RemoteException;

    // getContents returns a Vector of LineItem objects in the purchase
    public Vector getContents() throws RemoteException;

    //Add items to the purchase
    public void addItem (int sku, int count) throws RemoteException;
}
```

Primary Key

Each entity bean instance has a primary key that uniquely identifies it from other instances. You can define your primary key in one of two ways:

- Define the type of the primary key to be a well-known type, such as java.lang.String. If the primary key is a well-known data type, define the type in the <prim-key-class> in the deployment descriptor.

- Define the type of the primary key as a serializable object within <*name*>PK class. If the primary key is a complex data type, define the primary key in a class that is serializable. This class is declared in the <prim-key-class> element in the deployment descriptor.

Defining Primary Key as Well-known Type

Define your primary key to be a well-known type by defining the data type of the primary key within the deployment descriptor.

The purchase example defines its primary key as a `java.lang.String`.

```
<enterprise-beans>
     <entity>
        <ejb-name>test/purchase</ejb-name>
        <home>purchase.PurchaseOrderHome</home>
        <remote>purchase.PurchaseOrder</remote>
        <ejb-class>purchaseServer.PurchaseOrderBean</ejb-class>
        <persistence-type>Bean</persistence-type>
        <prim-key-class>java.lang.String</prim-key-class>
        <reentrant>False</reentrant>
     </entity>
...
</enterprise-beans>
```

Defining the Primary Key in a Class

If your primary key is more complex than a simple data type, your primary key must be a class that is serializable of the name `<name>PK`. Within this class, you should implement the `equals` and `hashCode` methods to provide for an implementation specific to this primary key.

The customer example declares its primary key—a customer identifier—within the `PurchaseOrderPK.java`.

```java
package purchase;

public class PurchaseOrderPK implements java.io.Serializable
{
  public int orderid;

  public boolean equals(Object obj) {
    if ((obj instanceof PurchaseOrderPK) &&
        (((PurchaseOrderPK)obj).orderid == this.orderid))
      return true;
    return false;
  }

  public int hashCode() {
    return orderid;
  }
}
```

The class that defines the primary key is declared within the deployment descriptor, as follows:

```
<enterprise-beans>
     <entity>
         <ejb-name>test/purchase</ejb-name>
         <home>purchase.PurchaseOrderHome</home>
         <remote>purchase.PurchaseOrder</remote>
         <ejb-class>purchaseServer.PurchaseOrderBean</ejb-class>
         <persistence-type>Bean</persistence-type>
         <prim-key-class>purchase.PurchaseOrderPK</prim-key-class>
         <reentrant>False</reentrant>
     </entity>
...
</enterprise-beans>
```

Manage the Primary Key

The `ejbCreate` method is responsible primarily for the creation of the primary key. This involves creating the primary key, creating the persistent data representation for the key, initializing the key to a unique value, and returning this key to the invoker. The `ejbFindByPrimaryKey` method is responsible for verifying that the primary key is still unique and returns it again to the container.

In the purchase order example, these methods perform the following:

- The `ejbCreate` method initializes the primary key, `ponumber`, to the next available number in the purchase order number sequence.

- The `ejbFindByPrimaryKey` method is implemented to check that the purchase order number is valid and returns this number to the container.

```
// The create methods takes care of generating a new PO and returns
// its primary key
public String ejbCreate () throws CreateException, RemoteException
{
  String ponumber = null;
  try {
    //retrieve the next available purchase order number
    #sql { select ponumber.nextval into :ponumber from dual };
    //assign this number as this instance's identification number
    #sql { insert into pos (ponumber, status) values (:ponumber, 'OPEN') };
  } catch (SQLException e) {
    throw new PurchaseException (this, "create", e);
  }
```

```
    return ponumber;
}

// The ejbFindByPrimaryKey method verifies that the POnumber exists. This
// method must return the primary key to the container.. which in turn
// retrieves the instance based on the primary key. So.. this method must
// only verify that the primary key is valid.
public String ejbFindByPrimaryKey (String ponumber)
    throws FinderException, RemoteException
{
    try {
      int count;
      #sql { select count (ponumber) into :count from pos
               where ponumber = :ponumber };

      // There has to be one
      if (count != 1)
        throw new FinderException ("Inexistent PO: " + ponumber);
    } catch (SQLException e) {
      throw new PurchaseException (this, "findByPrimaryKey", e);
    }
    // The ponumber is the primary key
    return ponumber;
}
```

Entity Bean Class

The entity bean class implements the following methods:

- An empty constructor for creating the bean instance.

- The target methods for the methods declared in the home interface, which includes the `ejbCreate` method and any finder methods, including `ejbFind-ByPrimaryKey`.

- The business logic methods declared in the remote interface.

- The methods declared in the `EntityBean` interface.

The following code implements methods of an entity bean called `PurchaseOrderBean`.

1. Declaring Variables

The purchase order bean declares a vector to store all of the items within the customer's shopping cart. In addition, to retrieve environment information for the entity bean, an entity context is defined.

```
#sql iterator ItemsIter (int skunumber, int count, String description,
                          float price);

public class PurchaseOrderBean implements EntityBean {
  EntityContext ctx;
  Vector items;            // The items in the PO (instances of LineItem)
```

2. Implementing Remote Interface Methods

The following is the implementation for the bean methods that were declared in the remote interface: **price**, **getContents**, and **addItem**.

```
public float price() throws RemoteException {
  float price = 0;
  Enumeration e = items.elements ();
  while (e.hasMoreElements ()) {
    LineItem item = (LineItem)e.nextElement ();
    price += item.quantity * item.price;
  }

  // 5% discount if buying more than 10 items
  if (items.size () > 10)
    price -= price * 0.05;

  // Shipping is a constant plus function of the number of items
  price += 10 + (items.size () * 2);

  return price;
}

// The getContents methods has to load the descriptions
public Vector getContents() throws RemoteException {
  return items;
}

// The add Item method gets the price and description
public void addItem (int sku, int count) throws RemoteException {
  try {
    String description;
    float price;
```

```
      #sql { select price, description into :price, :description
              from skus where skunumber = :sku };
      items.addElement (new LineItem (sku, count, description, price));
    } catch (SQLException e) {
      throw new PurchaseException (this, "addItem", e);
    }
  }
```

3. Implementing EntityBean Interface Methods

Once you have implemented the business logic methods, you also must provide the following:

- A public constructor for the bean instance—This constructor takes no arguments. The container invokes the constructor to create an instance of the entity bean class. The constructor can be an empty implementation.

- An `ejbCreate` method for each `create` method defined in the home interface.

- An `ejbFind<`*name*`>` method for each of the `find<`*name*`>` methods defined in the home interface. This includes at least an `ejbFindByPrimary` key method that returns the primary key to the container.

- An implementation for the `EntityBean` methods—These methods are callback methods that the container calls when necessary. Most of the callback methods pertain to managing the persistence of the entity bean's data.

Public Constructor The public constructor is called by the container to create the bean instance. The `ejbCreate` and `ejbPostCreate` methods are invoked to intialize this instance. The following is the purchase order constructor.

```
//provide an empty constructor for the creating the instance
public void PurchaseOrderBean() {}
```

The Create Methods: ejbCreate and ejbPostCreate As shown in Figure 4–2, the `ejbCreate` and `ejbPostCreate` methods are invoked when the corresponding `create` method—the methods all have the same arguments—is invoked. Typically, the `ejbCreate` method initializes all of the persistent data; the `ejbPostCreate` does any initialization that involves the entity's context. The context information is not available at `ejbCreate` time, but is available at `ejbPostCreate` time.

The following example shows the `ejbCreate` and `ejbPostCreate` for the purchase order example. The `ejbCreate` method initializes the primary key, which

is the purchase order number, and returns this key to the invoker. The purchase order line item vector is initialized within the `ejbPostCreate`.

```
// The create methods takes care of generating a new PO and returns
// its primary key
public String ejbCreate () throws CreateException, RemoteException
{
  String ponumber = null;
  try {
    //retrieve the next available purchase order number
    #sql { select ponumber.nextval into :ponumber from dual };
    //assign this number as this instance's identification number
    #sql { insert into pos (ponumber, status) values (:ponumber, 'OPEN') };
  } catch (SQLException e) {
    throw new PurchaseException (this, "create", e);
  }
  return ponumber;
}

// create a vector to contain the purchase order line items. since this
// is performed only once and needed for the lifetime of the object, it is
// appropriate to create the vector in either ejbCreate or ejbPostCreate.
public void ejbPostCreate () {
    items = new Vector ();
}
```

The Finder Methods All entity beans must provide an `ejbFindByPrimaryKey` method. You can also have other types of finder methods. Since the developer must implement any finder method declared within the home interface, there is no limitation on how many of these types of methods you can have. The only restrictions is that any finder method, other than the `ejbFindByPrimaryKey` method, must return either a reference to the remote interface or an `Enumeration` containing multiple references to remote interfaces. The `ejbFindByPrimaryKey` method must return the primary key.

> **Note:** The return type cannot be a `Collection`, as it is not currently supported.

In order to provide other finder methods, you must do the following:

1. Declare the method as `find<name>` in the home interface.

2. Implement the method as `ejbFind<name>` in the bean class.

The following is the `ejbFindByPrimaryKey` method for the purchase order example. It verifies that the primary key is valid. If so, it returns the key to the container. The container retrieves the correct bean instance for this key and returns the reference to the client.

```
// The findByPrimaryKey method verifies that the POnumber exists. This
// method must return the primary key to the container.. which in turn
// retrieves the instance based on the primary key. So.. this method must
// only verify that the primary key is valid.
public String ejbFindByPrimaryKey (String ponumber)
    throws FinderException, RemoteException
{
    try {
      int count;
      #sql { select count (ponumber) into :count from pos
              where ponumber = :ponumber };

      // There has to be one
      if (count != 1)
        throw new FinderException ("Inexistent PO: " + ponumber);
    } catch (SQLException e) {
      throw new PurchaseException (this, "findByPrimaryKey", e);
    }
    // The ponumber is the primary key
    return ponumber;
}
```

The EntityBean Methods: Load and Store The main difference between entity and session beans is that entity beans possess persistent data that must be managed. When data is defined as persistent, it must be continually saved to or restored from a resource, such as a database or file. If the bean is destroyed, the persistent data can be restored without any loss.

The `EntityBean` interface, which all entity beans implement, defines the following callback methods for managing the persistent data:

- `ejbStore`—saves the data to persistent storage

  The container always invokes `ejbStore` right before a transaction commits or the bean is removed to save the existing values of the persistent data.

- `ejbLoad`—loads the data saved within persistent storage into the bean

  The container invokes `ejbLoad` right after a bean is instantiated or a transaction begins.

Figure 4–4 shows how the persistent data within an entity bean can be saved to a database using ejbStore. In addition, the data is restored from the database through ejbLoad.

Figure 4–4   Persistent Data Management
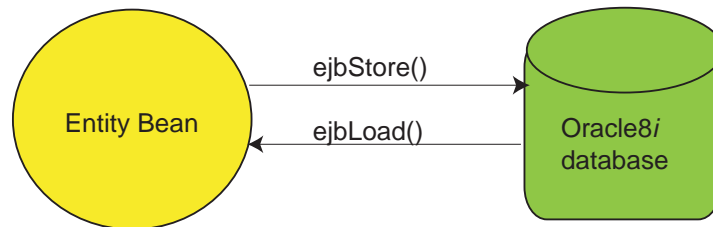
The following are the methods from the purchase order example. The ejbStore method saves the purchase order items to the database. The ejbLoad method restores the purchase order items from the database.

```
// The store method replaces all entries in the lineitems table with the
// new entries from the bean
public void ejbStore() throws RemoteException {
  // Get the purchase order number
  String ponumber = (String)ctx.getPrimaryKey();

  try {
    // Delete old entries in the database
    #sql { delete from lineitems where ponumber = :ponumber };

    // Insert new entries from the vector in the bean. Crude, but effective.
    Enumeration e = items.elements ();
    while (e.hasMoreElements ()) {
      LineItem item = (LineItem)e.nextElement ();
      #sql { insert into lineitems (ponumber, skunumber, count)
              values (:ponumber, :(item.sku), :(item.quantity))
      };
    }
  } catch (SQLException e) {
    throw new PurchaseException (this, "store", e);
  }
}

// The load method populates the items array with all the saved
// line items
public void ejbLoad() throws RemoteException {
```

```
// Get the purchase order number
String ponumber = (String)ctx.getPrimaryKey();

// Load all line items into a new vector.
try {
  items = new Vector ();
  ItemsIter iter = null;
  try {
    #sql iter = {
      select lineitems.skunumber, lineitems.count,
             skus.description, skus.price
        from lineitems, skus
        where ponumber = :ponumber and lineitems.skunumber = skus.skunumber
    };

    while (iter.next ()) {
      LineItem item =
        new LineItem (iter.skunumber(), iter.count(), iter.description(),
                      iter.price());
      items.addElement (item);
    }
  } finally {
    if (iter != null) iter.close ();
  }
} catch (SQLException e) {
  throw new PurchaseException (this, "load", e);
}
}
```

The EntityBean Methods: Remove The `ejbRemove` method is invoked when the client invokes the remove method. For a bean-managed persistent bean, you must remove the data from the database that is associated with the bean within this method.

The following example shows how the purchase order line items are removed from the database.

```
// The remove method deletes all line items belonging to the purchase order
public void ejbRemove() throws RemoteException {
  // Get the purchase order number from the session context
  String ponumber = (String)ctx.getPrimaryKey();
  try {
    //delete the line item vector for the purchase order
    #sql { delete from lineitems where ponumber = :ponumber };
    //delete the row associated with the purchase order
    #sql { delete from pos where ponumber = :ponumber };
```

```
  } catch (SQLException e) {
    throw new PurchaseException (this, "remove", e);
  }
}
```

The EntityBean Methods: Setting the Context If you want to access any information within the context during the lifetime of your application, you must save the context within the setEntityContext.

```
//Set the provided context to this.ctx
public void setEntityContext(EntityContext ctx)
{
 this.ctx = ctx;
}

//reinitialize the context to null
public void unsetEntityContext()
{
 this.ctx = null;
}
```

The Entity Bean Methods: Activate and Passivate Oracle does not currently support activation and passivation. However, you still must provide an empty implementation for these methods. ,

```
//There are no requirements for ejbActivate for this bean
public void ejbActivate()
{
}

//There are no requirements for ejbPassivate for this bean
public void ejbPassivate()
{
}
```

4. LineItem Class

The purchase order application persistently stores the individual orders within the purchase using a persistent Java object, called LineItem. That is, the entity bean delegates management of each item in the purchase order to a non-EJB Java object.

```
package purchase;

public class LineItem implements java.io.Serializable {
  public int sku;
  public int quantity;
```

```
public String description;
public float price;

//Persistently manage each line item within the purchase order.
public LineItem (int sku, int quantity, String description, float price) {
   //Each line item has the following information: SKU number, quantity,
   // description, and price.
   this.sku = sku;
   this.quantity = quantity;
   this.description = description;
   this.price = price;
 }
}
```

Create Database Table and Columns for Entity Data

If your entity bean stores its persistent data within a database, you need to create the appropriate table with the proper columns for the entity bean. This table must be created before the bean is loaded into the database.

In our purchase order example, you must create the following tables:

Table	Columns	Description
SKUS	■ skunumber: item number ■ description: item description ■ price: price of item	Each item in the warehouse is described in this table.
POS	■ ponumber:purchase order number ■ status: open, executed, or shipped	The table that manages the state of the purchase order for a customer. Contains the state of the order.
LINEITEMS	■ ponumber: purchase order number ■ skunumber: item number ■ count: number of items ordered.	The table that contains all of the individual items ordered by a customer.

The following shows the SQL commands that create these fields.

```
-- This sql scripts create the SQL tables used by the PurchaseOrder bean

-- The sku table lists all the items available for purchase
create table skus (skunumber number constraint pk_skus primary key,
                   description varchar2(2000),
```

```
                                price number);

            -- The pos table stores information about purchase orders
            -- The status column is 'OPEN', 'EXECUTED' or 'SHIPPED'
            create table pos (ponumber number constraint pk_pos primary key,
                              status varchar2(30));

            -- The ponumber sequence is used to generate PO ids
            create sequence ponumber;

            -- The lineitems table stores the contents of a po
            -- The skunumber is a reference into the skus table
            -- The ponumber is a reference into the pos table
            create table lineitems (ponumber number constraint fk_pos references pos,
                                    skunumber number constraint fk_skus references skus,
                                    count number);
            commit;

            exit;
```

Deploying the Entity Bean

You deploy the entity bean the same way as the session bean, which is detailed in "Deploying an EJB" on page 2-19. In the same manner, you must create the XML deployment descriptors for the bean, create a JAR file containing all of the bean's files, and use deployejb tool to load and publish the bean in the database.

The XML deployment descriptors are explained fully in Appendix A, "XML Deployment Descriptors". See the appendix for a full description on defining persistence for entity beans. For completeness, the following is how the purchase order example deployment descriptors are organized. Since the purchase order does not define any logical names that must be mapped and do not use the <run-as> option, only the EJB deployment descriptor is required.

Example 4–2   Purchase Order EJB Deployment Descriptor

```
<?xml version="1.0"?>
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems Inc.//DTD Enterprise JavaBeans 1.1
//EN" "ejb-jar.dtd">
<ejb-jar>
   <enterprise-beans>
      <entity>
         <description>no description</description>
         <ejb-name>test/purchase</ejb-name>
```

```
            <home>purchase.PurchaseOrderHome</home>
            <remote>purchase.PurchaseOrder</remote>
            <ejb-class>purchaseServer.PurchaseOrderBean</ejb-class>
            <persistence-type>Bean</persistence-type>
            <prim-key-class>java.lang.String</prim-key-class>
            <reentrant>False</reentrant>
        </entity>
    </enterprise-beans>
    <assembly-descriptor>
        <security-role>
            <description>no description</description>
            <role-name>PUBLIC</role-name>
        </security-role>
        <method-permission>
            <description>no description</description>
            <role-name>PUBLIC</role-name>
            <method>
                <ejb-name>test/purchase</ejb-name>
                <method-name>*</method-name>
            </method>
        </method-permission>
        <container-transaction>
            <description>no description</description>
            <method>
                <ejb-name>test/purchase</ejb-name>
                <method-name>*</method-name>
            </method>
            <trans-attribute>Required</trans-attribute>
        </container-transaction>
    </assembly-descriptor>
</ejb-jar>
```

Client Accessing Deployed Entity Bean

To access a deployed entity bean, the client does one of the following:

- Create a New Entity Bean

- Access an Existing Entity Bean

Create a New Entity Bean

When you access an entity bean, you must first locate the bean's home interface. You retrieve the home interface from the name space through JNDI. The URL must be of the following syntax:

```
<service_name>://<hostname>:<iiop_listener_port>:<SID>/<published_obj_name>
```

This syntax is described more in "Getting the Home Interface Object" on page 2-17.

Example 4–3   Retrieving the Home Interface from the JNDI Name Space

The following example retrieves the home interface of the EJB located published in `/test/purchase`. The host, port, and SID are localhost, 2471, and ORCL respectively.

```
String serviceURL = "sess_iiop://localhost:2471:ORCL";
String objectName = "/test/purchase";

Hashtable env = new Hashtable();
env.put(Context.URL_PKG_PREFIXES, "oracle.aurora.jndi");
env.put(Context.SECURITY_PRINCIPAL, user);
env.put(Context.SECURITY_CREDENTIALS, password);
env.put(Context.SECURITY_AUTHENTICATION, ServiceCtx.NON_SSL_LOGIN);
Context ic = new InitialContext (env);

CustomerHome ch = (CustomerHome)ic.lookup (serviceURL + objectName);
Customer myCust = (Customer) ch.create();
```

> **Note:**   Notice how the type casting on the lookup does not require the narrow method. The Oracle8*i* lookup method automatically performs the proper narrowing function for you. Although you still must provide the type that the returned object is cast to.

Access an Existing Entity Bean

A client can access an existing entity bean through one of the following methods:

- Receive the reference from another party—for example, as a returned parameter on a method call.

- Given a primary key for the entity bean, invoke the `findByPrimaryKey` method, which returns the entity bean's remote reference.

- Given a `Handle` object for the bean, invoke the `getEJBObject` method on the `Handle` object. This handle was created from the entity bean object using the `getHandle` method. The handle can be passed, as is, to another object or it can be serialized and stored to be used at a later date.

Difference Between Bean-Managed and Container-Managed Beans

There are two methods for managing the persistent data within an entity bean: bean-managed and container-managed persistence. The main difference between bean-managed and container-managed persistent beans is defined by who manages the persistence of the entity bean's data.

In practical terms, the following table provides a definition for both types and a summary of the programmatic and declarative differences between them:

	Bean-Managed Persistence	**Container-Managed Persistence**
Persistence management	You are required to implement the persistence management within the `ejbStore` and `ejbLoad` EntityBean methods. These methods must contain logic for saving and restoring the persistent data. For example, the `ejbStore` method must have logic in it to store the entity bean's data to the appropriate database. If it does not, the data can be lost. See "3. Implementing EntityBean Interface Methods" on page 4-17 for an example of implementing bean-managed persistence.	The management of the persistent data is done for you. That is, the container invokes a persistence manager on behalf of your bean. You use `ejbStore` and `ejbLoad` for preparing the data before the commit or for manipulating the data after it is refreshed from the database. The container always invokes the `ejbStore` method right before the commit. In addition, it always invokes the `ejbLoad` method right after reinstating CMP data from the database.
Finder methods allowed	The `findByPrimaryKey` method and any other finder method you wish to implement are allowed.	Only the `findByPrimaryKey` method and a finder method for the where clause are allowed.
Defining CMP fields	N/A	Required within the EJB deployment descriptor. The primary key must also be declared as a CMP field.
Mapping CMP fields to resource destination.	N/A	Required. Dependent on persistence manager.
Definition of persistence manager.	N/A	Required within the Oracle-specific deployment descriptor. See the next section for a description of a persistence manager.

Container-Managed Persistence

You can choose to have the container manage your persistent data for the bean. You have less to develop and manage, as the container stores and reloads your persistent data to and from the database.

When you use container-managed persistence, the container invokes a persistence manager class that provides the persistence management business logic. For this release, the only supported persistence manager is Oracle Persistence Service Interface Reference Implementation (PSI-RI).

To enable the container to manage your persistent data, you need to perform the following:

1. Modify the appropriate bean class callback methods

2. Define the primary key

3. Declare the container-managed persistent fields within the deployment descriptor

4. Declare the persistence manager class

5. Map container-managed persistent fields to a database

Modify Bean Class Callback Methods

If you do not want to manage your persistent data, choose to have your bean managed by the container. This means that you do not have to implement some of the callback methods as the container and the persistence manager performs the persistence and primary key management for you. The container will still call these methods—so you can add logic for other purposes. You still must provide at least an empty implementation for all callback methods.

The following table details the implementation requirements for the bean class' callback functions:

Callback Method	Functionality Required
ejbCreate	The same functionality as bean-managed persistent beans. You must initialize all container-managed persistent fields, including the primary key.
ejbPostCreate	The same functionality as bean-managed persistent beans. You have the option to provide any other initialization, which can involve the entity context.

Callback Method	Functionality Required
ejbRemove	No functionality for removing the persistent data from the outside resource is required. The persistent manager removes all persistent data associated with the entity bean from the database. You must at least provide an empty implementation for the callback, which means that you can add logic for performing any cleanup functionality you require.
ejbFindByPrimaryKey	No functionality is required for returning the primary key to the container. The container manages the primary key—after it is initialized by the ejbCreate method. Thus, the container performs the functionality normally required of this method. You still must provide an empty implementation for this method.
ejbStore	No functionaltiy is required for saving persistent data within this method. The persistent manager saves all persistent data to the database for you. However, you must provide at least an empty implementation as the container invokes the ejbStore method before invoking the persistent manager. This enables you to perform any data management or cleanup before the persistent data is saved.
ejbLoad	No functionality is required for restoring persistent data within this method. The persistence manager restores all persistent data for you. However, you must provide at least an empty implementation as the container invokes the ejbLoad method after invoking the persistent manager. This enables you to perform any logic to manipulate the persistent data after it is restored to the bean.
setEntityContext	Associates the bean instance with context information. The container calls this method after the bean creation. The enterprise bean can store the reference to the context object in an instance variable, for use in transaction management. Beans that manage their own transactions can use the session context to get the transaction context.
	You can also allocate any resources that will exist for the lifetime of the bean within this method. You should release these resources in unsetEntityContext.
unsetEntityContext	Unset the associated entity context and release any resources allocated in setEntityContext.

A Finder Method for the Where Clause Oracle enables you to perform a SQL query against the persistent data table through a CMP-only finder method with a find<*name*> naming syntax. This method takes a String that denotes the "where"

clause of a SQL query. Thus, the String would include the entire statement except for the `"select * from <table"`. If you supply an empty string, all values are selected from this table.

You must define any such finder method within the Home interface. The container will provide the implementation for satisfying the where clause for the finder method.

For example, the following defines finder methods within the home interface, where one retrieves all customers and the other retrieves a single customer. Notice how the `findAllCustomers` method, which retrieves all customers in the table, returns an `Enumeration`.

> **Note:**   Normally, the return type for multiple items in a finder method would be a `Collection`. However, `Collection` is not supported in this release. You must use the `Enumeration` type to receive multiple items from any finder method.

```
public Customer findByWhere(String whereString)
                          throws RemoteException, FinderException;

public java.util.Enumeration findMultipleCustomers(String whereString)
                          throws RemoteException, FinderException;
```

If you want to retrieve a single customer, provide the name, the SQL would be constructed as follows:

```
select * from customer where name = "Smith, John";
```

The `findbyWhere` finder method would include the entire statement except for the `"select * from customer"`. It assumes that you want to select against the persistence table, as follows:

```
Customer find_customer = findByWhere ("where name = ", + custname);
```

If you want to select a few rows from employee based upon a certain condition, the SQL would be constructed as follows:

```
select * from customer where item_bought = treadmill order by name;
```

The find<*name*> method, which in this example is findByWhere, would include the entire statement except for the `"select * from employee"`. It assumes that you want to select all matches against the persistence table. This is demonstrated below:

```
Enumeration customer_list = findMultipleCustomers(
                            "where item_bought = treadmill order by name");
```

Or, if you wanted a full customer listing, provide an empty string. The container will invoke a "`select * from customer`" and retrieve all records.

```
Enumeration customer_list = findMultipleCustomers();
```

Define Your Primary Key

The main difference between defining a bean-managed and container-managed persistent primary key is that the fields within the key must be declared as container-managed persistent fields in the deployment descriptor. All fields within the primary key are restricted to be either primitive, serializable, and types that can be mapped to SQL types. See "Persistence Fields" on page 4-17 for more information.

You can define your primary key in one of two ways:

- Defining A Single Object as your Primary Key
- Defining a Complex Primary Key Class

Defining A Single Object as your Primary Key Define your primary key as a container-managed persistent field and its type within the deployment descriptor. The following shows the primary key, custid, declared as a `<cmp-field>` and `<primkey-field>` and its type declared within the `<prim-key-class>`:

```
<enterprise-beans>
  <entity>
      <description>customer bean</description>
      <ejb-name>/test/customer</ejb-name>
      <home>customer.CustomerHome</home>
      <remote>customer.Customer</remote>
      <ejb-class>customerServer.CustomerBean</ejb-class>
      <persistence-type>Container</persistence-type>
      <prim-key-class>java.lang.String</prim-key-class>
      <primkey-field>custid</primkey-field>
      <reentrant>False</reentrant>
      <cmp-field><field-name>custid</field-name></cmp-field>
      <cmp-field><field-name>name</field-name></cmp-field>
      <cmp-field><field-name>addr</field-name></cmp-field>
  </entity>
</enterprise-beans>
```

The primary key variable declared within the bean class must be declared as `public`.

This should be a Java type that can be mapped to a SQL type through SQLJ. Also, this object must be serializable. See "Entity Bean Elements" on page 4-13 for a full description.

Defining a Complex Primary Key Class If your primary key is more complex than a simple data type, you define the fields that make up the primary key as container-managed fields within the deployment descriptor. Then, you declare these fields as the primary key within a class. This class must be serializable. Also, all primary key variables declared within the bean class must be types that can be mapped to SQL types.

Within the bean class, the primary key variables must be declared as `public`. Also, you must provide a constructor with no arguments for creating an empty primary key instance.

Within the serializable primary key class, you implement the `equals` and `hashCode` methods to provide for an implementation specific to this primary key.

The following example is a cruise ship cabin bean primary key that identifies each cabin with ship name, deck name, and cabin number.

```
package cruise;

public class CabinPK implements java.io.Serializable
{
//Ship names { Castaway, LightFantastic, FantasyRide }
  public String ship;

//Deck names { Upper Promendade, Promenade, Lower Promenade, Main Deck,
                  Lower Deck }
  public String deck;

  //Cabin numbers A100–N300
  public String cabin;

  //empty constructor
  public CabinPK ( ) { }

  public boolean equals(Object obj) {
    if ((obj instanceof CabinPK) &&
        (((CabinPK)obj).ship == this.ship) &&
         (((CabinPK)obj).deck == this.deck) &&
         (((CabinPK)obj).cabin == this.cabin))
```

```
      return true;
    return false;
  }

  public int hashCode() {
    return ((ship + deck + cabin).hash);
  }
```

The class that defines the primary key is declared within the XML deployment descriptor, as follows:

```
<enterprise-beans>
     <entity>
         <ejb-name>CabinBean</ejb-name>
         <home>cruise.CabinHome</home>
         <remote>cruise.Cabin</remote>
         <ejb-class>cruiseServer.CabinBean</ejb-class>
         <persistence-type>Container</persistence-type>
         <prim-key-class>cruise.CabinPK</prim-key-class>
         <reentrant>False</reentrant>
         <cmp-field><field-name>deck</field-name></cmp-field>
         <cmp-field><field-name>cabin</field-name></cmp-field>
...
</enterprise-beans>
```

See "Entity Bean Elements" on page 4-13 for a full description of the deployment descriptor definition.

Manage the Primary Key You must initialize all CMP fields, including the primary key, within the ejbCreate method. The ejbCreate method for the purchase order example initializes the primary key, ponumber, to the next available number in the purchase order number sequence.

```
// The create methods takes care of generating a new PO and returns
// its primary key
public String ejbCreate () throws CreateException, RemoteException
{
  String ponumber = null;
  try {
    //retrieve the next available purchase order number
    #sql { select ponumber.nextval into :ponumber from dual };
    //assign this number as this instance's identification number
    #sql { insert into pos (ponumber, status) values (:ponumber, 'OPEN') };
  } catch (SQLException e) {
    throw new PurchaseException (this, "create", e);
```

```
  }
  return ponumber;
}
```

Declare Persistence Fields

All container-managed persistent fields must be declared as public within your bean class. They cannot be transient. In addition, these fields are restricted to be either primitive, serializable, and types that can be mapped to SQL types. See Table A–2, "Unsupported Java Types for Persistent Variables" on page A-30 for more information.

Declare Persistence Provider

The container invokes a persistence provider for managing your CMP bean. This release supports Oracle Persistence Service Interface Reference Implementation (PSI-RI). For more information on defining the provider, see "Defining Container-Managed Persistence" on page A-28 for a full description.

> **Note:** Make sure that the class that contains the persistence manager is loaded within the database.

The following shows the portion of the Oracle-specific deployment descriptor that defines the persistence manager:

```
...
<persistence-provider>
  <description> specifies a type of persistence manager </description>
  <persistence-name>psi-ri</persistence-name>
  <persistence-deployer>oracle.aurora.ejb.persistence.ocmp.OcmpEntityDeployer</persistence-deployer>
</persistence-provider>

<persistence-descriptor>
  <description> This specifies a particular type of persistence manager to be used for a bean.  param is where you would put bean specific persistence info in the format of params.  The deployment process just passes what's in the param to the persistence deployer.  For the baby persistence, we do parse the persistence-mapping but for other persistence backend we don't do anything with the params
  </description>
  <ejb-name>customerbean</ejb-name>
  <persistence-name>psi-ri</persistence-name>
  <psi-ri>
```

```
       <schema>SCOTT</schema>
       <table>customers</table>
       <attr-mapping>
         <field-name>custid</field-name>
         <column-name>cust_id</column-name>
       </attr-mapping>
       <attr-mapping>
         <field-name>name</field-name>
         <column-name>cust_name</column-name>
       </attr-mapping>
       <attr-mapping>
         <field-name>addr</field-name>
         <column-name>cust_addr</column-name>
       </attr-mapping>
   </psi-ri>
</persistence-descriptor>
</oracle-descriptor>
```

Map Container-Managed Persistence Fields

All CMP data fields defined within your bean must be declared within the deployment descriptor in the `<cmp-field>` element. See "Entity Bean Elements" on page 4-13 for a full description. In addition, you must map these fields to the intended database table and respective columns. See "Persistence Fields" on page 4-17 for more information.

The following is a portion of the Oracle-specific deployment descriptor, which maps the primary key and any container-managed persistence fields from the customer example (with the single primary key field) to the database table and column that the persistence provider stores the values for these fields within. Specifically, the container-manager stores the persistent fields into the customers table in SCOTT's schema. The persistent fields are mapped as follows:

Persistent Field	Table Column
custid (primary key)	cust_id column in the customers table
name	cust_name column in the customers table
addr	cust_addr column in the customers table

```
<psi-ri>
     <schema>SCOTT</schema>
     <table>customers</table>
     <attr-mapping>
```

```
                    <field-name>custid</field-name>
                    <column-name>cust_id</column-name>
                </attr-mapping>
                <attr-mapping>
                    <field-name>name</field-name>
                    <column-name>cust_name</column-name>
                </attr-mapping>
                <attr-mapping>
                    <field-name>addr</field-name>
                    <column-name>cust_addr</column-name>
                </attr-mapping>
</psi-ri>
```

Accessing EJB References and JDBC DataSources

You can access EJB references and JDBC DataSources through your bean's environment.

- EJB References
- JDBC DataSources

EJB References

The entity bean may create an environment reference for a target bean within the deployment descriptors. The URL of an EJB environment reference should have the following syntax:

```
"java:comp/env/ejb/"<ejb-ref-name>
```

The `"java:comp/env/ejb"` prefix is a subcontext that instructs JNDI to locate the EJB reference within the EJB references defined in the deployment descriptor. The `<ejb-ref-name>` is the logical environment name of the EJB reference defined in the deployment descriptor. The following shows how a bean looks up another bean's reference within its deployed environment:

```
Context ic = new InitialContext ( );

CustomerHome ch = (CustomerHome)ic.lookup ("java:comp/env/ejb/PurchaseOrder");
```

See "Environment References To Other Enterprise JavaBeans" on page A-9 for a full description of EJB environment references.

JDBC DataSources

The entity bean had the option of creating an environment reference for JDBC
DataSources bound within JNDI. These were declared for the bean within the
deployment descriptors. If defined, the serviceURL and objectName contains
the URL of an EJB environment reference, which is of the following syntax:

```
"java:comp/env/jdbc/"<resource-ref-name>
```

The "java:comp/env/jdbc" prefix is a subcontext that instructs JNDI to locate
the JDBC DataSource within the <resource-ref> elements defined in the
deployment descriptor, which actually defines the logical environment name.

The following is the definition of the JDBC DataSource within the EJB deployment
descriptor for the purchase order:

```
<resource-ref>
          <res-ref-name>jdbc/EmployeeAppDB</res-ref-name>
          <res-type>javax.sql.DataSource</res-type>
          <res-auth>Application</res-auth>
</resource-ref>
```

> **Note:** Only the Application option is supported for
> <res-auth> for this release.

The following shows the JNDI context initialization for the purchase order example:

```
#import javax.sql.*

String dbURL = "java:comp/env/jdbc/EmployeeAppDB";

DataSource dbRes = (DataSource)ic.lookup (dbURL);
Connection conn = dbRes.getConnection;
```

See "Environment References To Resource Manager Connection Factory References"
on page A-12 for a full description of JDBC DataSource variables.

5

JNDI Connections and Session IIOP Service

This chapter describes in detail how clients connect to an Oracle8i server session and how they authenticate themselves to the server. The term *client,* as used in this chapter, includes client applications and applets running on a network PC or a workstation, as well as distributed objects such as EJBs and CORBA server objects that are calling other distributed server objects and, thus, acting as clients to these objects.

In addition to authentication, this chapter discusses security of access control to objects in the database. A published object in the data server has a set of permissions that determine who can access and modify the object. In addition, classes that are loaded in the data server are loaded into a particular schema, and the person who deploys the classes can control who can use them.

This chapter covers the following topics:

- JNDI Connection Basics
- The Name Space
- Execution Rights to Database Objects
- URL Syntax
- Using JNDI to Access Bound Objects
- Session IIOP Service
- Retrieving JServer Version Number
- Activating In-Session EJB Objects From Non-IIOP Presentations
- Invoking EJB Objects From Applets

JNDI Connection Basics

The client example in Chapter 2 showed how to connect to Oracle, start a database server session, and activate an object using a single URL specification. This was performed through the following steps:

```
1. Hashtable env = new Hashtable();
2. env.put(javax.naming.Context.URL_PKG_PREFIXES, "oracle.aurora.jndi");
3. env.put(javax.naming.Context.SECURITY_PRINCIPAL, username);
4. env.put(javax.naming.Context.SECURITY_CREDENTIALS, password);
5. env.put(javax.naming.Context.SECURITY_AUTHENTICATION,
                        ServiceCtx.NON_SSL_LOGIN);
6. Context ic = new InitialContext(env);
7. HelloHome hello_home =
   (HelloHome) ic.lookup("sess_iiop://localhost:2481:ORCL/test/myHello");
   myHello hello = hello_home.create ();
8. System.out.println(hello.helloWorld());
```

In this example, there are four basic operations:

- Lines 1-5 set up an environment for the JNDI initial context.

- Line 6 creates the JNDI initial context.

- Line 7 looks up a published object. (See "URL Syntax" on page 5-5 for a discussion of the URL syntax.)

- Line 8 invokes a method on the object.

When a client looks up an object through the JNDI lookup method, the client and server automatically perform the following logic:

- A session IIOP connection is created to the ORCL instance of the local host database.

- The server establishes a database session.

- The client is authenticated, using the NON_SSL_LOGIN protocol, with the username and password specified in the environment context.

- The published object, /test/myHello, is located in the session namespace and a reference to it is returned to the client.

When the client invokes a method—such as helloWorld()—on the returned reference, the server activates the object in the server.

The Name Space

The name space in the database looks just like a typical file system. You can examine and manipulate objects in the publishing name space using the session shell tool. See the "sess_sh" tool in the *Oracle8i Tools Reference Guide* for information about the session shell.

There is a root directory, indicated by a forward slash ('/'). The root directory is built to contain three other directories: bin, etc, and test. The /test directory is where most objects are published for the example programs. You can create new directories under root to hold objects for separate projects; however, you must have access as database user SYS to create new directories under the root.

There is no effective limit to the depth that you can nest directories.

> **Note:** The initial values in the publishing name space are set up when the JServer product for Oracle8*i* is installed.

The /etc directory contains objects the ORB uses. Do not delete objects in the /etc directory. The objects contained in /etc are:

deployejb   execute   loadjava   login   transactionFactory

The entries in the name space are represented by objects that are instances of the following classes:

- `oracle.aurora.AuroraServices.PublishingContext`—represents a class that can contain other objects (a directory)

- `oracle.aurora.AuroraServices.PublishedObject`—used for the leafs of the tree, that is the object names themselves.

These classes are documented in the JavaDoc on the product CD.

Published names for objects are stored in a database table. Each published object also has a set of associated permissions. Each class or resource file can have a combination (union) of the following permissions:

read The holder of read rights can list the class or the attributes of the class, such as its name, its helper class, and its owner.

write The holder of write for a context can bind new object names into a context. For an object (a leaf node of the tree), write allows the holder to republish the object under a different name.

execute You must have execute rights to resolve and activate an object represented by a context or published object name.

You use the `chmod` command of the session shell tool to view and change object rights.

Execution Rights to Database Objects

In addition to authentication and privacy, Oracle8*i* supports controlled access to the classes that make up CORBA and EJB objects. Only users or roles that have been granted execute rights to the Java class of an object stored in the database can activate the object and invoke methods on it.

You can control execute rights on Java classes with the following tools:

- At load time with the `-grant` argument to `loadjava`. See the *Oracle8i Java Developer's Guide* for more information about `loadjava` and execution rights on Java classes in the database.

- Using SQL commands—You use the SQL DDL GRANT EXECUTE command to grant execute permission on a Java class loaded in the database. For example, if SCOTT has loaded a class Hello, then SCOTT (or SYS) can grant execute privileges on that class to another user, say OTTO, by issuing the SQL command:

  ```
  SQL> GRANT EXECUTE ON "Hello" TO OTTO;
  ```

  Use the SQL command REVOKE EXECUTE to remove execute rights for a user from a loaded Java class.

- At publish time—Published objects are not restricted to a specific schema; they are potentially available to all users in the instance. Published objects have permissions that can differ from underlying classes. For example, if user SCOTT has execute permission on a published object name, but does not have execute permission on the class that the published object represents, SCOTT will not be able to activate the object.

  You can control permissions on a published object through the following:

  1. Using the `-grant` option with the `publish` tool.
  2. Using the `chmod` and `chown` commands within the Session Shell. You must be connected to the Session Shell as the user SYS to use the `chown` command.

Use the `ls -l` command in the session shell to view the permissions (EXECUTE, READ, and WRITE) and the owner of a published object.

There are three "built-in" server objects that a client can access without being authenticated, as shown below:
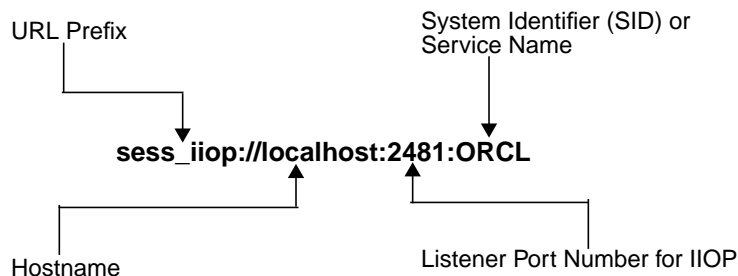
- the Name Service
- the `InitialReferences` object (the boot service)
- the `Login` object

You can activate these objects using `serviceCtx.lookup()` without authentication. See the "Logging In and Out of the JServer Session" on page 6-11 for an example that access the `Login` object explicitly.

URL Syntax

Oracle8*i* provides universal resource locator (URL) syntax to access services and sessions. The URL lets you use JNDI requests to start up services and sessions, and also to access components published in the database instance. An example service URL is shown in Figure 5–1.

Figure 5–1   Service URL

The service URL is composed of four components:

1. The URL prefix followed by a colon and two slashes: *sess_iiop://* for a session IIOP request.
2. The system name (the hostname). For example: `myPC-1`. You can also use `localhost` or the numeric form of the IP address for the host.
3. The listener port number for IIOP services. The default is 2481.

4. The system identifier (SID)—for example, `ORCL`—or the service name—for example, `mySID.myDomain`.

- SID—The system identifier is defined in your database initialization file as the db_name. This identifies the database instance you are connecting to. If you choose to add the SID to your service URL, the listener will load balance incoming requests across multiple dispatchers for the database instance.

- Service name—The service name is equivalent to either the `service_name` or the `db_name.db_domain` parameters defined in your database initialization file. If you use the service name within your service URL, the listener will load balance incoming requests across multiple database instances: that is, all database instances registered with the listener. This options is good when you are using parallel servers.

> **Note:** If you do use the service name, you must specify the `-useServiceName` flag on any tool that takes in the URL. If you do not specify this flag, the tool assumes that the last string is a SID.

Always use colons to separate the hostname, port, and SID or service name.

> **Note:** If you specify a dispatcher port instead of a listener port, and you specify a SID, an `ObjectNotFound` exception is thrown by the server. Because applications that connect directly to dispatcher ports do not scale well, Oracle does not recommend direct connection to dispatchers.

URL Components and Classes

When you make a connection to Oracle and look up a published object using JNDI, you use a URL that specifies the service (service name, host, port, and SID), as well as the name of a published object to look up and activate. For example, a complete URL could look like:

```
sess_iiop://localhost:2481:ORCL/:default/projectAurora/Plans816/getPlans
```

where `sess_iiop://localhost:2481:ORCL` specifies the service name, `:default` indicates the default session (when a session has already been established), `/projectAurora/Plans816` specifies a directory path in the namespace, and `getPlans` is the name of a published object to look up.

> **Note:** You do not specify the session name when no session has been established for that connection. That is, on the first look up there is no session active; therefore, `:default` as a session name has no meaning. In addition, `:default` is implied, so you can use a URL without a session name.

Each component of the URL represents a Java class. For example, the service name is represented by a `ServiceCtx` class instance, the session by a `SessionCtx` instance. See "Using JNDI to Access Bound Objects" and "Session IIOP Service" starting on page 5-7 for more information on the service and session names within the URL.

CosNaming Restriction for JNDI Name

The JNDI bound name for the published object must use JNDI syntax rules. The underlying naming service that JServer JNDI uses is CosNaming. Thus, if your name includes a dot (".") in one of the names, the behavior diverges from normal CosNaming rules, as follows:

- The substring before the dot is treated as a CosNaming `NameComponent` id.

- The substring after the dot is treated as a CosNaming `NameComponent` kind.

- Both id and kind are concatenated into a full JNDI name.

Normally, in retrieving a CosNaming object, you supply the id and kind as separate entities. The JServer implementation concatenates both id and kind. Thus, to retrieve the object, your application refers to the full name with the dot included as part of the JNDI name, rather than as a separator.

Using JNDI to Access Bound Objects

Clients use the *Java Naming and Directory Interface* (JNDI) interface to look up published objects in the JServer namespace. JNDI is an interface supplied by Sun Microsystems that gives the Java application developer a methodology to access name and directory services. This section discusses only those parts of the JNDI API that are needed to look up and activate published objects. To obtain a complete set of documentation for JNDI, see the web site URL
`http://java.sun.com/products/jndi/index.html`.

> **Note:**   It is also possible to access the session namespace without using JNDI. Instead, you can use CosNaming methods.

As described in "URL Syntax" on page 5-5, the JNDI URL required to access any bound name in the JServer namespace requires a compound name consisting of the following two components:

- Service name—the service name is used by JServer to retrieve a handle to the correct namespace.

  Several namespaces will exist within your network. The service specifies which namespace to retrieve the JNDI bound object from. Service names can be one of the following:

Service Name	Description
sess_iiop://<hostname>:<port>:<SID>	Specifies the host, port, and SID that the desired namespace is located. Specifying this service name only without a session name returns a ServiceCtx object. The session IIOP service is the main service used by IIOP applications. It retrieves objects and object references bound in JNDI namespaces on different database hosts. See "Session IIOP Service" on page 5-13 for a full description.
jdbc_access:	Specifies that the desired object exists in a well-known namespace. Used primarily to retrieve JTA UserTransaction and DataSource objects from the namespace.
java:	Used to specify that the bound name is actually an EJB environment variable that was specified within its deployment descriptor.

- Session name—the actual JNDI bound name of the object within the designated namespace. The syntax mimics a UNIX file system syntax. The session name can be represented by a `SessionCtx` object.

You can utilize the service and session contexts to perform some advanced techniques, such as opening different sessions within a database or enabling several clients to access an object in a single session. These are discussed further in the "Session IIOP Service" on page 5-13. However, for simple JNDI lookup invocations, you should use the URL syntax specified in "URL Syntax" on page 5-5.

Importing JNDI Support Classes

When you use JNDI in your client or server object implementations, be sure to include the following import statements in each source file:

```
import javax.naming.Context;      // the JNDI Context interface
import javax.naming.InitialContext;
import oracle.aurora.jndi.sess_iiop.ServiceCtx; // JNDI property constants
import java.util.Hashtable;       // hashtable for the initial context environment
```

Retrieving the JNDI InitialContext

`Context` is an interface in the `javax.naming` package that is used to retrieve the `InitialContext`. All Oracle8*i* EJB and CORBA clients use the `InitialContext` for JNDI `lookup()`. Before you perform a JNDI `lookup()`, you set the environment variables, such as authentication information into the `Context`. You can use a hash table or a properties list for the environment. Then, this information is made available to the naming service when the `lookup()` is performed. The examples in this guide always use a Java `Hashtable`, as follows:

```
Hashtable environment = new Hashtable();
```

Next, you set up properties in the hash table. You must always set the `Context` `URL_PKG_PREFIXES` property, whether you are on the client or server. The remaining properties are used for authentication, which primarily are used by clients or by a server authenticating itself as another user.

- `javax.naming.Context.URL_PKG_PREFIXES`

- `javax.naming.Context.SECURITY_PRINCIPAL`

- `javax.naming.Context.SECURITY_CREDENTIALS`

- `javax.naming.Context.SECURITY_ROLE`

- `javax.naming.Context.SECURITY_AUTHENTICATION`

- `USE_SERVICE_NAME`

URL_PKG_PREFIXES

`Context.URL_PKG_PREFIXES` holds the name of the environment property for specifying the list of package prefixes to use when loading in URL context factories. The value of the property should be a colon-separated list of package prefixes for the class name of the factory class that will create a URL context factory.

In the current implementation, you must always supply this property in the Context environment, and it must be set to the String "`oracle.aurora.jndi`".

SECURITY_PRINCIPAL

`Context.SECURITY_PRINCIPAL` holds the database username.

SECURITY_CREDENTIALS

`Context.SECURITY_CREDENTIAL` holds the clear-text password. This is the Oracle database password for the SECURITY_PRINCIPAL (the database user). In all of the three authentication methods mentioned in SECURITY_AUTHENTICATION below, the password is encrypted when it is transmitted to the server.

SECURITY_ROLE

`Context.SECURITY_ROLE` holds the Oracle8*i* database role with which the user is connecting. For example, "CLERK" or "MANAGER".

SECURITY_AUTHENTICATION

`Context.SECURITY_AUTHENTICATION` holds the name of the environment property that specifies the type of authentication to use. Values for this property provide for the authentication types supported by Oracle8*i*. There are four possible values, which are defined in the `ServiceCtx` class:

- `ServiceCtx.NON_SSL_LOGIN`: The client authenticates itself to the server with a username and password using the Login protocol over a standard TCP/IP connection (not a secure socket layer connection). The Login protocol provides for encryption of the password as it is transmitted from the client to the server. The server does not authenticate itself to the client. See "Providing Username and Password for Client-Side Authentication" on page 6-9 for more information about this protocol.

- `ServiceCtx.SSL_CREDENTIAL`: The client authenticates itself to the server providing a username and password that are encrypted over a secure socket layer (SSL) connection. The server authenticates itself to the client by providing credentials.

- `SSL_LOGIN`: The client authenticates itself to the server with a username and password within the Login protocol, over an SSL connection. The server does not authenticate itself to the client.

- `SSL_CLIENT_AUTH`: Both the client and the server authenticate themselves to each other by providing certificates to each other over an SSL connection.

> **Note:** To use an SSL connection, you must be able to access a
> listener that has an SSL port configured, and the listener must be
> able to redirect requests to an SSL-enabled database IIOP port. You
> must also include the following JAR files when you compile and
> build your application:
>
> - If your client uses JDK 1.1, import `jssl-1_1.jar` and
>   `javax-ssl-1_1.jar`.
> - If your client uses Java 2, import `jssl-1_2.jar` and
>   `javax-ssl-1_2.jar`.

USE_SERVICE_NAME

If you are using a service name instead of an SID in the URL, you set this property
to true. Otherwise, the last string in the URL must contain the SID. Given a
Hashtable within the variable, `env`, the following designates that the service name
is used instead of the SID within the URL:

```
Hashtable env = new Hashtable();
env.put(Context.URL_PKG_PREFIXES, "oracle.aurora.jndi");
env.put(Context.SECURITY_PRINCIPAL, "scott");
env.put(Context.SECURITY_CREDENTIALS, "tiger");
env.put(Context.SECURITY_AUTHENTICATION, ServiceCtx.NON_SSL_LOGIN);
env.put("USE_SERVICE_NAME", "true");
Context ic = new InitialContext(env);
```

The default is false.

The URL given within the lookup should contain a service name, instead of an SID.
The following URL contains the service name, `orasun12`:

```
myHello hello =
      (myHello) ic.lookup("sess_iiop://localhost:2481:orasun12/test/myHello");
```

The JNDI InitialContext Methods

`InitialContext` is a class in the `javax.naming` package that implements the
`Context` interface. All naming operations are relative to a context. The initial
context implements the `Context` interface and provides the starting point for
resolution of names.

Constructor

You construct a new initial context using the constructor:

```
public InitialContext(Hashtable environment)
```

It requires a Hashtable for the input parameter that contains the environment information described in "Retrieving the JNDI InitialContext" above. The following code fragment sets up an environment for a typical client, and creates a new initial context:

```
Hashtable env = new Hashtable();
env.put(Context.URL_PKG_PREFIXES, "oracle.aurora.jndi");
env.put(Context.SECURITY_PRINCIPAL, "scott");
env.put(Context.SECURITY_CREDENTIALS, "tiger");
env.put(Context.SECURITY_AUTHENTICATION, ServiceCtx.NON_SSL_LOGIN);
Context ic = new InitialContext(env);
```

lookup

This is the most common initial context class method that the CORBA or EJB application developer will use:

```
public Object lookup(String URL)
```

You use lookup() to retrieve an object instance or to create a new service context.

- To retrieve an object instance, specify a URL for the service name and append the JNDI bound name (the session name). The returned result must be cast to the expected object type. For example, to retrieve the Hello interface, you would do the following:

  ```
  HelloHome hello_home =
      (HelloHome) ic.lookup("sess_iiop://localhost:2481:ORCL/test/myHello");
  myHello hello = hello_home.create ();
  ```

  The service name is "sess_iiop://localhost:2481:ORCL"; the JNDI bound name for Hello's home interface is "/test/myHello".

- To retrieve a handle to a specific namespace, specify the desired service context. The return result must be cast to ServiceCtx when a new service context is being created. For example, if initContext is a JNDI initial context, the following statement creates a new service context:

  ```
  ServiceCtx service =
      (ServiceCtx) initContext.lookup("sess_iiop://localhost:2481:ORCL");
  ```

See "Session Management Scenarios" on page 5-18 for examples on how to use the JNDI lookup method within an EJB or CORBA application.

Session IIOP Service

All client/server network communications route requests over an accepted protocol between both entities. Most network communications to the Oracle8*i* database routed over the two-task common (TTC) layer. This is the service that processes incoming Net8 requests for database SQL services. However, with the addition of Java into the database, JServer required that clients communicate with the server over an IIOP transport that recognized database sessions. This is accomplished through the session IIOP service.

The session IIOP service is used for facilitating requests for IIOP applications, which includes CORBA and EJB applications. The following sections describe how to manage your applications within one or more database sessions:

- Session IIOP Service Overview
- Session Management
- Service Context Class
- Session Context Class
- Session Management Scenarios
- Setting Session Timeout

Session IIOP Service Overview

As discussed in the *Oracle8i Java Developer's Guide,* since the EJB is loaded into the database, your client application must start up the EJB within the context of a database session. Because beans are activated within a session, each client cannot see bean instances active in another session, unless given a handle to that session. Also, you can activate objects within the existing session or another session.

The session IIOP service session component tag—TAG_SESSION_IIOP— exists inside the IIOP profile—`SessionIIOP`. The value for this Oracle session IIOP component tag is 0x4f524100 and contains information that uniquely identifies the session in which the object was activated. The client ORB runtime uses this information to send requests to objects in a particular session.

Although the Oracle8*i* session IIOP service provides an enhancement of the standard IIOP protocol—it includes session ID information—it does not differ from standard IIOP in its on-the-wire data transfer protocol.

Client Requirements

Clients must have an ORB implementation that supports session IIOP to be able to access objects in different sessions simultaneously, from within the same program, and to be able to disconnect from and reconnect to the same session. The version of the Visigenic ORB that ships with Oracle8*i* has been extended to support session IIOP.

Session Routing

When a client makes an IIOP connection to the database, Oracle8*i* decides if a new session should be started to handle the request, or if the request should be routed to an existing session. If the client initializes a new request for a connection (using the `InitialContext.lookup()` method) and no session is active for that connection, a new session is automatically started. If a session has already been activated for the client, the session identifier is encoded into the object key of the object. This information enables the session IIOP service to route the request to the correct session. In addition, you can also use this session identifier to allow a single client to access multiple sessions. See "Session Management Scenarios" on page 5-18 for more information.

JServer Tools

When using the Oracle8*i* JServer tools, especially when developing EJB and CORBA applications, it is very important to distinguish the two network service protocol types: TTC and IIOP.

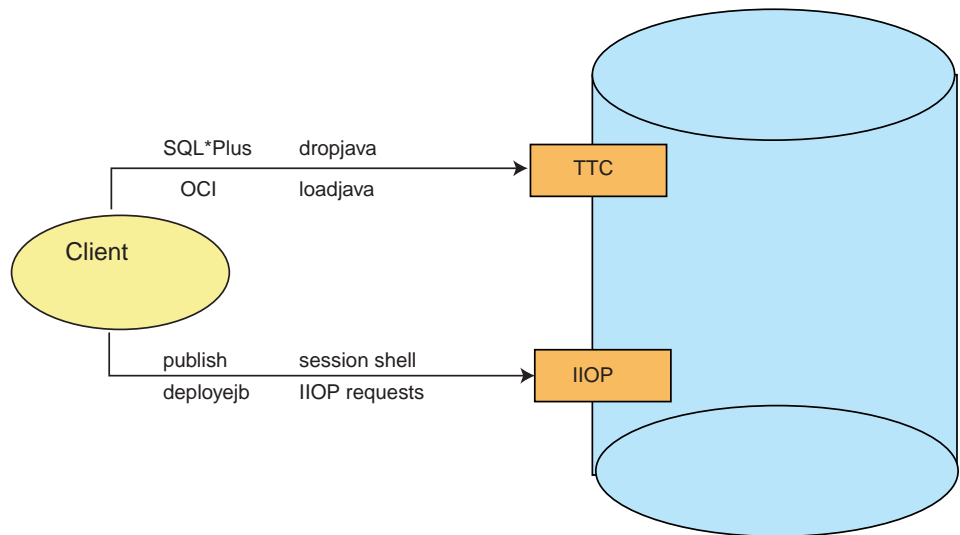**Figure 5–2   TTC and IIOP Services**

Figure 5–2 shows which tools and requests use TTC and which use IIOP database ports. 1521 is the default port number for TTC, and 2481 is the default for IIOP.

- Tools such as `publish`, `deployejb`, and the session shell access IIOP objects, and so must connect using an IIOP port. In addition, EJB and CORBA clients must use an IIOP port when sending requests to Oracle.

- Tools such as `loadjava` and `dropjava` connect using a TTC port.

Session Management

In the simple cases, a client (or a server object acting as a client) starts a new server session implicitly when it performs the lookup for a server object. Oracle8*i* also gives you the ability to control session start-up explicitly. Two Oracle-specific classes are provided that gives you control over the session IIOP service connection and over the sessions within the database.

- Service Context Class—Controls the session IIOP service connection to the database. Given a URL to that database, you can create a service context. Off of this service context, you can open one or more named sessions within the database.

■ Session Context Class—Controls named database sessions created off of a service context. Once created, you can activate CORBA or EJB objects within the session using the named session context object.

Service Context Class

Controls the session IIOP service connection to the database. Given a URL to that database, you can create a service context. Off of this service context, you can open one or more named sessions within the database. This Oracle-specific class extends the JNDI `Context` class.

Variables

The `ServiceCtx` class defines a number of final public static variables that you can use to define environment properties and other variables. Table 5–1 shows these.

Table 5–1   ServiceCtx Public Variables

String Name	Value
NON_SSL_CREDENTIAL	"Credential"
NON_SSL_LOGIN	"Login"
SSL_CREDENTIAL	"SecureCredential"
SSL_LOGIN	"SecureLogin"
SSL_CLIENT_AUTH	"SslClientAuth"
SSL_30	"30"
SSL_20	"20"
SSL_30_WITH_20_HELLO	"30_WITH_20_HELLO"
Integer Name	**Integer Constructor**
SESS_IIOP	new Integer(2)
IIOP	new Integer(1)

Methods

The public methods in this class that CORBA and EJB application developers can use are as follows:

```
public Context createSubcontext(String name)
```

This method takes a Java String as the parameter and returns a JNDI `Context` object representing a session in the database. The method creates a new named session. The parameter is the name of the session to be created, which must start with a colon (:).

The return result should be cast to a `SessionCtx` object.

Throws `javax.naming.NamingException`.

```
public Context createSubcontext(Name name)
```

(Each of the methods that takes a **String** parameter has a corresponding method that takes a `Name` parameter. The functionality is the same.)

```
public static org.omg.CORBA.ORB init(String username,
                                     String password,
                                     String role,
                                     boolean ssl,
                                     java.util.Properties props)
```

Gets access to the ORB created when you perform a look up. Set the `ssl` parameter to **true** for SSL authentication. Clients that do not use JNDI to access server objects should use this method.

See "sharedsession" in Appendix A of the *Oracle8i CORBA Developer's Guide* for a usage example.

```
public Object lookup(String string)
```

`lookup()` looks up a published object in the database instance associated with the service context, and either returns an activated instance of the object, or throws `javax.naming.NamingException`.

Session Context Class

Controls named database sessions created off of a service context. Once created, you can activate CORBA or EJB objects within the session using the named session context object. Session contexts represent sessions and contain methods that enable you to perform session operations such as authenticating the client to the session or activating objects. This class extends the JNDI `Context` class.

> **Note:** Creating a subcontext within the session context affects the object type returned on the final JNDI lookup. See "Lookup of Objects Off of JNDI Context" on page 5-25 for more information.

Methods

The session context methods that a client uses are the following:

```
public synchronized boolean login()
```

`login()` authenticates the client using the initial context environment properties passed in the `InitialContext` constructor: username, password, and role.

```
public synchronized boolean login(String username,
                                    String password,
                                  String role)
```

`login()` authenticates the client using the username, password, and optional database role supplied as parameters.

```
public Object activate(String name)
```

Looks up and activates a published object having the name.

Session Management Scenarios

The following sections describe the different scenarios for managing database sessions:

- Client Accessing a Single Session—client activates and accesses an object in the :default session.

- Ending a Session—discusses methods that explicitly terminate a session.

- Client Starting a Named Session—client activates and accesses one or more objects in a session other than the :default session. This session is identified by a name within a SessionCtx.

- Two Clients Accessing the Same Session—two or more clients can access an activated object within a session by providing x and y to both clients.

- In-Session Activation—a server object, acting as a client, activates another object within the same session.

- Lookup of Objects Off of JNDI Context—lookup of a partial JNDI name requires that you activate the bound object.

Client Accessing a Single Session In general, when you look up a published object from a client with a URL, hostname, and port, the object is activated in a new session. For example, a client would perform the following:

```
Hashtable env = new Hashtable();
env.put(Context.URL_PKG_PREFIXES, "oracle.aurora.jndi");
env.put(Context.SECURITY_PRINCIPAL, "scott");
env.put(Context.SECURITY_CREDENTIALS, "tiger");
env.put(Context.SECURITY_AUTHENTICATION, ServiceCtx.NON_SSL_LOGIN);
Context ic = new InitialContext(env);
SomeObjectHome myObj_home =
    (SomeObjectHome) ic.lookup("sess_iiop://localhost:2481:ORCL/test/myObj");
SomeObject myObj = myObj_home.create ();
```

Activating an object in a new session from a server object is identical to starting a session from an application client. If the lookup method is invoked within the server object, the second object instance is activated in a separate session from the originating session.

Ending a Session Normally, a session terminates when the client terminates. However, if you want to explicitly terminate a session, you can do one of the following:

Terminate A Session From The Server-Side Using The Endsession Method
The server can control session termination by executing the following method:

```
oracle.aurora.mts.session.Session.THIS_SESSION().endSession();
```

Terminate A Session From The Client-side Using The Logout Object
If the client wishes to exit the session, it can execute the logout method of the LogoutServer object, which is pre-published as "/etc/logout". Only the session owner is allowed to logout. Any other owner receives a NO_PERMISSION exception.

The LogoutServer object is analogous to the LoginServer object, which is pre-published as "/etc/login". You can use the LoginServer object to retrieve the Login object, which is used to authenticate to the server. This is an alternative method to using the Login object within the JNDI lookup.

The following example shows how a client can authenticate using the LoginServer object and can exit the session through the LogoutServer object.

```
import oracle.aurora.AuroraServices.LoginServer;
import oracle.aurora.AuroraServices.LogoutServer;
...
// To log in using the LoginServer object
LoginServer loginServer = (LoginServer)ic.lookup(serviceURL + "/etc/login");
Login login = new Login(loginServer);
System.out.println("Logging in ..");
login.authenticate(user, password, null);
...
//To logout using the LogoutServer
LogoutServer logout = (LogoutServer)ic.lookup(serviceURL + "/etc/logout");
logout.logout();
```

Client Starting a Named Session You can explicitly create multiple session on the database instance through the JNDI methods provided in the `ServiceCtx` and `SessionCtx` classes.

The following `lookup` method contains a URL that defines the IIOP service URL of `"sess_iiop://localhost:5521:ORCL"` and a default session context.

```
SomeObjectHome myObj_home =
    (SomeObjectHome) ic.lookup("sess_iiop://localhost:2481:ORCL/test/myObj");
SomeObject myObj = myObj_home.create ();
```

In this simple case, the JNDI initial context `lookup` method implicitly starts a session and authenticates the client. This session becomes the default session, which is identified by the name `":default"`. All sessions are named. However, in the default case, the client does not need to know the name of the session, because all requests go to this single session. Unless specified, all additional objects activated will be activated in the default session. Even if you create a new JNDI initial context and look up the same or a new object, the object is instantiated in the same session as the first object.

The only way to activate objects within another session is to create a named session. You can create other sessions in place of or in addition to the default session by creating session contexts off of the service context. Each session would be a named session, so that you can activate objects in different sessions within the database.

1. Instantiate a new hashtable for the environment properties to be passed to the server.

    ```
    Hashtable env = new Hashtable();
    env.put(Context.URL_PKG_PREFIXES, "oracle.aurora.jndi");
    ```

> **Note:** Only the `URL_PKG_PREFIXES` `Context` variable is filled in—the other information will be provided in the `login.authenticate()` method parameters.

2. Create a new JNDI Context.

```
Context ic = new InitialContext(env);
```

3. Use the JNDI `lookup` method on the initial context, passing in the service URL, to establish a service context. This example uses a service URL with the service prefix of hostname, listener port, and SID.

> **Note:** Provide only the service URL of hostname, listener port, and database SID. If you provide the JNDI name of the desired object, a default session will be created for you.

```
ServiceCtx service =
    (ServiceCtx) ic.lookup("sess_iiop://localhost:2481:ORCL");
```

4. Create a session by invoking the `createSubcontext` method on the service context object. Provide the name for the session as a parameter to the `createSubcontext` method. A new session is created within the database.

```
SessionCtx session = (SessionCtx) service.createSubcontext(":session1");
```

> **Note:** You must name a new session when you create it. The session name must start with a colon (:), and cannot contain a slash (/), but is not otherwise restricted.

5. Authenticate the client program to the database by invoking the `login` method on the session context object.

```
session.login("scott", "tiger", null);    // role is null
```

6. Activate the object, identified by its bound JNDI name, in the named session.

   The activation of an EJB returns the home interface for the bean. Basically, steps 1-6 performs the same functionality as performing a JNDI `lookup()` for the home interface. It simply allows you to specify the specific named session to activate the object in.

```
// Activate one Hello object in the session
HelloHome hello_home = (HelloHome)session.activate (objectName);
```

7. Retrieve the Remote interface and invoke a method on the Hello Bean.

```
Hello hello = hello_home.create ();
System.out.println (hello.helloWorld ());
```

Example 5–1   Activating Objects in Named Sessions

The following example creates two named sessions of the name :session1 and
:session2. Each one retrieves the Hello object separately. The client invokes both
Hello objects in each named session.

```
Hashtable env = new Hashtable ();
env.put (Context.URL_PKG_PREFIXES, "oracle.aurora.jndi");
Context ic = new InitialContext (env);

// Get a SessionCtx that represents a database instance
ServiceCtx service = (ServiceCtx) ic.lookup ("sess_iiop://localhost:2481:ORCL");

// Create and authenticate a first session in the instance.
SessionCtx session1 = (SessionCtx) service.createSubcontext (":session1");

// Authenticate
session1.login("scott", "tiger", null);

// Create and authenticate a second session in the instance.
SessionCtx session2 = (SessionCtx) service.createSubcontext (":session2");

// Authenticate using a login object (not required, just shown for example).
LoginServer login_server2 = (LoginServer)session2.activate ("/etc/login");
Login login2 = new Login (login_server2);
login2.authenticate ("scott", "tiger", null);

// Activate one Hello object in each session
HelloHome hello_home1 = (HelloHome)session1.activate (objectName);
HelloHome hello_home2 = (HelloHome)session2.activate (objectName);

//create the bean and retrieve the remote interface
Hello hello1 = hello_home1.create ();
Hello hello2 = hello_home2.create ();

// Verify that the objects are indeed different
System.out.println (hello1.helloWorld ());
System.out.println (hello2.helloWorld ());
```

Two Clients Accessing the Same Session When the client invokes the JNDI lookup method, JServer creates a session. If you want a second client to access the instantiated object in this session, you must do the following:

1. The first client saves both the object instance handle and a Login object reference.

2. The second client retrieves the handle and Login object reference and uses them to access the object instance.

Example 5–2   Two Clients Accessing a Single Instance

1. The first client authenticates itself to the database by providing a username and password through the authenticate method on a Login object.

2. The session is created through the lookup method given the URL.

3. The bean is instantiated with the create method of the home interface.

4. Both the LoginServer object and the server object instance handle are saved to a file for the second client to retrieve.

```
// Login to the 8i server
LoginServer lserver = (LoginServer)ic.lookup (serviceURL + "/etc/login");
new Login (lserver).authenticate (username, password, null);

// Activate a Hello in the 8i server
// This creates a first session in the server
HelloHome hello_home = (HelloHome)ic.lookup (serviceURL + objectName);
Hello hello = hello_home.create ();
hello.setMessage ("As created by Client1");
System.out.println ("Client1: " + hello.helloWorld ());

// save Login object into a file, loginFile, for Client2 to read
com.visigenic.vbroker.orb.ORB orb = oracle.aurora.jndi.orb_dep.Orb.init();
String log = orb.object_to_string (lserver);
OutputStream os = new FileOutputStream (loginFile);
os.write (log.getBytes ());
os.close ();

// save object instance handle into a file, helloFile,
// for Client2 to read
Handle obj_hndl = testBean.getHandle ();
OutputStream os = new FileOutputStream (helloFile);
os.write (obj_hndl.getBytes ());
os.close ();
```

The second client would access the `Hello` object instance in the active session by doing the following:

1. Retrieve the object handle and the `Login` object. This example uses implementation-defined methods of `readHandle` and `readLogin` to retrieve these objects from storage.

2. Retrieve the EJB reference through the `getEJBObject` method.

3. Authenticate to the database session with the same `Login` object as the first client through the `authenticate` method. You can recreate the `Login` object from the `LoginServer` object through the `Login` constructor.

```
FileInputStream finstream = new FileInputStream (hellofile);
ObjectInputStream istream = new ObjectInputStream (finstream);
javax.ejb.Handle handle  = (javax.ejb.Handle)istream.readObject ();
Hello hello = (Hello)helloHandle.getEJBObject ();
finstream.close ();

// Authenticate with the login Object
LoginServer lserver = (LoginServer) readLogin(loginFile);

//Set the VisiBroker bind options to specify that the
//login is to not try recursively, which means that if it
//fails on the first try, return with the error immediately.
//See VisiBroker manuals for more information.
lserver._bind_options (new BindOptions (false, false));

Login login = new Login (lserver);
login.authenticate (username, password, null);
```

In-Session Activation If the server object wants to look up and activate a new published object in the same session in which it is running, the server object can execute the following:

```
Context ic = new InitialContext( );
SomeObject myObj = (SomeObject) ic.lookup("/test/Hello");
```

Notice that there are no environment settings for authentication information in the environment or a session URL in the lookup. The authentication already succeeded in order to log into the session. Plus, the object exists on the local machine. So, any other object activation within the session can proceed without specifying authentication information or a target `sess_iiop` URL address.

Note: In-session activation as demonstrated in this section is valid for both IIOP and non-IIOP clients.

In-Session Activation in Pre-8.1.7 Releases In releases previous to Release 8.1.7, in-session activation was performed with the `thisServer/:thisSession` notation in place of the `hostname:port:SID` in the URL. This notation is still valid, but only for IIOP clients.

For example, to look up and activate an object in the same session, do the following:

```
Hashtable env = new Hashtable();
env.put(Context.URL_PKG_PREFIXES, "oracle.aurora.jndi");
Context ic = new InitialContext(env);
SomeObject myObj =
   (SomeObject) ic.lookup("sess_iiop://thisServer/:thisSession/test/Hello");
```

In this case, `myObj` is activated in the same session in which the invoking object is running. Note that there is no need to supply login authentication information, as the client (a server object in this case) is already authenticated to Oracle8*i*.

Realize that objects are not authenticated, instead, clients must be authenticated to a session. However, when a separate session is to be started, then some form of authentication must be done—either login or SSL credential authentication.

Note: You can only use the `thisServer` notation on the server side, that is, from server objects. You cannot use it in a client program.

Lookup of Objects Off of JNDI Context In the Sun Microsystems JNDI, if you bind a name of `"/test/myObject"`, you can retrieve an object from a `Context` when executing the following:

```
Context ctx = ic.lookup("/test");
MyObject myobj = ctx.lookup("myObject");
```

The returned object is activated and ready for you to perform method invocations off of it.

In Oracle8*i*, trying to retrieve an object from a `Context` results in an inactive object being returned. Instead, you must do the following:

1. Retrieve a `SessionCtx`, instead of a `Context`. You can retrieve the
   `SessionCtx` from a `ServiceCtx`, in one of the two following ways:

   - Retrieve the `ServiceCtx` first and the `SessionCtx` from the
     `ServiceCtx`, as follows:

     ```
     ServiceCtx service =
             (ServiceCtx) ic.lookup("sess_iiop://localhost:2481:ORCL");
     //Retrieve the ServiceCtx subcontext
     SessionCtx sess = (SessionCtx) service.lookup("/test");
     ```

   - Retrieve the `ServiceCtx` and `SessionCtx` in the same lookup, as
     follows:

     ```
     SessionCtx sess =
             (SessionCtx) ic.lookup("sess_iiop://localhost:2481:ORCL/test");
     ```

2. Execute the Oracle-specific `SessionCtx.activate` method for each object in
   the session that you want to retrieve. This method activates the object in the
   session and returns the object reference. You cannot just perform a `lookup` of
   the object, as it will return an inactive object. Instead, execute the `activate`
   method, as follows:

   ```
   MyObjectHome myObj_home = (MyObjectHome) sess.activate("myObject");
   ```

3. Finally, create the bean instance and invoke the bean's methods.

   ```
   //create the bean and retrieve the remote interface
   MyObject myObj = myObj_home.create ();

   // Verify that the objects are indeed different
   System.out.println (myObj.printMe ());
   ```

The JServer JNDI implementation provides two implementations of the `Context`
object:

- `ServiceCtx`—identifies the database instance through a `sess_iiop` URL

- `SessionCtx`—represents database session within the database

In performing a lookup, you must lookup both the `ServiceCtx` for identifying the
database and the `SessionCtx` for retrieving the actual JNDI bound object.
Normally, you supply the URLs for both objects within the JNDI URL given to the
`lookup` method. However, you can also retrieve each individually as demonstrated
above.

Setting Session Timeout

A session—with its state—normally exits when the last connection terminates. However, there are situations where you may want a session and its state to idle for a specified amount of time after the last connection terminates, such as the following:

- A middle-tier layer does not want to keep connections open to the session because connections are expensive; but, the middle-tier may want to keep the session open in case of another incoming client request.

- If you experience a network problem that abnormally terminates the connection, the session will stay around for the specified amount of time to allow the connection to be re-established.

- If your application passes a handle to an existing object within the session to another client before its connection terminates, the second client has time to access the session.

The timeout clock starts when the last connection to the session terminates. If another connection to the session starts within the timed window, the timeout clock is reset. If not, the session exits.

You can set the session idle timeout either from the client or from within a server object:

- Set the Session Timeout from the Client

- Set the Session Timeout from a Server Object

Set the Session Timeout from the Client

You can set the idle timeout on the client through the pre-published utility object—`oracle.aurora.AuroraServices.Timeout`. This object is pre-published under "`/etc/timeout`". Use the `setTimeout` method from this object.

1. Retrieve the `Timeout` object through a JNDI lookup of "`/etc/timeout`"

2. Set the timeout with the `setTimeout` method giving the number of seconds for session idle.

```
Timeout timeout = (Timeout)ic.lookup(serviceURL + "/etc/timeout");
System.out.println("Setting a timeout of 20 seconds ");
timeout.setTimeout(20);
```

Set the Session Timeout from a Server Object

A server object can control the session timeout by using the
`oracle.aurora.net.Presentation` object, which contains the
`sessionTimeout()` method. This method takes one parameter; the session
timeout value in seconds. For example:

```
int timeoutValue = 30;
...
// set the timeout to 30 seconds
oracle.aurora.net.Presentation.sessionTimeout(timeoutValue);
...
// set the timeout to a very long time
oracle.aurora.net.Presentation.sessionTimout(Integer.MAX_INT);
```

> **Note:** When you use the `sessionTimeout()` method, you must
> add `$(ORACLE_HOME)/javavm/lib/aurora.zip` to your
> CLASSPATH.

Retrieving JServer Version Number

You can retrieve the version of JServer that is installed in the database through the
pre-published `oracle.aurora.AuroraServices.Version` object, which is
published as "/etc/version" in the JNDI namespace. The `Version` object
contains the `getVersion` method, which returns a string that contains the version,
such as "8.1.7". You can retrieve the `Version` object by providing "/etc/version"
within the JNDI lookup. The following example retrieves the version number:

```
Version version = (Version)ic.lookup(serviceURL + "/etc/version");
System.out.println("The server version is : " + version.getVersion());
```

Activating In-Session EJB Objects From Non-IIOP Presentations

Non-IIOP server requests, such as HTTP or DCOM, can activate an EJB object
within the same session.

- HTTP   An HTTP client interacts with the JServer webserver and executes a
  JSP or servlet, which can activate the EJB object within the same
  session that it is running in.

- DCOM   A DCOM client uses a DCOM bridge to access JServer. While within
  the JServer session, the DCOM bridge session can activate the EJB
  object within the same session that it is running in.

If the non-IIOP server object wants to look up and activate a new published object in the *same session* in which it is running, the server object can execute the following:

```
Context ic = new InitialContext( );
SomeObject myObj = (SomeObject) ic.lookup("/test/Hello");
```

> **Note:** Once you retrieve the IIOP object reference through this method, you cannot pass this object to a remote client or server.

Notice that there are no environment settings for authentication information in the environment or a URL specified in the lookup. The authentication already succeeded in order to log into the session. Plus, the object exists on the local machine. So, any other object activation within the session can proceed without specifying authentication information or a target URL address.

Invoking EJB Objects From Applets

You invoke a server object from an applet in the same manner as from a client. The only differences are the following:

- You must conform to the applet standards.

- You must conform to the Java plug-in standards. The Java plug-ins that are supported are JDK 1.1, Java 2, and Oracle's JInitiator.

- You set the following properties within the initial context environment before the object lookup: ORBdisableLocator, ORBClass, and ORBSingletonClass.

Using Signed JAR Files to Conform to Sandbox Security

The security sandbox constricts your applet from accessing anything on the local disk or from connecting to a remote host other than the host that the applet was downloaded from. If you create a signed JAR file as a trusted party, you can bypass the sandbox security. See http://java.sun.com for more information on applet sandbox security and signed JAR files.

Performing Object Lookup in Applets

You perform the JNDI lookup within the applet the same as within any Oracle Java client, except that you set the following property within the initial context:

```
env.put(ServiceCtx.APPLET_CLASS, this);
```

By default, you do not need to install any JAR files on the client to run the applet. However, if you want to place the Oracle JAR files on the client machine, set the `ClassLoader` property in the `InitialContext` environment, as follows:

```
env.put('ClassLoader', this.getClass().getClassLoader());
```

The following shows the `init` method within an applet that invokes the Bank example. The applet sets up the initial context—including setting the `APPLET_CLASS` property—and performs the JNDI lookup giving the URL.

```
public void init() {
    // This GUI uses a 2 by 2 grid of widgets.
    setLayout(new GridLayout(2, 2, 5, 5));
    // Add the four widgets.
    add(new Label("Account Name"));
    add(_nameField = new TextField());
    add(_checkBalance = new Button("Check Balance"));
    add(_balanceField = new TextField());
    // make the balance text field non-editable.
    _balanceField.setEditable(false);
    try {
      // Initialize the ORB (using the Applet).
      Hashtable env = new Hashtable();
      env.put(Context.URL_PKG_PREFIXES, "oracle.aurora.jndi");
      env.put(Context.SECURITY_PRINCIPAL, "scott");
      env.put(Context.SECURITY_CREDENTIALS, "tiger");
      env.put(Context.SECURITY_AUTHENTICATION, ServiceCtx.NON_SSL_LOGIN);
      env.put(ServiceCtx.APPLET_CLASS, this);

      Context ic = new InitialContext(env);
      _manager = (AccountManager)ic.lookup
                        ("sess_iiop://hostfunk:2222/test/myBank");
    } catch (Exception e) {
      System.out.println(e.getMessage());
      e.printStackTrace();
      throw new RuntimeException();
    }
  }
```

Within the `action` method, the applet invokes methods off of the retrieved object. In this example, the `open` method of the retrieved `AccountManager` object is invoked.

```
  public boolean action(Event ev, Object arg) {
```

```
if(ev.target == _checkBalance) {
  // Request the account manager to open a named account.
  // Get the account name from the name text widget.
  Bank.Account account = _manager.open(_nameField.getText());
  // Set the balance text widget to the account's balance.
  _balanceField.setText(Float.toString(account.balance()));
  return true;
}
return false;
}
```

Modifying HTML for Applets that Access EJB Objects

Oracle8*i* supports only the following Java plug-ins for the HTML page that loads in the applet: JDK 1.1, Java 2, and Oracle JInitiator. Each plug-in contains different syntax for the applet information. However, each HTML page may contain definitions for the following two properties:

- `ORBdisableLocator` set to TRUE—Required for all applets.

- `ORBClass` and `ORBSingletonClass` definitions—Required for the applets that use the Java 2 or JInitiator plug-in.

> **Note:** Because of the sandbox security rules, you cannot set or read any system properties. Therefore, any values that you want to pass on to the ORB runtime, you may set within the applet parameters. This is the method used to set the `ORBdisableLocator`, `ORBClass` and `ORBSingletonClass` properties.

The examples in the following sections show how to create the correct HTML definition for each plug-in type. Each HTML definition defines the applet bank example.

- Example 5–3, "HTML Definition for JDK 1.1 Plug-in"

- Example 5–4, "HTML Definition for Java 2 Plug-in"

- Example 5–5, "HTML Definition for JInitiator Plug-in"

Example 5–3   HTML Definition for JDK 1.1 Plug-in

```
<pre>
<html>
```

```
                <title>Applet talking to 8i</title>
                <h1>applet talking to 8i using java plug in 1.1  </h1>
                <hr>
                The bank example
                Specify the plugin in codebase, the class within the CODE parameter, the JAR
                files in the ARCHIVE parameter, the plugin version in the type parameter, and
                set ORBdisableLocator to true.
                <OBJECT classid="clsid:8AD9C840-044E-11D1-B3E9-00805F499D93"
                        WIDTH = 500 HEIGHT = 50
                        codebase="http://java.sun.com/products/plugin/1.1/
                              jinstall-11-win32.cab#Version=1,1,0,0">
                        <PARAM NAME = CODE VALUE = OracleClientApplet.class >
                        <PARAM NAME = ARCHIVE VALUE = "oracleClient.jar,
                              aurora_client.jar,vbjorb.jar,vbjapp.jar" >
                        <PARAM NAME="type" VALUE="application/x-java-applet;version=1.1">
                        <PARAM NAME="ORBdisableLocator" VALUE="true">
                <COMMENT>
                Set the plugin version in the type, set ORBdisableLocator to true, the applet
                class within the java_CODE tag, the JAR files in the java_ARCHIVE tag, and the
                plug-in source site within the pluginspage tag.
                <EMBED type="application/x-java-applet;version=1.1"
                            ORBdisableLocator="true"
                            java_CODE = OracleClientApplet.class
                            java_ARCHIVE = "oracleClient.jar,
                            aurora_client.jar,vbjorb.jar,vbjapp.jar"
                            WIDTH = 500 HEIGHT = 50
                pluginspage="http://java.sun.com/products/plugin/1.1/plugin-install.html">
                <NOEMBED></COMMENT>
                </NOEMBED></EMBED>
                </OBJECT>

                </center>
                <hr>
                </pre>
```

Example 5–4   HTML Definition for Java 2 Plug-in

```
<pre>
<html>
<title>applet talking to 8i</title>
<h1>applet talking to 8i using Java plug in 1.2 </h1>
<hr>
The bank example
Specify the plugin in codebase, the class within the CODE parameter, the JAR
files in the ARCHIVE parameter, the plugin version in the type parameter, and
```

```
set ORBdisableLocator to true.
<OBJECT classid="clsid:8AD9C840-044E-11D1-B3E9-00805F499D93"
    WIDTH = 500 HEIGHT = 50
    codebase="http://java.sun.com/products/plugin/1.2/jinstall-11-win32.cab#
            Version=1,1,0,0">
    <PARAM NAME = CODE VALUE = OracleClientApplet.class >
    <PARAM NAME = ARCHIVE VALUE = "oracleClient.jar,
    aurora_client.jar,vbjorb.jar,vbjapp.jar" >
    <PARAM NAME="type" VALUE="application/x-java-applet;version=1.1.2">
    <PARAM NAME="ORBdisableLocator" VALUE="true">
    <PARAM NAME="org.omg.CORBA.ORBClass" VALUE="com.visigenic.vbroker.orb.ORB">
    <PARAM NAME="org.omg.CORBA.ORBSingletonClass"
            VALUE="com.visigenic.vbroker.orb.ORB">
<COMMENT>
Set the plugin version in the type, set ORBdisableLocator to true, the ORBClass
and ORBSingletonClass to the correct ORB class, the applet
class within the java_CODE tag, the JAR files in the java_ARCHIVE tag, and the
plug-in source site within the pluginspage tag.
<EMBED type="application/x-java-applet;version=1.1.2"
        ORBdisableLocator="true"
        org.omg.CORBA.ORBClass="com.visigenic.vbroker.orb.ORB"
        org.omg.CORBA.ORBSingletonClass="com.visigenic.vbroker.orb.ORB"
        java_CODE = OracleClientApplet.class
        java_ARCHIVE = "oracleClient.jar,
            aurora_client.jar,vbjorb.jar,vbjapp.jar"
        WIDTH = 500 HEIGHT = 50
pluginspage="http://java.sun.com/products/plugin/1.2/plugin-install.html">
<NOEMBED></COMMENT>
</NOEMBED></EMBED>
</OBJECT>

</center>
<hr>
</pre>
```

Example 5–5   HTML Definition for JInitiator Plug-in

```
<h1> applet talking to 8i using JInitiator 1.1.7.18</h1>
    <COMMENT>
    Set the plugin version in the type, set ORBdisableLocator to true, the
    ORBClass and ORBSingletonClass to the correct ORB class, the applet
    class within the java_CODE tag, the source of the applet in the java_CODEBASE
    and the JAR files in the java_ARCHIVE tag.
    <EMBED   type="application/x-jinit-applet;version=1.1.7.18"
        java_CODE="OracleClientApplet"
```

```
java_CODEBASE="http://hostfunk:8080/applets/bank"
java_ARCHIVE="oracleClient.jar,aurora_client.jar,vbjorb.jar,vbjapp.jar"
WIDTH=400
HEIGHT=100
ORBdisableLocator="true"
org.omg.CORBA.ORBClass="com.visigenic.vbroker.orb.ORB"
org.omg.CORBA.ORBSingletonClass="com.visigenic.vbroker.orb.ORB"
serverHost="orasundb"
serverPort=8080
<NOEMBED>
</COMMENT>
</NOEMBED>
</EMBED>
```

6

IIOP Security

Security involves data integrity, authentication, and authorization.

- For data integrity, Oracle8*i* enables your application to use IIOP over a secure socket layer (SSL).

- For authentication, your application can choose between providing a username/password combination or a certificate.

- For authorization, you can choose the level of trust points that any incoming clients will be required to give.

The following sections explain these subjects in detail:

- Overview
- Data Integrity
- Authentication
- Client-side Authentication
- Server-Side Authentication
- Authorization

Overview

As discussed in the *Oracle8i Java Developer's Guide*, there are several security issues you must think about for your application. The *Oracle8i Java Developer's Guide* divides security into network connection, database contents, and JVM security issues. All these issues are pertain to IIOP. However, IIOP has specific implementation issues for both the networking and the JVM security, as listed below:

- JVM security includes both utilizing Java2 permissions and granting execution rights. For IIOP, you can grant execution privileges in one of two ways:

  * CORBA—The owner grants execution rights to CORBA objects with an option on the loadjava tool. See the loadjava discussion in the *Oracle8i Java Developer's Guide* for information on granting execution rights when loading the CORBA classes.

  * EJB—The owner grants execution rights to EJB objects and, potentially, methods within the deployment descriptor. See the section on "Access Control" in the *Oracle8i Enterprise JavaBeans Developer's Guide* for more information on defining execution rights within your deployment descriptor.

- Network connection security includes the following issues:

  * Data Integrity—To prevent a sniffer from reading the transmission directly off the wire, all transmissions are encoded. Oracle supports Secure Socket Layer (SSL) for encryption.

  * Authentication—To prevent an invalid user from impersonating a valid user, the client or server provides authentication information. This information can take the form of a username/password combination or certificates.

  * Authorization—To prove that the user is allowed access to the object, two types of authorization are performed:

    - Session authorization—The session is authorized to the user. In this case, the client is authorized to access the server through validating either the username or certificate provided.

    - User authorization—The client or server can perform authorization on a provided certificate. This type of authorization can be performed only when the client or server authenticates itself by providing a certificate.

This section describes fully the network connection security issues that IIOP applications must consider.

Data Integrity

Do you want your transport line to be encrypted? Do you want data integrity and confidentiality? If you believe that the physical connection can be tampered with, you can consider encrypting all transmissions by using the secure socket layer (SSL) encryption technology. However, because adding encryption to your transmission affects your connection performance, if you do not have any transport security issues, you should transmit unencrypted.

Figure 6–1   Data Integrity Decision Tree
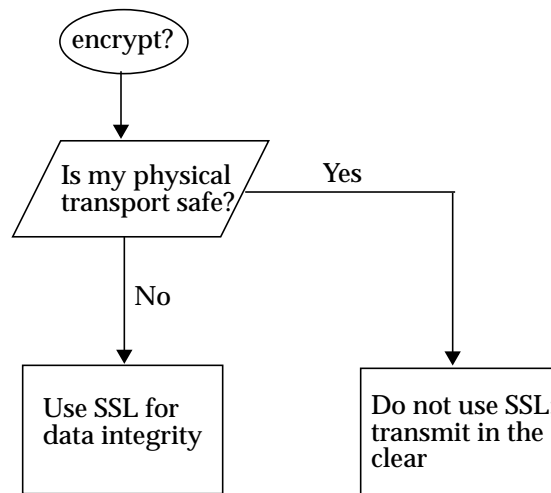
Using the Secure Socket Layer

JServer's CORBA and EJB implementations rely on the Secure Socket Layer (SSL) for data integrity and authentication. SSL is a secure networking protocol, originally defined by Netscape Communications, Inc. Oracle8*i* JServer supports SSL over the IIOP protocol used for the ORB.

When a connection is requested between a client and the server, the SSL layer within both parties negotiate during the connection handshake to verify if the connection is allowed. The connection is verified at several levels:

1. The SSL version on both the client and the server must agree for the transport to be guaranteed for data integrity.

2. If server-side authentication with certificates is requested, the certificates provided by the server are verified by the client at the SSL layer. This means that the server is guaranteed to be itself. That is, it is not a third party pretending to be the server.

3. If client-side authentication with certificates is requested, the certificates provided by the client are verified at the SSL layer. The server receives the client's certificates for authentication or authorization of the client.

> **Note:** Normally, client-side authentication means only that the server verifies that the client is not an impersonator and is trusted. However, when you specify SSL_CLIENT_AUTH in JServer, you are requesting both server-side and client-side authentication.

The SSL layer performs authentication between the peers. After the handshake, you can be assured that the peers are authenticated to be who they say they are. You can perform additional tests on their certificate chain to authorize that this user can access your application. See "Authorization" on page 6-26 on how to go beyond authentication.

> **Note:** If you decide to use SSL, your client must import the following JAR files:
>
> - If your client uses JDK 1.1, import `jssl-1_1.jar` and `javax-ssl-1_1.jar`.
>
> - If your client uses Java 2, import `jssl-1_2.jar` and `javax-ssl-1_2.jar`.

SSL Version Negotiation

SSL makes sure that both the client and server side agree on an SSL protocol version number. The values that you can specify are as follows:

- Undetermined: `SSL_UNDETERMINED`. This is the default setting.

- 3.0 with 2.0 Hello: This setting is not supported.

- 3.0: `SSL_30`.

- 2.0: This setting is not supported.

On the database, the default is "Undetermined". The database does not support 2.0 or 3.0 with 2.0 Hello. Thus, you can use only the Undetermined or 3.0 setting for the client.

- The server's version is set within the database SQLNET.ORA file, using the SSL_VERSION parameter. For example, SSL_VERSION = 3.0.

- For the client, you set the SSL client version number in the client's JNDI environment, as follows:

```
environment.put("CLIENT_SSL_VERSION", ServiceCtx.SSL_30);
```

Table 6–1 shows which handshakes resolve to depending on SSL version settings on both the client and the server. The star sign "✷" indicates cases where the handshake fails.

Table 6–1   SSL Version Numbers

Client Setting	Server Setting			
	Undetermined	3.0 W/2.0 Hello (not supported)	3.0	2.0 (not supported)
Undetermined	3.0	✷	✷	✷
3.0 W/2.0 Hello (not supported)	✷	✷	✷	✷
3.0	3.0	✷	3.0	✷
2.0 (not supported)	✷	✷	✷	✷

Authentication

Authentication is the process where one party supplies to a requesting party information that identifies itself. This information guarantees that the originator is not an imposter. In the client/server distributed environment, authentication can be required from the client or the server:

- Server-side authentication—The server sends identifying information to authenticate itself. The client uses this information to verify that the server is itself and not an imposter. If you request SSL, the server will always send certificate-based authentication information.

- Client-side authentication—For the same reasons, the client sends identifying information to the server, which includes either a username/password combination or certificates. Since the client is logging on to a database, the client must always authenticate itself to the database.

- Callout authentication—The server initiates a call to another object. This causes the server to act as a client; as such, the server cannot use the database authentication information, but must provide information and authenticate itself as an independent party.

- Callback authentication—The server is given either a CORBA IOR or an EJB handle for calling back to an object that exists on the client. In this scenario, the server is acting as a client; as such, the server cannot use the database authentication information, but must provide information and authenticate itself as an independent party.

Client-side Authentication

The Oracle data server is a secure server; a client application cannot access data stored in the database without first being authenticated by the database server. Oracle8*i* CORBA server objects and Enterprise JavaBeans execute in the database server. For a client to activate such an object and invoke methods on it, the client must authenticate itself to the server. The client authenticates itself when a CORBA or EJB object starts a new session. The following are examples of how each IIOP client must authenticate itself to the database:

- When a client initially starts a new session, it must authenticate itself to the database.

- When a client passes an object reference (a CORBA IOR or an EJB bean handle) to a second client, the second client connects to the session specified in the object reference. The second client authenticates itself to the server.

The client authenticates itself by providing one of the following types:

Authentication type	Definition
Certificates	You can provide the user certificate, the Certificate Authority certificate (or a chain that contains both, including other identifying certificates), and a private key.
Username and password combination	You can provide the username and password through either credentials or the login protocol. In addition, you can pass a database role to the server, along with the username and password.

The type of client-side authentication can be determined by the server's configuration. If, within the SQLNET.ORA file, the SSL_CLIENT_AUTHENTICATION parameter is TRUE, the client must provide certificate-based authentication. If SSL_CLIENT_AUTHENTICATION is FALSE, the client authenticates itself with a username/password combination. If SSL_CLIENT_AUTHENTICATION is TRUE and the client provides a username/password, the connection handshake will fail.

The following table gives a brief overview of the options that the client has for authentication.

- The columns represent the options available if you have chosen to use SSL for data integrity.

- The rows demonstrate the three authentication vehicles: login protocol, credentials, and certificates.

- The table entries detail the different methods you must employ when implementing the client-side authentication type.

Authentication vehicle	NON-SSL transport	SSL transport
Providing username and password using the login protocol	■ Implicit method: Set JNDI property to NON_SSL_LOGIN; provide username and password in JNDI properties. ■ Explicit method: Create a Login object with username and password.	■ Implicit method: Set JNDI property to SSL_LOGIN; provide username and password in JNDI properties. ■ Explicit method: Create a Login object with username and password.
Providing username and password using credentials	Not supported because the password would transmit in the clear.	Set JNDI property to SSL_CREDENTIAL; username and password are implicitly sent to the server in the handshake.
Providing certificates	Not supported because certificates require an SSL transport.	Set JNDI property to SSL_CLIENT_AUTH; provide client certificate, CA certificate, and private key in JNDI properties. Pure CORBA objects use AuroraCertificateManager class to specify certificates, CA certificate, and private key.

As the table demonstrates, most of the authentication options include setting an appropriate value in JNDI properties.

Using JNDI for Authentication

To set up client-side authentication using JNDI, you set the `javax.naming.Context.SECURITY_AUTHENTICATION` attribute to one of the following values:

- `ServiceCtx.NON_SSL_LOGIN`—A plain IIOP connection is used. Because SSL is not used, all data flowing over the line is not encrypted. Thus, to protect the password, the client uses the login protocol to authenticate itself. In addition, the server does not provide SSL certificates to the client to identify itself.

- `ServiceCtx.SSL_LOGIN`—An SSL-enabled IIOP connection is used. All data flowing over the transport is encrypted. If you do not want to provide a certificate for the client authentication, use the login protocol to provide the username and password.

  Because this is an SSL connection, the server sends its certificate identity to the client. The client is responsible for verifying the server's certificate, if interested, for server authentication. Optionally, the client can set up trust points for the server's certificate to be verified against.

- `ServiceCtx.SSL_CREDENTIAL`—An SSL-enabled IIOP connection is used. All data flowing over the transport is encrypted. The client provides the username and password without using the login protocol for client authentication to the server. The username and password are automatically passed to the server in a security context, on the first message.

  > **Note:** The client's password is not encrypted, as it is with SSL. It might be slightly more efficient than SSL_LOGIN, where encrypting a password over an SSL connection is redundant.

  The server provides its certificate identity to the client. The client is responsible for verifying the server's certificate, if interested, for server authentication.

- `ServiceCtx.SSL_CLIENT_AUTH`—An SSL-enabled IIOP connection is used. All data flowing over the transport is encrypted. The client provides appropriate certificates for client-side authentication to the server. In addition, the server provides its certificate identity to the client. If interested, the client is responsible for authorizing the server's certificate.

■ Nothing is specified. The client must activate the login protocol explicitly before activating and invoking methods on a server-side object. Use this method when a client must connect to an existing session and invoke methods on an existing object. See the `demo/examples/corba/session/sharedsession` example for more information. The username and password in the initial context environment are automatically passed as parameters to the login object's `authenticate()` method.

Within each of these options, you choose to do one or more of the following:

Client authentication	■	authenticate itself to the server using login protocol
	■	authenticate itself to the server using straight username and password
	■	authenticate itself to the server using SSL certificates
Server authentication	■	authenticate itself to the client using SSL certificates

For information on how to implement each of these methods for client or server authentication, see the following sections:

■ Providing Username and Password for Client-Side Authentication

■ Using Certificates for Client Authentication

■ Server-Side Authentication

Providing Username and Password for Client-Side Authentication

The client authenticates itself to the database server either through a username/password or by supplying appropriate certificates. The username/password can be supplied either through Oracle's login protocol or credentials over the SSL transport connection.

■ Provide a username and password by setting JNDI properties, which implicitly sets these values in a login protocol. Set SECURITY_AUTHENTICATION to `ServiceCtx.SSL_LOGIN` or `ServiceCtx.NON_SSL_LOGIN`.

■ Provide a username and password through credentials. The username and password are provided implicitly and are shipped to the server over the encrypted SSL transport. Set SECURITY_AUTHENTICATION to `serviceCtx.SSL_CREDENTIAL`.

■ Provide a username and password in an explicitly activated login protocol.

> **Note:** The Login class serves as an implementation of the client side of the login handshaking protocol and as a proxy object for calling the server login object. This component is packaged in the `aurora_client.jar` file. All Oracle8*i* ORB applications must import this library.

Username Sent by Setting JNDI Properties for the Login Protocol

A client can use the login protocol to authenticate itself to the Oracle8*i* data server. You can use the login protocol either with or without SSL encryption, because a secure handshaking encryption protocol is built in to the login protocol.

If your application requires an SSL connection for client-server data security, specify the **SSL_LOGIN** service context value for the SECURITY_AUTHENTICATION property that is passed when the JNDI initial context is obtained. The following example defines the connection to be SSL-enabled for the login protocol. Notice that the username and password are set.

```
Hashtable env = new Hashtable();
env.put(javax.naming.Context.URL_PKG_PREFIXES, "oracle.aurora.jndi");
env.put(javax.naming.Context.SECURITY_PRINCIPAL, username);
env.put(javax.naming.Context.SECURITY_CREDENTIALS, password);
env.put(javax.naming.Context.SECURITY_AUTHENTICATION, ServiceCtx.SSL_LOGIN);
Context ic = new InitialContext(env);
...
```

If your application does not use an SSL connection, specify **NON_SSL_LOGIN** within the SECURITY_AUTHENTICATION parameter as shown below:

> **Note:** The login handshaking is secured by encryption, but the remainder of the client-server interaction is not secure.

```
env.put(javax.naming.Context.SECURITY_AUTHENTICATION, ServiceCtx.NON_SSL_LOGIN);
```

When you specify values for all four JNDI Context variables—URL_PKG_PREFIXES, SECURITY_PRINCIPAL, SECURITY_CREDENTIALS, and SECURITY_AUTHENTICATION—the first invocation of the `Context.lookup()` method performs a login automatically.

If the client setting up the connection is not using JNDI look up because it already has an IOR, the user that gave them the IOR for the object should have also passed in a Login object that exists in the same session as the active object. You must

provide the username and password in the `authenticate` method of the Login object, before invoking the methods on the active object.

Logging In and Out of the JServer Session If the session owner wishes to exit the session, the owner can use the `logout` method of the `LogoutServer` object, which is pre-published as "/etc/logout". You use the `LogoutServer` object to exit the session. Only the session owner is allowed to logout. Any other owner receives a NO_PERMISSION exception.

The `LogoutServer` object is analogous to the `LoginServer` object, which is pre-published as "/etc/login". You can use the `LoginServer` object to retrieve the `Login` object, which is used to authenticate to the server. This is an alternative method to using the `Login` object within the JNDI lookup.

The following example shows how a client can authenticate using the `LoginServer` object and can exit the session through the `LogoutServer` object.

```
import oracle.aurora.AuroraServices.LoginServer;
import oracle.aurora.AuroraServices.LogoutServer;
...
// To log in using the LoginServer object
LoginServer loginServer = (LoginServer)ic.lookup(serviceURL + "/etc/login");
Login login = new Login(loginServer);
System.out.println("Logging in ..");
login.authenticate(user, password, null);
...
//To logout using the LogoutServer
LogoutServer logout = (LogoutServer)ic.lookup(serviceURL + "/etc/logout");
logout.logout();
```

Username Sent Implicitly by using Credentials

Using the `ServiceCtx.SSL_CREDENTIAL` authentication type means that the username, password, and, potentially, a role are passed to the server on the first request. Because this information is passed over an SSL connection, the password is encrypted by the transfer protocol, and there is no need for the handshaking that the Login protocol uses. This is slightly more efficient and is recommended for SSL connections.

Username Sent by Explicitly Activating a Login Object

You can explicitly create and populate a Login object for the database login. Typically, you would do this if you wanted to create and use more than a single

session from a client. The following example shows a client creating and logging on to two different sessions. To do this, you must perform the following steps:

1. Create the initial context.

2. Perform a look up on a URL for the destination database.

3. On this database service context, create two subcontexts—one for each session.

4. Login to each session using a Login object, providing a username and password.

> **Note:** The username and password for both sessions are identical because the destination database is the same database. If the client were to connect to two different databases, the username and password may need to be different for logging on.

```
// Prepare a simplified Initial Context as we are going to do
// everything by hand
Hashtable env = new Hashtable ();
env.put (Context.URL_PKG_PREFIXES, "oracle.aurora.jndi");
Context ic = new InitialContext (env);

// Get a SessionCtx that represents a database instance
ServiceCtx service = (ServiceCtx)ic.lookup (serviceURL);

// Create and authenticate a first session in the instance.
SessionCtx session1 = (SessionCtx)service.createSubcontext (":session1");
LoginServer login_server1 = (LoginServer)session1.activate ("etc/login");
Login login1 = new Login (login_server1);
login1.authenticate (user, password, null);

// Create and authenticate a second session in the instance.
SessionCtx session2 = (SessionCtx)service.createSubcontext (":session2");
LoginServer login_server2 = (LoginServer)session2.activate ("etc/login");
Login login2 = new Login (login_server2);
login2.authenticate (user, password, null);

// Activate one Hello object in each session
Hello hello1 = (Hello)session1.activate (objectName);
Hello hello2 = (Hello)session2.activate (objectName);
```

Using Certificates for Client Authentication

Client authentication through certificates requires the client sending a certificate or certificate chain to the server; the server verifies that the client is truly who the client said it was and that it is trusted.

> **Note:** All certificates, trustpoints, and the private key should be in base-64 encoded format.

You set up the client for certificate authentication through one of the following methods:

- Specifying Certificates in a File
- Specifying Certificates in Individual JNDI Properties
- Specifying Certificates using AuroraCertificateManager

Specifying Certificates in a File

You can set up a file that contains the user certificate, the issuer certificate, the entire certificate chain, an encrypted private key, and the trustpoints. Once created, you can specify that the client use the file during connection handshake for client authentication.

1. Create the client certificate file—This file can be created through an export feature in the Wallet Manager. The Oracle Wallet Manager has an option that creates this file. You must populate a wallet using the Wallet Manager before requesting that the file is created.

   After you create a valid wallet, bring up the Wallet Manager and perform the following:

   - From the menu bar pull down, click on Operations > Export Wallet.
   - Within the filename field, enter the name that you want the certificate file known as.

   This creates a base-64 encoded file that contains all certificates, keys, and trustpoints that you added within your wallet. For information on how to create the wallet, see the *Oracle Advanced Security Administrator's Guide.*

2. Specify the client certificates file for the connection—Within the client code, set the `SECURITY_AUTHENTICATION` property to `ServiceCtx.SSL_CLIENT_AUTH`. Provide the appropriate certificates and

trustpoints for the server to authenticate against. Specify the filename and decrypting key in the JNDI properties, as follows:

Values	Set in JNDI Property
Name of the certificate file	`SECURITY_PRINCIPAL`
Key for decrypting the private key	`SECURITY_CREDENTIAL`

The following code is an example of how to set up the JNDI properties to define the client certificate file:

```
Hashtable env = new Hashtable();
env.put(javax.naming.Context.URL_PKG_PREFIXES, "oracle.aurora.jndi");
env.put(javax.naming.Context.SECURITY_PRINCIPAL, <filename>);
env.put(javax.naming.Context.SECURITY_CREDENTIAL, <decrypting_key>);
env.put(javax.naming.Context.SECURITY_AUTHENTICATION,
                                    ServiceCtx.SSL_CLIENT_AUTH);
Context ic = new InitialContext(env);
...
```

For example, if your decrypting key is `welcome12` and the certificate file is `credsFile`, the following two lines would specify these values within the JNDI context:

```
env.put(Context.SECURITY_CREDENTIALS, "welcome12");
env.put(Context.SECURITY_PRINCIPAL, "credsFile");
```

Specifying Certificates in Individual JNDI Properties

You can provide each certificate, private key, and trust point programmatically, by setting each item individually within JNDI properties. Once you populate the JNDI properties with the user certificate, issuer (Certificate Authority) certificate, encrypted private key, and trust points, they are used during connection handshake for authentication. To identify client-side authentication, set the `SECURITY_AUTHENTICATION` property to `serviceCtx.SSL_CLIENT_AUTH`.

> **Note:** Only a single issuer certificate can be set through JNDI properties.

You can choose any method for setting up your certificates within the JNDI properties. All authorization information values must be set up before initializing the context.

The following example declares the certificates as a static variable. However, this is just one of many options. Your certificate must be base-64 encoded. For example, in the following code, the **testCert_base64** is a base-64 encoded client certificate declared as a static variable. The other variables for CA certificate, private key, and so on, are not shown, but they are defined similarly.

> **Note:** When setting individual certificates as static variables, any certificates for Oracle8*i* parties do not have any separators. However, if you are setting a certificate for a Visigenic ORB (as the client callback object does in a callback scenario), the certificate must be delineated by "BEGIN CERTIFICATE" and "END CERTIFICATE" identifying lines. See the Visigenic documentation for the format of these strings.

```
final private static String testCert_base64 =
  "MIICejCCAeOgAwIBAgICAmowDQYJKoZIhvcNAQEEBQAwazELMAkGA1UEBhMCVVMx" +
  "DzANBgNVBAoTBk9yYWNsZTEoMCYGA1UECxMfRW50ZXJwcmlzZSBBcHBsaWNhdGlv" +
  "biBTZXJ2aWNlczEhMB8GA1UEAxMYRUFTTUEgQ2VydGlmaWNhdGUgU2VydmVyMB4X" +
  "DTk5MDgxNzE2MjIxMloXDTAwMDIxMzE2MjIxMlowgYUxCzAJBgNVBAYTAlVTMRsw" +
  "GQYDVQQKExJPcmFjbGUgQ29ycG9yYXRpb24xPDA6BgNVBAsUMyoqIFNlY3VyaXR5" +
  "IFRFU1RJTkcgQU5EIEVWWQUxVQVRJT04gT05MWSB2ZXJzaW9uMiAqKjEbMBkGA1UE" +
  "AxQSdGVzdEB1cy5vcmFjbGUuY29tMHwwDQYJKoZIhvcNAQEBBQADawAwaAJhANG1" +
  "Kk2K7uOOtI/UBYrmTe89LVRrG83Eb0/wY3xWGelkBeEUTwW57a26u2M9LZAfmT91" +
  "e8Afksqc4qQW23Sjxyo4ObQK3Kth6y1NJgovBgfMu1YGtDHaSn2VEg8p58g+nwID" +
  "AQABozYwNDARBglghkgBhvhCAQEEBAMCAMAwHwYDVR0jBBgwFoAUDCHwEuJfIFXD" +
  "a7tuYNO8bOw1EYwwDQYJKoZIhvcNAQEBQADgYEARC5rWKge5trqqgZ18onldinCg" +
  "Fof6D/qFT9b6Cex5JK3a2dEekg/P/KqDINyifIZL0DV7z/XCK6PQDLwYcVqSSK/m" +
  "487qjdH+zM5X+1DaJ+ROhqOOX54UpiAhAleRMdLT5KuXV6AtAx6Q2mc8k9bzFzwq" +
  "eR3uI+i5Tn0dKgxhCZU=\n";

Hashtable env = new Hashtable();
env.put(Context.URL_PKG_PREFIXES, "oracle.aurora.jndi");
env.put(Context.SECURITY_AUTHENTICATION, ServiceCtx.SSL_CLIENT_AUTH);
//decrypting key
env.put(Context.SECURITY_CREDENTIALS, "welcome12");

// you may also set the certificates individually, as shown bellow.
//User certificate
env.put(ServiceCtx.SECURITY_USER_CERT, testCert_base64);
//Certificate Authority's certificate
env.put(ServiceCtx.SECURITY_CA_CERT, caCert_base64);
//Private key
```

```
env.put(ServiceCtx.SECURITY_ENCRYPTED_PKEY, encryptedPrivateKey_base64);
// setup the trust point
env.put(ServiceCtx.SECURITY_TRUSTED_CERT, trustedCert);

Context ic = new InitialContext(env);
```

Specifying Certificates using AuroraCertificateManager

CORBA clients that do not use JNDI can use `AuroraCertificateManager` for setting the user and issuer certificates, the encrypted private key, and the trust points.

`AuroraCertificateManager` maintains certificates for your application. For the certificates to be passed on the SSL handshake for the connection, you must set the certificates before an SSL connection is made. Setting up a certificate in this manner is only required if the following is true:

- The client sets its certificates through `AuroraCertificateManager` if client-side authentication is required, and the client does not want to use JNDI properties for setting certificates.

- The server sets its certificates through `AuroraCertficateManager` if it is executing a callout or a callback. The typical server-side authentication for a simple client/server exchange is taken care of by the database wallet. However, if this server intends to act as a client by executing a callout or callback, it needs to set certificates identifying itself; it cannot use the database certificate that is contained in the wallet.

AuroraCertificateManager Class

The methods offered by this object allow you to:

- Set the SSL protocol version. The default is Undetermined.

- Set the private key and certificate chain.

- Require that client applications authenticate themselves by presenting their certificate chain. This method is used only by servers.

Invoking the `ORB.resolve_initial_references` method with the parameter `SSLCertificateManager` will return an object that can be narrowed to a `AuroraCertificateManager`. Example 6–1 shows a code example of the following methods.

addTrustedCertificate

This method adds the specified certificate as a trusted certificate. The certificate must be in DER encoded format. The client adds trustpoints through this method for server-side authentication.

When your client wants to authenticate a server, the server sends its certificate chain to the client. You might not want to check every certificate in the chain. For example, you have a chain composed of the following certificates: Certificate Authority, enterprise, business unit, a company site, and a user. If you trust the company site, you would check the user's certificate, but you might stop checking the chain when you get to the company site's certificate, because you accept the certificates above the company sites in the hierarchical chain.

Syntax

```
void addTrustedCertificate(byte[] derCert);
```

Parameter	Description
derCert	The DER encoded byte array containing the certificate.

requestClientCertificate

This method is invoked by servers that wish to require certificates from client applications. This method is not intended for use by client applications.

> **Note:** The `requestClientCertificate` method is not currently required, because the SQLNET.ORA and LISTENER.ORA configuration parameter SSL_CLIENT_AUTHENTICATION performs its function.

Syntax

```
void requestClientCertificate(boolean need);
```

Parameter	Description
need	If true, the client must send a certificate for authentication. If false, no certificate is requested from the client.

setCertificateChain

This method sets the certificate chain for your client application or server object and can be invoked by clients or by servers. The certificate chain always starts with the Certificate Authority certificate. Each subsequent certificate is for the issuer of the preceding certificate. The last certificate in the chain is the certificate for the user or process.

Syntax

```
void setCertificateChain(byte[][] derCertChain)
```

Parameter	Description
derCertChain	A byte array containing an array of certificates.

setEncryptedPrivateKey

This method sets the private key for your client application or server object. You must specify the key in PKCS5 or PKCS8 format.

Syntax

```
void setEncryptedPrivateKey(byte[] key, String password);
```

Parameter	Description
key	The byte array that contains the encrypted private key.
password	A string containing a password for decrypting the private key.

setProtocolVersion

This method sets the SSL protocol version that can be used for the connection. A 2.0 Client trying to establish an SSL connection with a 3.0 Server will fail and the converse. We recommend using Version_Undetermined, because it lets the peers establish an SSL connection whether they are using the same protocol version or not. `SSL_Version_Undetermined` is the default value.

Syntax

```
void setProtocolVersion(int protocolVersion);
```

Parameter	Description
protocolVersion	The protocol version being specified. The value you supply is defined in `oracle.security.SSL.OracleSSLProtocolVersion`. This class defines the following values:

- `SSL_Version_Undetermined`: Version is undetermined. This is used to connect to SSL 2.0 and SSL 3.0 peers. This is the default version.
- `SSL_Version_3_0_With_2_0_Hello`: Not supported.
- `SSL_Version_3_0`: Used to connect to 3.0 peers only.
- `SSL_Version_2_0`: Not supported.

Example 6–1   Setting SSL Security Information Using AuroraCertificateManager

This example does the following:

1. Retrieve the **AuroraCertificateManager**.

2. Initialize this client's SSL information:

    a. Set the certificate chain through **setCertificateChain**.

    b. Set the trustpoint through **addTrustedCertificate**.

    c. Set the private key through **setEncryptedPrivateKey**.

```
// Get the certificate manager
AuroraCertificateManager cm = AuroraCertificateManagerHelper.narrow(
                orb.resolve_initial_references("AuroraSSLCertificateManager"));

BASE64Decoder decoder = new BASE64Decoder();
byte[] userCert = decoder.decodeBuffer(testCert_base64);
byte[] caCert = decoder.decodeBuffer(caCert_base64);

// Set my certificate chain, ordered from CA to user.
byte[][] certificates = {
        caCert, userCert
};
cm.setCertificateChain(certificates);
cm.addTrustedCertificate(caCert);

// Set my private key.
byte[] encryptedPrivateKey =
decoder.decodeBuffer(encryptedPrivateKey_base64);

cm.setEncryptedPrivateKey(encryptedPrivateKey, "welcome12");
```

Server-Side Authentication

The server can require a different type of authentication depending on its role. If you are utilizing the database as a server in a typical client/server environment, you use certificates that are set within a wallet for the database for server-side authentication. However, if you are using the server to callout to another object or callback to an object on the client, the server is now acting as a client and so requires its own identifying certificates. That is, in a callout or callback scenario, the server cannot use the wallet generated for database server-side authentication.

Server activity	Authentication method
Typical client/server	Use database wallet generated by Oracle Wallet Manager
Callout to another object	Set identifying certificates using either JNDI properties or `AuroraCurrentManager` class.
Callback to client object	Set identifying certificates using `AuroraCurrentManager` class.

The following sections describe this in more detail:

- Typical Client/Server
- Callouts using Security
- Callbacks using Security

Typical Client/Server

Server-side authentication takes place when the server provides certificates for authentication to the client. When requested, the server will authenticate itself to the client, also known as server-side authentication, by providing certificates to the client. The SSL layer authenticates both peers during the connection handshake. The client requests server-side authentication by setting any of the SSL_* values in the JNDI property. See "Using JNDI for Authentication" on page 6-8 for more information on these JNDI values.

For server-side authentication, you must set up a database wallet with the appropriate certificates, using the Wallet Manager. See the *Oracle Advanced Security Administrator's Guide* for information on how to create a wallet.

Callouts using Security

A callout is when a Java object loaded within the database invokes a method within another Java object. If the original call from the client required a certain level of security—certificate-based or username/password security—the server object is also required to provide the same level of security information for itself before invoking the method on the second server object.

Figure 6–2   Server callout requires security
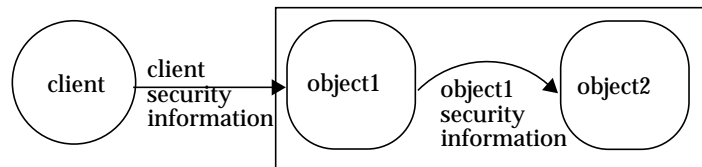
- Username/password: If the client sent a username/password combination for authenticating to the database, the server object is also required to send its own username/password combination to the second object. The server object cannot forward along the client's username/password combination, but must supply its own. You can set the username/password combination in the same manner as the client. See "Providing Username and Password for Client-Side Authentication" on page 6-9 for more information.

- Certificate-based: Similarly, if the client sent certificates for authentication, the server object must do the same. Additionally, the server must create and send its own certificates, it cannot forward on the client's certificates for authentication. You set up your server object certificates using either the appropriate JNDI properties or the `AuroraCertificateManager` as discussed in "Using Certificates for Client Authentication" on page 6-13.

Callbacks using Security

A callback is when the client passes the server object an object reference to an object that exists on the client. As shown in Figure 6–3, the server object receives the object

reference and invokes methods. This effectively calls out of the server and back to an object located in the client. See "Debugging Techniques" on page 2-28 for more information on callbacks.

Figure 6–3   Server callout requires security
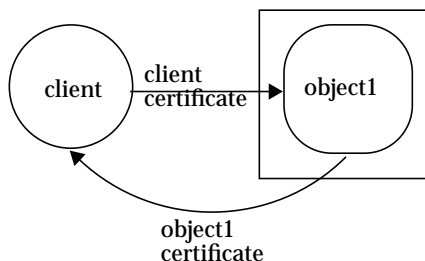
The type of security you can use for callbacks is certificate-based security over SSL. When you add SSL security to callbacks, you can have one of two situations:

1. Server-side authentication only
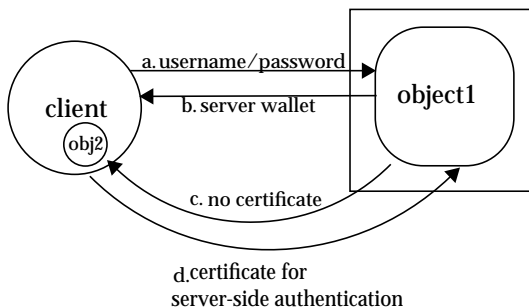
a. The client is not required to authenticate itself with a certificate. However, it must still authenticate itself to the database using a username/password combination.

b. The server, since server-side authentication is always required with SSL, authenticates itself to the client by providing certificates contained in the database wallet.

c. When the server calls back to the client, it acts as a client; thus, it is not required to provide certificates for authentication.

d. The called object, although contained in the client, is the server object in the callback scenario. Thus, since server-side authentication rules hold, the callback object must provide certificates to authenticate itself.

Example 6–2   Callback code with server-side authentication only

The following code shows the client code that performs (a) and (d) steps above. The first half of the client code sets up a username and password for authenticating itself to the database. It retrieves the server object. However, before it invokes the server's method, the last half of the code sets up the client callback object by setting certificates, initializing the BOA, and instantiating the callback object. Finally, the server method is invoked.

```
public static void main (String[] args) throws Exception {
String serviceURL = args [0];
String objectName = args [1];
String user = args [2];
String password = args [3];

//set up username/password for authentication to database. Set up
//security to be SSL_LOGIN - login authentication for client and server-side
//authentication.
Hashtable env = new Hashtable ();
env.put (Context.URL_PKG_PREFIXES, "oracle.aurora.jndi");
env.put (Context.SECURITY_PRINCIPAL, user);
env.put (Context.SECURITY_CREDENTIALS, password);
env.put (Context.SECURITY_AUTHENTICATION, ServiceCtx.SSL_LOGIN);
Context ic = new InitialContext (env);

// Get the server object before preparing the client object.
// You have to do it in this order to get the ORB initialized correctly
Server server = (Server)ic.lookup (serviceURL + objectName);

// Create the client object and export it to the ORB in the client
// First, set up the ORB properties for the callback object
java.util.Properties props = new java.util.Properties();
props.put("ORBservices", "oracle.aurora.ssl");

// Initialize the ORB.
com.visigenic.vbroker.orb.ORB orb = (com.visigenic.vbroker.orb.ORB)
        oracle.aurora.jndi.orb_dep.Orb.init(args, props);

// Get the certificate manager
AuroraCertificateManager certificateManager =
        AuroraCertificateManagerHelper.narrow(
```

```
                    orb.resolve_initial_references("AuroraSSLCertificateManager"));

        // Set up client callback certificate chain, ordered from user to CA.
        byte[] testCert = new byte[testCert_base64.length()];
        testCert_base64.getBytes(0, testCert_base64.length(), testCert, 0);

        byte[] caCert = new byte[caCert_base64.length()];
        caCert_base64.getBytes(0, caCert_base64.length(), caCert, 0);

        // Set my certificate chain, ordered from user to CA.
        byte[][] certificates = {
             testCert, caCert
           };
        certificateManager.setCertificateChain(certificates);

        // Set client callback object's private key.
        byte[] encryptedPrivateKey = new byte[encryptedPrivateKey_base64.length()];
        encryptedPrivateKey_base64.getBytes(0, encryptedPrivateKey_base64.length(),
                                          encryptedPrivateKey, 0);

        certificateManager.setEncryptedPrivateKey(encryptedPrivateKey,"welcome12");

        // Initialize the BOA with SSL
        org.omg.CORBA.BOA boa = orb.BOA_init("AuroraSSLTSession", null);

        //Instantiate the client callback object
        ClientImpl client = new ClientImpl ();

        //register callback object with BOA
        boa.obj_is_ready (client);

        // Invoke the server method, passing the client to call us back
        System.out.println (server.hello (client));
         }
        }
```

2. Client-side and Server-side authentication
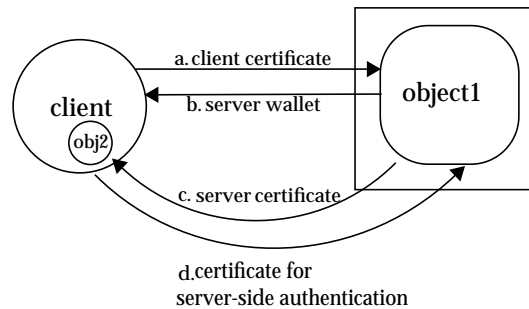
- **a.** The client is required to authenticate itself with a certificate.

- **b.** The server, since server-side authentication is always required with SSL, authenticates itself to the client by providing certificates contained in the database wallet.

- **c.** When the server calls back to the client, it acts as a client; thus, it is required to provide its own certificates for authentication.

- **d.** The called object, although contained in the client, is the server object in the callback scenario. Thus, since server-side authentication rules hold, the callback object must provide certificates to authenticate itself.

The code for the client shown in Example 6–2 is the same for this scenario, except that instead of providing a username and password, the client provides certificates.

Since client-side authentication is required and because the server is acting as a client, the server code sets up identifying certificates for itself before invoking the callback object. The server must create and send its own certificates, it cannot forward on the client's certificates for authentication. You set up your server object certificates using either the appropriate JNDI properties or the `AuroraCertificateManager` as discussed in "Using Certificates for Client Authentication" on page 6-13.

Example 6–3   Server code in callback with client-side authentication

The following server code does the following:

1. Retrieve the Oracle8*i* ORB reference by invoking the init method.

2. Retrieve the `AuroraCertificateManager`

3. Set certificates and key through `AuroraCertificateManager` methods.

4. Invoke the client callback method, `hello`.

```java
public String hello (Client client) {
BASE64Decoder decoder = new BASE64Decoder();
com.visigenic.vbroker.orb.ORB orb = (com.visigenic.vbroker.orb.ORB)
                         oracle.aurora.jndi.orb_dep.Orb.init();

try {
// Get the certificate manager
    AuroraCertificateManager cm = AuroraCertificateManagerHelper.narrow(
           orb.resolve_initial_references("AuroraSSLCertificateManager"));

     byte[] userCert = decoder.decodeBuffer(testCert_base64);
     byte[] caCert = decoder.decodeBuffer(caCert_base64);

  // Set my certificate chain, ordered from CA to user.
     byte[][] certificates = { caCert, userCert };
     cm.setCertificateChain(certificates);

     // Set my private key.
     byte[] encryptedPrivateKey =
              decoder.decodeBuffer(encryptedPrivateKey_base64);

     cm.setEncryptedPrivateKey(encryptedPrivateKey, "welcome12");

  } catch (Exception e) {
     e.printStackTrace();
     throw new org.omg.CORBA.INITIALIZE( "Couldn't initialize SSL context");
  }

  return "I Called back and got: " + client.helloBack ();
}
```

Authorization

The SSL layer authenticates the peers during the connect handshake. After the handshake, you can be assured that the peer is authenticated to be who they said they are. In addition, since the server has specified, within an Oracle wallet, its trustpoints, the SSL adapter on the server will authorize the client. However, the client has the option of how much authorization is done against the server.

- The client can direct the SSL layer to authorize the server by setting up trustpoints.

- The client can authorize the server itself by extracting the server's certificate chain and parsing through the chain.

Setting up Trust Points

The server automatically has trustpoints established through the installed Oracle Wallet. The trustpoints in the wallet are used to verify the client's certificates. However, if the client wants to verify the server's certificates against certain trustpoints, it can set up its these trustpoints, as follows:

- If server-side authentication is requested, the client does not have any certificates set. Thus, to verify the server's certificates, the client can set a single trustpoint through JNDI, or if it is a pure CORBA application—that does not use JNDI—can add trustpoints through the `AuroraCertificateManager.addTrustedCertificate` method. See Example 6–4 on how to set a single trustpoint through JNDI.

- If client-side authentication is requested, the client has set up certificates. Thus, the client can add trustpoints to the file that contains its certificates, can add a single trustpoint through JNDI, or if it is a pure CORBA application—that does not use JNDI—can add trustpoints through the `AuroraCertificateManager.addTrustedCertificate` method.

If the client does not set up trust points, it does not hinder the authorization. That is, JServer assumes that the client trusts the server.

Example 6–4 Verifying Trustpoints

The following example shows how the client sets up its trustpoints through JNDI. The JNDI `SECURITY_TRUSTED_CERT` property can take only a single certificate.

```
// setup the trust point
env.put(ServiceCtx.SECURITY_TRUSTED_CERT, trustedCert);
```

Parsing through the Server's Certificate Chain

The client retrieves the certificates to perform any authorization checks. In the past, you could retrieve the single issuer certificate. Now, you receive the entire issuer certificate chain. You must parse the certificate chain for the information that you need. You can parse the chain through the `AuroraCurrent` object.

> **Note:** You must configure the database and listener to be SSL-enabled, as described in Chapter 3, "Configuring IIOP Applications".

> **Note:** JDK 1.1 certificate classes were contained within `javax.security.cert`. In JDK 1.2, these classes moved to `java.security.cert`.

`AuroraCurrent` contains three methods for retrieving and managing the certificate chain. For creating and parsing the certificate chain, you can use the `X509Cert` class methods. For information on this class, see Sun Microsystems's JDK documentation. Note that the `X509Cert` class manipulates the certificate chain differently in JDK 1.1 than in Java 2.

The `AuroraCurrent` class methods are as follows:

- `getPeerDERCertChain`—obtain the peer's certificate chain, which enables you to verify that the peer is authorized to access your application methods.

- `getNegotiatedProtocolVersion`—obtain the SSL protocol version being used by the connection, to verify the versioning.

- `getNegotiatedCipherSuite`—obtain the cipher suite used to encrypt messages passed over the connection, to verify that the encryption is strong enough for your purposes.

When the handshake occurs, the protocol version and the type of encryption used is negotiated. The type of encryption can be full or limited encryption, which complies with the United States legal restrictions. After the handshake completes, the AuroraCurrent can retrieve what was resolved in the negotiation.

AuroraCurrent Class

The following describes the methods contained within `AuroraCurrent`. See Example 6–5 for a code example of these methods.

getNegotiatedCipherSuite

This method obtains the type of encryption negotiated in the handshake with the peer.

Syntax

```
String getNegotiatedCipherSuite(org.omg.CORBA.Object peer);
```

Parameter	Description
peer	The peer from which you obtain the negotiated cipher.

Returns

A string one of the following values:

Export ciphers:

- SSL_RSA_EXPORT_WITH_RC4_40_MD5

- SSL_RSA_EXPORT_WITH_DES40_CBC_SHA

- SSL_DH_anon_EXPORT_WITH_RC4_40_MD5

- SSL_DH_anon_EXPORT_WITH_DES40_CBC_SHA

- SSL_RSA_WITH_NULL_SHA

- SSL_RSA_WITH_NULL_MD5

Domestic ciphers

- SSL_RSA_WITH_3DES_EDE_CBC_SHA

- SSL_RSA_WITH_RC4_128_SHA

- SSL_RSA_WITH_RC4_128_MD5

- SSL_RSA_WITH_DES_CBC_SHA

- SSL_DH_anon_WITH_3DES_EDE_CBC_SHA

- SSL_DH_anon_WITH_RC4_128_MD5

- SSL_DH_anon_WITH_DES_CBC_SH

getPeerDERCertificateChain

This method obtains the peer's certificate chain. After retrieving the chain, you can parse through the certificates within the chain, to authorize the peer to your application.

Syntax

```
byte [] [] getPeerDERCertificateChain(org.omg.CORBA.Object peer);
```

Parameter	Description
peer	The peer from which you obtain its certificate chain.

Returns

A byte array containing an array of certificates.

getNegotiatedProtocolVersion
This method obtains the negotiated SSL protocol version of a peer.

Syntax

```
String getNegoriatedProtocolVersion(org.omg.CORBA.Object peer);
```

Parameter	Description
peer	The peer from which you obtain the negotiated protocol version.

Returns

A string with one of the following values:

- SSL_Version_Undetermined
- SSL_Version_3_0

Example 6–5   Retrieving a Peer's SSL information for Authorization

This example shows how to authorize a peer by retrieving the certificate information using the AuroraCurrent object.

1. To retrieve an **AuroraCurrent** object, invoke the ORB.resolve_initial_references method with **AuroraSSLCurrent** as the argument.

2. Retrieve the SSL information from the peer through **AuroraCurrent** methods: **getNegotiatedCipherSuite**, **getNegotiatedProtocolVersion**, and **getPeerDERCertChain**.

3. Authorize the peer. You can authorize the peer based on its certificate chain.

> **Note:** This example uses the x509Certificate class methods for parsing the certificate chain and is specific to Java 2. If you are using Java 1.1, you must use the x509Certificate class methods specific to Java 1.1.

```java
static boolean verifyPeerCert(org.omg.CORBA.Object obj) throws Exception
{
   org.omg.CORBA.ORB orb = oracle.aurora.jndi.orb_dep.Orb.init();

   // Get the SSL current
   AuroraCurrent current = AuroraCurrentHelper.narrow
       (orb.resolve_initial_references("AuroraSSLCurrent"));

   // Check the cipher
   System.out.println("Negotiated Cipher:  " +
                      current.getNegotiatedCipherSuite(obj));
   // Check the protocol version
   System.out.println("Protocol Version:   " +
                      current.getNegotiatedProtocolVersion(obj));
   // Check the peer's certificate
   System.out.println("Peer's certificate chain : ");
   byte [] [] certChain = current.getPeerDERCertChain(obj);

   //Parse through the certificate chain using the X509Certificate methods
   System.out.println("length : " + certChain.length);
   System.out.println("Certificates: ");
   CertificateFactory cf = CertificateFactory.getInstance("X.509");

   //For each certificate in the chain
   for(int i = 0; i < certChain.length; i++) {
     ByteArrayInputStream bais = new ByteArrayInputStream(certChain[i]);
     Certificate  xcert = cf.generateCertificate(bais);
     System.out.println(xcert);
     if(xcert instanceof X509Certificate)
     {
       X509Certificate x509Cert = (X509Certificate)xcert;
       String globalUser = x509Cert.getSubjectDN().getName();
       System.out.println("DN out of the cert : " + globalUser);
     }
   }

   return true;
}
```

Note: The x509Certificate class is a Java 2 class. See Sun Microsystems's documentation for more information. In addition, you can find information in the javadoc for javax.net.ssl.

7

Transaction Handling

In Oracle8*i*, Enterprise JavaBeans use Java Transaction API (JTA) 1.0.1 for managing transactions. JTA provides the ability for both bean-managed and container-managed transactions:

- Bean-managed transactions are programmatically demarcated within your bean implementation. The transaction is completely controlled by the application.

- Container-managed transactions are controlled by the container. That is, the container either joins the client's transaction or starts the transaction for the application—as defined within the deployment descriptor—and ends the transaction when the bean completes. Your implementation does not need to provide code for managing the transaction.

This chapter assumes that you have a working knowledge of JTA. The discussion focuses mostly on examples and explaining the differences between the Sun Microsystems JTA specification and the Oracle JTA implementation. See `http://www.javasoft.com` for the Sun Microsystems JTA specification.

- Transaction Overview
- JTA Server-Side Demarcation
- JTA Client-Side Demarcation
- Configuring Two-Phase Commit Engine
- Creating DataSource Objects Dynamically
- Setting the Transaction Timeout
- JDBC Restrictions

Transaction Overview

Transactions manage changes to multiple databases within a single application as a unit of work. That is, if you have an application that manages data within one or more databases, you can ensure that all changes in all databases are committed at the same time if they are managed within a transaction.

Transactions are described in terms of ACID properties, which are as follows:

- *Atomic*: all changes to the database made in a transaction are rolled back if any change fails.

- *Consistent*: the effects of a transaction take the database from one consistent state to another consistent state.

- *Isolated*: the intermediate steps in a transaction are not visible to other users of the database.

- *Durable*: when a transaction is completed (committed or rolled back), its effects persist in the database.

The JTA implementation, specified by Sun Microsystems, relies heavily on the JDBC 2.0 specification and XA architecture. The result is a complex requirement on applications in order to ensure that the transaction is managed completely across all databases. Sun Microsystems's specifies Java Transaction API (JTA) 1.0.1 and JDBC 2.0 on `http://www.javasoft.com`.

You should be aware of the following when using JTA within the Oracle8*i* environment:

- Global and Local Transactions

- Demarcating Transactions

- Transaction Context Propagation

- Two-Phase Commit

- Enlisting Resources

- JTA Limitations

Global and Local Transactions

Whenever your application connected to a database using JDBC or a SQL server, you were creating a transaction. However, the transaction involved only the single database and all updates made to the database were committed at the end of these changes. This is referred to as a local transaction.

A global transaction involves a complicated set of management objects—objects that track all of the objects and databases involved in the transaction. These global transaction objects—`TransactionManager` and `Transaction`—track all objects and resources involved in the global transaction. At the end of the transaction, the `TransactionManager` and `Transaction` objects ensure that all database changes are atomically committed at the same time.

Demarcating Transactions

A transaction is said to be demarcated, which means that each transaction has a definite start and stop point. For example, in an interactive tool such as SQL*Plus, each SQL DML statement implicitly begins a new transaction, if it is not already part of a transaction. A transaction ends when a SQL COMMIT or ROLLBACK statement is issued.

The important designation for a transaction depends on the originator of the transaction and whether explicitly or implicitly demarcated:

- Transaction originator—In a distributed object application, transactions are demarcated differently if the originator is the client or the server. Where the transaction originates defines the transaction as *client-side demarcated* or *server-side demarcated.* See "JTA Client-Side Demarcation" on page 7-16 and "JTA Server-Side Demarcation" on page 7-13 for more information.

- Explicit or implicit demarcation

    * Explicit demarcation means that either the client or bean-managed transactional bean programmatically demarcates the transaction by executing the appropriate `begin` or `commit` methods. Explicit demarcation is discussed further in both "JTA Client-Side Demarcation" on page 7-16 and "JTA Server-Side Demarcation" on page 7-13.

    * Implicit demarcation is specified within the deployment descriptor for container-managed transactional beans. The container begins and ends the transaction depending on the configuration for the bean within the deployment descriptor. See "JTA Server-Side Demarcation" on page 7-13 for more information on implicit demarcation.

> **Note:** The originating client or object that starts the transaction must also end the transaction with a commit or rollback. However, the originator can end the transaction in a different method than the originating method. For example, if the client begins the transaction, calls out to a server object, the client must end the transaction after the invoked method returns. The invoked server object cannot end the transaction.

Container or Bean Managed Transactions

Enterprise JavaBeans can specify whether the bean demarcates and manages any transactions within itself or whether the container should demarcate and manage the transaction.

Container-Managed Transactional

The bean specifies itself as container-managed transactional; it does not have any transactional implementation within its methods. All transactional logic is executed by the container based on the transactional attribute specified in the EJB deployment descriptor. See the following for more information:

- "Defining Transactions" on page A-19 describes the EJB deployment descriptor specification for container-managed transactions.

- "Propagating the Transactional Context to Container-Managed Transactional Beans" on page 7-6 describes what the container performs depending on the deployment descriptor specified transactional attribute.

Bean-Managed Transactional

If the bean specifies itself as bean-managed transactional, it has the following responsibility:

- All global transactions are started within itself. It cannot accept a transactional context from any other object—client or server. Thus, a bean-managed transactional bean is not involved in any other object's global transaction. If the invoking object is involved in a transaction, that transaction is suspended while the bean-managed transactional bean is executing. The invoking object's transaction is resumed when the bean-managed transactional bean returns.

- If the bean wants another object involved in the global transaction, it must invoke a container-managed bean with a transactional attribute that accepts propagated transaction contexts.

- The bean can request resources to be automatically enlisted in the same manner as detailed in "Enlisting Resources" on page 7-7.

Transaction Context Propagation

When you begin a transaction within either a client or a server instance, JTA denotes the originator in the transaction manager. As the transaction involves more objects and resources, the transaction manager tracks all of these objects and resources in the transaction and manages the transaction for these entities.

When an object calls another object, in order for the invoked object to be included in the transaction, JTA propagates the transaction context to the invoked object. Propagation of the transaction context is necessary for including the invoked object into the global transaction.

As shown in Figure 7–1, if the client begins a global transaction, calls a server object in the database, the transaction context is propagated to the server object. If the server object supports transactions, this object is attached to the transaction manager as involved in the global transaction. If this server object invokes another server object, within the same or a remote database, the transaction context is propagated to this object as well. This ensures that all objects that are supposed to be involved in the global transaction are tracked by the transaction manager.

> **Note:** Since all server objects are loaded within a database, JTA automatically enlists the database as a resource included in the global transaction along with the server object.

Figure 7–1   Connection to an Object over IIOP
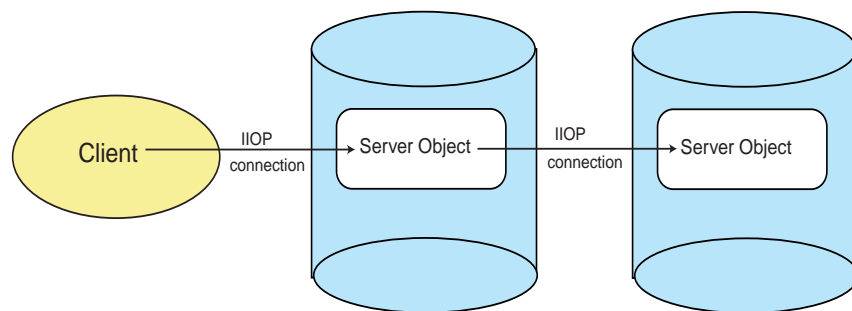
Propagating the Transactional Context to Container-Managed Transactional Beans

The definition of the transaction attribute within the EJB deployment descriptor for container-managed transactional beans determines whether the global transaction context is propagated to the server object. The following table lists each transaction attribute and the behavior that occurs for each server object type and for any resources.

Table 7–1 Effect of Transactional Attributes for Container-Managed Transactional Beans

Deployment Transaction Attribute	Client Transaction Demarcation	Behavior for Target Server Object or Resource (Database)
NotSupported	Does not start transaction	No transaction started
	Starts transaction	Invoker's transaction is suspended while the bean executes, resumed when control returns to the invoker.
Required	Does not start transaction	A new transaction is started
	Starts transaction	Invoker's transaction context is propagated. Server object and the local resource joins the transaction.
Supports	Does not start transaction	No transaction started
	Starts transaction	Invoker's transaction context is propagated. Server object and the local resource joins the transaction.
RequiresNew	Does not start transaction	A new transaction is started
	Starts transaction	Invoker's transaction is suspended. A new transaction is started and committed before returning to the invoker. The invoker's transaction is resumed when control is returned to it.

Table 7–1   Effect of Transactional Attributes for Container-Managed Transactional Beans

Deployment Transaction Attribute	Client Transaction Demarcation	Behavior for Target Server Object or Resource (Database)
Mandatory	Does not start transaction	Error is returned. This object requires a transactional context.
	Starts transaction	Invoker's transaction context is propagated. Server object and the local resource joins the transaction.
Never	Does not start transaction	No transaction is started
	Starts transaction	An error is returned. This object cannot be called from any object—client or server—that is involved in a transaction.

Enlisting Resources

Each resource, including databases, that you want managed in the global transaction must be enlisted. The Oracle8*i* JTA implementation automatically enlists all databases if you open a JDBC connection to the database within the context of a global transaction. The *Oracle8i JDBC Developer's Guide* contains more information on how to open a JDBC connection to a database within a transaction.

JTA automatically enlists a database resource when one of the following occurs:

Enlisting the Local Database

The following JDBC methods are used within a local transaction for accessing the local database. In order to ensure that the statements are included within a global transaction, execute these methods and subsequent SQL statements after the global transaction has started.

Table 7–2   JDBC Methods Used For Enlisting Databases

Retrieval Method	Description
OracleDriver(). defaultConnection()	Pre-JDBC 2.0 method for retrieving the local connection. If using the #sqlj macro for updating your database tables, note that this macro invokes the defaultConnection method for you.
DriverManager.getConnection ("jdbc:oracle:kprb:")	Pre-JDBC 2.0 method for retrieving the local connection.

Table 7–2   JDBC Methods Used For Enlisting Databases

Retrieval Method	Description
`DataSource.getConnection` `("jdbc:oracle:kprb:")`	JDBC 2.0 method for retrieving connections to the local databases.

> **Note:**   You should use either the JDBC 2.0 method or the pre-2.0 JDBC methods for retrieving connections to databases. You must not mix both methods within the same bean.

An example for each of these methods of automatic enlistment are detailed in "Enlisting Resources on the Server-side" on page 7-25.

Enlisting a Remote Database

An object—client or server object—can only enlist a database within the global transaction through JDBC 2.0 methods within the `DataSource` object. The `getConnection` method must be invoked after the `begin` method of the `UserTransaction` object.

If your transaction involves more than one database, you must specify an Oracle8*i* database as the two-phase commit engine. See "Configuring Two-Phase Commit Engine" on page 7-29 for more information.

Two-Phase Commit

One of the primary advantages for a global transaction is the number of objects and database resources managed as a single unit within the transaction. If your global transaction involves more than one database resource, you must specify a two-phase commit engine, which is an Oracle8*i* database designated to manage the changes to all databases within the transaction. The two-phase commit engine is responsible for ensuring that when the transaction ends, all changes to all databases are either totally committed or fully rolled back.

On the other hand, if your global transaction has multiple server objects, but only a single database resource, you do not need to specify a two-phase commit engine. The two-phase commit engine is required only to synchronize the changes for multiple databases. If you have only a single database, single-phase commit can be performed by the transaction manager.

> **Note:** Your two-phase commit engine can be any Oracle8*i*
> database. It can be the database where your server object exists, or
> even a database that is not involved in the transaction at all. See
> "Configuring Two-Phase Commit Engine" on page 7-29 for a full
> explanation of the two-phase commit engine setup.

Figure 7–2 shows three databases enlisted in a global transaction and another
database that is designated as the two-phase commit engine. All databases,
including the local database, are automatically enlisted when a JDBC connection is
opened after the global transaction starts. See "Enlisting Resources" on page 7-7 for
more information on database enlistment.

When the global transaction ends, the two-phase commit engine ensures that all
changes made to the databases A, B, and the local are committed or rolled back
simultaneously.

Figure 7–2   Two-Phase Commit for Global Transactions
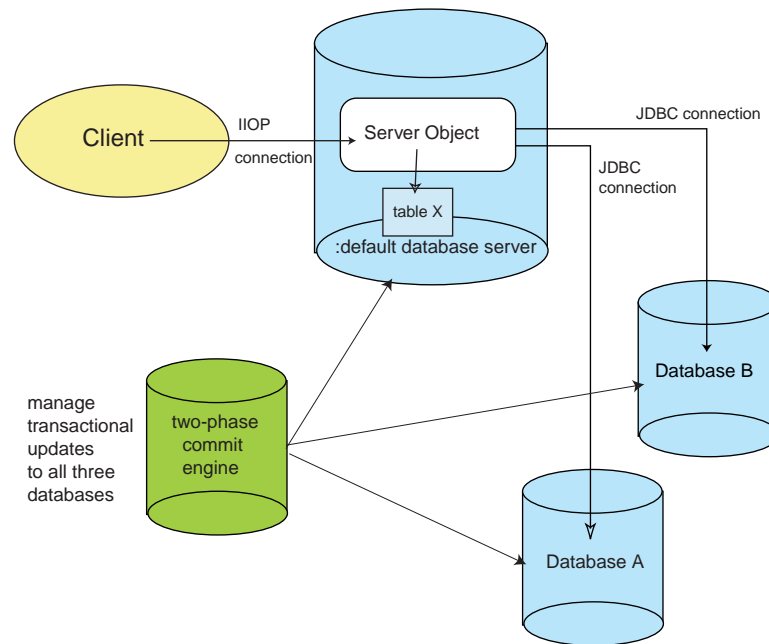
JTA Summary

The following sections summarize the details for demarcating the transaction and enlisting the database in the transaction. These details are explained and demonstrated in the rest of the chapter. However, these tables provide a reference point for you.

Table 7–3   Environment Setup For Transactional Object Retrieval

Source	Qualifiers	Environment Setup
		Setup includes authentication information, namespace URL, and OracleDriver registration.
Client	retrieve remote object or remote database connection.	Always
Server	use in-session activation; retrieving a local object or local database connection	Never
	retrieve remote object or remote database connection	Always

Table 7–4   Differences Between Using JDBC 2.0 DataSource or Pre-2.0 JDBC Drivers

	JDBC 2.0 DataSource	Pre-2.0 JDBC Drivers: OracleDriver or DriverManager
Binding Differences	You must bind the `DataSource` into the namespace with the `bindds` command.	No binding of any JDBC objects is required.
Code Differences You perform a JNDI lookup, as follows: Client	1. Provide the following environment setup: the environment `Hashtable` contains authentication information and namespace URL, and register the `OracleDriver`. 2. Retrieve the `DataSource` object through a JNDI lookup that contains the `"jdbc_access://"` prefix.	Cannot be used in the client.
Server	Do one of the following: ■ Retrieve the `DataSource` object using in-session activation. Environment setup and `"jdbc_access://"` prefix is not required. ■ Lookup a database through an environment variable that was previously specified in the deployment descriptor. This uses the `"java:comp/env"` prefix.	The pre-2.0 JDBC drivers—`OracleDriver` and `DriverManager`—can only be used within a server object. Retrieval of JDBC connections is the same as in previous releases.

Table 7–5   Differences Between Container and Bean-Managed Transactions

		Container-Managed Transaction	Bean-Managed Transaction
Single-Phase Commit	Deployment Descriptor	■ Define that this is container-managed in the `<transaction-type>` element. ■ Define the type of container management attribute, one of the values described in Table 7–1, within the `<container-transaction>` element in the XML deployment descriptor.	■ Define that this is bean-managed in the `<transaction-type>` element.
	Binding	■ No binding required for `UserTransaction`. The `UserTransaction` object is created for you. ■ If using a `DataSource` object in the transaction, bind it using the `bindds` command. However, you do not need any database links.	■ No binding required for `UserTransaction`. The `UserTransaction` object is created for you. ■ If using a `DataSource` object in the transaction, bind it using the `bindds` command. However, you do not need any database links.
	Runtime	■ Do not retrieve the `UserTransaction`. The container manages the `UserTransaction` for you. ■ If using the `DataSource` object to manage SQL DML statements within the transaction, retrieve the `DataSource`.	■ Retrieve the `UserTransaction` either through the EJB 1.0 `getUserTransaction` method of `SessionCtx` or the EJB 1.1 method of a JNDI `lookup` with the `"java:comp/ UserTransaction"` string. ■ Your runtime is responsible for starting and terminating the transaction. ■ If using the `DataSource` object to manage SQL DML statements within the transaction, retrieve the `DataSource`.

Table 7–5   Differences Between Container and Bean-Managed Transactions (Cont.)

		Container-Managed Transaction	**Bean-Managed Transaction**
Two-Phase Commit	Deployment Descriptor	In addition to the single-phase requirements, you must also add the JNDI bound name for the `UserTransaction` object in the `<transaction-manager>` element in the Oracle-specific deployment descriptor.	Same requirements as the container-managed transaction.
	Binding	■ You must bind a `UserTransaction` object with the fully-qualified database address of the two-phase commit engine and its username and password. ■ The user that was bound within the `UserTransaction` is the one that must have the privilege to commit the transaction. Thus, make sure that this user has the "`FORCE ANY TRANSACTION`" privilege. ■ You must bind `DataSource` objects for each database involved in the transaction with a database link from the two-phase commit engine to itself.	■ You must bind a `UserTransaction` object with the fully-qualified database address of the two-phase commit engine and its username and password. ■ You must bind `DataSource` objects for each database involved in the transaction with a database link from the two-phase commit engine to itself.
	Runtime	■ Do not retrieve the `UserTransaction`. The container manages the `UserTransaction` for you. ■ Retrieve the `DataSource`.	■ Retrieve the `UserTransaction` either through the EJB 1.0 `getUserTransaction` method of `SessionCtx` or the EJB 1.1 method of a JNDI `lookup` with the "`java:comp/ UserTransaction`" string. ■ Your runtime is responsible for starting and terminating the transaction. ■ If using the `DataSource` object to manage SQL DML statements within the transaction, retrieve the `DataSource`.

JTA Limitations

The following are the portions of the JTA specification that Oracle8*i* does not support.

Nested Transactions

Nested transactions are not supported in this release. If you attempt to begin a new transaction before committing or rolling back any existing transaction, the transaction service throws a `NotSupportedException` exception.

Interoperability

The transaction services supplied with this release do not interoperate with other JTA implementations.

JTA Server-Side Demarcation

Server-side demarcation can occur either implicitly through container-managed transaction or explicitly through bean-managed transactions. A container-managed transaction is specified within the EJB deployment descriptor for the bean. Most EJBs use container-managed transactions as it requires no transaction implementation within the application.

No matter which way that the server object decides to demarcate transactions, the enlistment of the database is the same for both container and bean-managed transactions.

- Container-Managed Transactions
- Bean-Managed Transactions
- Enlisting Resources on the Server-side

Container-Managed Transactions

You can declare that the container manages the transaction for the bean within the EJB deployment descriptor. This requires that you specify "`Container`" within the `<transaction-type>` element and the container-managed transaction attribute (see Table 7–1) within the `<container-transaction>` element. See "Defining Transactions" on page A-19 for full instructions.

Based upon the transaction attribute you specify within the EJB deployment descriptor, the container will begin, commit, or rollback global transactions when a method in the bean instance is invoked. Each bean can be specified with different

transaction attributes. The transaction attributes specify how the transaction demarcation is handled. This means that your bean does not retrieve the UserTransaction object, nor invokes any of its methods. The container does this for you.

The container does not enlist the database resources for you. Bind each DataSource within the namespace as indicated within Table 7–5 on page 7-11. Your code enlists each database by retrieving the database connection through one of the methods listed in Table 7–2 on page 7-7. A full example using container-managed transactional beans and database enlistment is shown in the section "Configuring Two-Phase Commit Engine" on page 7-29.

Bean-Managed Transactions

Only session beans have the option to use bean-managed transactions. This means that only session beans can demarcate the transaction with the begin, commit, and rollback methods. In order to specify that this session bean is going to programmatically demarcate its transaction, it must specify the following in its EJB deployment descriptor:

```
<enterprise-beans>
 <session>
  . . .
  <transaction-type>Bean</transaction-type>
 </session>
</enterprise-beans>
```

> **Note:** See "Defining Transactions" on page 7-13 for a full description of defining transaction management in the EJB deployment descriptor.

The session bean implementation demarcates transactions in the same manner as a client would. It invokes the begin, commit, and rollback methods off of the UserTransaction object. The only difference is how the UserTransaction object is retrieved. There are two methods for retrieving the UserTransaction object on the server side in a bean-managed transactional bean:

- SessionContext getUserTransaction method
- JNDI lookup

SessionContext getUserTransaction method

The bean retrieves the UserTransaction object from the SessionContext object, which was set within the setSessionContext method. This is the EJB 1.0 methodology for retrieving the UserTransaction object and can only be used in a single-phase commit environment.

The container has already retrieved the UserTransaction object for you. The following shows that the session context is saved in the ctx variable.

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

Within the bean implementation, retrieve the UserTransaction from the session context and begin the transaction.

```
UserTransaction ut = ctx.getUserTransaction();
ut.begin();
...
ut.commit();
```

JNDI lookup

The EJB 1.1 method retrieves the UserTransaction object by performing an in-session lookup with the following JNDI name:
"java:comp/UserTransaction". In a single-phase commit environment, the container will create the UserTransaction object for you; in a two-phase environment, you must have already bound a UserTransaction object with the two-phase commit information (username, password, and two-phase commit URL) into the namespace. After retrieval, you use this object to demarcate your global transaction.

The following demonstrates retrieving a UserTransaction object from the namespace:

```
ic = new InitialContext ( );

// lookup the usertransaction
UserTransaction ut = (UserTransaction)ic.lookup ("java:comp/UserTransaction");
ut.begin ();
...
ut.commit();
```

JTA Client-Side Demarcation

For JTA, client-side demarcated transactions are explicitly demarcated programmatically through the `UserTransaction` object, which must be bound with the `bindut` command into the namespace. With client-side transaction demarcation, the client controls the transaction. The client starts a global transaction by invoking the `UserTransaction begin` method; it ends the transaction by invoking either the `commit` or `rollback` methods.

In addition, the client must always set up an environment including a `Hashtable` with authentication information and namespace location URL. It must also register an `OracleDriver` when retrieving the transaction objects from the namespace.

Figure 7–3 shows a client invoking a server object—which, in this example, is a container-managed transactional bean. The client starts a global transaction, then invokes the bean. Since the bean is a container-managed transactional bean and is specified with the transactional attribute of "Supports", the transactional context is propagated to include the server object.

Figure 7–3   Client Demarcated Global Transaction
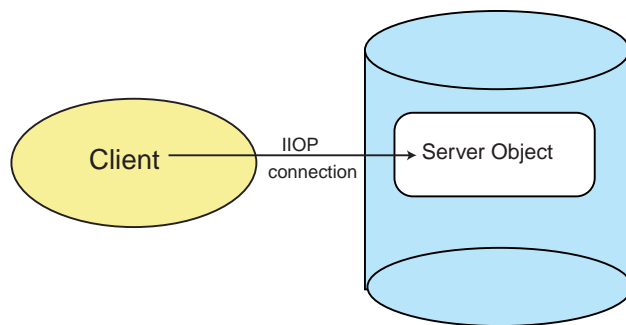
The following must occur for the client to demarcate the transaction:

1. Initialize a `Hashtable` environment with the namespace address and authentication information.

2. Register the `OracleDriver`.

3. Retrieve the `UserTransaction` object from the namespace within the client logic. When you retrieve the `UserTransaction` object from any client, the URL must consist of "`jdbc_access://`" prefix before the JNDI name.

4. Start the global transaction within the client using `UserTransaction.begin()`.

5. Retrieve the server bean.

6. Invoke any object methods to be included in the transaction.

7. End the transaction through `UserTransaction.commit()` or `UserTransaction.rollback()`.

Example 7–1 shows a client that invokes a server bean within the transaction.

Example 7–1   Employee Client Code for Client Demarcated Transaction

Before starting the client, you must first bind the `UserTransaction` object in the namespace.

Bind UserTransaction Object in the Namespace

You bind the `UserTransaction` object in the namespace through the `bindut` command of the `sess_sh` tool. To bind a `UserTransaction` object to the name "`/test/myUT`" in the namespace located on `nsHost`, execute the following:

```
sess_sh -service jdbc:oracle:thin:@nsHost:5521:ORCL -user SCOTT -password TIGER
& bindut /test/myUT
```

Verify that the user bound with the `UserTransaction` has FORCE ANY TRANSACTION granted to the user that bound this object. This privilege enables the user to commit this transaction. In this example, you would execute the following:

```
GRANT FORCE ANY TRANSACTION TO SCOTT
```

> **Note:**   The client needs the same information to retrieve the `UserTransaction` as you give within the `bindut` command.

Developing the Client Application

After binding the `UserTransaction` object, your client code can retrieve the `UserTransaction` object and start a global transaction. Since the client is retrieving the `UserTransaction` object from a remote site, the lookup requires authentication information, location of the namespace, the `OracleDriver` registration, and the "`jdbc_access://`" prefix.

```
import employee.Employee;
```

```
            import employee.EmployeeHome;
            import employee.EmployeeInfo;

            import javax.naming.Context;
            import javax.naming.InitialContext;
            import java.util.Hashtable;
            import javax.transaction.*;

            public class Client
            {
              public static void main (String[] args) throws Exception {

                //Set up the service URL to where the UserTransaction object
                //is bound. Since from the client, the connection to the database
                //where the namespace is located can be communicated with over either
                //a Thin or OCI8 JDBC driver. This example uses a Thin JDBC driver.
                String namespaceURL = "jdbc:oracle:thin:@nsHost:1521:ORCL";

                //User and password are case sensitive.
                String user = "SCOTT";
                String password = "TIGER";

                //1.(a) Authenticate to the database.
                // create InitialContext and initialize for authenticating client
                Hashtable env = new Hashtable ();
                env.put (Context.URL_PKG_PREFIXES, "oracle.aurora.jndi");
                env.put (Context.SECURITY_PRINCIPAL, user);
                env.put (Context.SECURITY_CREDENTIALS, password);
                env.put (Context.SECURITY_AUTHENTICATION, ServiceCtx.NON_SSL_LOGIN);
                //1.(b) Specify the location of the namespace where the transaction objects
                //    are bound.
                env.put(jdbc_accessURLContextFactory.CONNECTION_URL_PROP, namespaceURL);
                Context ic = new InitialContext (env);

                //2. Register a JDBC OracleDriver. Requirement for opening JDBC connection
                // to retrieve UserTransaction object from namespace
                DriverManager.registerDriver (new oracle.jdbc.driver.OracleDriver());

                //3. Retrieve the UserTransaction object from JNDI namespace
                UserTransaction ut = (UserTransaction)ic.lookup ("jdbc_access://test/myUT");

                //4. Start the transaction
                ut.begin();

                //5. Retrieve the EJB
```

```
      // get an handle to the employee_home object
      EmployeeHome employee_home =
        (EmployeeHome)ic.lookup ("sess_iiop://myhost:1521:orcl/test/employee");

      // get an handle to the remote bean
      Employee employee = employee_home.create ();

      //6. Perform bean business logic.
      // get an info of an employee
      EmployeeInfo info = employee.getEmployee ("SCOTT");
      System.out.println ("Beginning salary = " + info.salary);

      // do work on the info-object
      info.salary += (info.salary * 10) / 100;

      // call update on the server-side
      employee.updateEmployee (info);

      //7. End the transaction
      //Commit the updated value
      ut.commit();
  }
}
```

Example 7–2   Container-Managed Transactional Bean with Supports Attribute

In the EJB deployment descriptor, the container-managed transactional session bean
is specified with the Supports attribute. These transactional specifications are
defined in the `<transaction-type>` and `<container-transaction>`
elements.

XML Deployment Descriptor

```
<?xml version="1.0"?>
<!DOCTYPE ejb-jar PUBLIC "-//Oracle Corporation//DTD Enterprise JavaBeans 1.1
//EN" "ejb-jar.dtd">
<ejb-jar>
   <enterprise-beans>
      <session>
         <ejb-name>test/myEmployee</ejb-name>
         <home>employee.EmployeeHome</home>
         <remote>employee.Employee</remote>
         <ejb-class>employeeServer.EmployeeBean</ejb-class>
         <session-type>Stateful</session-type>
         <transaction-type>Container</transaction-type>
```

```
        </session>
    </enterprise-beans>
    <assembly-descriptor>
      <container-transaction>
        <method>
           <ejb-name>EmployeeBean</ejb-name>
           <method-name>*</method-name>
        </method>
        <trans-attribute>Supports</trans-attribute>
      </container-transaction>
    </assembly-descriptor>
</ejb-jar>
```

Since the session bean allows the container to manage the transaction and the bean is specified as supporting existing transactions, the transaction context is propagated to the bean when the client invokes it. The bean itself contains no transactional logic within it. It only contains the bean implementation code, as follows:

> **Note:** The database that this bean resides on is automatically enlisted. Thus, any SQLJ statement is executed against the database and its changes are included in the global transaction. See "Local Database Enlistment" on page 7-25 for full details on the automatic enlistment of a local database.

```
package employeeServer;

import employee.*;

import javax.ejb.SessionBean;
import javax.ejb.CreateException;
import javax.ejb.SessionContext;
import java.rmi.RemoteException;

import java.sql.SQLException;

public class EmployeeBean implements SessionBean
{
  // Methods of the Employee interface
  public EmployeeInfo getEmployee (String name)
                           throws RemoteException, SQLException
  {
    int empno = 0;
```

```
  double salary = 0.0;
  #sql { select empno, sal into :empno, :salary from emp
              where ename = :name };

  return new EmployeeInfo (name, empno, salary);
}

public void updateEmployee (EmployeeInfo employee)
    throws RemoteException, SQLException
{
  #sql { update emp set ename = :(employee.name),
              sal = :(employee.salary)
              where empno = :(employee.number) };
  return;
}

// Methods of the SessionBean
public void ejbCreate () throws RemoteException, CreateException {}
public void ejbRemove () {}
public void setSessionContext (SessionContext ctx) {}
public void ejbActivate () {}
public void ejbPassivate () {}
}
```

JTA Client-Side Demarcation Including Databases

The previous example showed how a transaction context was propagated to server objects from a client within the JTA global transaction. When you execute the server object, the transaction is propagated over the IIOP transport layer. In addition to invoking IIOP server objects, you may wish to update databases over JDBC connections. This section shows how you enlist databases using a JDBC connection in tandem with the IIOP server object propagation.

To include a remote database within the transaction from a client, you must use a DataSource object, which has been bound in the namespace as a JTA DataSource. Then, invoke the getConnection method of the DataSource object after the transaction has started, and the database is included in the global transaction. See "Enlisting Resources" on page 7-7 for more information.

The following must occur in the client runtime to demarcate the transaction:

1. Initialize a Hashtable environment with the namespace address and authentication information.

2. Register the OracleDriver.

3. Retrieve the `UserTransaction` object from the namespace within the client logic. When you retrieve the `UserTransaction` object from the client, the URL must consist of `"jdbc_access://"` prefix before the JNDI name.

4. Start the global transaction within the client using `UserTransaction.begin()`.

5. Enlist any database resources to be included in the transaction by opening a connection to the specified database, as follows:

    a. Retrieve the `DataSource` object from the namespace within the client logic. When you retrieve the `DataSource` object from any client, the URL must consist of `"jdbc_access://"` prefix before the JNDI name.

    b. Open a connection to the database through `DataSource.getConnection` method.

6. Retrieve the bean reference.

7. Invoke any object methods to be included in the transaction.

8. Invoke SQL DML statements against any enlisted databases.

9. End the transaction through `UserTransaction.commit()` or `UserTransaction.rollback()`.

Example 7–3 shows a client that invokes a bean and enlists a single database within the transaction.

Example 7–3   Employee Client Code for Client Demarcated Transaction

Before starting the client, you must first bind the `UserTransaction` and `DataSource` objects in the namespace. See "Bind UserTransaction Object in the Namespace" on page 7-17 for directions on the binding the `UserTransaction` object.

Bind DataSource Object in the Namespace

Use the `bindds` command of the `sess_sh` tool to bind an `DataSource` object in the namespace. The full command is detailed in the *Oracle8i Java Tools Reference*.

To bind a `DataSource` object for a single-phase commit transaction with the `empHost` database to the name `"/test/empDatabase"` in the namespace located on `nsHost`, execute the following:

```
sess_sh -service jdbc:oracle:thin:@nsHost:5521:ORCL -user SCOTT -password TIGER
& bindds /test/empDatabase -url jdbc:oracle:thin:@empHost:5521:ORCL -dstype jta
```

After binding the `DataSource` object in the namespace, the server can enlist the database within a global transaction.

> **Note:** If using more than one database, you will need to setup for a two-phase commit. See "Configuring Two-Phase Commit Engine" on page 7-29 for more information.

Developing the Client Application

The following example follows the steps listed in "JTA Client-Side Demarcation Including Databases" on page 7-21.

```
import employee.Employee;
import employee.EmployeeHome;
import employee.EmployeeInfo;

import javax.naming.Context;
import javax.naming.InitialContext;
import java.util.Hashtable;
import javax.transaction.*;

public class Client
{
  public static void main (String[] args) throws Exception {

    //Set up the service URL to where the UserTransaction object
    //is bound. Since from the client, the connection to the database
    //where the namespace is located can be communicated with over either
    //a Thin or OCI8 JDBC driver. This example uses a Thin JDBC driver.
    String namespaceURL = "jdbc:oracle:thin:@nsHost:1521:ORCL";

    //User and password are case sensitive.
    String user = "SCOTT";
    String password = "TIGER";

    //1.(a) Authenticate to the database.
    // create InitialContext and initialize for authenticating client
    Hashtable env = new Hashtable ();
    env.put (Context.URL_PKG_PREFIXES, "oracle.aurora.jndi");
    env.put (Context.SECURITY_PRINCIPAL, user);
    env.put (Context.SECURITY_CREDENTIALS, password);
    env.put (Context.SECURITY_AUTHENTICATION, ServiceCtx.NON_SSL_LOGIN);
    //1.(b) Specify the location of the namespace where the transaction objects
    //    are bound.
```

```
env.put(jdbc_accessURLContextFactory.CONNECTION_URL_PROP, namespaceURL);
Context ic = new InitialContext (env);

//2. Register a JDBC OracleDriver. Required for retrieving UserTransaction
// and the DataSource objects from namespace over JDBC connection.
DriverManager.registerDriver (new oracle.jdbc.driver.OracleDriver());

//3. Retrieve the UserTransaction object from JNDI namespace
UserTransaction ut;
ut = (UserTransaction)ic.lookup ("jdbc_access://test/myUT");

//4. Start the transaction
ut.begin();

//5.(a) Retrieve the DataSource (that was previously bound with bindds in
// the namespace. After retrieving the DataSource...
// get a connection to a database. You need to provide authentication info
// for a remote database lookup, similar to what you would do from a client.
// In addition, if this was a two-phase commit transaction, you must provide
// the username and password.
DataSource ds = (DataSource)ic.lookup ("jdbc_access://test/empDB");

// 5.(b) Get connection to the database through DataSource.getConnection
// in this case, the database requires the same username and password as
// set in the environment.
Connection conn = ds.getConnection ();

//6. Retrieve the EJB
// get an handle to the employee_home object
EmployeeHome employee_home =
  (EmployeeHome)ic.lookup (serviceURL + objectName);

// get an handle to the remote bean
Employee employee = employee_home.create ();

//7. (a) Perform bean business logic.
// get an info of an employee
EmployeeInfo info = employee.getEmployee ("SCOTT");
System.out.println ("Beginning salary = " + info.salary);

// do work on the info-object
info.salary += (info.salary * 10) / 100;

// call update on the server-side
employee.updateEmployee (info);
```

```
//7. (b) Execute SQL statements against the enlisted database.
Statement stmt = conn.createStatement ();
int cnt = stmt.executeUpdate ("insert into my_tab values (39304)");

//8. Close the database connection.
conn.close ();

//9. End the transaction
//Commit the updated value
ut.commit();
  }
}
```

The deployment descriptor and bean implementation code is the same as described in Example 7–2.

Enlisting Resources on the Server-side

Whether your bean instance uses bean-managed or container-managed transactions, the databases that the bean accesses must be enlisted to be included within the global transaction. This is discussed more in "Enlisting Resources" on page 7-7 and "Bind DataSource Object in the Namespace" on page 7-22.

Once bound, you can enlist the database after the transaction begins.

- Local Database Enlistment

- Remote Oracle8i Database Enlistment

Local Database Enlistment

Because your bean has been deployed into an Oracle8*i* database, the container enlists this database automatically within the transaction. This database is known as the local database. Thus, if you execute any SQL against this database, the results are committed when the transaction is committed.

- ■ SQLJ   As discussed in the *Oracle8i SQLJ Developer's Guide and Reference,* an implicit local connection is supplied to the database that the object is running in. Any statements executed within the SQLJ statement is executed against the local database. However, in order for the statement results to be part of the transaction, you must execute the SQLJ statement within an open global transaction. That is, the SQLJ statement must be executed after the `UserTransaction.begin` method is invoked. However, do not commit the transaction within a SQLJ statement. Commitment of the transaction should only occur within by the `UserTransaction.commit` method.

- ■ JDBC   You can create a local connection by executing the methods specified in Table 7–6.

Table 7–6   Local Database Retrieval Methods

Retrieval Method	Description
`OracleDriver().` `    defaultConnection()`	See Example 7–6
`DriverManager.getConnection` `    ("jdbc:oracle:kprb:")`	See Example 7–5
`DataSource.getConnection` `    ("jdbc:oracle:kprb:")`	See Example 7–4

Always execute these methods and subsequent SQL statements after the global transaction has started.

Example 7–4   DataSource.getConnection Method Example

The following example is a container-managed transactional bean with RequiresNew attribute, so the global transaction is initialized when the bean is first invoked. The local connection is retrieved through the `DataSource.getConnection` method, which enlists the database in the transaction. SQL statements are executed against the local database. These statements are committed when the global transaction is committed by the container when the bean exits.

```
public EmpRecord query (int empNumber) throws SQLException, RemoteException
  {
    //Retrieving the UserTransaction and DataSource using in-session activation
    Context ic = new InitialContext ( );
```

```
//Retrieve the DataSource using in-session activation
DataSource ds = (DataSource) ic.lookup("/test/myDB");

//Retrieve the local connection object to the local database
Connection conn = ds.getConnection ();

//prepare and execute a sql statement against the local database.
PreparedStatement ps =
  conn.prepareStatement ("select ename, sal from emp where empno = ?");
try {
  ps.setInt (1, empNumber);
  ResultSet rset = ps.executeQuery ();
  if (!rset.next ())
    throw new RemoteException ("no employee with ID " + empNumber);
  return new EmpRecord (rset.getString (1), empNumber, rset.getFloat (2));
} finally {
  ps.close();
}

//close the database connection
conn.close();
}
```

Example 7–5   DriverManager.getConnection Method Example

The following example is the same as Example 7–4, except for the database retrieval method. Substitute the following to retrieve the connection:

```
//Retrieve the local connection object to the local database
    Connection conn =
        DriverManager.getConnection ("jdbc:oracle:kprb:");
```

Example 7–6   OracleDriver().defaultConnection Method Example

The following example is the same as Example 7–4, except for the database retrieval method. Substitute the following to retrieve the connection:

```
//Retrieve the local connection object to the local database
    Connection conn =
        new oracle.jdbc.driver.OracleDriver().defaultConnection ();
```

Remote Oracle8*i* Database Enlistment

If you access a remote Oracle8*i* database from the server that should be included in the transaction, you must open the connection to the database after the global transaction starts.

> **Note:** At this time, the Oracle JTA implementation does not
> support including non-Oracle databases in a global transaction.

Example 7–7   Enlist Database in Single Phase Transaction

The following example enlists a database in the global transaction within a
bean-managed transactional bean.

The following example is a container-managed transactional bean, so it does not
retrieve the `UserTransaction`. However, it does retrieve a `DataSource` to start a
JDBC 2.0 connection.

```
package employeeServer;

import employee.*;

import javax.ejb.SessionBean;
import javax.ejb.CreateException;
import javax.ejb.SessionContext;
import java.rmi.RemoteException;
import java.sql.SQLException;
import javax.transaction.*;


public class EmployeeBean implements SessionBean
{
  SessionContext ctx;

  // Methods of the Employee interface
  public EmployeeInfo getEmployee (String name)
       throws RemoteException, SQLException
  {
    // get a connection to the local database. If this was a two-phase commit
    // transaction, you would provide the username and password
    // for the 2pc engine
    DataSource ds = (DataSource)ic.lookup ("/test/myDS");

    // get connection to the local database through DataSource.getConnection
    Connection conn = ds.getConnection ();

    //perform your SQL against the database.
    //prepare and execute a sql statement.
    //retrieve the employee's selected benefits
    PreparedStatement ps =
```

```
      conn.prepareStatement ("update emp set ename = :(employee.name),
          sal = :(employee.salary) where empno = :(employee.number)");
    try {
      ps.setInt (1, empNumber);
      ResultSet rset = ps.executeQuery ();
      if (!rset.next ())
        throw new RemoteException ("no employee with ID " + empNumber);
      return new EmpRecord (rset.getString (1), empNumber, rset.getFloat (2));
    } finally {
      ps.close();
  }

  //close the connection
   conn.close();

   return new EmployeeInfo (name, empno, salary);

  }

  // Methods of the SessionBean
  public void ejbCreate () throws RemoteException, CreateException {}
  public void ejbRemove() {}
  public void setSessionContext (SessionContext ctx) {
    this.ctx = ctx;
  }
  public void ejbActivate () {}
  public void ejbPassivate () {}
}
```

Configuring Two-Phase Commit Engine

If you have more than a single database involved in your transaction, you must designate a two-phase commit engine for managing all changes to all databases involved in the transaction. A two-phase commit engine is responsible for contacting all of the databases at the end of the transaction and managing the commit or rollback of all updates to all included databases. Thus, this two-phase commit engine must have access to database links to each database included within the transaction.

To configure for a two-phase commit, your system administrator must do the following:

1. Designate an Oracle8*i* database as the two-phase commit engine.

2. Configure database links from the two-phase commit engine to each database that may be involved in the global transaction. This is necessary for the two-phase commit engine to communicate with each database at the end of the transaction.

3. Provide the database link name in the `-dblink` option of `binds` for each individual database when binding that database's `DataSource` into the namespace.

```
binds /test/empDatabase -url jdbc:oracle:thin:@empHost:5521:ORCL
        -dstype jta -dblink 2pcToEmp.oracle.com
```

4. Provide the two-phase commit engine's fully-qualified database address, username, and password when binding the `UserTransaction` into the namespace.

```
bindut /test/myUT -url sess_iiop://dbsun.mycompany.com:2481:ORCL
                -user SCOTT -password TIGER
```

> **Note:** Verify that the user bound with the `UserTransaction` has FORCE ANY TRANSACTION granted to the user that bound this object. This privilege enables the user to commit this transaction. In this example, you would execute the following:
>
> ```
> GRANT FORCE ANY TRANSACTION TO SCOTT
> ```

5. Update the Oracle-specific deployment descriptor with the JNDI name for the two-phase commit engine. This name should be included in the `<transaction-manager>` element. This element only needs to be defined in the bean where the transaction is started. See "Defining Two Phase Commit Engine for Transactions" on page A-26 for more information.

Once all of this configuration is complete, your application differs from the single-phase commit scenario in the following aspects:

- If you are demarcating your transaction from the client, you can chose to not supply the username and password in the `UserTransaction` binding; but instead, provide the username/password when retrieving the `UserTransaction` object within the `Hashtable` in the `InitialContext` used when looking up the `UserTransaction`.

- You can only retrieve database connections through `DataSource.getConnection`. Since this object is bound with the database

link, which the two-phase commit engine needs to manage the changes within the database, you can only use the DataSource to enlist database resources.

Figure 7–4 shows the client invoking EmployeeBean, which opens a connection to its local database and a connection to a remote database.

Figure 7–4   Including Remote Oracle8i Databases in a Global Transaction
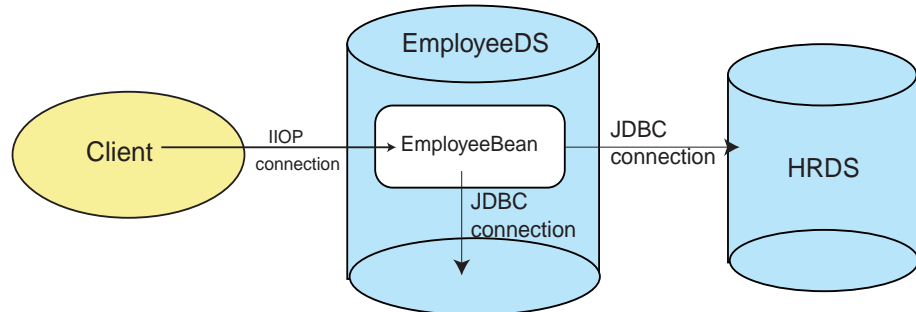
Example 7–8   Client Code

The only difference between this client and the client in Example 7–1 is that this client does not start the transaction. When the client invokes the EmployeeBean, the container starts the transaction.

```
package client;
import common.*;
import java.util.Hashtable;
import javax.naming.Context;
import javax.naming.InitialContext;
import oracle.aurora.jndi.sess_iiop.ServiceCtx;

public class Client
{
  public static void main (String[] args) throws Exception
  {
    if (args.length != 6)
    {
      System.out.println ("usage: Client serviceURL jdbcURL objectName " +
              "user password");
      System.exit (1);
    }
    String serviceURL = args [0];
    String jdbcURL = args [1];
```

```
        String objectName = args [2];
        String user = args [3];
        String password = args [4];

        // set up the initial context
        Hashtable env = new Hashtable ();
        env.put (Context.URL_PKG_PREFIXES, "oracle.aurora.jndi");
        env.put (Context.SECURITY_PRINCIPAL, user);
        env.put (Context.SECURITY_CREDENTIALS, password);
        env.put (Context.SECURITY_AUTHENTICATION, ServiceCtx.NON_SSL_LOGIN);
        Context ic = new InitialContext (env);

        // lookup the home and remote interfaces fro the employee
        EmployeeHome home = (EmployeeHome)ic.lookup (serviceURL + objectName);
        EmployeeRemote remote = home.create ();
        Employee employee = null;

        // retrieve info abount this employee in this session
        employee = remote.getEmployeeForUpdate ("SCOTT");
        System.out.println ("Beginning salary for " + employee.name + " is " +
                employee.salary);

        // increase salary
        // employee.salary += 0.1 * employee.salary;
        employee.salary += 100;

        // update the infomation in the transaction
        remote.updateEmployee (employee);

        // Get and print the info in the transaction
        employee = remote.getEmployee ("SCOTT");
        System.out.println ("End salary for " + employee.name + " is " +
                employee.salary);
    }
}
```

Example 7–9   EmployeeBean EJB Deployment Descriptor

The `EmployeeBean` is specified as a container-managed transaction bean with the RequiresNew attribute. Also, the JDBC 2.0 and pre-2.0 objects are defined as environment variables.

```
<?xml version="1.0"?>
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems Inc.//DTD Enterprise JavaBeans 1.1
//EN" "ejb-jar.dtd">
```

```
<ejb-jar>
   <enterprise-beans>
      <session>
         <ejb-name>test/EmployeeBean</ejb-name>
         <home>common.EmployeeHome</home>
         <remote>common.EmployeeRemote</remote>
         <ejb-class>server.EmployeeBean</ejb-class>
         <session-type>Stateful</session-type>
         <transaction-type>Container</transaction-type>
         <env-entry>
            <env-entry-name>EmployeeBean.KPRB_URL</env-entry-name>
            <env-entry-type>java.lang.String</env-entry-type>
            <env-entry-value>jdbc:oracle:kprb:</env-entry-value>
         </env-entry>
         <env-entry>
            <env-entry-name>EmployeeBean.JDBCDriverName</env-entry-name>
            <env-entry-type>java.lang.String</env-entry-type>
            <env-entry-value>oracle.jdbc.driver.OracleDriver</env-entry-value>
         </env-entry>
         <env-entry>
            <env-entry-name>EmployeeBean.JDBC_URL</env-entry-name>
            <env-entry-type>java.lang.String</env-entry-type>
            <env-entry-value>
               JDBC_URL=jdbc:oracle:thin:@localhost:5521:jis1
            </env-entry-value>
         </env-entry>
         <env-entry>
            <env-entry-name>EmployeeBean.DB_LINK</env-entry-name>
            <env-entry-type>java.lang.String</env-entry-type>
            <env-entry-value>
               LOOP2.REGRESS.RDBMS.DEV.US.ORACLE.COM
            </env-entry-value>
         </env-entry>
         <resource-ref>
            <res-ref-name>jdbc/EmployeeDS</res-ref-name>
            <res-type>javax.sql.DataSource</res-type>
            <res-auth>Application</res-auth>
         </resource-ref>
         <resource-ref>
            <res-ref-name>jdbc/HRDS</res-ref-name>
            <res-type>javax.sql.DataSource</res-type>
            <res-auth>Application</res-auth>
         </resource-ref>
      </session>
   </enterprise-beans>
```

```
                    <assembly-descriptor>
                       <security-role>
                          <description>no description</description>
                          <role-name>PUBLIC</role-name>
                       </security-role>
                       <method-permission>
                          <description>no description</description>
                          <role-name>PUBLIC</role-name>
                          <method>
                             <ejb-name>test/EmployeeBean</ejb-name>
                             <method-name>*</method-name>
                          </method>
                       </method-permission>
                       <container-transaction>
                          <description>no description</description>
                          <method>
                             <ejb-name>test/EmployeeBean</ejb-name>
                             <method-name>*</method-name>
                          </method>
                          <trans-attribute>RequiresNew</trans-attribute>
                       </container-transaction>
                    </assembly-descriptor>
</ejb-jar>
```

Example 7–10   EmployeeBean Oracle-Specific Deployment Descriptor

The environment variables for the JDBC objects are mapped to the JNDI bound names. The `UserTransaction` that is bound with the two-phase commit engine URL is specified in the `<transaction-manager>` element.

```
<?xml version="1.0"?>
<!DOCTYPE oracle-descriptor PUBLIC "-//Sun Microsystems Inc.//DTD Enterprise Jav
aBeans 1.1//EN" "oracle-ejb-jar.dtd">
<oracle-descriptor>
  <mappings>
    <ejb-mapping>
      <ejb-name>EmployeeBean</ejb-name>
      <jndi-name>test/EmployeeBean</jndi-name>
    </ejb-mapping>
    <security-role-mapping>
       <security-role>
         <description>just a role</description>
         <role-name>SECURITY_CLERK</role-name>
       </security-role>
       <oracle-role>CLERK</oracle-role>
```

```
      </security-role-mapping>
      <resource-ref-mapping>
        <res-ref-name>jdbc/EmployeeDS</res-ref-name>
        <jndi-name>test/DataSource/empDS</jndi-name>
      </resource-ref-mapping>
      <resource-ref-mapping>
        <res-ref-name>jdbc/HRDS</res-ref-name>
        <jndi-name>test/DataSource/hrDS</jndi-name>
      </resource-ref-mapping>
      <transaction-manager>
        <jndi-name>test/UserTransaction/testut</jndi-name>
      </transaction-manager>
    </mappings>
</oracle-descriptor>
```

Example 7–11   EmployeeBean Using Bean-Managed Transactions

The container-managed transactional `EmployeeBean` retrieves connections to both the local and a remote Oracle8*i* database.

```
package server;

import common.*;
import java.sql.*;
import java.util.Hashtable;
import java.rmi.RemoteException;
import javax.sql.*;
import javax.naming.*;
import javax.ejb.*;
import oracle.aurora.jndi.sess_iiop.*;
import sqlj.runtime.ref.DefaultContext;
import javax.transaction.*;

import oracle.aurora.transaction.xa.OracleJTADataSource;

public class EmployeeBean implements SessionBean
{
  SessionContext ctx;
  Context ic = null; // inSession Lookup Context
  Connection localConn = null;
  Connection remoteConn = null;
  DefaultContext remoteCtx = null;
  Employee remoteEmployee = null;
  String remoteEmpName = "SMITH";
```

```
          private void setupDSConnections ()
               throws SQLError, RemoteException
          {
            try {
              if (ic == null)
                ic = new InitialContext ();

              //retrieve the remote connection
              remoteConn = getRemoteDSConnection ();
              if (remoteConn == null)
                System.out.println ("remote connection is NULL");
              else
                System.out.println ("got remote connection");

              //setup the context for issuing SQLJ against the remote database
              remoteCtx = new DefaultContext (remoteConn);

              //retrieve the local connection
              localConn = getLocalDSConnection ();
              if (localConn == null)
                  System.out.println ("local connection is NULL");
              else
                  System.out.println ("got local connection");
            } catch (NamingException e) {
              e.printStackTrace ();
              throw new SQLError ("setupDSConnection failed:" + e.toString ());
            } catch (SQLException e) {
              e.printStackTrace ();
              throw new SQLError ("setupDSConnection failed:" + e.toString ());
            }
          }

          private Connection getLocalDSConnection ()
               throws SQLException, SQLError
          {
            try {
              System.out.println ("looking up EmployeeDS in JNDI");
              // get a connection to the local DB using an environment variable
              //specified in the deployment descriptor
              DataSource localDS = (DataSource)ic.lookup
                    ("java:comp/env/jdbc/EmployeeDS");
              System.out.println ("getLocalDSConnection: " +
                      ((OracleJTADataSource)localDS).getURL());

              // get a connectoin to the local DB
```

```
      return localDS.getConnection ();
    } catch (NamingException e) {
      e.printStackTrace ();
      throw new SQLError ("getLocalDSConnection failed:" + e.toString ());
    }
  }

  private Connection getRemoteDSConnection ()
        throws SQLException, SQLError
  {
    try {
      // get a connection to the remopte DB
      System.out.println ("looking up HRDS in JNDI");
      //retrieve the remote database DataSource HRDS using the environment
      //variable specified in the deployment descriptor
      DataSource remoteDS = (DataSource)ic.lookup ("java:comp/env/jdbc/HRDS");
      System.out.println ("getRemoteDSConnection: "+
              ((OracleJTADataSource)remoteDS).getURL());

      // get a connection to the remote DB passing in the usernamd and
      // password for this database. (Otherwise, it would have had to be
      // specified in the Context environment.
      return remoteDS.getConnection ("scott", "tiger");
    } catch (NamingException e) {
      e.printStackTrace ();
      throw new SQLError ("getRemoteDSConnection failed:" + e.toString ());
    }
  }

  public void updateEmployee (Employee employee)
    throws SQLError, RemoteException
  {
    try {
      setupDSConnections ();

      //issue SQL DML statements against the local database
      #sql { update emp set ename = :(employee.name), sal = :(employee.salary)
                    where empno = :(employee.number) };

      remoteEmployee.salary += 200;

      //issue SQL DML statemetns against the remote database
      #sql [remoteCtx] { update emp set ename = :(remoteEmployee.name),
            sal = :(remoteEmployee.salary)
              where empno = :(remoteEmployee.number) };
```

```
          //close both database connections
          localConn.close();
          remoteConn.close ();
      } catch (SQLException e) {
          e.printStackTrace ();
          throw new SQLError ("updateEmployee failed: " + e.toString ());
      }
  }

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

  public void ejbCreate() throws CreateException, RemoteException {}
  public void ejbActivate() {}
  public void ejbPassivate() {}
  public void ejbRemove() {}

}
```

Creating DataSource Objects Dynamically

If you want to bind only a single DataSource object in the namespace to be used for multiple database resources, you must do the following:

1. Bind the DataSource without specifying the URL, host, port, SID, or driver type. Thus, you execute the bindds tool with only the -dstype jta option, as follows:

   ```
   sess_sh -service jdbc:oracle:thin:@nsHost:5521:ORCL -user SCOTT -password
   TIGER
   & bindds /test/empDatabase -dstype jta
   ```

2. Retrieve the DataSource in your code. When you perform the lookup, you must cast the returned object to OracleJTADataSource instead of DataSource. The Oracle-specific version of the DataSource class contains methods to set the DataSource properties.

3. Set the following properties:

   ■ URL with the OracleJTADataSource.setURL method

- Host, port, SID, and driver type if you did not set the URL with the following `OracleJTADataSource` methods: `setURL`, `setDatabaseName`, `setPortNumber`, and `setDriverType`

- Database link if using two-phase commit engine with the `OracleJTADataSource.setDBLink` method

- Username and password if need to provide authentication information for a two-phase commit engine. This information could have been provided on the initial context environment or can be provided in the `getConnection` method. However, if you want to set it with the `OracleJTADataSource` methods, you can through the `setUser` and `setPassword` methods.

4. Retrieve the connection through the `OracleJTADataSource.getConnection` method as indicated in the other examples.

Example 7–12   Retrieving Generic DataSource

The following example retrieves a generically bound `DataSource` from the namespace using in-session lookup and initializes all relevant fields.

```
//retrieve an in-session generic DataSource object
OracleJTADataSource ds = (OracleJTADataSource)ic.lookup ("/test/genericDS");

//set all relevant properties for my database
//URL is for a local database so use the KPRB URL
ds.setURL ("jdbc:oracle:kprb:");
//Used in two-phase commit, so provide the fully qualified database link that
//was created from the two-phase commit engine to this database
ds.setDBLink("localDB.oracle.com");

//Finally, retrieve a connection to the local database using the DataSource
Connection conn = ds.getConnection ();
```

Setting the Transaction Timeout

A global transaction automatically has an idle timeout of 60 seconds. If the object attached to the transaction is idle for over the timeout limit, the transaction is rolled back. To initialize a different timeout, set the timeout value—in seconds—through the `setTransactionTimeout` method before the transaction is begun. If you change the timeout value after the transaction begins, it will not affect the current transaction. The following example sets the timeout to 2 minutes (120 seconds) before the transaction begins.

```
//create the initial context
InitialContext ic = new InitialContext ( );

//retrieve the UserTransaction object
ut = (UserTransaction)ic.lookup ("/test/myUT");

//set the timeout value to 2 minutes
ut.setTransactionTimeout (120);

//begin the transaction
ut.begin

//Update employee table with new employees
updateEmployees(emp, newEmp);

//end the transaction.
ut.commit ();
```

Using the Session Synchronization Interface

An EJB that is a session bean can optionally implement the session synchronization interface, to be notified by the container of the transactional state of the bean. The following methods are specified in the `javax.ejb.SessionSynchronization` interface:

afterBegin

```
public abstract void afterBegin() throws RemoteException
```

The `afterBegin()` method notifies a session Bean instance that a new transaction has started, and that the subsequent methods on the instance are invoked in the context of the transaction.

A bean can use this method to read data from a database and cache the data in the bean's fields.

This method executes in the proper transaction context.

beforeCompletion

```
public abstract void beforeCompletion() throws RemoteException
```

The container calls the `beforeCompletion()` method to notify a session bean that a transaction is about to be committed. You can implement this method to, for example, write any cached data to the database.

afterCompletion

```
public abstract void afterCompletion(boolean committed) throws RemoteException
```

The container calls afterCompletion() to notify a session bean that a transaction commit protocol has completed. The parameter tells the bean whether the transaction has been committed or rolled back.

This method executes with no transaction context.

Example 7–13   SessionSynch Example

In order for the container to invoke your bean implementation before and after every transaction, your bean must implement the SessionSynch interface.

```
package employeeServer;

import employee.*;

import javax.ejb.SessionBean;
import javax.ejb.CreateException;
import javax.ejb.SessionContext;
import java.rmi.RemoteException;

import java.sql.SQLException;

public class EmployeeBean implements SessionBean implements SessionSynch
{
  // Methods of the Employee interface
  public EmployeeInfo getEmployee (String name)
      throws RemoteException, SQLException
  {
    int empno = 0;
    double salary = 0.0;
    #sql { select empno, sal into :empno, :salary from emp
                where ename = :name };

    return new EmployeeInfo (name, empno, salary);
  }

  public void updateEmployee (EmployeeInfo employee)
      throws RemoteException, SQLException
  {
    #sql { update emp set ename = :(employee.name),
                sal = :(employee.salary)
                where empno = :(employee.number) };
```

```
      return;
  }

  // Methods of the SessionBean
  public void ejbCreate () throws RemoteException, CreateException {}
  public void ejbRemove () {}
  public void setSessionContext (SessionContext ctx) {}
  public void ejbActivate () {}
  public void ejbPassivate () {}

  public void beforeBegin()
  {
    ... perform work ...
  }
  public void afterCompletion()
  {
    ... perform work ...
  }
}
```

JDBC Restrictions

If you are using JDBC calls in your bean to update a database, and you have an
active transaction context, you should *not* also use JDBC to perform transaction
services, by calling methods on the JDBC connection. Do *not* code JDBC transaction
management methods. For example:

```
Connection conn = ...
...
conn.commit();  // DO NOT DO THIS!!
```

Doing so will cause a SQLException to be thrown. Instead, you must commit
using the UserTransaction object retrieved to handle the global transaction.
When you commit using the JDBC connection, you are instructing a local
transaction to commit, not the global transaction. When the connection is involved
in a global transaction, trying to commit a local transaction within the global
transaction causes an error to occur.

In the same manner, you must also avoid doing direct SQL commits or rollbacks
through JDBC. Code the bean to either handle transactions directly using the
UserTransaction interface or let the bean container manage the
bean transactions.

A

XML Deployment Descriptors

To deploy your Enterprise JavaBean component into the database, you must provide the appropriate deployment descriptors. There are two possible deployment descriptors. Both deployment descriptors are written using XML notation.

> **Note:** All values entered in either deployment descriptor are case sensitive. This includes usernames and passwords.

The following table describes both deployment descriptors:

Deployment Descriptor	Required/ Optional	Description
Enterprise JavaBean Deployment Descriptor	Required	This deployment descriptor is defined by Sun Microsystems's Enterprise JavaBeans 1.1 specification. It contains information on the names and features for the bean.

Oracle-Specific Deployment Descriptor	Optional	The Oracle-specific deployment descriptor contains information specific to beans deployed in an Oracle8*i* database. It is required only if one of the following is true:
		■ The names provided within the XML deployment descriptor can be logical names and not the actual names.
		■ If you need to specify the `<run-as>` element. This was an EJB 1.0 feature that is supported within the Oracle-specific deployment descriptor.
		■ If you use container-managed persistence for your bean, the persistence manager and container managed fields are defined within the Oracle-specific deployment descriptor.

This appendix briefly describes the sections within both of the deployment descriptors. For more information on the Sun Microsystems's XML deployment descriptor, see the "Deployment Descriptor" chapter in the Enterprise Javabeans 1.1 specification located at `http://www.javasoft.com`. For more information on the Oracle-specific deployment descriptor, see "DTD for Oracle-Specific Deployment Descriptor" on page A-33.

Enterprise JavaBean Deployment Descriptor

This is the main deployment descriptor that contains most of the information about the bean: the bean identification, security roles, transaction demarcation, and any optional environment definition.

- Header
- JAR file
- Enterprise JavaBeans Descriptor
- Application Assembler Section
- EJB Client JAR Section

Header

The following is the required header for all Oracle8*i* EJB deployment descriptors. It details the XML version and the XML DTD file, which details the syntax required for the descriptor.

XML Version Number

```
<?xml version="1.0"?>
```

DTD Filename

```
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems Inc.//DTD Enterprise JavaBeans 1.1
//EN" "ejb-jar.dtd">
```

JAR file

The first element to be declared is the `<ejb-jar>` element. Within this element, you define two sections: the `<enterprise-beans>` section and the `<assembly-descriptor>` section.

Deployment Section	Description
`<enterprise-beans>`	This section defines each bean and each bean's environment.
`<assembly-descriptor>`	This section defines security roles and transactional attributes for the beans. If you decide to use the defaults, this section is optional.

The overall structure for the EJB deployment descriptor follows this syntax:

```
<ejb-jar>                              //Start of JAR file descriptor
 <description> </description>          //Description of JAR file
 <enterprise-beans>                    //EJB Descriptor section
  . . .
 </enterprise-beans>
 <assembly-descriptor>                 //Application Descriptor section
  . . .
 </assembly-descriptor>
 <ejb-client-jar>                      //specify output JAR file for
 ...                                   //client stubs
 </ejb-client-jar>
</ejb-jar>
```

Enterprise JavaBeans Descriptor

The beans are described within the `<enterprise-beans>` section. This section contains information such as the type of bean—entity or session, the home interface name, the remote interface name, the bean class name, and the type of persistence—container-managed or bean-managed. The following shows the elements contained within the `<enterprise-beans>` section.

```
<enterprise-beans>                            //beginning of the EJB descriptor
 <entity> or <session>                        //define EJB type: entity or session
  <description> </description>                //text display description
  <ejb-name> </ejb-name>                      //logical name for the bean
  <home> </home>                              //home interface name
  <remote> </remote>                          //remote interface name
  <ejb-class> </ejb-class>                    //bean class name
  <persistence-type> </persistence-type>      //For entity beans: container or
                                              //bean-managed?
  <prim-key-class> </prim-key-class>          //For entity beans: primary key class
  <primkey-field> </prim-key-field>          //For entity beans: primary key field
  <reentrant> </reentrant>                    //Reentrant boolean: True or False
  <cmp-field> </cmp-field>                    //Container-managed
                                              //fields for entity beans
  <transaction-type> </transaction-type>      //transaction information for bean
  <env-entry> </env-entry>                    //bean environment definition
  <ejb-ref> </ejb-ref>                        //EJB environment definition
  <resource-ref> </resource-ref>              //database resource environment
  <security-role-ref> </security-role-ref>   //security role for bean
 </session> or </entity>
</enterprise-beans>
```

Type of Bean
The first item you must define is whether the bean is a session or an entity bean. You do this with the `<entity>` or `<session>` element.

Bean Names
There are four names necessary to define the bean within the descriptor:

- The home interface is defined within the `<home>` element.

- The remote interface is defined within the `<remote>` element.

- The bean class is defined within the `<ejb-class>` element.

- The logical name within the JAR file is defined within the `<ejb-name>` element. You can do one of two things with this element. You can declare the

actual JNDI name within this element or you can define a logical name that identifies your bean within the deployment descriptor. Ultimately, the <ejb-name> must resolve to the JNDI bound name for the bean. So, if you use a logical name, this name must be mapped to the JNDI name within the Oracle-specific deployment descriptor.

Example A–1   Purchase Order Bean Descriptor

The following defines the purchase order bean as an entity bean with the following components:

- home interface—purchase.PurchaseOrderHome

- remote interface—purchase.PurchaseOrder

- bean implementation—purchaseServer.PurchaseOrderBean

- logical name for the bean within the descriptor—/test/purchase

> **Note:**   This example used the JNDI name, "/test/purchase", within the <ejb-name> element. If a logical name had been used, such as PurchaseOrder, then PurchaseOrder would have to be mapped to "/test/purchase" within the <mappings> element in the Oracle-specific deployment descriptor. See "Defining Mappings" on page A-23 for more information.

```
<enterprise-beans>
   <entity>
      <description>no description</description>
      <ejb-name>test/purchase</ejb-name>
      <home>purchase.PurchaseOrderHome</home>
      <remote>purchase.PurchaseOrder</remote>
      <ejb-class>purchaseServer.PurchaseOrderBean</ejb-class>
      <persistence-type>Container</persistence-type>
      <prim-key-class>java.lang.String</prim-key-class>
      <reentrant>False</reentrant>
   </entity>
</enterprise-beans>
```

Entity Bean Elements

Certain elements define items that pertain only to entity beans. These are as follows:

- Define the bean as container-managed or bean-managed through the `<persistence-type>` element. Value can be either "`Container`" or "`Bean`".

- Define the data fields for a container-managed persistent (CMP) entity bean with the `<cmp-field>` element. Each data field is listed within its own `<cmp-field>` `<field-name>` section.

- Define the primary key for the entity bean. Note that this is only required for CMP beans.

  The primary key can be declared as a single field of a Java type that is consistent with SQL types, such as `java.lang.String` or declared as a combination of several container-managed fields. The one restriction is that you cannot define the primary key to be a byte array that is mapped to a Long Raw column in the database.

Primary Key Declaration	Methodology Required
Java data type consistent with SQL types	**1.** You declare the data type for the primary key within the `<prim-key-class>` element. **2.** Define the field that will become the primary key as CMP within a `<cmp-field>` element. **3.** Designate the CMP field by defining it within the `<primkey-field>` element.
Combination of container-managed persistent fields for primary key	**1.** Define all fields within the primary key as CMP within the `<cmp-field>` element. **2.** Define a serializable class of the name *<bean_name>*PK that contains the names of the `<cmp-field>` elements that are included in the primary key. **3.** List the primary key class within the `<prim-key-class>` element.

Example A–2   CMP Entity Bean

In this example, the customer bean is a CMP entity bean with a customer number as its primary key and two persistent fields: customer name and address.

- The bean is defined as container-managed persistent within the `<persistence-type>` element.

- The primary key is a customer id, `custid`, which is declared as `java.lang.String`.

  1. The `custid` is defined in a `<cmp-field>`.

2. Map the `custid` to the primary key within `<primkey-field>`

3. Declare the `custid`'s data type within the `<prim-key-class>` element.

- The other container-managed persistent fields are defined within the `<cmp-field>` elements as `name` and `addr`.

Note: The method for translating the container-managed fields into actual persistent storage is dependent on the persistent manager that you choose. See "Defining Container-Managed Persistence" on page A-28 for more information.

```
<enterprise-beans>
  <entity>
      <description>customer bean</description>
      <ejb-name>/test/customer</ejb-name>
      <home>customer.CustomerHome</home>
      <remote>customer.Customer</remote>
      <ejb-class>customerServer.CustomerBean</ejb-class>
      <persistence-type>Container</persistence-type>
      <prim-key-class>java.lang.String</prim-key-class>
      <primkey-field>custid</primkey-field>
      <reentrant>False</reentrant>
      <cmp-field><field-name>custid</field-name></cmp-field>
      <cmp-field><field-name>name</field-name></cmp-field>
      <cmp-field><field-name>addr</field-name></cmp-field>
  </entity>
</enterprise-beans>
```

Example A–3   CMP Entity Bean with Complex Primary Key

In this example, the customer bean is a CMP entity bean with a complex primary key defined within its own serializable class. In addition, the CMP bean declares two persistent fields: customer name and address.

- The bean is defined as container-managed persistent within the `<persistence-type>` element.

- The primary key is defined within the `customer.CustomerPK` class.

    1. The elements of the primary key are a customer last name and date of birth: `custname` and `dobirth`. Both fields are defined in a `<cmp-field>`.

    2. Declare the custid's data type within the `<prim-key-class>` element.

- The other container-managed persistent fields are defined within the
  `<cmp-field>` elements as `name` and `addr`.

> **Note:** The method for translating the container-managed fields
> into actual persistent storage is dependent on the persistent
> manager that you choose. See "Defining Container-Managed
> Persistence" on page A-28 for more information.

```
<enterprise-beans>
  <entity>
    <description>customer bean</description>
    <ejb-name>/test/customer</ejb-name>
    <home>customer.CustomerHome</home>
    <remote>customer.Customer</remote>
    <ejb-class>customerServer.CustomerBean</ejb-class>
    <persistence-type>Container</persistence-type>
    <prim-key-class>customer.CustomerPK</prim-key-class>
    <reentrant>False</reentrant>
    <cmp-field><field-name>custname</field-name></cmp-field>
    <cmp-field><field-name>dobirth</field-name></cmp-field>
    <cmp-field><field-name>name</field-name></cmp-field>
    <cmp-field><field-name>addr</field-name></cmp-field>
  </entity>
</enterprise-beans>
```

Environment Elements

You can create three types of environment elements that are accessible to your bean
during runtime: environment variables, EJB references, and resource managers
(JDBC `DataSource`). These environment elements are static and can not be
changed by the bean.

ISVs typically develop EJBs that are independent from the EJB container. In order to
distance the bean implementation from the container specifics, you can create
environment elements that map to one of the following: defined variables, entity
beans, or resource managers. This indirection enables the bean developer to refer to
existing variables, EJBs, and a JDBC `DataSource` without specifying the actual
name. These names are defined in the deployment descriptor and are linked to the
actual names within the Oracle-specific deployment descriptor.

Environment variables you can create environment variables that your bean can access
through a lookup on the `InitialContext`. These variables are defined within an

`<env-entry>` element and can be of the following types: `String`, `Integer`, `Boolean`, `Double`, `Byte`, `Short`, `Long`, and `Float`. The name of the environment variable is defined within `<env-entry-name>`, the type is defined in `<env-entry-type>`, and its initialized value is defined in `<env-entry-value>`. The `<env-entry-name>` is relative to the `"java:comp/env"` context.

For example, the following two environment variables are declared within the XML deployment descriptor for `java:comp/env/minBalance` and `java:comp/env/maxCreditBalance`.

```
<env-entry>
    <env-entry-name>minBalance</env-entry-name>
    <env-entry-type>java.lang.Integer</env-entry-type>
    <env-entry-value>500</env-entry-value>
   </env-entry>
<env-entry>
    <env-entry-name>maxCreditBalance</env-entry-name>
    <env-entry-type>java.lang.Integer</env-entry-type>
    <env-entry-value>10000</env-entry-value>
</env-entry>
```

Within the bean's code, you would access these environment variables through the `InitialContext`, as follows:

```
InitialContext ic = new InitialContext();
Integer min = (Integer)ic.lookup("java:comp/env/minBalance");
Integer max = (Integer) ic.lookup("java:comp/env/maxCreditBalance"));
```

Notice that to retrieve the values of the environment variables, you prefix each environment element with `"java:comp/env/"`, which is the location that the container stored the environment variable.

Environment References To Other Enterprise JavaBeans You can define a reference to an EJB within the deployment descriptor. If you know that your bean will call out to another bean, you can define this reference within the deployment descriptors. The `deployejb` tool binds the EJB reference to the bean's home interface, which enables the target bean to be retrieved without the originating bean needing to know the exact JNDI name for the target. Either way, you use JNDI to retrieve the target bean's home. Once retrieved, the bean referenced by the returned EJB reference is instantiated in the same session.

Declaring the target bean as an environment EJB reference provides a level of indirection, so that the originating bean can refer to the target bean with a logical

name. This is useful when the target bean's JNDI name within the name space may be different depending on the operating environment.

To define an EJB within the environment, you provide the following:

1. Name—provide a logical name for the target bean. This name is what the bean uses within the JNDI URL for accessing the target bean. The name should begin with "ejb/", such as "ejb/myEmployee", and will be available within the "java:comp/env/ejb" context.

2. Type—define whether the bean is a session or an entity bean. Value should be either "Session" or "Entity".

3. Home—provide the fully qualified home interface name.

4. Remote—provide the fully qualified remote interface name.

5. Link —provide a name that links this EJB reference with the actual JNDI URL.

   As shown in Figure A–1, the logical name for the bean is mapped to the JNDI name by providing the same link name, "HelloWorldBean", in both the <ejb-link> in the EJB deployment descriptor and the <ejb-name> within the <ejb-mapping> element in the Oracle-specific deployment descriptor.

Note: Using the <ejb-link> field to map a logical name to a JNDI URL defined within the Oracle-specific deployment descriptor is a different implementation than stated within the Enterprise JavaBeans 1.1 specification.

Figure A–1   EJB Reference Mapping
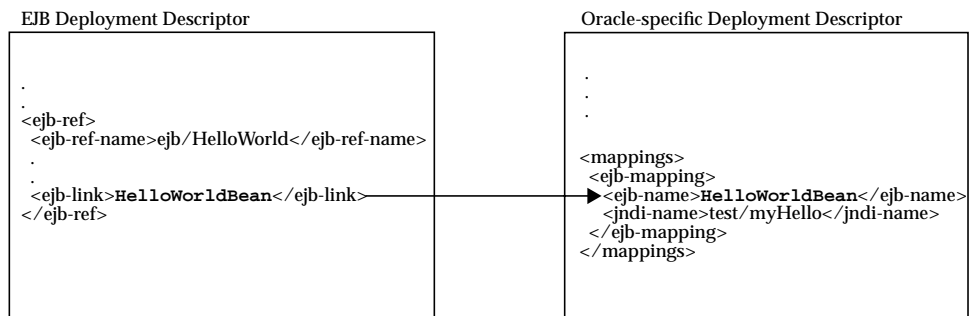
Example A–4   Defining an EJB Reference Within the Environment

The following example defines a reference to the `Hello` bean, as follows:

1. The logical name used for the target bean within the originating bean is `"java:comp/env/ejb/HelloWorld"`.

2. The target bean is a session bean.

3. Its home interface is `hello.HelloHome`; its remote interface is `hello.Hello`.

4. The link to the JNDI URL for this bean is defined in the Oracle-specific deployment descriptor under the `"HelloWorldBean"` name.

EJB
Deployment
Descriptor

```
<ejb-ref>
  <description>Hello World Bean</description>
  <ejb-ref-name>ejb/HelloWorld</ejb-ref-name>
  <ejb-ref-type>Session</ejb-ref-type>
  <home>hello.HelloHome</home>
  <remote>hello.Hello</remote>
  <ejb-link>HelloWorldBean</ejb-link>
</ejb-ref>
```

As shown in Figure A–1, the `<ejb-link>` is mapped to `<ejb-name>` within the `<ejb-mapping>` element in the Oracle-specific deployment descriptor by providing the same logical name in both elements. The Oracle-specific deployment descriptor would have the following definition to map the logical bean name of `"java:comp/env/ejb/HelloWorld"` to the JNDI URL `"/test/myHello"`.

Oracle-specific
Deployment
Descriptor

```
<mappings>
  <ejb-mapping>
    <ejb-name>HelloWorldBean</ejb-name>
    <jndi-name>/test/myHello</jndi-name>
  </ejb-mapping>
</mappings>
. . .
</ejb-ref>
```

> **Note:** For more information on the Oracle-specific deployment
> descriptor, see "Oracle-Specific Deployment Descriptor" on
> page A-22.

To invoke this bean from within your implementation, you use the
`<ejb-ref-name>` defined in the deployment descriptor. You prefix this name with
`"java:comp/env/ejb/"`, which is where the container places the EJB references
defined in the deployment descriptor.

```
InitialContext ic = new InitialContext();
HelloHome hh = (HelloHome)ic.lookup("java:comp/env/ejb/HelloWorld");
```

Environment References To Resource Manager Connection Factory References   The
resource manager connection factory references can include resource managers such
as JMS, mail, XML, and JDBC DataSource objects. In this release, the only resource
manager connection factory that Oracle8*i* supports is the JDBC `DataSource`.

Similar to the EJB references, you can access a database through JDBC either using
the traditional method or by creating an environment element for a JDBC
`DataSource`.

In order to create an environment element for your JDBC `DataSource`, you must
do the following:

1. Bind your JDBC `DataSource` within the JNDI name space.

2. Create a logical name within the EJB deployment descriptor. This name should
   always start with `"java:comp/env/jdbc"`.

3. Map the logical name within the EJB deployment descriptor to the JNDI name,
   created in step 1, within the Oracle-specific deployment descriptor.

As shown in Figure A–2, the JDBC `DataSource` was bound to the JNDI name
`"test/OrderDataSource"`. The logical name that the bean knows this resource as
is `"jdbc/OrderDB"`. These names are mapped together within the Oracle-specific
deployment descriptor. Thus, within the bean's implementation, the bean can
retrieve the connection to `OrderDataSource` by using the
`"java:comp/env/jdbc/OrderDB"` environment element.

Figure A–2   JDBC Resource Manager Mapping

EJB Deployment Descriptor

Oracle-specific Deployment Descriptor

```
<enterprise-beans>
.
<resource-ref>
 <res-ref-name>jdbc/OrderDB</res-ref-name>
 <res-type>javax.sql.DataSource</res-type>
 <res-auth>Application</res-auth>
</resource-ref>
</enterprise-beans>
```

```
<mappings>
 <resource-ref-mapping>
 ►<res-ref-name>jdbc/OrderDB</res-ref-name>
   <jndi-name>test/OrderDataSource</jndi-name>
 </resource-ref-mapping>
</mappings>
```
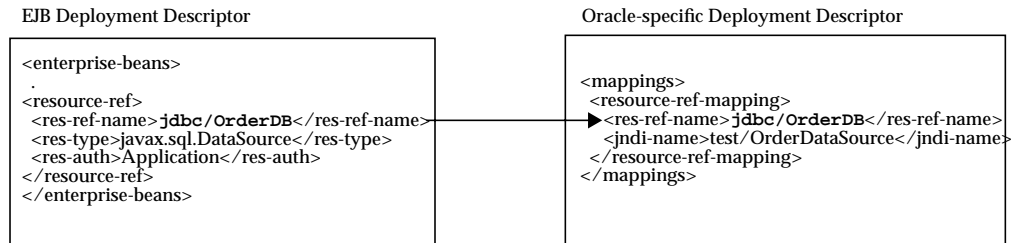
The JDBC `DataSource` environment element is defined with the following information:

Element	Description
`<res-ref-name>`	The logical name of the JDBC DataSource to be used within the originating bean. The name should be prefixed with "`jdbc/`". In our example, the logical name for our ordering database is "`jdbc/OrderDB`".
`<res-type>`	The Java type of the resource. For JDBC, this is `javax.sql.DataSource`.
`<res-auth>`	Define who is responsible for signing on to the database. At this time, the only value supported is "`Application`". The application provides the authentication information for the database by providing the username and password within the DataSource.

Example A–5   Defining an environment element for JDBC Connection

The environment element is defined within the EJB deployment descriptor by providing the logical name, "`jdbc/OrderDB`", its type of `javax.sql.DataSource`, and the authenticator of "`Application`".

EJB
Deployment
Descriptor

```
<resource-ref>
  <res-ref-name>jdbc/OrderDB</res-ref-name>
  <res-type>javax.sql.DataSource</res-type>
  <res-auth>Application</res-auth>
</resource-ref>
```

The environment element of `"jdbc/OrderDB"` is mapped to the JNDI bound name for the connection, `"test/OrderDataSource"` within the Oracle-specific deployment descriptor.

Oracle-specific
Deployment
Descriptor

```
<mappings>
  <resource-ref-mapping>
    <res-ref-name>jdbc/OrderDB</res-ref-name>
    <jndi-name>test/OrderDataSource</jndi-name>
  </resource-ref-mapping>
</mappings
```

Once deployed, your application can retrieve the JDBC DataSource as follows:

```
javax.sql.DataSource db;
java.sql.Connection conn;
.
.
.
db = (javax.sql.DataSource) initCtx.lookup("java:comp/env/jdbc/OrderDB");
conn = db.getConnection();
```

Bean Services

The transaction, security, and reentrancy for the bean is defined by the following elements:

- Define reentrancy for the bean through the `<reentrant>` element. Value should be either `"True"` or `"False"`.

- Define the type of transaction demarcation for a session bean through the `<transaction-type>` element. Value should be `"Bean"` if the session bean uses bean demarcation or `"Container"` if the session bean uses container demarcation for its transaction control.

  Note that the transaction information—for both session and entity beans—is defined in more detail within the `<assembly-descriptor>` section. See "Defining Transactions" on page A-19 for more information.

- Define the security role logical name for the bean with the `<security-role-ref>` element. This refers to a security role name used within the bean and should map to an actual security role defined within the `<assembly-descriptor>` section. See "Defining Security" on page A-15 for more information.

Example A–6   Reentrancy, Transaction Demarcation, and Security Role

The following example defines a customer bean that is not reentrant, uses
bean-demarcated transactions, and uses the "CustRole" logical name within the
bean implementation when referring to its security role. This role name is mapped
to SCOTT.

```
<enterprise-beans>
  <session>
     <ejb-name>CustomerBean</ejb-name>
     <home>customer.CustomerHome</home>
     <remote>customer.Customer</remote>
     <ejb-class>customerServer.CustomerBean</ejb-class>
     <reentrant>False</reentrant>
     <transaction-type>Bean</transaction-type>
     <security-role-ref>
       <role-name>CustRole</role-name>
       <role-link>SCOTT</role-link>
     </security-role-ref>
  </session>
</enterprise-beans>
```

Application Assembler Section

The application assembler adds generic information about all of the beans in this
descriptor in the <assembly-descriptor> section. This section has the following
structure:

```
<assembly-descriptor>
 <security-role> </security-role>
 <method-permission> </method-permission>
 <container-transaction>  </container-transaction>
</assembly-descriptor>
```

These sections describe the security and transaction attributes.

Defining Security

You can manage some of your security for the user and method permissions from
within the deployment descriptor. If you do not specify any method permissions,
the default is that no users are allowed access.

In addition, as shown in Figure A–3, you can use a logical name for a role within
your bean implementation, and map this logical name to the correct database role
or user. The mapping of the logical name to a database role is specified in the

Oracle-specific deployment descriptor. See "Security Role" on page A-25 for more information.

Figure A–3   Security Mapping
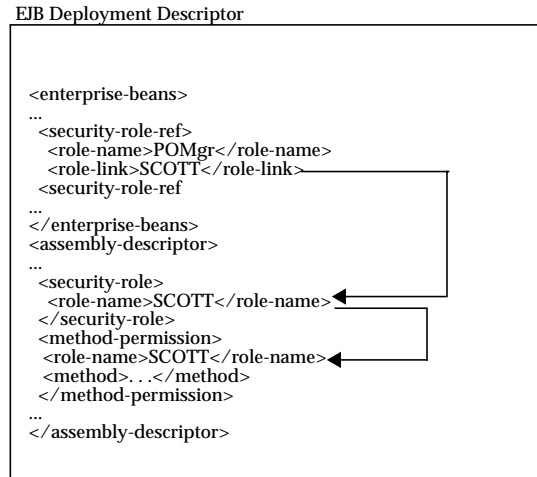
If you use a logical name for a database role within your bean implementation for methods such as isCallerInRole, you can map the logical name to an actual database role by doing the following:

1. Declare the logical name within the `<enterprise-beans>` section `<security-role-ref>` element. For example, to define a role used within the purchase order example, you may have checked, within the bean's implementation, to see if the caller had authorization to sign a purchase order. Thus, the caller would have to be signed in under a correct role. In order for the bean to not need to be aware of database roles, you can check `isCallerInRole` on a logical name, such as POMgr, since only purchase order managers can sign off on the order. Thus, you would define the logical security role, POMgr within the `<security-role-ref><role-name>` element within the `<enterprise-beans>` section, as follows:

```
<enterprise-beans>
...
  <security-role-ref>
   <role-name>POMgr</role-name>
   <role-link>SCOTT</role-link>
  </security-role-ref>
```

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

The `<role-link>` element within the `<security-role-ref>` element can be the actual database role, which is defined further within the `<assembly-descriptor>` section. Alternatively, it can be another logical name, which is still defined more in the `<assembly-descriptor>` section and is mapped to an actual database role within the Oracle-specific deployment descriptor.

2. Define the role and the methods that it applies to. In the purchase order example, any method executed within the `PurchaseOrder` bean must have authorized itself as SCOTT. Note that `PurchaseOrder` is the name declared in the `<entity | session><ejb-name>` element.

   Thus, the following defines the role as SCOTT, the EJB as `PurchaseOrder`, and all methods by denoting the '*' symbol.

   > **Note:** SCOTT is the same as the `<role-link>` element within the `<enterprise-beans>` section. This ties the logical name of `POMgr` to the SCOTT definition.

```
<assembly-descriptor>
 <security-role>
  <description>Role needed purchase order authorization</description>
  <role-name>SCOTT</role-name>
 </security-role>
 <method-permission>
  <role-name>SCOTT</role-name>
  <method>
   <ejb-name>PurchaseOrder</ejb-name>
   <method-name>*</method-name>
  </method>
 </method-permission>
...
</assembly-descriptor>
```

After performing both steps, you can refer to POMgr within the bean's implementation and the container translates POMgr to SCOTT.

The `<method>` element is used to define one or more methods within an interface or implementation. According to the EJB specification, this definition can be of one of three forms:

1. Defining all methods within a bean by specifying the bean name and using the '*' character to denote all methods within the bean, as follows:

```
<method>
    <ejb-name>EJBNAME</ejb-name>
    <method-name>*</method-name>
</method>
```

2. Defining a specific method that is uniquely identified within the bean. Use the appropriate interface name and method name, as follows:

```
<method>
    <ejb-name>myBean</ejb-name>
    <method-name>myMethodInMyBean</method-name>
</method>>
```

> **Note:** If there are multiple methods with the same overloaded name, the element of this style refers to all the methods with the overloaded name.

3. Defining a method with a specific signature among many overloaded versions, as follows:

```
<method>
    <ejb-name>myBean</ejb-name>
    <method-name>myMethod</method-name>
    <method-params>
        <method-param>javax.lang.String</method-param>
        <method-param>javax.lang.String</method-param>
    <method-params>
<method>
```

The parameters are the fully-qualified Java types of the method's input parameters. If the method has no input arguments, the `<method-params>` element contains no elements. Arrays are specified by the array element's type, followed by one or more pair of square brackets, such as int[][].

4. Defining a specific method name that is defined in two or more interfaces. You could have the same method name defined both in the bean implementation and within either the remote or home interface. In this case, you must specify which method you are defining by using the `<method-intf>` element, as follows:

```
<method>
```

```
        <ejb-name>EmployeeService</ejb-name>
        <method-intf>Remote</method-intf>
        <method-name>create</method-name>
        <method-params>
            <method-param>java.lang.String</method-param>
            <method-param>java.lang.String</method-param>
        </method-params>
    </method>
```

Defining the remote interface in the `<method-intf>` element differentiates the `create(String, String)` method defined in the remote interface from the `create(String, String)` method defined in the home interface, which would be defined as follows:

```
<method>
    <ejb-name>EmployeeService</ejb-name>
    <method-intf>Home</method-intf>
    <method-name>create</method-name>
    <method-params>
        <method-param>java.lang.String</method-param>
        <method-param>java.lang.String</method-param>
    </method-params>
</method>
```

> **Note:** The `<method-intf>` element must be defined before the `<method-name>` element.

Defining Transactions

Entity beans can only use container-managed transactions; session beans can use either bean-managed or container-managed transactions. If you have a session bean, you define whether it uses bean or container-managed transactions within the `<enterprise-beans>` section with the `<transaction-type>` element. Values should be either `"Bean"` or `"Container"`. For example, the following defines a session bean as a bean-managed transactional bean:

```
<enterprise-beans>
 <session>
  . . .
  <transaction-type>Bean</transaction-type>
 </session>
</enterprise-beans>
```

For container-managed beans, you define the how the container maintains the transactions for the bean through transaction attributes. The transaction attributes define in what situations to start a new transaction, to continue an existing transaction, and others. If you have a bean-managed bean, you do not define any of these attributes for the bean. However, if your bean-managed bean invokes another bean, the target bean can be either bean or container-managed.

The transaction attributes are specified in the `<trans-attribute>` element and are detailed in Table A–1.

Table A–1   Transaction Attributes

Transaction Attribute	Description
NotSupported	The bean is not involved in a transaction. If the client calls the bean while involved in a transaction, the client's transaction is suspended, the bean executes, and when the bean returns to the client, the client's transaction is resumed.
Required	The bean must be involved in a transaction. If the client is involved in a transaction, the bean uses the client's transaction. If the client is not involved in a transaction, the container starts a new transaction for the bean.
Supports	Whatever transactional state that the client is involved in is used for the bean. If the client has begun a transaction, the client's transaction context is used by the bean. If the client is not involved in a transaction, neither is the bean.
RequiresNew	Whether the client is involved in a transaction or not, this bean requires a new transaction that exists only for itself. If the client calls while involved in a transaction, the client's transaction is suspended until the bean completes.
Mandatory	The client must be involved in a transaction before invoking this bean. The bean uses the client's transaction context.
Never	The bean is not involved in a transaction. Furthermore, the client cannot be involved in a transaction when calling the bean. If the client is involved in a transaction, a RemoteException is thrown.

You can define the transaction attributes on a whole bean or on an individual method within the bean. Each definition is contained within the `<container-transaction>` element. If you are defining an attribute for the entire bean, you supply the bean name (the `<ejb-name>` defined within the

<session> or <entity> element) within the <container-transaction> <method> <ejb-name> element.

The following example defines the following:

- All methods (denoted by the '*' character) within the PurchaseOrder bean must have a transaction (Required).

  > **Note:** The methods can be specified exactly as within the <method-permission> element. See "Defining Security" on page A-15 for more information.

- The price method within the PurchaseOrder bean must always have a new transaction (RequiresNew).

```
<assembly-descriptor>
...
 <container-transaction>
  <method>
    <ejb-name>PurchaseOrder</ejb-name>
    <method-name>*</method-name>
  </method>
  <trans-attribute>Required</trans-attribute>
 </container-transaction>
 <container-transaction>
  <method>
    <ejb-name>PurchaseOrder</ejb-name>
    <method-name>price</method-name>
  </method>
  <trans-attribute>RequiresNew</trans-attribute>
 </container-transaction>
 . . .
</assembly-descriptor>
```

Finally, if you are using two-phase commit for your global transaction, you must provide the UserTransaction object's JNDI name to the <transaction-manager> element within the Oracle-specific deployment descriptor. See "Defining Two Phase Commit Engine for Transactions" on page A-26 for more information.

Oracle-Specific Deployment Descriptor

The Oracle-specific deployment descriptor is used for the following:

- Defining Mappings

- Defining Two Phase Commit Engine for Transactions

- Defining Run-As Identity

- Defining Container-Managed Persistence

This deployment descriptor has the following structure:

```
<?xml version="1.0"?>
<!DOCTYPE oracle-descriptor PUBLIC "-//Oracle Corporation//DTD Enterprise
JavaBeans 1.1//EN" "oracle-ejb-jar.dtd">

<oracle-descriptor>
  <mappings>
    <ejb-mapping> </ejb-mapping>
    <resource-ref-mapping> </resource-ref-mapping>
  </mappings>
  <run-as> </run-as>
  <persistence-provider> </persistence-provider>
  <persistence-descriptor> </persistence-descriptor>
</oracle-descriptor>
```

> **Note:** The entire DTD is listed "DTD for Oracle-Specific
> Deployment Descriptor" on page A-33.

Header

The following is the required header for all Oracle8*i* EJB deployment descriptors. It
details the XML version and the XML DTD file.

XML Version Number

```
<?xml version="1.0"?>
```

DTD Filename

```
<!DOCTYPE oracle-descriptor PUBLIC "-//Oracle Corporation.//DTD Oracle 1.1//EN"
"oracle-ejb-jar.dtd">
```

Defining Mappings

The primary use of the Oracle-specific deployment descriptor is to map logical names used within the EJB deployment descriptor to actual names used within the Oracle8*i* database. The different names that can be mapped are as follows:

EJB Logical Name	Description
Bean name defined in `<session	entity>` `<ejb-name>`	If a logical name was used within the `<ejb-name>` element instead of the JNDI name, this name should be mapped to the actual JNDI name.
EJB reference environment elements	If you provided a logical name for the target bean within `<ejb-ref>`, you must map this to the actual JNDI name for the EJB.
JDBC DataSource environment element	If you defined a JDBC DataSource as an environment element, you must map the `<res-ref-name>` to the actual JNDI name.
Security roles	If you used a logical name within the `<security-role><role-name>` element within the `<assembly-descriptor>`, this logical name must be mapped to the actual user name used within the database.
Transaction Manager	If you are using a two phase commit engine for your global transaction, specify the JNDI name for the OracleJTADataSource object that represents this database.

Bean Name

As described in "Bean Names" on page A-4, the bean name can be either a logical name or a JNDI name. If you defined a logical name, as demonstrated in the following example, you must map this name to the actual JNDI name within the `<mappings>` section.

Example A–7   EJB Name

The following defines a logical name of `PurchaseOrder` for the bean with in the EJB deployment descriptor.

```
<enterprise-beans>
   <entity>
      <description>no description</description>
      <ejb-name>PurchaseOrder</ejb-name>
      <home>purchase.PurchaseOrderHome</home>
      <remote>purchase.PurchaseOrder</remote>
      <ejb-class>purchaseServer.PurchaseOrderBean</ejb-class>
      <persistence-type>Bean</persistence-type>
```

```
        <prim-key-class>java.lang.String</prim-key-class>
        <reentrant>False</reentrant>
     </entity>
  </enterprise-beans>
```

The `PurchaseOrder` logical name is mapped to its JNDI name within the
`<mappings>` section of the Oracle-specific deployment descriptor. The following
example shows how the bean name is defined in the `<ejb-name>` element and the
corresponding JNDI name is defined in the `<jndi-name>` element.

```
<oracle-descriptor>
  <mappings>
    <ejb-mapping>
      <ejb-name>PurchaseOrder</ejb-name>
      <jndi-name>/test/purchase</jndi-name>
    </ejb-mapping>
  </mappings>
. . .
</oracle-descriptor>
```

EJB Reference

The EJB reference mapping is described in "Environment References To Other
Enterprise JavaBeans" on page A-9. The structure of the mapping is the same as the
bean name, demonstrated in "Bean Name" on page A-23.

JDBC DataSource

The JDBC `DataSource` connection mapping is described in "Environment
References To Resource Manager Connection Factory References" on page A-12. You
map the JDBC DataSource to the bound JNDI name with the
`<resource-ref-mapping>` element, as follows:

```
<oracle-descriptor>
  <mappings>
    <resource-ref-mapping>
      <res-ref-name>jdbc/EmployeeAppDB</res-ref-name>
      <jndi-name>jdbc/OracleDataSource</jndi-name>
    </resource-ref-mapping>
  </mappings>
  ...
</oracle-descriptor>
```

The `<res-ref-name>` element contains the JDBC `DataSource` defined in the EJB
deployment descriptor. The `<jndi-name>` element contains the JNDI name bound

to the JDBC DataSource within the name space. This example maps the `"jdbc:comp/env/jdbc/EmployeeAppDB"` environment name to the JNDI name `"jdbc/OracleDataSource"`.

Security Role

If you defined a logical name for the `<role-name>` within the `<assembly-descriptor>`, this logical name must be mapped to an actual database role or user within the Oracle-specific deployment descriptor.

For example, you define a logical role of POMGR for the role necessary for purchase order authorization within the `<assembly-descriptor>`. The actual database role that is allowed to make this authorization is SCOTT. The following would be the necessary mapping definition within the Oracle-specific deployment descriptor:

```
<mappings>
 <security-role-mapping>
   <security-role>
     <description>POMGR role mapping</description>
     <role-name>POMGR</role-name>
   </security-role>
   <oracle-role>SCOTT</oracle-role>
 </security-role-mapping>
</mappings>
```

As shown in Figure A–4, the `<role-name>` of POMGR defined in the `<assembly-descriptor><security-role><role-name>` is mapped to SCOTT within the Oracle-specific deployment descriptor in the `<security-role-mapping>` element.

Figure A–4   Security Mapping
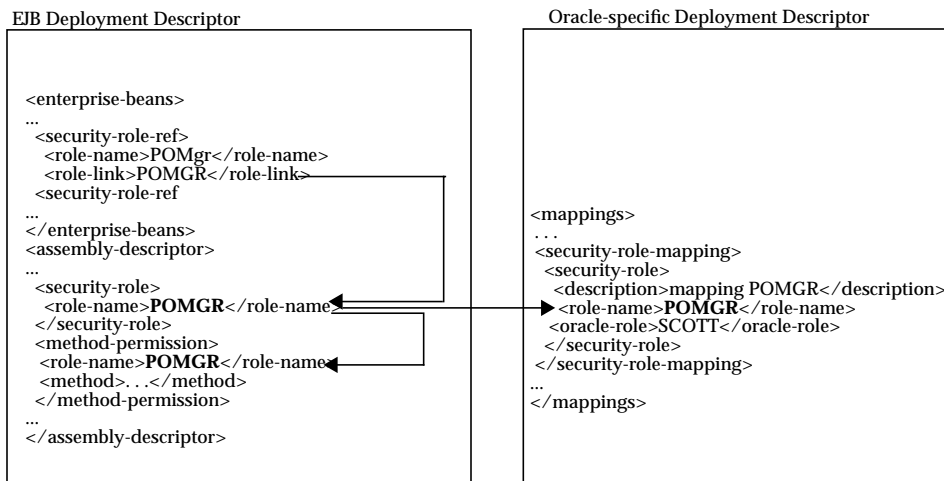
Defining Two Phase Commit Engine for Transactions

If you are using two-phase commit for your global transaction, you must provide the `UserTransaction` object's JNDI name to the `<transaction-manager>` element. The following example specifies the `UserTransaction` object `"/test/myUTFor2pc"`:

```
<mappings>
 ...
 <transaction-manager>
   <jndi-name>/test/myUTFor2pc<jndi-name>
 </transaction-manager>
</mappings>
```

Defining Run-As Identity

The `<run-as>` element is an EJB 1.0 element that Oracle still supports within the Oracle-specific deployment descriptor. You can define that a particular EJB will run under the specified identity with the `<run-as>` element. There are three types of identity that you can specify for your bean. They are as follows:

Run-As Identity	Description
CLIENT_IDENTITY	The bean runs under the client's identity. This is the default.

Run-As Identity	Description
SYSTEM_IDENTITY	The bean runs under the SYSTEM identity. This is specific to the type of environment the bean is running on. For example, on an Oracle8*i* database, the SYSTEM identity is equivalent to SYS.
SPECIFIED_IDENTITY	You can specify a specific identity or role that the bean runs as when invoked. This identity is specified within the `<security-role>` element.

The `<method>` element specifies the methods that the `<run-as>` definition applies to. It takes in the following definitions:

- `<ejb-name>`: The logical name of the bean defined in the XML deployment descriptor.

- `<method-name>`: The method name (* implies all methods) that this mode applies to within the bean.

- `<method-params>`: If you have more than one method of this name with overloaded parameters and you want the `<run-as>` to apply to only one of them, specify the parameters within this element.

For example, the following defines that the `price` method, which requires two String parameters, in the `PurchaseOrder` EJB will always run as SCOTT when invoked.

```
<run-as>
  <description>no description</description>
  <mode>SPECIFIED_IDENTITY</mode>
  <security-role>
    <role-name>SCOTT</role-name>
  </security-role>
  <method>
    <ejb-name>PurchaseOrder</ejb-name>
    <method-name>price</method-name>
    <methodl-params>
      <method-param>java.lang.String<method-param>
      <method-param>java.lang.String<method-param>
  </method>
</run-as>
```

> **Note:** Specify the `<security-role>` element only for the
> SPECIFIED_IDENTITY value. The `<security-role>` element is
> ignored for both CLIENT_IDENTITY and SYSTEM_IDENTITY
> values.

Defining Container-Managed Persistence

If you have chosen to have the container manage the persistent variables for your
entity bean, you need to provide the name of the persistence provider and map each
persistence provider to the container-managed persistent bean. This version
supports the Oracle Persistence Service Interface Reference Implementation (PSI-RI)
persistence manager. Using this manager, you must define the mappings of the
persistent fields to the database.

Persistence Provider The persistence provider controls the management of all
container-managed fields. It defines how to map the fields within the bean to
corresponding persistent storage. It also defines how to save and update these fields
from the bean to the storage.

You define the Oracle PSI-RI persistence provider is defined within the
`<persistence-provider>` element. The following elements describe the
persistence provider:

Persistence Provider Elements	Description
`<description>`	Gives a description of the provider.
`<persistence-name>`	Defines a logical name for the provider. In this release, only Oracle PSI-RI is supported.
`<persistence-deployer>`	Provides the Java class used for the persistence maintenance. This class is called by Oracle's container when persistence is required.

The following defines the Oracle PSI-RI persistence provider. The logical name is
defined as `"PSI-RI"` and the class called for persistence management is
`oracle.aurora.ejb.persistence.ocmp.OcmpEntityDeployer`.

```
<persistence-provider>
  <description> specifies a type of persistence manager </description>
  <persistence-name>psi-ri</persistence-name>
  <persistence-deployer>oracle.aurora.ejb.persistence.ocmp.OcmpEntityDeployer
```

```
    </persistence-deployer>
</persistence-provider>
```

Persistence Fields The persistent fields were defined in the EJB deployment descriptor in the `<cmp-field>` elements. Each of these `<cmp-field>` elements need to be persistently managed by your provider. You define a single provider for all `<cmp-field>` elements within a single bean through the `<persistence-descriptor>` element. You define the provider for the bean through the combination of the logical names for the bean and for the provider within the `<persistence-descriptor>`. These fields are as follows:

Persistence Elements	Description
`<description>`	Provides a text description of the persistence elements.
`<ejb-name>`	The EJB that contains the persistent fields. This would be the same `<ejb-name>` for the entity bean that defined the `<cmp-field>` elements.
`<persistence-name>`	The logical name of the persistence provider defined within the `<persistence-provider>` `<persistence-name>` element.

For example, to define that Oracle PSI-RI manages all persistent fields within the purchase order bean, you specify the logical bean name, `PurchaseOrder`, and the logical persistence provider name, Oracle PSI-RI, within the following fields:

```
<persistence-descriptor>
  <ejb-name>PurchaseOrder</ejb-name>
  <persistence-name>psi-ri</persistence-name>
  . . .
</persistence-descriptor>
```

The Oracle PSI-RI maps the persistent data types to database rows. It uses SQL to update the persistent fields between the bean and the database. Each persistent data type is mapped to the Oracle SQLJ data types documented in the Oracle8i SQLJ Developer's Guide and Reference. The data type mapping is documented fully on Table 5-1, "Type Mappings for Supported Host Expression Types" on page 5-2. The only restrictions is that the following data types are not supported:

Table A–2   Unsupported Java Types for Persistent Variables

	Java Type	Oracle Types Definition	Oracle Datatype
SQLJ stream classes	▪ sqlj.runtime.BinaryStream	▪ LONGVARBINARY	▪ LONG RAW
	▪ sqlj.runtime.AsciiStream	▪ LONGVARCHAR	▪ LONG
	▪ sqlj.runtime.UnicodeStream	▪ LONGVARCHAR	▪ LONG
Oracle extensions	▪ oracle.sql.ROWID	▪ ROWID	▪ ROWID
	▪ oracle.sql.BLOB	▪ BLOB	▪ BLOB
	▪ oracle.sql.CLOB	▪ CLOB	▪ CLOB
Query result objects	▪ java.sql.ResultSet	▪ CURSOR	▪ CURSOR
	▪ SQLJ iterator objects	▪ CURSOR	▪ CURSOR

If you decide to use PSI-RI, you must define the following within the `<psi-ri>` element, which exists within the `<persistence-descriptor>`.

Persistence Elements	Description
`<schema>`	Define the schema where the database table resides.
`<table>`	Define the table where the persistent fields are saved.
`<attr-mapping>`	Define each container-managed field and its corresponding table row within this element.
`<field-name>`	Define the container-managed field name as defined within the bean.
`<column-name>`	Define the database row where the `<field-name>` is saved.

> **Note:** Before you can deploy this bean, the table and its appropriate rows must exist within the schema.

For example, the following `<psi-ri>` maps the customer bean persistence fields—primary key, customer name, and address—to the customers table within SCOTT's schema

```
<persistence-descriptor>
  <ejb-name>PurchaseOrder</ejb-name>
  <persistence-name>psi-ri</persistence-name>
```

```
  <psi-ri>
      <schema>SCOTT</schema>
      <table>customers</table>
      <attr-mapping>
        <field-name>custid</field-name>
        <column-name>cust_id</column-name>
      </attr-mapping>
      <attr-mapping>
        <field-name>name</field-name>
        <column-name>cust_name</column-name>
      </attr-mapping>
      <attr-mapping>
        <field-name>addr</field-name>
        <column-name>cust_addr</column-name>
      </attr-mapping>
  </psi-ri>
</persistence-descriptor>
```

If the persistent fields that you define are objects, you can ask Oracle PSI-RI to serialize some or all fields defined as persistent into a single database column. This column must be defined as either long Raw or Raw.

> **Note:** There is a limitation of 4 KB for any Raw columns.

You define the fields and the destination column within the `<serialize-mapping>` element. The persistent fields are mapped either within the `<attr-mapping>` or the `<serialize-mapping>` elements, but not in both.

The following example serializes the customer bean persistent fields—customer name, and address—and save them into the `custinfo` column. Normally, you would do this if you wanted to serialize an object.

> **Note:** There is one restriction: you cannot include the primary key within the `<serialize-mapping>` element. It hinders finding an object through `findByPrimaryKey`.

```
<persistence-descriptor>
  <ejb-name>PurchaseOrder</ejb-name>
  <persistence-name>psi-ri</persistence-name>
  <psi-ri>
   <schema>SCOTT</schema>
```

```
 <table>customers</table>
 <serialize-mapping>
  <field-name>name</field-name>
  <field-name>addr</field-name>
  <column-name>custinfo</column-name>
 </serialize-mapping>
 </psi-ri>
</persistence-descriptor>
```

Example A–8   Container-Managed Persistence

The following is the full example for defining container-managed persistence for the customer example.

```
<persistence-provider>
  <description>use the simple CMP provider</description>
  <persistence-name>psi-ri</persistence-name>
  <persistence-deployer>oracle.aurora.ejb.persistence.ocmp.OcmpEntityDeployer
  </persistence-deployer>
</persistence-provider>

<persistence-descriptor>
  <ejb-name>customerbean</ejb-name>
  <persistence-name>psi-ri</persistence-name>
  <psi-ri>
      <schema>SCOTT</schema>
      <table>customers</table>
      <attr-mapping>
        <field-name>custid</field-name>
        <column-name>cust_id</column-name>
      </attr-mapping>
      <attr-mapping>
        <field-name>name</field-name>
        <column-name>cust_name</column-name>
      </attr-mapping>
      <attr-mapping>
        <field-name>addr</field-name>
        <column-name>cust_addr</column-name>
      </attr-mapping>
  </psi-ri>
</persistence-descriptor>
</oracle-descriptor>
```

EJB Client JAR Section

One of the tasks that the deployejb tool performs is to create a JAR file with the required stubs and skeletons, which is used by the client at execution time. Unless this JAR file name is specified in the <ejb-client-jar> element in the XML deployment descriptor, the default name for this JAR file is server_ generated.jar. This JAR file must be included in the CLASSPATH for client execution.

The following defines that deployejb should create the client's stubs and skeletons in myClient.jar:

```
<ejb-client-jar>myClient.jar</ejb-client-jar>
```

DTD for Oracle-Specific Deployment Descriptor

```
<!--
This is the XML DTD for the Oracle Specific EJB deployment descriptor
-->
<!--
The oracle-descriptor element is the root element of the Oracle-specific
deployment descriptor. It contains an optional description, optional structural
information about logical name mappings, optional definitions for run-as
beans and/or methods, and definitions of container-managed persistence. -->
<!ELEMENT oracle-descriptor (mappings*, run-as*, persistence-provider*,
persistence-descriptor*)>
<!--
The mappings section enables you to map logical names defined in the XML
deployment descriptor to actual names. Bean logical names are mapped to JNDI
names; security logical names are mapped to database roles or users. -->
<!ELEMENT mappings (ejb-mapping*, security-role-mapping*,
resource-ref-mapping*, transaction-manager*)>
<!--
The ejb-mapping element maps an EJB to its bound JNDI name -->
<!ELEMENT ejb-mapping (ejb-name, jndi-name)>
<!--
The security-role-mapping element maps security logical names to a database
role or user -->
<!ELEMENT security-role-mapping (security-role, oracle-role)>
<!--
The resource-ref-mapping element maps any environment variable defined in the
XML deployment descriptor to the JNDI name for the target object -->
<!ELEMENT resource-ref-mapping (res-ref-name, jndi-name)>
<!--
The transaction manager defines the UserTransaction JNDI name that manages
```

the global transaction. This is only required for transactions that use
two-phase commit -->
```
<!ELEMENT transaction-manager (description?, jndi-name)>
<!--
The jndi-name element specifies a JNDI name for a bound object -->
<!ELEMENT jndi-name (#PCDATA)>
<!--
The run-as element enables you to specify a bean or certain methods within
a bean to run with an identity other than its own. The modes allowed are
CLIENT_IDENTITY (default), SYSTEM_IDENTITY, and SPECIFIED_IDENTITY. With the
SPECIFIED_IDENTITY mode, you must provide the identity within the
<security-role> element.
-->
<!ELEMENT run-as (description?, mode, security-role, method)>
<!--
The mode element specifies the type of <run-as> identity. The values
can be one of the following:SYSTEM_IDENTITY, SPECIFIED_IDENTITY, CLIENT_IDENTITY
if mode is SPECIFIED_IDENTITY, security-role must be specified
if mode is SYSTEM_IDENTITY or CLIENT_IDENTITY and security-role is
specified, security-role is ignored
-->
<!ELEMENT mode (#PCDATA)>
<!--
The security-role element specifies a database role -->
<!ELEMENT security-role (description?, role-name)>
<!-- The role-name element specifies a database role or user -->
<!ELEMENT role-name (#PCDATA)>
<!--
The method element defines a method by the bean's logical name, optionally
adding the interface name, the method name, and if overloading is present for
this method, the parameters of the method you are indicating.
-->
<!ELEMENT method (description?, ejb-name, method-intf?, method-name,
method-params?)>
<!--
The ejb-name element defines the logical name for the bean that was used in the
XML deployment descriptor -->
<!ELEMENT ejb-name (#PCDATA)>
<!--The method interface defines where the method is specified-->
<!ELEMENT method-intf (#PCDATA)>
<!--
The method name element takes in the actual name of a method defined. -->
<!ELEMENT method-name (#PCDATA)>
<!--
The method-params element specifies one or more parameters for a method. -->
```

```
<!ELEMENT method-params (method-param*)>
<!--
The method-param defines a single parameter for a method by its class type-->
<!ELEMENT method-param (#PCDATA)>
<!-- The oracle-role element specifies a database role or user -->
<!ELEMENT oracle-role (#PCDATA)>
<!-- The ejb-ref-name is the logical name for the EJB reference specified
in the XML deployment descriptor -->
<!ELEMENT ejb-ref-name (#PCDATA)>
<!-- The res-ref-name element is the logical name for the resource reference
specified in the XML deployment descriptor -->
<!ELEMENT res-ref-name (#PCDATA)>


<!---
persistence-provider describes the container managed persistence
-->
<!--
The persistence-provider element specifies the CMP provider that you are using.
At this time, only Oracle8i's PSI-RI is supported. -->
<!ELEMENT persistence-provider (description?, persistence-name,
persistence-deployer)>
<!ELEMENT description (#PCDATA)>
<!--
The persistence-name element defines the name of the provider. For Oracle8i,
this should be psi-ri -->
<!ELEMENT persistence-name (#PCDATA)>
<!-- The persistence-deployer is the class of the CMP provider. This should be
oracle.aurora.ejb.persistence.ocmp.OcmpEntityDeployer. -->
<!ELEMENT persistence-deployer (#PCDATA)>
<!-- The persistence-descriptor element defines the persistence fields in the
bean that must be managed by the CMP provider -->
<!ELEMENT persistence-descriptor (description?, ejb-name,
persistence-name, persistence-param*, psi-ri*)>
<!ELEMENT persistence-param (#PCDATA)>
<!-- The psi-ri element defines how the persistence fields int he beans are
mapped to database tables and columns. -->
<!ELEMENT psi-ri (schema, table, attr-mapping+, serialize-mapping?)>
<!-- The schema element specifies the schema where the table exists -->
<!ELEMENT schema (#PCDATA)>
<!-- The table element specifies the table where to store the persistent fields
-->
<!ELEMENT table (#PCDATA)>
<!-- The attr-mapping element specifies how each persistent field is mapped to a
corresponding column in the table -->
<!ELEMENT attr-mapping (field-name, column-name)>
```

```
<!-- If you serialize all persistent fields into a single column, use the
serialize-mapping element -->
<!ELEMENT serialize-mapping (field-name+, column-name)>
<!-- The field-name element specifies the persistent variable in the bean -->
<!ELEMENT field-name (#PCDATA)>
<!-- The column-name element specifies the column for a single persistent field
-->
<!ELEMENT column-name (#PCDATA)>
```

B

Example Code: EJB

Oracle8*i* JServer installs several samples under the `demo` directory. Some of these samples are included in this appendix for your perusal.

The examples in the `demo` directory include a UNIX makefile and Windows NT batch file to compile and run each example. You need a Java-enabled Oracle8*i* database with the standard EMP and DEPT demo tables to run the examples.

The emphasis in these short examples is on demonstrating features of the ORB and CORBA, not on elaborate Java coding techniques. Each of the examples includes a README file that tell you what files the example contains, what the example does, and how to compile and run the example.

- Basic Example
- SQLJ Example
- Bean Inheritance Example
- Entity Bean Examples
- Session Example
- SSL Examples

Basic Example

README

```
Overview
========

This is the most basic program that you can create for the Orcale8i
EJB server. One bean, HelloBean, is implemented. The bean and
associated classes are loaded into the database, and the bean home
interface is published as /test/myHello, as specified in the bean
deployment descriptor hello.ejb.

The bean contains a single method: helloWorld, which simply returns a
String containing the JavaVM version number to the client that invokes
it.

This example shows the minimum number of files that you must provide
to implement an EJB application: five. The five are:

(1) the bean implementation:    helloServer/HelloBean.java in this example
(2) the bean remote interface:  hello/Hello.java
(3) the bean home interface:    hello/HelloHome.java
(4) the deployment descriptor:  hello.ejb
(5) a client app or applet:     Client.java is the application in this example


Source Files
============


Client.java
-----------

You invoke the client program from a command prompt, and pass it four
arguments, the

   - service URL (service ID, hostname, port, and SID if port is a listener)
   - name of the published bean to lookup and instantiate
   - username
   - password that authenticates the client to the Oracle8i database server

For example:
```

```
% java -classpath LIBs Client sess_iiop://localhost:2222 /test/myHello scott
tiger
```

where LIBs is the classpath that must include

```
$ORACLE_HOME/lib/aurora_client.jar
#If using Java 2, use classes12.zip instead of classes111.zip
$ORACLE_HOME/jdbc/lib/classes111.zip
$ORACLE_HOME/lib/vbjorb.jar
$ORACLE_HOME/lib/vbjapp.jar
$JAVA_HOME/lib/classes.zip
```

(Note: for NT users, the environment variables would be %ORACLE_HOME% and
%JAVA_HOME%.)

The client code performs the following steps:

- gets the arguments passed on the command line
- creates a new JNDI Context (InitialContext())
- looks up the published bean to find and activate its home interface
- using the home interface, instantiates through its create()
  method a new bean object, hello
- invokes the helloWorld() method on the hello object and prints the results

The printed output is:

Hello client, your javavm version is 8.1.5.

hello.ejb

The bean deployment descriptor. This source file does the following:

- shows the class name of the bean implementation in the deployment name:
    helloServer.HelloBean
- names the published bean "/test/myHello"
- declares the remote interface implementation: hello.Hello
- declares the home interface: hello.HelloHome
- sets RunAsMode to the client's identity (SCOTT in this case)
- allows all members of the group PUBLIC to run the bean
- sets the transaction attribute to TX_SUPPORTS

The deployement descriptor is read by the deployejb tool, which uses
it to load the required classes, and publish the bean home

interface. (Deployejb does much else also. See the Tools chapter in
the Oracle8i EJB and CORBA Developer's Guide for more information.)

helloServer/HelloBean.java

This is the EJB implementation. Note that the bean class is public,
and that it implements the SessionBean interface, as required by the
EJB specification.

The bean implements the one method specified in the remote interface:
helloWorld(). This method gets the system property associated with
"oracle.server.version" as a String, and returns a greeting plus the
version number as a String to the invoking client.

The bean implementation also implements ejbCreate() with no parameters,
following the specification of the create() method in hello/HelloHome.java.

Finally, the methods ejbRemove(), setSessionContext(), ejbActivate(), and
ejbPassivate() are implemented as required by the SessionBean interface. In
this simple case, the methods are implemented with null bodies.

(Note that ejbActivate() and ejbPassivate() are never called in the
8.1.5 release of the EJB server, but they must be implemented as
required by the interface.)

hello/Hello.java

This is the bean remote interface. In this example, it specifies only
one method: helloWorld(), which returns a String object. Note the two
import statements, which are required, and that the helloWorld()
method must be declared as throwing RemoteException. All bean methods
must be capable of throwing this exception. If you omit the
declaration, the deployejb tool will catch it and error when you try
to deploy the bean.

hello/HelloHome.java

This is the bean home interface. In this example, a single create()
method is declared. It returns a Hello object, as you saw in the

Client.java code.

Note especially that the create() method must be declared as able to
throw RemoteException and CreateException. These are required. If you
do not declare these, the deployejb tool will catch it and error when
you try to deploy the bean.

Compiling and Running the Example
=================================

UNIX

Enter the command 'make all' or simply 'make' in the shell to compile,
load, and deploy the objects, and run the client program. Other
targets are 'run' and 'clean'.

Make sure that a shell environment variable ORACLE_HOME is set to
point to the home location of the Oracle installation. This is
operating system dependent, so see the Installation documentation that
came with your system for the location. Also, review the README file
for the Oracle database, and the README file for the CORBA/EJB server
(the Oracle8i ORB), for additional up-to-date information.

Windows NT

On Windows NT, run the batch file makeit.bat from a DOS command prompt
to compile, load, and deploy the objects. Run the batch file runit.bat
to run the client program, and see the results.

Make sure that the environment variables %ORACLE_HOME%, %CLASSPATH%,
and %SERVICE% are set appropriately for the DOS command window. You
can set these as either user or system environment variables from the
Control Panel. Double click on System in the Control Panel then on
the Environment tab to set these variables. Start a new DOS window
after setting environment variable values.

See the Installation documentation that came with your Oracle8i system

for the values of these variables. Also, review the README file for the Oracle database, and the README file for the CORBA/EJB server (the Oracle8i ORB), for additional up-to-date information.

You can also set an environment variable %JAVA_HOME% to point to the root of your Java JDK. For example, SET JAVA_HOME=C:\JDK1.1.6.

Client

```
import hello.Hello;
import hello.HelloHome;

import oracle.aurora.jndi.sess_iiop.ServiceCtx;

import javax.naming.Context;
import javax.naming.InitialContext;
import java.util.Hashtable;

public class Client
{
  public static void main (String[] args) throws Exception {
    if (args.length != 4) {
      System.out.println ("usage: Client serviceURL objectName user password");
      System.exit (1);
    }
    String serviceURL = args [0];
    String objectName = args [1];
    String user = args [2];
    String password = args [3];

    Hashtable env = new Hashtable ();
    env.put (Context.URL_PKG_PREFIXES, "oracle.aurora.jndi");
    env.put (Context.SECURITY_PRINCIPAL, user);
    env.put (Context.SECURITY_CREDENTIALS, password);
    env.put (Context.SECURITY_AUTHENTICATION, ServiceCtx.NON_SSL_LOGIN);
    Context ic = new InitialContext (env);

    HelloHome hello_home = (HelloHome)ic.lookup (serviceURL + objectName);
    Hello hello = hello_home.create ();
    System.out.println (hello.helloWorld ());
  }
}
```

Home Interface for Hello

```
package hello;

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

public interface HelloHome extends EJBHome
{
  public Hello create () throws RemoteException, CreateException;
}
```

Remote Interface for Hello

```
package hello;

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

public interface Hello extends EJBObject
{
  public String helloWorld () throws RemoteException;
}
```

Bean Implementation for Hello

```
package helloServer;

import javax.ejb.SessionBean;
import javax.ejb.CreateException;
import javax.ejb.SessionContext;
import java.rmi.RemoteException;

public class HelloBean implements SessionBean
{
  // Methods of the Hello interface
  public String helloWorld () throws RemoteException {
    String v = System.getProperty("oracle.server.version");
    return "Hello client, your javavm version is " + v + ".";
  }

  // Methods of the SessionBean
  public void ejbCreate () throws RemoteException, CreateException {}
  public void ejbRemove() {}
```

```
      public void setSessionContext (SessionContext ctx) {}
      public void ejbActivate () {}
      public void ejbPassivate () {}
}
```

SQLJ Example

README

```
Overview
========

This example demonstrates doing a database query using SQLJ. pay
attention to the makefile (UNIX) or the makeit.bat batch file (Windows
NT), and note that the files that SQLJ generates (SER files converted
to class files) must be loaded into the database with deployejb also.

Compare this example with the jdbcimpl basic EJB example, which uses
JDBC instead of SQLJ to perform exactly the same query.


Source files
============

Client.java
-----------

Invoke the client program from the command line, passing it four
arguments:
    - the name of the service URL, e.g. sess_iiop://localhost:2222
    - the path and name of the published bean, e.g. /test/employeeBean
    - the username for db authentication
    - the password (you wouldn't do this in a production program, of course)

For example

% java Client -classpath LIBs sess_iiop://localhost:2222 /test/employeeBean
   scott tiger

The client looks up and activates the bean, then invokes the query() method on
the bean. query() returns an EmpRecord structure with the salary and the name
of the employee whose ID number was passed to query().
```

There is no error checking in this code. See the User's Guide for more
information about the appropriate kinds of error checking in this kind of
client code.

The client prints:

Emp name is ALLEN
Emp sal is 3100.0

employeeServer/employeeBean.sqlj

This class is the bean implementation. A SQLJ named iterator is declared to
hold the results of the query. The myIter.next(); statement is used as is to
keep the code simple: after all the parameter passed in is a known valid
primary key for the EMP table. (See what happens if you try an empno that is
not in the table.)

The EmpIter getter methods are used to retrieve the query results into the
EmpRecord object, which is then returned *by value*, as a serialized object,
to the client.

employeeServer/EmpRecord.java

A class that is in essence a struct to contain the employee name and salary,
as well as the ID number.

Note that the class *must* be defined as implementing the java.rmi.Serializable
interface, to make it a valid serializable RMI object that can be passed from
server to the client.

employee/employee.java

The bean remote interface.

employee/employeeHome.java

The bean home interface.

```
Compiling and Running the Example
=================================

UNIX
----

Enter the command 'make all' or simply 'make' in the shell to compile,
load, and deploy the objects, and run the client program.  Other
targets are 'run' and 'clean'.

Make sure that a shell environment variable ORACLE_HOME is set to
point to the home location of the Oracle installation. This is
operating system dependent, so see the Installation documentation that
came with your system for the location. Also, review the README file
for the Oracle database, and the README file for the CORBA/EJB server
(the Oracle8i ORB), for additional up-to-date information.


Windows NT
----------

On Windows NT, run the batch file makeit.bat from a DOS command prompt
to compile, load, and deploy the objects. Run the batch file runit.bat
to run the client program, and see the results.


Make sure that the environment variables %ORACLE_HOME%, %CLASSPATH%,
and %SERVICE% are set appropriately for the DOS command window. You
can set these as either user or system environment variables from the
Control Panel. Double click on System in the Control Panel then on
the Environment tab to set these variables. Start a new DOS window
after setting environment variable values.


See the Installation documentation that came with your Oracle8i system
for the values of these variables. Also, review the README file for
the Oracle database, and the README file for the CORBA/EJB server (the
Oracle8i ORB), for additional up-to-date information.

You can also set an environment variable %JAVA_HOME% to point to the
root of your Java JDK. For example, SET JAVA_HOME=C:\JDK1.1.6.
```

Client

```
import employee.Employee;
import employee.EmployeeHome;
import employee.EmpRecord;

import oracle.aurora.jndi.sess_iiop.ServiceCtx;

import javax.naming.Context;
import javax.naming.InitialContext;
import java.util.Hashtable;

public class Client {
  public static void main (String [] args) throws Exception {
    if (args.length != 4) {
      System.out.println("usage: Client serviceURL objectName user password");
      System.exit(1);
    }
    String serviceURL = args [0];
    String objectName = args [1];
    String user = args [2];
    String password = args [3];

    Hashtable env = new Hashtable();
    env.put(Context.URL_PKG_PREFIXES, "oracle.aurora.jndi");
    env.put(Context.SECURITY_PRINCIPAL, user);
    env.put(Context.SECURITY_CREDENTIALS, password);
    env.put(Context.SECURITY_AUTHENTICATION, ServiceCtx.NON_SSL_LOGIN);
    Context ic = new InitialContext (env);

    EmployeeHome home = (EmployeeHome)ic.lookup (serviceURL + objectName);
    Employee testBean = home.create();
    EmpRecord empRec = empRec = testBean.query (7499);
    System.out.println ("Emp name is " + empRec.ename);
    System.out.println ("Emp sal  is " + empRec.sal);
  }
}
```

Home Interface

```
package employee;

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

```
public interface EmployeeHome extends EJBHome {
  public Employee create()
       throws CreateException, RemoteException;
}
```

Remote Interface

```
package employee;

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

public interface Employee extends EJBObject  {
  public EmpRecord query (int empNumber)
       throws java.sql.SQLException, RemoteException;
}
```

Bean Implementation

EmployeeBean.sqlj

```
package employeeServer;

import employee.EmpRecord;

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

public class EmployeeBean implements SessionBean {
  //SessionContext ctx;

  public void ejbCreate() throws CreateException, RemoteException {}
  public void ejbActivate() {}
  public void ejbPassivate() {}
  public void ejbRemove() {}
  public void setSessionContext(SessionContext ctx) {
    //this.ctx = ctx;
  }

  public EmpRecord query (int empNumber) throws SQLException, RemoteException
  {
    String ename;
    double sal;
```

```
      #sql { select ename, sal into :ename, :sal from emp
                   where empno = :empNumber };
System.out.println ("ename = " + ename);
System.out.println ("sal   = " + sal);

    return new EmpRecord (ename, empNumber, sal);
  }
}
```

EmpRecord.java

```
package employee;

public class EmpRecord implements java.io.Serializable {
  public String ename;
  public int empno;
  public double sal;

  public EmpRecord (String ename, int empno, double sal) {
    this.ename = ename;
    this.empno = empno;
    this.sal = sal;
  }
}
```

Bean Inheritance Example

README

```
Overview
========

This example show two beans: Foo and Bar. In the example, the Bar bean
inherits from the Foo bean. The required coding and the effects of
this bean inheritance are demonstrated in this example.


Source Files
============


Client.java
-----------
```

You invoke the client program from a command prompt, and pass it four
arguments, the

- service URL (service ID, hostname, port, and SID if port is a listener)
- name of the published bean to lookup and instantiate
- username
- password that authenticates the client to the Oracle8i database server

For example:
```
% java -classpath LIBs Client sess_iiop://localhost:2222 /test/myHello scott
tiger
```

where LIBs is the classpath that must include

```
$ORACLE_HOME/lib/aurora_client.jar
#If using Java 2, use classes12.zip instead of classes111.zip
$ORACLE_HOME/jdbc/lib/classes111.zip
$ORACLE_HOME/lib/vbjorb.jar
$ORACLE_HOME/lib/vbjapp.jar
$JAVA_HOME/lib/classes.zip
```

(Note: for NT users, the environment variables would be %ORACLE_HOME% and
%JAVA_HOME%.)

The client code performs the following steps:

- gets the arguments passed on the command line
- creates a new JNDI Context (InitialContext())
- looks up the published bean to find and activate its home interface
- using the home interface, instantiates through its create()
  method a new bean object, hello
- invokes the helloWorld() method on the hello object and prints the results

The printed output is:

```
Hello World
Hello World from bar
Hello World 2 from bar
Hello World from bar
```

```
foo.ejb
-------
```

The Foo bean deployment descriptor. See ../helloworld/readme.txt for a
more complete description of a typical example deployment descriptor.

bar.ejb

The bar bean deployment descriptor.

inheritance/FooHome.java

The Foo bean home interface. Specifies a single no-parameter create() method.

inheritance/Foo.java

The Foo remote interface. Note that only a single method, hello(), is
specified.

inheritance/BarHome.java

The Bar bean home interface. Specifies a single no-parameter create() method.

inheritance/Bar.java

The Bar remote interface. Note that only a single method, hello2(), is
specified.

inheritanceServer/FooBean.java

The Foo bean implementation. Implements the hello() method of
inheritance/Foo.java, returning a String greeting.

inheritanceServer/BarBean.java

The Bar bean implementation. Implements both the hello() method inherited from
FooBean, as well as the hello2() method specified in inheritance/Bar.java.

Note that this bean extends FooBean, so it does not implement SessionBean or
any of its methods, such as ejbRemove(0, ejbActivate(), and so on, which is
normally a requirement of a session bean. This is because BarBeam inherits the
implementation of these from FooBean.

Compiling and Running the Example
=================================

UNIX

Enter the command 'make all' or simply 'make' in the shell to compile,
load, and deploy the objects, and run the client program. Other
targets are 'run' and 'clean'.

Make sure that a shell environment variable ORACLE_HOME is set to
point to the home location of the Oracle installation. This is
operating system dependent, so see the Installation documentation that
came with your system for the location. Also, review the README file
for the Oracle database, and the README file for the CORBA/EJB server
(the Oracle8i ORB), for additional up-to-date information.

Windows NT

On Windows NT, run the batch file makeit.bat from a DOS command prompt
to compile, load, and deploy the objects. Run the batch file runit.bat
to run the client program, and see the results.

Make sure that the environment variables %ORACLE_HOME%, %CLASSPATH%,
and %SERVICE% are set appropriately for the DOS command window. You
can set these as either user or system environment variables from the
Control Panel. Double click on System in the Control Panel then on
the Environment tab to set these variables. Start a new DOS window
after setting environment variable values.

See the Installation documentation that came with your Oracle8i system
for the values of these variables. Also, review the README file for
the Oracle database, and the README file for the CORBA/EJB server (the
Oracle8i ORB), for additional up-to-date information.

You can also set an environment variable %JAVA_HOME% to point to the
root of your Java JDK. For example, SET JAVA_HOME=C:\JDK1.1.6.

Client

```
import inheritanceServer.*;
import inheritance.*;

import oracle.aurora.jndi.sess_iiop.ServiceCtx;

import javax.naming.Context;
import javax.naming.InitialContext;
import java.util.Hashtable;

public class Client
{
  public static void main(String[] args) throws Exception {
    if (args.length != 5) {
      System.out.println("usage: Client serviceURL fooBeanName "
  + "barBeanName username password");
      System.exit(1);
    }

    String serviceURL = args [0];
    String fooBeanName = args [1];
    String barBeanName = args[2];
    String username = args[3];
    String password = args[4];

    Hashtable env = new Hashtable();
    env.put(Context.URL_PKG_PREFIXES, "oracle.aurora.jndi");
    env.put(Context.SECURITY_PRINCIPAL, username);
    env.put(Context.SECURITY_CREDENTIALS, password);
    env.put(Context.SECURITY_AUTHENTICATION, ServiceCtx.NON_SSL_LOGIN);
    Context ic = new InitialContext(env);


    // Get a foo object from a foo published bean
    FooHome home = (FooHome) ic.lookup(serviceURL + fooBeanName);
    Foo foo = home.create();
```

```
                      System.out.println(foo.hello());

                      // Get a bar object from a bar published bean
                      BarHome barHome = (BarHome) ic.lookup(serviceURL + barBeanName);
                      Bar bar = barHome.create();
                      System.out.println(bar.hello());
                      System.out.println(bar.hello2());


                      // Get a foo object from a bar published bean
                      BarHome fooBarHome = (BarHome)ic.lookup(serviceURL + barBeanName);
                      Foo fooBar = (Foo) fooBarHome.create();
                      System.out.println(fooBar.hello());
                 }
          }
```

Home Interface

BarHome.java

```
package inheritance;

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

public interface FooHome extends EJBHome
{
  public Foo create () throws RemoteException, CreateException;
}
```

FooHome.java

```
package inheritance;

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

public interface FooHome extends EJBHome
{
  public Foo create () throws RemoteException, CreateException;
}
```

Remote Interface

Bar.java

```
package inheritance;

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

public interface Foo extends EJBObject
{
  public String hello () throws RemoteException;
}
```

Foo.java

```
package inheritance;

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

public interface Foo extends EJBObject
{
  public String hello () throws RemoteException;
}
```

Bean Implementation

BarBean.java

```
package inheritanceServer;

import java.rmi.RemoteException;
import javax.ejb.*;
import oracle.aurora.jndi.sess_iiop.*;

public class FooBean implements SessionBean
{
  // Methods of the interface
  public String hello () throws RemoteException {
    return "Hello World";
  }

  // Methods of the SessionBean
```

```
      public void ejbCreate () throws RemoteException, CreateException {
      }

      public void ejbRemove() {
      }

      public void setSessionContext (SessionContext ctx) {
      }

      public void ejbActivate () {
      }

      public void ejbPassivate () {
      }
}
```

FooBean.java

```
package inheritanceServer;

import java.rmi.RemoteException;
import javax.ejb.*;
import oracle.aurora.jndi.sess_iiop.*;

public class FooBean implements SessionBean
{
  // Methods of the interface
  public String hello () throws RemoteException {
    return "Hello World";
  }

  // Methods of the SessionBean
  public void ejbCreate () throws RemoteException, CreateException {
  }

  public void ejbRemove() {
  }

  public void setSessionContext (SessionContext ctx) {
  }

  public void ejbActivate () {
  }

  public void ejbPassivate () {
```

```
      }
    }
```

Entity Bean Examples

The following two examples show how to implement entity beans either using bean-managed or container-managed options:

- Bean-Managed Entity Bean Example
- Container-Managed Entity Bean Example

Bean-Managed Entity Bean Example

Client

```
import purchase.PurchaseOrder;
import purchase.PurchaseOrderHome;
import purchase.LineItem;

import oracle.aurora.jndi.sess_iiop.ServiceCtx;

import javax.naming.Context;
import javax.naming.InitialContext;
import javax.transaction.UserTransaction;
import java.util.*;

public class Client {
  public static void main (String [] args) throws Exception {
    System.out.println("Running client");
    if (args.length != 4) {
      System.out.println("usage: Client serviceURL objectName user password");
      System.exit(1);
    }
    String serviceURL = args [0];
    String objectName = args [1];
    String user = args [2];
    String password = args [3];

    Hashtable env = new Hashtable();
    env.put(Context.URL_PKG_PREFIXES, "oracle.aurora.jndi");
    env.put(Context.SECURITY_PRINCIPAL, user);
    env.put(Context.SECURITY_CREDENTIALS, password);
```

```
env.put(Context.SECURITY_AUTHENTICATION, ServiceCtx.NON_SSL_LOGIN);
Context ic = new InitialContext (env);

PurchaseOrderHome home =
  (PurchaseOrderHome)ic.lookup (serviceURL + objectName);

// Begin a transaction to create a new PO
UserTransaction ut;
ut = (UserTransaction)ic.lookup ("local://corbaut");
ut.begin();

// Create a new PO and add items to it
PurchaseOrder po = home.create();
po.addItem (111111, 2);
po.addItem (333333, 4);

// Price the PO
System.out.println ("PO price $" + po.price ());

// Get the po number for future reference
String ponumber = (String)po.getPrimaryKey ();

// Commit the transaction
ut.commit();

// This is now the future:

// Start another transaction
ut.begin ();

// Retrieve the PO from its primary key
PurchaseOrder po2 = home.findByPrimaryKey(ponumber);

// Add another item
po.addItem (222222, 1);

// Check the PO contents
System.out.println ("Contents of the PO:");
Vector items = po.getContents ();
Enumeration e = items.elements ();
while (e.hasMoreElements ()) {
  LineItem item = (LineItem)e.nextElement ();
  System.out.println (item.quantity + " " + item.description + " at $"
+ (int)item.price + " each");
  }
```

```
        // Compute the price again
        System.out.println ("PO price $" + po.price ());

        // Rollback the change
        ut.rollback ();
      }

    }
```

Home Interface

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

public interface PurchaseOrderHome extends EJBHome {
  // Create a new PO
  public PurchaseOrder create() throws CreateException, RemoteException;

  // Find an existing one
  public PurchaseOrder findByPrimaryKey (String POnumber)
    throws FinderException, RemoteException;
}
```

Remote Interface

```
package purchase;

/*
   PurchaseOrder is an entity bean.

   It is a remote interface to a purchasing application.

   The remote interface lets you manage PO contents (getContents,
   addItem) and execute complex logic on the server side (price).

*/

import javax.ejb.EJBObject;
import java.rmi.RemoteException;
import java.sql.SQLException;
import java.util.Vector;
```

```
public interface PurchaseOrder extends EJBObject  {
  // Price the PO
  public float price() throws RemoteException;

  // Manage contents

  // getContents returns a Vector of LineItem objects
  public Vector getContents() throws RemoteException;

  public void addItem (int sku, int count) throws RemoteException;
}
```

Bean Implementation

PurchaseOrderBean.sqlj

```
package purchaseServer;

/*
   This is the PurchaseOrder Bean.

   The bean manages its own persistence.  The PO line items are stored
   in the LINEITEMS table.  This table references the SKUS table that
   contain individual pricing information.

*/

import purchase.*;
import java.sql.*;
import java.rmi.RemoteException;
import javax.ejb.*;
import java.util.*;

#sql iterator ItemsIter (int skunumber, int count, String description,
 float price);

public class PurchaseOrderBean implements EntityBean {
  EntityContext ctx;

  Vector items;// The items in the PO (instances of LineItem)

  public void PurchaseOrderBean() {}

  // Bean Managed Persistence methods
```

```java
// The create methods takes care of generating a new PO and returns
// its primary key
public String ejbCreate () throws CreateException, RemoteException
{
  String ponumber = null;
  try {
    #sql { select ponumber.nextval into :ponumber from dual };
    #sql { insert into pos (ponumber, status) values (:ponumber, 'OPEN') };
  } catch (SQLException e) {
    throw new PurchaseException (this, "create", e);
  }
  return ponumber;
}

// Nothing to do here
public void ejbPostCreate () {
  items = new Vector ();
}

// The remove method deletes all line items belonging to the PO
public void ejbRemove() throws RemoteException {
  // Get the PO number and delete
  String ponumber = (String)ctx.getPrimaryKey();
  try {
    #sql { delete from lineitems where ponumber = :ponumber };
    #sql { delete from pos where ponumber = :ponumber };
  } catch (SQLException e) {
    throw new PurchaseException (this, "remove", e);
  }
}

// The load method populates the items array with all the existing
// line items
public void ejbLoad() throws RemoteException {
  // Get the PO number
  String ponumber = (String)ctx.getPrimaryKey();

  // Load all line items.
  try {
    items = new Vector ();
    ItemsIter iter = null;
    try {
      #sql iter = {
select lineitems.skunumber, lineitems.count,
```

```
          skus.description, skus.price
              from lineitems, skus
              where ponumber = :ponumber and lineitems.skunumber = skus.skunumber
};

        while (iter.next ()) {
  LineItem item =
    new LineItem (iter.skunumber(), iter.count(), iter.description(),
  iter.price());
  items.addElement (item);
}
      } finally {
if (iter != null) iter.close ();
      }
    } catch (SQLException e) {
      throw new PurchaseException (this, "load", e);
    }
  }

  // The store method replaces all entries in the lineitems table with the
  // new entries from the bean
  public void ejbStore() throws RemoteException {
    // Get the PO number
    String ponumber = (String)ctx.getPrimaryKey();

    try {
      // Delete old entries
      #sql { delete from lineitems where ponumber = :ponumber };

      // Insert new entries
      Enumeration e = items.elements ();
      while (e.hasMoreElements ()) {
LineItem item = (LineItem)e.nextElement ();
        #sql { insert into lineitems (ponumber, skunumber, count)
                values (:ponumber, :(item.sku), :(item.quantity))
};
      }
    } catch (SQLException e) {
      throw new PurchaseException (this, "store", e);
    }
  }

  // The findByPrimaryKey method verifies that the POnumber exists
  public String ejbFindByPrimaryKey (String ponumber)
    throws FinderException, RemoteException
```

```
        {
          try {
            int count;
            #sql { select count (ponumber) into :count from pos
             where ponumber = :ponumber };

            // There has to be one
            if (count != 1)
throw new FinderException ("Inexistent PO: " + ponumber);
          } catch (SQLException e) {
            throw new PurchaseException (this, "findByPrimaryKey", e);
          }
          // The ponumber is the primary key
          return ponumber;
        }

        // Business Methods

        // Price the PO
        public float price() throws RemoteException {
          float price = 0;
          Enumeration e = items.elements ();
          while (e.hasMoreElements ()) {
            LineItem item = (LineItem)e.nextElement ();
            price += item.quantity * item.price;
          }

          // 5% discount if buying more than 10 items
          if (items.size () > 10)
            price -= price * 0.05;

          // Shipping is a constant plus function of the number of items
          price += 10 + (items.size () * 2);

          return price;
        }

        // The getContents methods has to load the descriptions
        public Vector getContents() throws RemoteException {
          return items;
        }

        // The add Item method gets the price and description
        public void addItem (int sku, int count) throws RemoteException {
          try {
```

```
        String description;
        float price;
        #sql { select price, description into :price, :description
              from skus where skunumber = :sku };
        items.addElement (new LineItem (sku, count, description, price));
      } catch (SQLException e) {
        throw new PurchaseException (this, "addItem", e);
      }
    }

    // EntityBean Methods
    public void setEntityContext(EntityContext ctx) { this.ctx = ctx; }
    public void unsetEntityContext() {}
    public void ejbActivate() {}
    public void ejbPassivate() {}
}
```

LineItem.java

```
package purchase;

/*

   This class represents one line item

*/

public class LineItem implements java.io.Serializable {
  public int sku;
  public int quantity;
  public String description;
  public float price;

  public LineItem (int sku, int quantity, String description, float price) {
    this.sku = sku;
    this.quantity = quantity;
    this.description = description;
    this.price = price;
  }
}
```

Deployment Descriptor

```
<?xml version="1.0"?>
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems Inc.//DTD Enterprise JavaBeans 1.1
```

```
//EN" "ejb-jar.dtd">
<ejb-jar>
   <enterprise-beans>
      <entity>
         <description>no description</description>
         <ejb-name>test/purchase</ejb-name>
         <home>purchase.PurchaseOrderHome</home>
         <remote>purchase.PurchaseOrder</remote>
         <ejb-class>purchaseServer.PurchaseOrderBean</ejb-class>
         <persistence-type>Bean</persistence-type>
         <prim-key-class>java.lang.String</prim-key-class>
         <reentrant>False</reentrant>
      </entity>
   </enterprise-beans>
   <assembly-descriptor>
      <security-role>
         <description>no description</description>
         <role-name>PUBLIC</role-name>
      </security-role>
      <method-permission>
         <description>no description</description>
         <role-name>PUBLIC</role-name>
         <method>
            <ejb-name>test/purchase</ejb-name>
            <method-name>*</method-name>
         </method>
      </method-permission>
      <container-transaction>
         <description>no description</description>
         <method>
            <ejb-name>test/purchase</ejb-name>
            <method-name>*</method-name>
         </method>
         <trans-attribute>Required</trans-attribute>
      </container-transaction>
   </assembly-descriptor>
</ejb-jar>
```

Container-Managed Entity Bean Example

Client

```
import customer.Customer;
```

```
import customer.CustomerHome;

import javax.naming.InitialContext;
import javax.naming.NamingException;
import java.rmi.RemoteException;
import javax.ejb.RemoveException;
import javax.ejb.CreateException;
import javax.ejb.FinderException;
import java.util.Enumeration;
import oracle.aurora.jndi.sess_iiop.ServiceCtx;
import java.util.Hashtable;
import javax.naming.Context;

public class CustomerClient
{
  public static void main(String[] argv)
  {

    System.out.println("client is running");
    try
      {
      if (argv.length != 4) {
        System.out.println("usage: Client serviceURL
               objectName user password");
        System.exit(1);
      }
      String serviceURL = argv [0];
      String objectName = argv [1];
      String user = argv [2];
      String password = argv [3];

      Hashtable env = new Hashtable();
      env.put(Context.URL_PKG_PREFIXES, "oracle.aurora.jndi");
      env.put(Context.SECURITY_PRINCIPAL, user);
      env.put(Context.SECURITY_CREDENTIALS, password);
      env.put(Context.SECURITY_AUTHENTICATION, ServiceCtx.NON_SSL_LOGIN);
      Context ic = new InitialContext (env);
      CustomerHome ch = (CustomerHome)ic.lookup (serviceURL + objectName);

      Customer cust = ch.create("Jake Terwilliger", "Pine Drive");
      System.out.println("created " + cust.getName());
      System.out.println (" address = " + cust.getAddress());
      String pk = (String) cust.getPrimaryKey();
      System.out.println("Primarykey = " + pk);
```

```
      //imagine that time passes here, or this program is
      //finished, and a later program wants to use the
      //primary key
      Customer cust1 = ch.create("Al Smith", "Sesame Street");

      Customer cust2 = ch.create("Bob Davidson", "Elm Street");

      Customer cust3 = ch.create("Carol Fernandez", "Cedar Blvd");

      cust = null;
      cust = ch.findByPrimaryKey(pk);
      System.out.println("Found by primary key lookup");
      System.out.println (" name = " + cust.getName());
      System.out.println (" address = " + cust.getAddress());
      cust.remove();
      System.out.println("removed bean");

      cust = ch.findByWhere("where cust_addr = 'Elm Street'");
      System.out.println("Found by findByWhere");
      System.out.println (" name = " + cust.getName());
      System.out.println (" address = " + cust.getAddress());
      cust.remove();
      System.out.println("removed bean");

      Enumeration e = ch.findAllCustomers("");
      while(e.hasMoreElements())
        {
          cust = (Customer) e.nextElement();
          System.out.println (" name = " + cust.getName());
          System.out.println (" address = " + cust.getAddress());
        }

  }
catch (RemoveException e)
  {
  System.out.println("RemoveException caught:" + e);
  e.printStackTrace();
  }
catch (NamingException e)
  {
  System.out.println("NamingException caught:" + e);
  e.printStackTrace();
  }
catch (FinderException e)
  {
```

```
            System.out.println("FinderException caught:" + e);
            e.printStackTrace();
            }

        catch (CreateException e)
          {
          System.out.println("CreateException caught:" + e);
          e.printStackTrace();
          }

        catch (RemoteException e)
          {
          System.out.println("RemoteException caught:" + e);
          e.printStackTrace();
          }
      }
    }
```

Home Interface

```
package customer;
import javax.ejb.EJBHome;
import java.rmi.RemoteException;
import javax.ejb.CreateException;
import javax.ejb.FinderException;

public interface CustomerHome extends EJBHome
{
  public Customer findByPrimaryKey(String pk) throws RemoteException,
          FinderException;
  public Customer findByWhere(String whereString) throws RemoteException,
          FinderException;
  public java.util.Enumeration findAllCustomers(String whereString) throws
          RemoteException, FinderException;
  public Customer create(String custname, String custaddr) throws
          RemoteException, CreateException;
}
```

Remote Interface

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

public interface Customer extends EJBObject
```

```
{
  public String getName() throws RemoteException;
  public String getAddress() throws RemoteException;
  public void setAddress(String addr) throws RemoteException;
}
```

Bean Implementation

```
package customerServer;
import customer.*;
import javax.ejb.*;
import java.sql.*;
import java.util.*;
import java.rmi.RemoteException;
import java.io.Serializable;

public class CustomerBean implements EntityBean
{
  private transient EntityContext ctx;
  public String name;
  public String addr;

  public String getName() throws RemoteException
  {
    return name;
  }
  public void setName(String name) throws RemoteException
  {
    this.name = name;
  }
  public String getAddress() throws RemoteException
  {
    return addr;
  }
  public void setAddress(String addr) throws RemoteException
  {
    this.addr = addr;
  }
  public void setEntityContext(EntityContext ctx)
  {
    this.ctx = ctx;
    Properties props = ctx.getEnvironment();

  }
  public void unsetEntityContext()
```

```
        {
          this.ctx = null;
        }

        public String ejbCreate(String custname, String custaddr) throws
      CreateException, RemoteException
        {
          try {
            setName(custname);
            setAddress(custaddr);
          } catch (java.rmi.RemoteException e) {
            throw new CreateException();
          }
            return null;
        }

      public String ejbFindByPrimaryKey(String pk) throws RemoteException,
      FinderException {
        return null;
      }


        public void ejbPostCreate(String custname, String custaddr) throws
      CreateException
        {
          // get primarykey
          String pk = (String)ctx.getPrimaryKey();
        }


        public void ejbActivate()
        {

        }

        public void ejbPassivate()
        {

        }

        public void ejbRemove()
        {
        }

        public void ejbLoad()
```

```
      {
        // You can get to the primary key
        String pk = (String)ctx.getPrimaryKey();
      }

      public void ejbStore()
      {
      }

}
```

XML Deployment Descriptor

```xml
<?xml version="1.0"?>
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems Inc.//DTD Enterprise JavaBeans 1.1
//EN" "ejb-jar.dtd">
<ejb-jar>
   <enterprise-beans>
      <entity>
         <ejb-name>customerbean</ejb-name>
         <home>customer.CustomerHome</home>
         <remote>customer.Customer</remote>
         <ejb-class>customerServer.CustomerBean</ejb-class>
         <persistence-type>Container</persistence-type>
         <prim-key-class>java.lang.String</prim-key-class>
         <reentrant>False</reentrant>
         <cmp-field><field-name>name</field-name></cmp-field>
         <cmp-field><field-name>addr</field-name></cmp-field>
         <primkey-field>name</primkey-field>
         <env-entry>
           <env-entry-name>realmName</env-entry-name>
           <env-entry-type>java.lang.String</env-entry-type>
           <env-entry-value>my.realm</env-entry-value>
         </env-entry>
         <env-entry>
           <env-entry-name>customerBean.databaseURL</env-entry-name>
           <env-entry-type>java.lang.String</env-entry-type>
           <env-entry-value>jdbc:oracle:kprb:</env-entry-value>
         </env-entry>
         <env-entry>
           <env-entry-name>customerBean.JDBCDriverName</env-entry-name>
           <env-entry-type>java.lang.String</env-entry-type>
           <env-entry-value>oracle.jdbc.driver.OracleDriver</env-entry-value>
         </env-entry>
      </entity>
```

```
            </enterprise-beans>
            <assembly-descriptor>
                <security-role>
                    <role-name>PUBLIC</role-name>
                </security-role>
                <method-permission>
                    <role-name>PUBLIC</role-name>
                    <method>
                        <ejb-name>test/customer</ejb-name>
                        <method-name>*</method-name>
                    </method>
                </method-permission>
                <container-transaction>
                    <method>
                        <ejb-name>test/customer</ejb-name>
                        <method-name>*</method-name>
                    </method>
                    <trans-attribute>RequiresNew</trans-attribute>
                </container-transaction>
            </assembly-descriptor>
        </ejb-jar>
```

Oracle-Specific Deployment Descriptor

```
            <?xml version="1.0"?>
            <!DOCTYPE oracle-descriptor PUBLIC "-//Oracle Corporation.//DTD Oracle 1.1//EN"
            "oracle-ejb-jar.dtd">
            <oracle-descriptor>
            <mappings>
              <ejb-mapping>
                  <ejb-name>customerbean</ejb-name>
                  <jndi-name>test/customer2</jndi-name>
              </ejb-mapping>
            </mappings>

            <persistence-provider>
              <persistence-name>psi-ri</persistence-name>
              <persistence-deployer>oracle.aurora.ejb.persistence.ocmp.OcmpEntityDeployer
              </persistence-deployer>
            </persistence-provider>

            <persistence-descriptor>
              <ejb-name>customerbean</ejb-name>
              <persistence-name>psi-ri</persistence-name>
              <psi-ri>
```

```
        <schema>SCOTT</schema>
        <table>customers</table>
        <attr-mapping>
          <field-name>name</field-name>
          <column-name>cust_name</column-name>
        </attr-mapping>
        <attr-mapping>
          <field-name>addr</field-name>
          <column-name>cust_addr</column-name>
        </attr-mapping>
    </psi-ri>
</persistence-descriptor>
</oracle-descriptor>
```

Database Table Updates

```
connect scott/tiger

drop table CUSTOMERS;

create table CUSTOMERS (CUST_NAME VARCHAR(64) NOT NULL, CUST_ADDR VARCHAR(64),
            CUST_SERIALIZE LONG RAW );
```

Session Example

README

```
Overview
========

This EJB example shows how you can create a second EJB in
the same server, but in a different session. The same username and
password are used to create the second object, and it accesses the
same published EJB.

Source Files
============


Client.java
-----------

You invoke the client program from a command prompt, and pass it four
```

```
arguments, the

    - service URL (service ID, hostname, port, and SID if port is a listener)
    - name of the published bean to lookup and instantiate
    - username
    - password that authenticates the client to the Oracle8i database server

For example:
% java -classpath LIBs Client sess_iiop://localhost:2481:ORCL |
     /test/myHello scott tiger

where LIBs is the classpath that must include

$ORACLE_HOME/lib/aurora_client.jar
#If using Java 2, use classes12.zip instead of classes111.zip
$ORACLE_HOME/jdbc/lib/classes111.zip
$ORACLE_HOME/lib/vbjorb.jar
$ORACLE_HOME/lib/vbjapp.jar
$JAVA_HOME/lib/classes.zip

The client code performs the following steps:

    - gets the arguments passed on the command line
    - creates a new JNDI Context (InitialContext())
    - looks up the published bean to find and activate its home interface
    - using the home interface, instantiates through its create()
      method a new bean object, hello
    - sets the hello bean's message to "Hello World!"
    - asks the first hello bean to create another bean, by invoking the
        getOtherHello() method, passing it the authentication, service URL,
        and bean name parameters
    - invokes otherHelloWorld() on the first bean, and printing its
        return value, which is derived from the second created bean


The printed output is:

Hello World!
Hello from the Other HelloBean Object


hello.ejb
---------

The bean deployment descriptor.
```

```
helloServer/HelloBean.java
--------------------------
```

The EJB implementation.

```
hello/Hello.java
----------------
```

The bean remote interface.

```
hello/HelloHome.java
--------------------
```

The bean's home interface.

```
Compiling and Running the Example
=================================
Before running this example, the user 'scott' needs to have
javauserpriv. This can be enabled by doing:
$ svrmgrl
SVRMGRL> connect internal
SVRMGRL> grant javauserpriv to scott;
SVRMGRL> quit
$
```

The configuration file INITSID.ORA must also specify that at least two
MTS servers can be activated. That is, the parameters MTS_SERVERS and
MTS_MAX_SERVERS must be set to at least the following:

```
mts_servers=2
mts_max_servers=2
```

```
UNIX
----
```

Enter the command 'make all' or simply 'make' in the shell to compile,
load, and deploy the objects, and run the client program. Other
targets are 'run' and 'clean'.

Make sure that a shell environment variable ORACLE_HOME is set to
point to the home location of the Oracle installation. This is
operating system dependent, so see the Installation documentation that
came with your system for the location. Also, review the README file
for the Oracle database, and the README file for the CORBA/EJB server
(the Oracle8i ORB), for additional up-to-date information.

```
Windows NT
----------
```

On Windows NT, run the batch file makeit.bat from a DOS command prompt
to compile, load, and deploy the objects. Run the batch file runit.bat
to run the client program, and see the results.

Make sure that the environment variables %ORACLE_HOME%, %CLASSPATH%,
and %SERVICE% are set appropriately for the DOS command window. You
can set these as either user or system environment variables from the
Control Panel. Double click on System in the Control Panel then on
the Environment tab to set these variables. Start a new DOS window
after setting environment variable values.

See the Installation documentation that came with your Oracle8i system
for the values of these variables. Also, review the README file for
the Oracle database, and the README file for the CORBA/EJB server (the
Oracle8i ORB), for additional up-to-date information.

You can also set an environment variable %JAVA_HOME% to point to the
root of your Java JDK. For example, SET JAVA_HOME=C:\JDK1.1.6.

Client

```java
import hello.Hello;
import hello.HelloHome;

import oracle.aurora.jndi.sess_iiop.ServiceCtx;

import javax.naming.Context;
import javax.naming.InitialContext;
import java.util.Hashtable;

public class Client
{
```

```
          public static void main (String[] args) throws Exception {
            if (args.length != 4) {
              System.out.println ("usage: Client serviceURL objectName user password");
              System.exit (1);
            }
            String serviceURL = args [0];
            String objectName = args [1];
            String user = args [2];
            String password = args [3];

            Hashtable env = new Hashtable ();
            env.put (Context.URL_PKG_PREFIXES, "oracle.aurora.jndi");
            env.put (Context.SECURITY_PRINCIPAL, user);
            env.put (Context.SECURITY_CREDENTIALS, password);
            env.put (Context.SECURITY_AUTHENTICATION, ServiceCtx.NON_SSL_LOGIN);
            Context ic = new InitialContext (env);

            // Activate a Hello in the 8i server
            // This creates a first session in the server
            HelloHome hello_home = (HelloHome)ic.lookup (serviceURL + objectName);
            Hello hello = hello_home.create ();
            hello.setMessage ("Hello World!");
            System.out.println (hello.helloWorld ());

            // Ask the first Hello to activate another Hello in the same server
            // This creates Another SESSION used by the first session
            hello.getOtherHello (user, password, serviceURL + objectName);
            System.out.println (hello.otherHelloWorld ());
          }
        }
```

Home Interface

```
        package hello;

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

        public interface HelloHome extends EJBHome
        {
          public Hello create () throws RemoteException, CreateException;
        }
```

Remote Interface

```
package hello;

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

public interface Hello extends EJBObject
{
  public String helloWorld () throws RemoteException;

  public void setMessage (String message) throws RemoteException;

  public void getOtherHello (String user, String password, String otherBeanURL)
       throws RemoteException, CreateException;

  public String otherHelloWorld () throws RemoteException;
}
```

Bean Implementation

```
package helloServer;

import hello.*;

import javax.ejb.SessionBean;
import javax.ejb.SessionContext;

import javax.ejb.CreateException;
import java.rmi.RemoteException;
import javax.naming.NamingException;

import javax.naming.Context;
import javax.naming.InitialContext;
import java.util.Hashtable;
import oracle.aurora.jndi.sess_iiop.ServiceCtx;

public class HelloBean implements SessionBean
{
  String message;
  Hello otherHello;

  // Methods of the Hello interface
  public String helloWorld () throws RemoteException {
```

```
      return message;
  }

  public void setMessage (String message) throws RemoteException {
    this.message = message;
  }

  public void getOtherHello (String user, String password, String otherBeanURL)
        throws RemoteException, CreateException
  {
    try {
      // start a new session
      Hashtable env = new Hashtable ();
      env.put (Context.URL_PKG_PREFIXES, "oracle.aurora.jndi");
      env.put (Context.SECURITY_PRINCIPAL, user);
      env.put (Context.SECURITY_CREDENTIALS, password);
      env.put (Context.SECURITY_AUTHENTICATION, ServiceCtx.NON_SSL_LOGIN);
      Context ic = new InitialContext (env);

      // create the other Bean instance
      HelloHome other_HelloHome = (HelloHome)ic.lookup (otherBeanURL);
      otherHello = other_HelloHome.create ();
      otherHello.setMessage ("Hello from the Other HelloBean Object");
    } catch (NamingException e) {
      e.printStackTrace ();
    }
  }

  public String otherHelloWorld () throws RemoteException {
    if (otherHello != null)
      return otherHello.helloWorld ();
    else
      return "otherBean is not accessed yet";
  }

  // Methods of the SessionBean
  public void ejbCreate () throws RemoteException, CreateException {}
  public void ejbRemove () {}
  public void setSessionContext (SessionContext ctx) {}
  public void ejbActivate () {}
  public void ejbPassivate () {}
}
```

SSL Examples

Client-Side Authentication Example

README

```
Overview
========

This is the exact same example as under examples/ejb/basic/helloworld,
except that this example is using SSL client auth. So, except for the
SSL details, please refer to the readme file under
examples/ejb/basic/helloworld for other details.

The purpose of the example is to show how to use ssl client side
authentication instead of username/password combination.

Setup required
--------------

You need to open the encrypted wallet(ewallet.der) provided in this directory
using the wallet manager tool provided by Oracle, and save it as clear
text wallet (cwallet.sso). The password is welcome12.
Copy the generated cwallet.sso into TNS_ADMIN directory.

The encrypted wallet(ewallet.der) is only valid for Solaris platforms. For
other platforms, you should generate the wallet using Oracle's owm tool.

This test also requires B64 encoded wallet(cert.txt) which is already present
in this directory.  You can also generate your own using Oracle's owmgui tool
and using export option in the tool.

The parameter SSL_CLIENT_AUTHENTICATION in $TNSADMIN/sqlnet.ora should be set
to TRUE.

Restart the database.

The setup also requires creation of a global user, say guest. The script to
create
global user is in this directory(create.sh).  This script prompts for username
and password of a privileged user as input to this script.
```

Client

```
import hello.Hello;
import hello.HelloHome;

import oracle.aurora.jndi.sess_iiop.ServiceCtx;

import javax.naming.Context;
import javax.naming.InitialContext;
import java.util.Hashtable;

public class Client
{
  public static void main (String[] args) throws Exception {
    if (args.length != 4) {
      System.out.println ("usage: Client serviceURL objectName credentials_file
password");
      System.exit (1);
    }
    String serviceURL = args [0];
    String objectName = args [1];
    String credsFile = args [2];
    String password = args [3];

    Hashtable env = new Hashtable ();
    env.put(Context.URL_PKG_PREFIXES, "oracle.aurora.jndi");
    env.put(Context.SECURITY_AUTHENTICATION, ServiceCtx.SSL_CLIENT_AUTH);
    env.put(Context.SECURITY_CREDENTIALS, password);
    // Simply specify  a file that contains all the credential info. This is
    // the file generated by the wallet manager tool.
    env.put(Context.SECURITY_PRINCIPAL, credsFile);

    Context ic = new InitialContext (env);

    HelloHome hello_home = (HelloHome)ic.lookup (serviceURL + objectName);
    Hello hello = hello_home.create ();
    System.out.println (hello.helloWorld ());
  }
}
```

Home Interface

```
package hello;

import javax.ejb.EJBHome;
```

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

public interface HelloHome extends EJBHome
{
  public Hello create () throws RemoteException, CreateException;
}
```

Remote Interface

```
package hello;

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

public interface Hello extends EJBObject
{
  public String helloWorld () throws RemoteException;
}
```

Bean Implementation

```
package helloServer;

import javax.ejb.SessionBean;
import javax.ejb.CreateException;
import javax.ejb.SessionContext;
import java.rmi.RemoteException;

public class HelloBean implements SessionBean
{
  // Methods of the Hello interface
  public String helloWorld () throws RemoteException {
    String v = System.getProperty("oracle.server.version");
    return "Hello client, your javavm version is " + v + ".";
  }

  // Methods of the SessionBean
  public void ejbCreate () throws RemoteException, CreateException {}
  public void ejbRemove() {}
  public void setSessionContext (SessionContext ctx) {}
  public void ejbActivate () {}
  public void ejbPassivate () {}
}
```

Server-Side Authentication Example

README

```
Overview
========

This is the exact same example as under examples/ejb/basic/helloworld,
except that this example is using SSL server side auth. So, except for the
SSL details, please refer to the readme file under
examples/ejb/basic/helloworld for other details.

The purpose of the example is to show how to use ssl server side
authentication. Since the client doesn't have certificate in this
case, it still passes username/password.

Setup required
--------------

You need to open the encrypted wallet(ewallet.der) provided in this directory
using the wallet manager tool provided by Oracle, and save it as clear
text wallet (cwallet.sso). The password is welcome12.
Copy the generated cwallet.sso into TNS_ADMIN directory.

The encrypted wallet(ewallet.der) is only valid for Solaris platforms. For
other platforms, you should generate the wallet using Oracle's owm tool.

The parameter SSL_CLIENT_AUTHENTICATION in $TNSADMIN/sqlnet.ora should be set
to FALSE.

Restart the database.
```

Client

```
import hello.Hello;
import hello.HelloHome;

import oracle.aurora.jndi.sess_iiop.ServiceCtx;

import javax.naming.Context;
import javax.naming.InitialContext;
import java.util.Hashtable;

public class Client
```

```
      {
        public static void main (String[] args) throws Exception {
          if (args.length != 4) {
            System.out.println ("usage: Client serviceURL objectName user password");
            System.exit (1);
          }
          String serviceURL = args [0];
          String objectName = args [1];
          String user = args [2];
          String password = args [3];

          Hashtable env = new Hashtable ();
          env.put (Context.URL_PKG_PREFIXES, "oracle.aurora.jndi");
          env.put (Context.SECURITY_PRINCIPAL, user);
          env.put (Context.SECURITY_CREDENTIALS, password);
          env.put (Context.SECURITY_AUTHENTICATION, ServiceCtx.SSL_CREDENTIAL);
          Context ic = new InitialContext (env);

          HelloHome hello_home = (HelloHome)ic.lookup (serviceURL + objectName);
          Hello hello = hello_home.create ();
          System.out.println (hello.helloWorld ());
        }
      }
```

Home Interface

```
      package hello;

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

      public interface HelloHome extends EJBHome
      {
        public Hello create () throws RemoteException, CreateException;
      }
```

Remote Interface

```
      package hello;

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

      public interface Hello extends EJBObject
```

```
{
  public String helloWorld () throws RemoteException;
}
```

Bean Implementation

```
package helloServer;

import javax.ejb.SessionBean;
import javax.ejb.CreateException;
import javax.ejb.SessionContext;
import java.rmi.RemoteException;

public class HelloBean implements SessionBean
{
  // Methods of the Hello interface
  public String helloWorld () throws RemoteException {
    String v = System.getProperty("oracle.server.version");
    return "Hello client, your javavm version is " + v + ".";
  }

  // Methods of the SessionBean
  public void ejbCreate () throws RemoteException, CreateException {}
  public void ejbRemove() {}
  public void setSessionContext (SessionContext ctx) {}
  public void ejbActivate () {}
  public void ejbPassivate () {}
}
```

C

Abbreviations and Acronyms

This appendix lists some of the most common acronyms that you will find in the areas of networks, distributed object development, and Java. In cases where an acronym refers to a product or a concept that is associated with a specific group, company or product, the group, company, or product is indicated in brackets following the acronym expansion. For example: CORBA ... [OMG].

This acronym list is intended as a helpful guide only. There are no guarantees that it is complete or even completely accurate.

3GL	third generation language
4GL	fourth generation language
ACID	atomicity, consistency, isolation, durability
ACL	access control list
ADT	abstract datatype
AFC	application foundation classes [Microsoft]
ANSI	American National Standards Institute
API	application program interface
AQ	advanced queuing [Oracle8]
ASCII	American standard code for information interchange
ASP	active server pages [Microsoft] application service provider
AWT	abstract windowing toolkit [Java]
BDK	beans developer kit [Java]
BLOB	binary large object

BOA	basic object adapter [CORBA]
BSD	Berkeley system distribution [UNIX]
C/S	client/server
CGI	common gateway interface
CICS	customer information control system [IBM]
CLI	call level interface [SAG]
CLOB	character large object
COM	common object model [Microsoft]
COM+	common object model, extended [Microsoft]
CORBA	common object request broker architecture [OMG]
DB	database
DBA	database administrator, database administration
DBMS	database management system
DCE	distributed computing environment [OSF]
DCOM	distributed common object model [Microsoft]
DDCF	distributed document component facility
DDE	dynamic data exchange [Microsoft]
DDL	data definition language [SQL]
DLL	dynamic link library [Microsoft]
DLM	distributed lock manager [Oracle8]
DML	data manipulation language [SQL]
DOS	disk operating system
DSOM	distributed system object model [IBM]
DSS	decision support system
DTP	distributed transaction processing
EBCDIC	extended binary-coded decimal interchange code [IBM]
EJB	Enterprise JavaBean
ERP	enterprise resource planning
ESIOP	environment-specific inter-orb protocol

FTP	file transfer protocol
GB	gigabyte
GIF	graphics interchange format
GIOP	general inter-orb protocol
GUI	graphical user interface
GUID	globally-unique identifier
HTML	hypertext markup language
HTTP	hypertext transfer protocol
IDE	integrated development environment interactive development environment
IDL	interface definition language
IEEE	Institute of Electrical and Electronics Engineers
IIOP	internet inter-ORB protocol
IIS	Internet information server [Microsoft]
IP	internet protocol
IPC	interprocess communication
IS	information services
ISAM	indexed sequential access method
ISAPI	Internet server API [Microsoft]
ISO	international standards organization (translation)
ISP	internet service provider
ISQL	interactive SQL [Interbase]
ISV	independent software vendor
IT	information technology
J2EE	Java 2 Enterprise Edition [Sun]
JAR	Java archive (on analogy with tar, q.v.)
JCK	Java compatibility kit [Sun]
JDBC	"Java database connectivity"
JDK	Java developer kit

JFC	Java foundation classes
JIT	just in time
JLS	Java language specification
JMF	Java media framework
JMS	Java messaging service
JNDI	Java naming and directory interface
JNI	Java native interface
JOB	Java Objects for Business [Sun]
JPEG	joint photographic experts group
JRMP	Java remote ??message protocol
JSP	Java server pages [Sun] (sometimes used for Java Stored Procedure [Oracle])
JTA	Java transaction API
JTS	Java transaction service
JWS	Java Web Server [Sun]
KB	kilobyte
LAN	local area network
LDAP	lightweight directory access protocol
LDIF	LDPA data interchange format
LOB	large object
MB	megabyte
MIME	multipurpose Internet mail extensions
MIS	management information services
MOM	message-oriented middleware
MPEG	motion picture experts group
MTS	multi-threaded server [Oracle]
MTS	Microsoft Transaction Server [Microsoft]
NCLOB	national character large object
NIC	network information center [internet]

NNTP	net news transfer protocol
NSAPI	Netscape server application programming interface
NSP	network service provider
NT	New Technology [Microsoft]
OCI	Oracle call interface
OCX	OLE common control [Microsoft]
ODBC	open database connectivity [Microsoft]
ODBMS	object database management system
ODL	object definition language [Microsoft]
ODMG	Object Database Management Group
OEM	original equipment manufacturer
OID	object identifier
OLE	object linking and embedding
OLTP	on line transaction processing
OMA	object management architecture [OMG]
OMG	Object Management Group
OO	object-oriented, object orientation
OODBMS	object-oriented database management system
OQL	object query language
ORB	object request broker
ORDBMS	object-relational database management system
OS	operating system
OSF	Open System Foundation
OSI	open systems interconnect
OSQL	object SQL
OTM	object transaction monitor
OTS	object transaction service
OWS	Oracle Web Server
PB	petabyte

PDF	portable document format [Adobe]
PGP	pretty good privacy
PL/SQL	procedural language/SQL [Oracle]
POA	portable object adapter [CORBA]
RAM	random access memory
RAS	remote access service [Microsoft]
RCS	revision control system
RDBMS	relational database management system
RFC	request for comments
RFP	request for proposal
RMI	remote method invocation [Sun]
ROM	read only memory
RPC	remote procedure call
RTF	rich text file
SAF	server application function [Netscape]
SAG	SQL Access Group
SCSI	small computer system interface
SDK	software developer kit
SET	secure electronic transaction
SGML	standard generalized markup language
SID	system identifier [Oracle]
SLAPD	standalone LDAP daemon
SMP	symmetric multiprocessing
SMTP	simple mail transfer protocol
SPI	service provider interface
SQL	structured query language
SQLJ	SQL for Java
SRAM	static (or synchronous) random access memory
SSL	secure socket layer

TB	terabyte
TCPS	TCP for SSL
TCP/IP	transmission control protocol/internet protocol
TP	transaction processing
TPC	Transaction Processing Council
TPCW	TPC web benchmark
TPF	transaction processing facility
TPM	transaction processing monitor
UCS	universal character set [ISO 10646]
UDP	user datagram protocol
UI	user interface
UML	unified modeling language [Rational]
URI	uniform resource identifier
URL	universal resource locator
URN	universal resource name
VAR	value-added reseller
VB	Visual Basic [Microsoft]
VRML	virtual reality modeling language
WAI	web application interface [Netscape]
WAN	wide area network
WIPS	web interactions per second [TPCW]
WWW	world wide web
XA	extended architecture [X/Open]
XML	extended markup language
jdb	Java debugger [Sun]
tar	tape archive, tape archiver [UNIX]
tps	transactions per second

Index