Oracle8*i*

Enterprise JavaBeans and CORBA Developer’s Guide

Release 2 (8.1.6)

December 1999

Part No. A81356-01

Enterprise JavaBeans and CORBA Developer’s Guide, Release 2 (8.1.6)

Part No. A81356-01

Release 2 (8.1.6)

Copyright © 1998, 1999, Oracle Corporation. All rights reserved.

Primary Authors: Tim Smith, Sheryl Maring

Contributors: 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

The programs are not intended for use in any nuclear, aviation, mass transit, medical, or other inherently dangerous applications. It shall be licensee's responsibility to take all appropriate fail-safe, back up, redundancy and other measures to ensure the safe use of such applications if the Programs are used for such purposes, and Oracle disclaims liability for any damages caused by such use of the Programs.

This Program contains proprietary information of Oracle Corporation; it is provided under a license agreement containing restrictions on use and disclosure and is also protected by copyright, patent and other intellectual property law. Reverse engineering of the software is prohibited.

The information contained in this document is subject to change without notice. If you find any problems in the documentation, please report them to us in writing. Oracle Corporation does not warrant that this document is error free. No part of this document may be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose, without the express written permission of Oracle Corporation

If this Program is delivered to a U.S. Government Agency of the Department of Defense, then it is delivered with Restricted Rights and the following legend is applicable:

Restricted Rights Legend Programs delivered subject to the DOD FAR Supplement are "commercial computer software" and use, duplication and disclosure of the Programs shall be subject to the licensing restrictions set forth in the applicable Oracle license agreement. Otherwise, Programs delivered subject to the Federal Acquisition Regulations are "restricted computer software" and use, duplication and disclosure of the Programs shall be subject to the restrictions in FAR 52.227-14, Rights in Data -- General, including Alternate III (June 1987). Oracle Corporation, 500 Oracle Parkway, Redwood City, CA 94065.

Oracle is a registered trademark, and Oracle products mentioned herein are trademarks or registered trademarks of Oracle Corporation. All other company or product names mentioned are used for identification purposes only and may be trademarks of their respective owners.

Contents

Send Us Your Comments

Enterprise JavaBeans and CORBA Developer's Guide, Release 2 (8.1.6)

Part No. A81356-01

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 and CORBA applications 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 or CORBA applications 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*. If you are developing CORBA applications, this guide assumes that you have some familiarity with CORBA; if you are developing EJB applications, read the EJB 1.0 specification to supplement this Guide. See "Suggested Reading" on page iii.

How This Guide Is Organized

This guide consists of six chapters and four appendices:

Chapter 1, "Overview", presents a brief overview of the EJB and CORBA development models 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.

Chapter 3, "Developing CORBA Applications", describes techniques for developing CORBA server objects that run in the Oracle8*i* data server.

Chapter 4, "Connections and Security", covers more advanced information than that in Chapters 2 and 3, including session management and alternative authentication procedures.

Chapter 5, "Transaction Handling", documents the transaction interfaces that you can use when developing both EJB and CORBA applications.

Chapter 6, "Tools", documents the command-line tools that are necessary when developing a CORBA or EJB application.

Appendix A, "Example Code: CORBA", includes examples of CORBA applications. Each example contains both the Java and IDL source code.

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

Appendix C, "Comparing the Oracle8i JServer and VisiBroker VBJ ORBs", discusses some of the fundamental differences between developing CORBA applications for VisiBroker and the Oracle8*i* JServer.

Appendix D, "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.

Suggested Reading

Programming with VisiBroker, by D. Pedrick et al. (John Wiley and Sons, 1998) provides a good introduction to CORBA development from the VisiBroker point of view.

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.

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`

Another popular Java Web site is:

`http://www.gamelan.com`

For Java API documentation, see:

`http://www.javasoft.com`

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

Oracle8i SQLJ Developer's Guide and Reference

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, 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. As with the more specific chapters that follow, this overview focuses on the aspects of Enterprise JavaBeans and CORBA development that are particular to JServer, giving a brief general description of these standard development models.

This chapter covers the following topics:

- Prerequisite Reading
- About Enterprise JavaBeans
- About CORBA
- Common Features
- Tools
- Caffeine
- Example Code

Prerequisite Reading

Before consulting this Guide, you should read the *Oracle8i Java Developer's Guide*, which 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.

In addition, the *Oracle8i Java Developer's Guide* describes the strategic advantages of the distributed component development model that is implemented by both EJBs and CORBA.

About Enterprise JavaBeans

Enterprise JavaBeans (EJB) is an architecture for developing transactional applications as distributed components in Java, and it is a powerful development methodology for distributed application development. Neither the bean developer nor the client application programmer need to be concerned with details such as transaction support, security, remote object access, and many other complicated and error-prone issues when developing with EJB. The EJB server and container furnish these features transparently for you.

Additionally, EJB applications are developed entirely in Java. It is not necessary for developers to learn a new language such as IDL.

Because of this simplicity, you can quickly develop applications that use EJBs; furthermore, EJBs offer portability. A bean developed on one EJB server should run on other EJB servers that meet the EJB specification. Portability has not currently been tested for most servers, but it is a promise for the future.

The Oracle8*i* JServer implements the EJB version 1.0 specification, providing a server and a container that hosts 1.0-compatible enterprise beans. The current release of JServer supports session beans only, as required by the specification. Entity beans will be supported in a future release.

EJB specifies Java Remote Method Invocation (RMI) as the transport protocol. Oracle8*i* JServer implements RMI over IIOP. 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.

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 but 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. EJB activates both types of bean on demand in a new session. Stateful beans can offer the same performance as stateless beans, while preserving the advantages of stateful beans (their "conversational state").

Deployment Descriptor

EJB deployment is simplified by the following:

- a text form deployment descriptor
- a tool that verifies the bean interfaces, generates and compiles the required infrastructure classes for the bean, and loads these classes into the database. The deploy tool then publishes the bean home interface in the database so that the client applications can access it.

Oracle8*i* JServer complies with the EJB 1.0 specification and offers a highly scalable and high-performance execution environment for EJBs. 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 Service (JTS) for client-side transaction demarcation
- a UserTransaction interface for bean-managed transactions

- tools that assist you in developing deployment descriptors and deploying your EJB application

About CORBA

CORBA, the Common Object Request Broker Architecture model, offers a well-supported international standard for cross-platform, cross-language development. CORBA supports cross-language development by specifying a neutral language, Interface Definition Language (IDL), in which you develop specifications for the interfaces that the application objects expose.

CORBA 2.0 supports cross-platform development by specifying a transport mechanism, IIOP, that allows different operating systems running on very different hardware to interoperate. IIOP supplies a common "software" bus that, together with an ORB running on each system, makes data and request transfer transparent to the application developer.

Although the CORBA standard was developed and promulgated just before the advent of Java and is a standard focused on component development in a heterogeneous application development environment, incorporating systems and languages of varying age and sophistication, it is perfectly possible to develop CORBA applications solely in Java. CORBA and Java are a good match.

For CORBA developers, JServer offers the following services and tools:

- a Java Transaction Service (JTS) interface to the OMG Object Transaction Service (OTS)
- a CosNaming implementation used when publishing objects to an Oracle8*i* database, retrieving the object references, and activating objects
- a version of the IIOP protocol that supports the JServer session-based ORB, which is compatible with standard IIOP
- a wide range of tools, which assist in developing CORBA applications, that do the following:
  - load Java classes and resource files to the database
  - drop loaded classes
  - publish objects to the CosNaming service
  - manage the session name space

Common Features

CORBA and EJB have different strengths. CORBA was designed to support a heterogeneous application development environment, incorporating systems and languages of varying age and sophistication. The EJB specification was designed to bring Java within the realm of enterprise application development and to automate the most error-prone features of large-scale development.

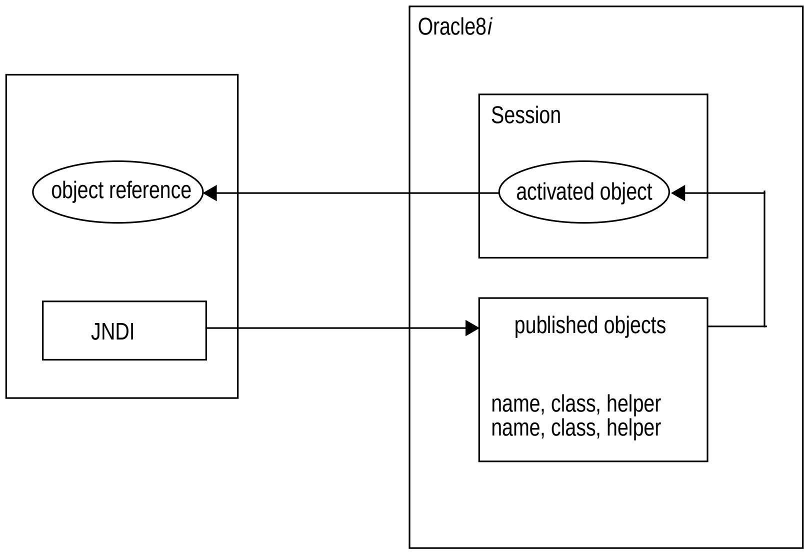
Although they represent different development models, you will find that developing for either CORBA or EJB within the Oracle8*i* JServer framework offers a large degree of conceptual similarity.

Both EJB and CORBA leverage the capabilities of the Oracle8*i* database server, in particular the multi-threaded server. The threading model offered by the server simplifies not only the implementation of the ORB but the user's view of it.

With both EJB and CORBA applications, access to server-side objects is similar. You publish objects in the Oracle database using the OMG CosNaming service, which you can access using Oracle's JNDI interface to CosNaming. CORBA developers can also use the pure CosNaming approach. EJB developers must follow the EJB specification and use JNDI for access.

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

Figure 1–1 Remote Object Access



The organization of this Guide reflects the partial similarity between EJB and CORBA. Chapter 2 covers EJB development, and Chapter 3 discusses CORBA. The important issues of session management, security, and client-side transaction control are not described independently for EJBs and CORBA because there are many similar aspects to them. Chapter 4 discusses the connection and authentication aspects of EJB and CORBA. Chapter 5 explains transactions.

IIOP

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 and CORBA programming models.

Tools

Oracle8*i* JServer comes with a complete set of tools for use in developing EJB and CORBA applications. You run these command-line tools from a UNIX shell or at a Windows NT DOS prompt. The tools allow you to compile IDL specifications, load Java classes or source files into the Oracle database, publish objects in the session name space, and display and manipulate published object names.

Chapter 6 of this Guide covers the tools.

In addition to the command-line tools JServer offers, you can use Oracle's JDeveloper tool suite to develop your distributed object applications.

Caffeine

JServer incorporates the Inprise (Visigenic) Caffeine tools that allow you to code object interfaces directly in Java and generate the infrastructure necessary to support distributed object invocation. These tools include:

- *`java2rmi_iiop`* generates the infrastructure EJB requires to call other remote objects. `java2rmi_iiop` is an extension of the Inprise `java2iiop` tool.
- *`java2idl`* compiles Java interfaces to IDL code, for cases where IDL is required.

Chapter 6 describes these tools.

Example Code

JServer comes with approximately 40 EJB and CORBA sample programs. These brief examples demonstrate all the major features of the product, including:

- database access using both SQLJ and JDBC
- security
- session management
- transaction control

The examples come complete and ready to run, including a UNIX makefile and Windows NT batch file to compile and run each example. All you need is a Java-enabled Oracle8*i* database with the standard EMP and DEPT demo tables for some of the examples, and you can run the examples right out of the box.

The emphasis in these short examples is on demonstrating features of the ORB, EJBs, and CORBA, not on elaborate Java coding techniques. Even Java novices will be able to understand these examples with only brief study.

In addition to these sample programs, there are longer demos that contain more complete examples, including an example (acctMgmt) that uses several objects and is implemented using both EJB and CORBA, so that you can compare the two models.

The example and demo code are available on the distribution Compact Disc. See the README file that comes with JServer for the location and name of the archive file that contains the examples. 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.

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:

Defining Enterprise JavaBeans

Enterprise JavaBeans 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. Portability has not been tested yet for most servers, but it is a bright promise for the future.

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. See "Deploying an EJB" on page 2-21 for specific information about this phase of development.

  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*.

EJBs as Distributed Components

Although the EJB specification is based on concepts developed for the Remote Method Invocation interface (RMI), EJB server vendors are not required to use the RMI transport. Oracle8*i* uses the Internet Inter-ORB Protocol (IIOP). Using IIOP means that a server can support EJBs whose methods can be invoked by other IIOP clients.

Enterprise beans can also call out to CORBA objects. See Figure 2–2 on page 2-8.

Enterprise JavaBeans As Components

An EJB is a software component that runs in a server. This runtime environment is one factor that distinguishes an enterprise bean from a JavaBean. The JavaBean usually runs on a client system, such as a network computer, a PC, or a workstation, and it typically performs presentation tasks, such as implementing GUI widgets.

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 implements a business entity. A session bean might contain methods that query and update data in a relational table; an entity bean represents business data directly. For example, an entity bean can represent a row in a relational table.

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.

Persistence

Session beans are not inherently *persistent*. Be careful about this word. 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 saves its state in an Oracle8*i* database, if required, but it does not directly represent business data. Entity beans persist the business data either automatically (in a container-managed entity bean) or by way of methods that use JDBC or SQLJ and are coded into the bean (bean-managed).

Implementing the synchronization interface can make data storage and retrieval automatic for session beans. See "Using Session Synchronization" on page 2-27.

EJB Support in Oracle8*i*

The version 1.0 of the EJB specification requires that the EJB server support session beans. Entity bean support is optional. In the current release, the Oracle8*i* EJB server does not support entity beans. Entity beans will be supported in a future release.

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. When a session bean is re-instantiated, state of previous instances is not automatically restored.

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.

Stateless Session Beans

In most EJB implementations, a stateless session bean is used for short transactions with a client. In these implementations, the major difference between stateful and stateless session beans is that a stateless bean can change identity between method calls; a stateful bean maintains identity. If the client calls Method A in a stateless bean, then calls Method B in the same stateless bean class, the second method might be called on a separate instance of the bean.

In the Oracle8*i* implementation, stateless and stateful beans are identical. The inherent multi-threaded nature of the Oracle8*i* MTS data server makes stateful session beans functionally identical to stateless beans. There is no difference between the two for Oracle8*i*.

For example, a typical use of stateless session beans is a server maintaining a pool of beans ready to serve clients that are performing short OLTP-like transactions. But this is not required in the Oracle8*i* architecture for performance. Stateful beans can serve just as well in this situation.

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

The home interface is an interface to an object that the container itself implements: the *home object*. The home interface has `create()` methods that specify how a bean is created. The home interface, with the home object, actually serves as a factory object for EJBs.

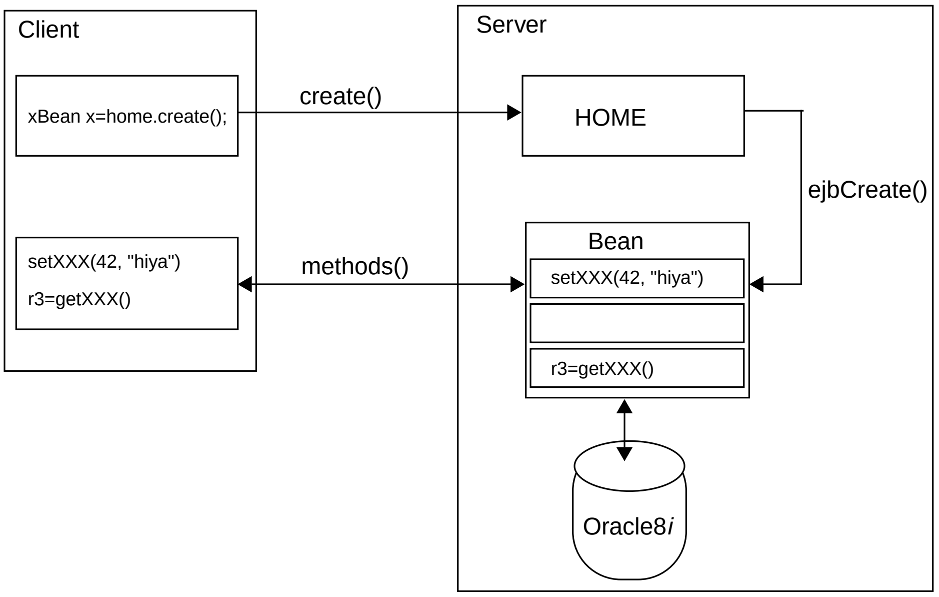
The remote interface specifies the methods that you implement in the bean. These methods perform the business logic of the bean. The bean must also implement additional service methods that the EJB container calls at various times in the life cycle of a bean. See Basic Concepts on page 2-8 for more information about these service methods.

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.

Why is this level of indirection necessary? Remember that the container provides services transparently for the bean. For example, if you deploy the bean with a transaction attribute that declares that the bean must run in its own transaction context, then the container can start up the transaction before the message is passed to the bean and can do a commit or rollback, as required, before return messages or data is sent back to the client.

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

Figure 2–1 Basic EJB Component Relationships



The bean implementation contains the Java code that implements the remote interface and the required container methods.

The deployment descriptor is an object that specifies attributes of the bean. 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.

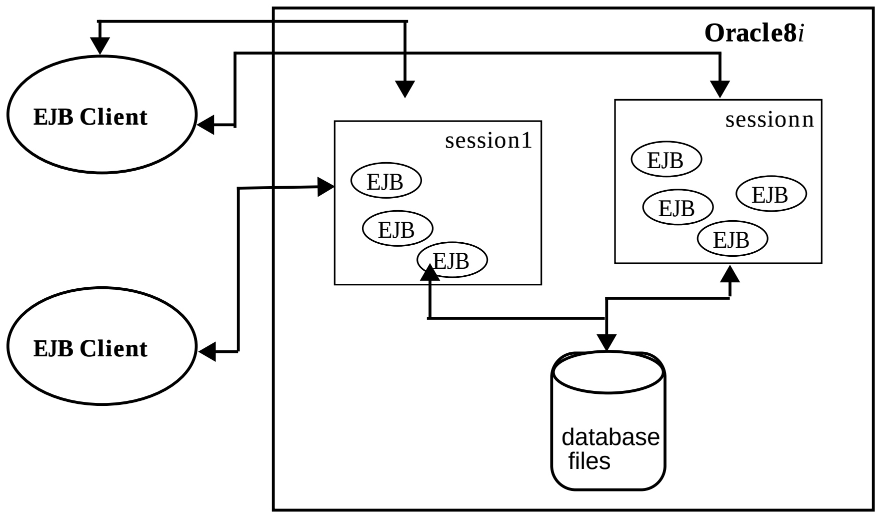
The EJB Architecture

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—a practice that is becoming common among server vendors.

Figure 2–2 shows the basic EJB architecture.

Figure 2–2 EJB Architecture



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.

First, the client actually calls a local proxy stub for the remote method. The stub marshalls any parameter data, and then calls a remote skeleton on the server. The skeleton unmarshalls 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.

In the second step, the bean container gets 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 (see Using Session Synchronization on page 2-27)
- identity checks and switch

The container then delegates the method call to the bean. The bean method executes. 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.

Then the bean container calls the skeleton, which marshalls 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.

The Home Interface

When a client needs to create a bean instance, it does so through the home interface, which specifies one or more `create()` methods. A `create()` method can take parameters passed in from the client when the bean is created.

For each create method in the home interface, there must be a corresponding `ejbCreate()` method specified in the remote interface, with the same signature. The only difference is that `create()` is specified to return the bean type; `ejbCreate()` is a void method. When a client invokes `create()` on the home interface, the container interposes whatever services are necessary at that point and then calls the corresponding `ejbCreate()` method in the bean itself.

The `deployejb` tool publishes a reference to the home object in the database. See "deployejb" on page 6-22. This is the object that the client looks up to create instances of the bean.

The Remote Interface

The bean developer writes a remote interface for each EJB in the application, which specifies the business methods that the bean contains. Each method in the bean that the client accesses must be specified in the remote interface. Private methods in the bean are not specified in the remote interface.

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

PL/SQL developers will recognize that the remote interface is similar to a package specification, and the remote interface implementation resembles the package body. However, the remote interface does not declare public variables, but declares only the methods that are implemented by the bean.

The remote interface must be public, and it must subclass `javax.ejb.EJBObject`. For example, you could write a remote interface for an `employeeManagement` bean as follows:

```
public interface employeeManagement extends javax.ejb.EJBObject {

  public void hire(int empNumber, String startDate, double salary)
        throws java.rmi.RemoteException;
  public double getCommission(int empNumber) throws java.rmi.RemoteException;
  // empRecord is a class that is defined separately as part of the bean
  public empRecord getEmpInfo(int empNumber) throws java.rmi.RemoteException;
  ...
}
```

All methods in the remote interface are declared as throwing `RemoteException`. This is the usual mechanism for notifying the client of runtime errors in the bean. However, the bean container can throw other exceptions, such as `SQLException`. 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, which contains the stack trace of the remote exception.

Accessing the Bean Methods

Invoking a bean’s methods is a two-step process.

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. For those who know CORBA, the bean home interface is acting similar to a CORBA factory object, able to produce new CORBA objects on demand.

Once you have the home interface and then the bean reference returned by the home interface `create()` method, you call the bean methods using the normal Java syntax: `bean.method()`.

These steps are completely illustrated by example in "A First EJB Application" on page 2-12.

As a quick first example, suppose that `myBeanHome` is a reference that you have obtained to the home interface of a bean called `myBean`. `myBean` 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:

```
myBean home =
  (myBean) initialContext.lookup(URL);  // get the home interface using JNDI
myBean tester = home.create();          // create a new bean of type myBean
```

and then call myBean's methods using the usual syntax

```
tester.method1(p1, p2);
```

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 (int, double) are serializable. Any non-remote object that implements the `java.io.Serializable` interface can also be passed.

A non-remote object passed as a parameter to a bean or returned from a bean is passed by *copy*, 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 gets 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 the pass-by-copy semantics.

If the non-remote object is complex, for example 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 First EJB Application

This section demonstrates a complete example of an application, including:

- home and remote interface code
- the bean implementation code
- the deployment descriptor
- client-side code

This example has a single EJB, which queries an Oracle8*i* database to get name and salary information about an employee. The example is identical in functionality to the first CORBA example presented in Chapter 3, "Developing CORBA Applications".

In this example, the client code is an application running on a client system. For an applet example, see the `EJBClubMed` example under the basic EJB examples that are shipped with this product.

The Interfaces

The first task of the bean provider is to design and code the home and remote interfaces. The home interface specifies how the server will create the bean, using the `EJBCreate()` method of the bean implementation. This example creates a stateful session bean that takes no parameters, because there is no initial state for the bean.

How is it known that the bean is stateful? Although this is a design property of the bean, the statefulness of the bean is declared in the deployment descriptor. See "Deployment Steps" on page 2-27 for more information.

The remote interface specifies the methods of the bean. In this example, a single method, `getEmployee()`, takes an **`int`** as its single parameter, thereby returning an `EmpRecord` class.

Home Interface

As required by the EJB specification, you must declare that any home interface `create()` method throws the `javax.ejb.CreateException` and `java.rmi.RemoteException` exceptions. When you try to deploy the bean, the `deployejb` verifier will exit with an error if this is not the case.

```
package employee;

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

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

Remote Interface

The remote interface declares that the bean can throw a `RemoteException` (required by the specification), and a `java.sql.SQLException`, which is particular to this bean. Exceptions, such as `SQLException`, that are thrown to the bean by JDBC or other methods that it calls are propagated back to client if the remote interface declares that the bean throws them.

Here is the code for the remote interface for this example EJB:

```
package employee;

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

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

The Bean Implementation

The bean implementation simply fills in the Java code, including appropriate JDBC methods, to perform the work of the `getEmployee()` method. Note that 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. It is the *only* way that a JDBC connection can be opened in server-side JDBC code.

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;
  public EmpRecord getEmployee (int empNumber)
       throws SQLException, RemoteException {

    EmpRecord empRec = new EmpRecord();

    Connection conn =
      new oracle.jdbc.driver.OracleDriver().defaultConnection();
    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;
  }

  public void ejbCreate() throws CreateException, RemoteException {
  }
  public void ejbActivate() {
  }
  public void ejbPassivate() {
  }
  public void ejbRemove() {
  }
  public void setSessionContext(SessionContext ctx) {
    this.ctx = ctx;
  }
}
```

This remote interface implementation illustrates the minimum methods required for an EJB implementation. 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)`	Set the associated session context. 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.

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 Deployment Descriptor

The most convenient way to implement the deployment descriptor for a bean is to write a descriptor file in text form. The EJB deployment tool can read the text form descriptor, parse it, signal parse errors, and then verify that the descriptor itself, and the interface and bean implementation declarations meet the standard. For example, bean implementations and interface specifications must be declared as throwing certain specified exceptions. If they do not, the deployment tool (see "deployejb" on page 6-22) lists the errors and exits.

The text form deployment descriptor is usually stored in a file with a `.ejb` extension, though this naming convention is not required. In the EJB examples that are shipped with this product, the deployment descriptors are in the base directory of the example, along with the client application implementations and the Makefile and Windows NT batch files.

Here is the deployment descriptor for this example. For a complete description of the deployment descriptor attributes, see "Deploying an EJB" on page 2-21.

```
SessionBean employeeServer.EmployeeBean {
  BeanHomeName = "test/employeeJDBCBean";
  RemoteInterfaceClassName = employee.Employee;
  HomeInterfaceClassName = employee.EmployeeHome;

  AllowedIdentities = {SCOTT};
  StateManagementType = STATEFUL_SESSION;
  RunAsMode = CLIENT_IDENTITY;
  TransactionAttribute = TX_REQUIRED;
}
```

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 the URLs in "For More Information" on page 2-32.

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 "publish" on page 6-3 for more details about publishing EJB home interfaces and about the Oracle8*i* published object directory structure.

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 2–1 contains the six properties that you can set in the hashtable that are passed to the `javax.naming.Context.`

Table 2–1 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 three 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`

Table 2–1 Context Properties

Property	Purpose
`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 "SYSDBA" to connect with the SYSDBA role.
`oracle.aurora.sess_iiop.ServiceCtx.SSL_VERSION`	The client-side SSL version number.

See Chapter 4, "Connections and Security", 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 4, "Connections and Security" 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. 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 = new EmpRecord();  // create a slot for the incoming data
    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:

- Get the beans from the EJB developer. In the typical case, JServer compiles and puts into a JAR file—one JAR file for each bean—the beans and their accompanying classes, including the home and remote interfaces and any classes dependent on the bean.
- Develop a deployment descriptor for each bean.
- Run the `deployejb` tool, which:
  - reads the deployment descriptor and the bean JAR file
  - loads the bean classes into the Oracle8*i* database
  - publishes the bean home interface
- Make sure that the application developer has the information necessary about the bean remote interface and the name of the published beans.

Write the Deployment Descriptor

The enterprise bean deployer supplies a deployment descriptor for each EJB in the application.

There is a text form of the descriptor, which is described in this section. You can also use the Oracle8*i* JServer `ejbdescriptor` command-line tool to convert a text form of the deployment descriptor to the serialized class, or to convert back from the serialized object to a text file. The `ejbdescriptor` is documented in "ejbdescriptor" on page 6-24.

Note: You can only use 7-bit ASCII characters in the deployment descriptor. Do not use ISO Latin-1 or other non-ASCII characters.

Text Format

The text form of the session bean descriptor follows the conventions of Java code—the descriptor has the syntax of a Java class. The text format always begins with a `SessionBean` keyword, which is followed by the fully-qualified class name of the bean. The body of the declaration contains a list of descriptor attributes and their values. For example:

```
SessionBean ejb.test.server.ExampleBeanImpl
{
    <attribute>=<value>
    ...
}
```

In this example, `ejb.test.server` is the name of the package that contains the implementation of the bean class `ExampleBean`.

Note: Bean deployment will change significantly for the next release of the EJB specification. The preliminary version of that specification calls for the bean deployment information to be specified using XML.

There are three different types of bean attributes:

- Attributes of the bean itself, such as:
  - its published name
  - the names of its home and remote interfaces
  - miscellaneous attributes that can apply only to the bean as a whole, such as `SessionTimeout`
- Each bean method can have a specific set of attributes, which defines security and transaction support. For example, you can specify that a method of a bean runs with a different identity (user or schema name) from other methods in the same bean. Or, you can set transaction properties on a method so that it runs with different transaction properties from the rest of the bean.

  The transaction and security properties can apply to the bean as a whole or be specified on a method-by-method basis.
- Any environment properties to be passed to the bean.

The attributes of a session bean descriptor correspond to the attributes of the class `javax.ejb.deployment.SessionDescriptor` and its super class `javax.ejb.deployment.DeploymentDescriptor.`

Table 2–2 on page 2-23 lists the attributes that you can use in the deployment descriptor.

Table 2–2 Deployment Descriptor Attributes (Cont.)
Table 2–3

Attribute Name	Values	Required?
`BeanHomeName`	A Java String that represents the published name of the bean.	Yes
`HomeInterfaceClassName`	The fully-qualified name of the bean home interface class.	Yes
`RemoteInterfaceClassName`	The fully-qualified name of the bean remote interface class.	Yes
`Reentrant`	The literal "true" or "false". For entity beans.	No
`SessionTimeout`	In seconds from the time that the last bean client disconnects. The default value is 0, which means that the session terminates when the last connection has terminated.	No

Table 2–3

Attribute Name	Values	Required?
`StateManagementType`	STATEFUL_SESSION | STATELESS_SESSION Determines whether a session bean is stateful or stateless. Not relevant for the Oracle8*i* implementation. The default is STATEFUL_SESSION, which you should always use.	No
`TransactionAttribute`	TX_BEAN_MANAGED | TX_MANDATORY | TX_NOT_SUPPORTED | TX_REQUIRED | `TX_REQUIRES_NEW | TX_SUPPORTS` (the default). See Transaction Management for EJBs on page 5-11 for the semantics of the transaction attributes.	No
`IsolationLevel`	TRANSACTION_READ_COMMITTED | TRANSACTION_READ_UNCOMMITTED | TRANSACTION_REPEATABLE_READ | TRANSACTION_SERIALIZABLE This is not supported in the Oracle8*i* EJB server.	No
`RunAsMode`	CLIENT_IDENTITY | SPECIFIED_IDENTITY | SYSTEM_IDENTITY	No
`RunAsIdentity`	A username in the database. Cannot be a role.	Yes, if RunAsMode is used.
`AllowedIdentities`	A list of usernames or roles in the database, enclosed in braces. Example: {SCOTT, WENDY, OTTO}.	No

The example below demonstrates a deployment descriptor:

```
SessionBean ejb.test.server.DatabaseWorkImpl
{
  BeanHomeName = "test/dbwork";  // this is the published name of the bean
  RemoteInterfaceClassName = ejb.test.DatabaseWork;
  HomeInterfaceClassName = ejb.test.DatabaseWorkHome;

  AllowedIdentities = {SCOTT};

  SessionTimeout = 30;   // in seconds
  StateManagementType = STATEFUL_SESSION;

  RunAsMode = CLIENT_IDENTITY;

  TransactionAttribute = TX_REQUIRES_NEW;

  // Add any environment properties that the bean requires
  EnvironmentProperties {
    prop1 = value1;
    prop2 = "value two";
  }

  public ejb.test.EmpRecord getEmployee(int e) throws TestException{
    RunAsMode = CLIENT_IDENTITY;
    AllowedIdentities = { SCOTT };
  }
  public void update(int e, double s) throws TestException{
    RunAsMode = SPECIFIED_IDENTITY;
    AllowedIdentities = { OTTO };
  }
}
```

Create a JAR File

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

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. In JServer, you publish the bean home interface in an Oracle8*i* database with the `deployejb` command-line tool. It publishes the bean in the instance CosNaming namespace under the name that you specify in the `BeanHomeName` attribute of the deployment descriptor.

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 Chapter 6, "Tools", for documentation of the `dropjava` and session shell tools.

Implementing Transactions

Enterprise JavaBeans are inherently transactional. Transaction support is supplied for the bean by the EJB container. You do not need to implement explicit transaction methods in the bean or call transaction services from the client.

EJB supports *declarative* transactions. The bean deployer can specify, in the deployment descriptor, the transaction attributes for a bean or for an individual method in a bean. For example, if the deployment descriptor for a bean declares that the bean has the transaction attribute TX_REQUIRES_NEW, the bean container starts a transaction before each method call to the bean and attempts to commit the transaction when the method ends.

TransactionAttribute

The bean deployer declares the transaction handling characteristics of a bean in the deployment descriptor, which is specified in the transaction attribute. The attribute has six possible values:

- TX_NOT_SUPPORTED
- TX_REQUIRED
- TX_SUPPORTS
- TX_REQUIRES_NEW
- TX_MANADATORY
- TX_BEAN_MANAGED

"Transaction Management for EJBs" on page 5-11 describes the semantics of these attribute values.

Access Control

The EJB deployment descriptor enables you to specify access control lists. You can specify access control either on the entire bean or on individual methods of the bean.

In the text form of the deployment descriptor, specify the `AllowedIdentities` attribute with a list containing usernames, roles, or a mixture of the two. Only users, or users with the roles specified in the `AllowedIdentities` attribute, can access the bean, or the methods that are restricted by the attribute. For example:

```
AllowedIdentities = {SCOTT};  // only SCOTT can access the bean
AllowedIdentities = {PUBLIC}; // all users can access the bean


public int Rmethod (int p1, double p2) throws TestException{
    RunAsMode = CLIENT_IDENTITY;
    AllowedIdentities = { ROGERS };   // only ROGERS can invoke this method
}
```

When you specify method access control, the method must be either a public business method of the bean or the `ejbCreate()` method.

Transaction Isolation Level

The current release does not support the transaction isolation level attribute.

Using Session Synchronization

An EJB can optionally implement the session synchronization interface, to be notified by the container of the transactional state of the bean. Use this interface to save the bean state in the database at transaction boundaries. "Session Synchronization Interface" on page 5-16 describes this interface.

Deployment Steps

The format used to package EJBs is defined by the EJB specification. The format is adaptable—you can use it to distribute a single EJB or to distribute a complete server-side application made up of tens or even hundreds of beans. 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 "deployejb" on page 6-22.

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 version 1.1.6 compiler. Although, you might be able to use another JCK-tested Java compiler to create EJBs to run in the Oracle8*i* server, Oracle supports JDK 1.1.6 or later.
2. Write a deployment descriptor for the EJB. See Programming Restrictions on page 2-31 for specific information about creating deployment descriptors.
3. 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.
4. Call the `deployejb` tool (see "deployejb" on page 6-22) to load and publish the JAR'd bean.

Programming Techniques

This section describes some of the programming techniques you can use when developing EJB session beans. You can use all the capabilities supplied by the Oracle8*i* multi-threaded server to manage multiple sessions, use SQLJ to simplify data acquisition and update, and employ the UserTransaction interface to manage transactions.

Using SQLJ

The bean developer can employ the Oracle8*i* SQLJ translator to simplify EJBs that access the database using static SQL statements. For example, "The Bean Implementation" on page 2-14 required about seven JDBC calls. The same bean implementation can use SQLJ, which requires only two major SQLJ statements, as shown below:

```
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 };

    return new EmpRecord (ename, empNumber, sal);
  }
}
```

The complete example is available on the distribution CD in the `demo.tar` file, as `sqljimpl` in the `examples/ejb/basic` directory.

Setting a Session Timeout

The session timeout value in the deployment descriptor determines how long a session stays active after the last bean client disconnects. It can be important to keep a session alive in at least two cases:

- if the connection might be interrupted, and the client has an expectation of being able to reconnect and resume processing
- if a second client might need to connect to the session and access its EJBs after the originating client has exited

The EJB deployer can set a session timeout value, using the SessionTimeout attribute in the bean deployment descriptor (see "The Deployment Descriptor" on page 2-16).

Saving an EJB Handle

Using the Oracle8*i* EJB server, it is possible for a client to connect to a session that was started by another client, and to access a bean in that session. This is true as long as the second client can authenticate as a valid user of the database.

However, to access a session established by another user, the client must have access to a handle for a bean in that session. A client can provide such a handle to another client using the `getHandle()` method, which returns a bean object reference.

The following code demonstrates one way to get a bean handle and save it to a file using an output stream. You can also use Java streams to write the bean handle to another reader object.

First, get a reference to a bean, in the usual way:

```
saveHandleHome home =
 (saveHandleHome) ic.lookup("sess_iiop://localhost:2481:ORCL/test/myEmployee");
saveHandle testBean = home.create();
```

Next, create an object output stream from a file stream:

```
FileOutputStream fostream = new FileOutputStream(handlefile);
ObjectOutputStream ostream = new ObjectOutputStream(fostream);
```

Then get the bean handle using getHandle(), and write it to the output stream:

```
ostream.writeObject(testBean.getHandle());
```

Finally, clean up the streams:

```
ostream.flush();
fostream.close();
```

See the complete example in `examples/ejb/basic/saveHandle` in the `demo.tar` file on the distribution CD.

EJB as Client

It is possible for an EJB to serve as a client to another EJB. In this case, the client EJB simply looks up the other EJB in the same way that a Java non-EJB client would.

See the example in the examples `/ejb/session/clientserverserver` directory in the demo file (`demo.tar` or `demo.zip`).

Programming Restrictions

The EJB 1.0 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 is not allowed to use the calls to an underlying transaction manager directly. The only exception is employing enterprise Beans with the

TX_BEAN_MANAGED transaction attribute. These beans can use the `javax.jts.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 1.0 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*.

For More Information

Here are some references to specifications and other material that provides more information about EJBs and related services.

- The current 1.0 EJB specification is available at:

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

- 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`

- The *Developer's Guide to Understanding Enterprise JavaBeans*, an overview of EJBs, is available at `http://www.Nova-Labs.com.`
- *Core Java: Volume II—Advanced Features* by Horstmann and Cornell, Sunsoft Press, has a chapter on RMI. Because RMI provides much of the conceptual foundation for the EJB architecture, this is a valuable chapter to read.

3 Developing CORBA Applications

This chapter describes how to develop CORBA applications for Oracle8*i*. CORBA is a powerful distributed application development architecture. Although it is a powerful tool, you can still start to develop useful applications quickly, using Oracle8*i* CORBA. The emphasis in this chapter is practical, not conceptual. The first few sections of this chapter present the conceptual basis for CORBA application development.

This chapter is based on examples, showing you how to use Oracle8*i* CORBA by developing examples—from the simple to the slightly more complex. You can expand these examples into full-fledged applications for your enterprise.

This chapter covers the following topics:

Terminology

This section defines some of the basic terms used in this chapter. See also Appendix D, "Abbreviations and Acronyms" for a list of common acronyms used in Java and distributed object computing.

client

A client is an object, an application, or an applet that makes a request of a server object. Remember that a client need not be a Java application running on a workstation or a network computer, nor an applet downloaded by a web browser. A server object can be a client of another server object. "Client" refers to a role in a requestor/server relationship, not to a physical location or a type of computer system.

marshalling

In distributed object computing, marshalling refers to the process by which the ORB passes requests and data between clients and server objects.

object adapter

Each CORBA ORB implements an object adapter (OA), which is the interface between the ORB and the message-passing objects. CORBA 2.0 specifies that a basic object adapter (BOA) must exist, but most of the details of its interface are left up to individual CORBA vendors. Future CORBA standards will require a vendor-neutral *portable object adapter* (POA). Oracle intends to support a POA in a future release.

request

A request is a method invocation. Other names sometimes used in its stead are *method call* and *message*.

server object

A CORBA server object is a Java object activated by the server, typically on a first request from a client.

session

A *session* always means a database session. Although it is conceptually the same kind of session as that established when a tool such as SQL*Plus connects to Oracle, there are differences in the CORBA case, as follows:

- You establish the database session using the IIOP protocol; you establish a SQL*Plus session using the Net8 TTC protocol.
- A Java virtual machine (JVM) that runs in the database server controls an IIOP session.

Note: To use CORBA with Oracle8*i*, you must configure the database so that the listener can recognize incoming IIOP requests, in addition to TTC requests. DBAs and system administrators should see the "Configuring CORBA and EJB in JServer" on page 4-11 for information on setting up the database and the listener to accept incoming IIOP requests.

See Chapter 4, "Connections and Security", for more information about sessions.

About CORBA

This section provides a short introduction to CORBA, and should give you some idea of how you typically use CORBA in the Oracle8*i* server environment. Providing a complete introduction to CORBA is beyond the scope of this Guide. See the references in "For More Information" on page 3-42 for suggested further reading. This first section gives a very high-level overview of CORBA itself.

CORBA stands for *Common Object Request Broker Architecture*. What is *common* about CORBA is that it integrates ideas from several of the original proposers. CORBA did not just follow the lead of a single large corporation, and it is very deliberately vendor neutral. The CORBA *architecture* specifies a software component, a *broker*, that mediates and directs *requests* to *objects* that are distributed across a network (or several networks), which might have been written in a different language from that of the requestor, and which might be (and in fact, usually are) running on a completely different hardware architecture from that of the requestor.

You can begin to get an idea of the tremendous advantages of CORBA from the preceding paragraph. CORBA enables your application to tie together components from various sources. Also, and unlike a typical client/server application, a CORBA application is not inherently synchronous. It is not necessarily typical that a CORBA requestor (a client) invokes a method on a server component and waits for a result. Using asynchronous method invocations, event interfaces and callbacks from server object to the client ORB, you can construct elaborate applications that link together many interacting objects and that access one or many data sources and other resources under transactional control. CORBA enables you to go beyond the bounds of the traditional client/server application in many imaginative ways.

CORBA Features

CORBA achieves its flexibility in several ways:

- It specifies an *interface description language* (IDL) that allows you to specify the interfaces to objects. IDL object interfaces describe, among other things:
    - The data that the object makes public.
    - The operations that the object can respond to, including the complete signature of the operation. CORBA operations map to Java methods, and the IDL operation parameter types map to Java datatypes.
    - Exceptions that the object can throw. IDL exceptions also map to Java exceptions, and the mapping is very direct.

      CORBA provides bindings for many languages, including both non-object languages such as COBOL and C, and object-oriented languages such as Smalltalk and Java.
- All CORBA implementations provide an *object request broker* (ORB), that handles the routing of object requests in a way that is largely transparent to the application developer. For example, requests (method invocations) on remote objects that appear in the client code look just like local method invocations. The remote call functionality, including marshalling of parameter and return data, is performed for the programmer by the ORB.
- CORBA specifies a network protocol, the *Internet Inter-ORB Protocol* (IIOP), that provides for transmission of ORB requests and data over a widely-available transport protocol: TCP/IP, the Internet standard.
- A set of fully-specified *services* eases the burden of application development by making it unnecessary for the developer to constantly reinvent the wheel. Among these services are:
    - Naming. One or more services that let you resolve names that are bound to CORBA server objects.
    - Transactions. Services that let you manage transaction control of data resources in a flexible and portable way.
    - Events.

CORBA specifies over 12 services. Most of these are not yet implemented by CORBA ORB vendors.

The remainder of this section introduces some of the essential building blocks of an Oracle8*i* JServer CORBA application. These include:

- the ORB—how to talk to remote objects
- IDL—how to write a portable interface
- the naming service (and JNDI)—how to locate a persistent object
- object adapters—how to register a transient object

Note: The Java code examples in this chapter are available on line. You can study the complete examples (see Appendix A, "Example Code: CORBA"), compile and run them, and then modify them for your own use. All examples are installed in the $ORACLE_HOME/javavm/demo/demo.zip file.

About the ORB

The object request broker, or ORB, is the fundamental part of a CORBA implementation. The ORB makes it possible for a client to send messages to a server, and the server to return values to the client. The ORB handles all communication between a client and a server object.

The JServer ORB is based on code from Inprise's VisiBroker 3.4 for Java. The ORB that executes on the server side has been slightly modified from the VisiBroker code, to accommodate the different Oracle8*i* object location and activation model. The client-side ORB has been changed very little.

In some CORBA implementations, the application programmer and the server object developer must be aware of the details of how the ORB is activated on the client and the server, and they must include code in their objects to start up the ORBs and activate objects. The Oracle8*i* ORB, on the other hand, makes these details largely transparent to the application developer. As you will see from the Java code examples later in this chapter and in Appendix A, only in certain circumstances does the developer need to control the ORB directly. These occur, for example, when coding callback mechanisms, or when there is a need to register transient objects with the basic object adapter.

The Interface Definition Language (IDL)

One of the key factors in the success of CORBA is language independence. CORBA objects written in one language can send requests to objects implemented in a different language. Objects implemented in an object-oriented language such as Java or Smalltalk can talk to objects written in C or COBOL, and the converse .

Language independence is achieved through the use of a specification meta-language that defines the interfaces that an object (or a piece of legacy code wrappered to look like an object) presents to the outside world. As in any object-oriented system, a CORBA object can have its own private data and its own private methods. The specification of the public data and methods is the interface that the object presents to the outside world.

IDL is the language that CORBA uses to specify its objects. You do not write procedural code in IDL—its only use is to specify data, methods, and exceptions.

Each CORBA vendor supplies a compiler that translates IDL specifications into language code. Oracle8*i* JServer uses the `idl2java` compiler from Inprise (see "Miscellaneous Tools" on page 6-26**)**. The `idl2java` compiler translates your IDL interface specifications into Java classes, which are compiled by the Java compiler into byte codes to be loaded into the Oracle8*i* database for execution.

Note: The `idl2java` compiler accepts only ASCII characters. Do not use ISO Latin-1 or other non-ASCII NLS characters in IDL files.

Using IDL

Here is an example of a short IDL file. It is the IDL for the `HelloWorld` example (see "helloworld" on page A-4 for the complete example):

```
module hello {
  interface Hello {
    wstring helloWorld();
  };
};
```

The IDL consists of a *module*, which contains a group of usually related object interfaces. By default, the IDL compiler uses the module name to name a directory where the IDL compiler puts the Java classes that it generates, and this maps to a Java package.

This module has only a single interface: `Hello`. The `Hello` interface defines a single operation: `helloWorld`, which takes no parameters and returns a `wstring` (a wide string, which is mapped to a Java **String**).

> **Note:** IDL data and exception types, such the `wstring` shown in the preceding example, are not specified in this guide. Although some of the IDL to Java bindings are listed in this guide (for example see "IDL Types" on page 3-10), CORBA developers should have access to the OMG specifications for complete information about IDL and IDL types. See "For More Information" on page 3-42.

The module and interface names must be valid Java identifiers and also valid file names for your operating system. When naming interfaces and modules, remember that both Java and CORBA objects are portable, and that some operating systems are case sensitive and some are not, so be sure to keep names distinct in your project.

Nested Modules

You can nest modules. For example, an IDL file that specifies

```
module org {
  module omg {
     module CORBA {
       ...
     };
    ...
  };
   ...
};
```

would map to the Java package hierarchy `package org.omg.CORBA`.

Running the IDL Compiler

Assume that the HelloWorld IDL is saved in a file called `hello.idl`. When you run `idl2java` to compile the `hello` module, eight Java class files are generated and are put in a subdirectory named `hello` in the same directory as the IDL file:

```
% idl2java hello.idl
Traversing hello.idl
Creating: hello/Hello.java
Creating: hello/HelloHolder.java
```

```
Creating: hello/HelloHelper.java
Creating: hello/_st_Hello.java
Creating: hello/_HelloImplBase.java
Creating: hello/HelloOperations.java
Creating: hello/_tie_Hello.java
Creating: hello/_example_Hello.java
```

The ORB uses eight Java classes to invoke a remote object, pass and return parameters, and perform various other things. You can control the files generated, where they are put, and other aspects of IDL compiling (such as whether the IDL compiler generates comments in the Java files). See the complete description of the `idl2java` compiler in Chapter 6, "Tools".

Each of the eight files the compiler generates is described briefly below.

`Hello` — This is the interface file that specifies in Java what the interface to a Hello object looks like. In this case, the interface is:

```
package hello;
public interface Hello extends org.omg.CORBA.Object {
  public java.lang.String helloWorld();
}
```

Note that because the file is put in a `hello` directory, it takes the package spec from that name. All CORBA basic interface classes subclass, directly or indirectly, `org.omg.CORBA.Object`.

The server object developer must implement the methods in the interface. Typical of the examples in this guide is that the implementation class for an interface named `hello.java` would be named `helloImpl`, but this naming convention is not a requirement.

`HelloHolder` — The application uses the holder class when parameters in the interface operation are of types `out` or `inout`. Because the ORB passes Java parameters by value, special holder classes are necessary to provide for parameter return values.

`HelloHelper`	The helper classes contain methods that read and write the object to a stream, and cast the object to and from the type of the base class. For example, the helper class has a `narrow()` method that is used to cast an object to the appropriate type, as in the following code:

```
LoginServer lserver = LoginServerHelper.narrow
      (orb.string_to_object (loginIOR));
```

	Note that when you get an object reference using the JNDI `InitialContext lookup()` method, you do not have to call the helper `narrow()` method. The ORB calls it automatically for you.
`_st_Hello`	The generated files that have _st_ prefixed to the interface name are the *stub* files or client proxy objects. (`_st_` is a VisiBroker-specific prefix.) These classes are installed on the client that calls the remote object (the `hello` object, in this example). In effect, when a client calls a method on the remote object, it is really calling into the stub, which then performs the operations necessary to perform a remote method invocation. For example, it must marshall parameter data for transport to the remote host.
`_HelloImplBase`	Generated source files of the form `_<interfaceName>ImplBase` are the *skeleton* files. A skeleton file is installed on the server and communicates with the stub file on the client, in that it receives the message on the ORB from the client and upcalls to the server. The skeleton file also returns parameters and return values to the client. In earlier CORBA implementations, the skeleton files were named `_sk_<interfaceName>`, but this is now deprecated.
`HelloOperations` `_tie_Hello`	The server uses these two classes for Tie implementations of server objects. See "Using the CORBA Tie Mechanism" on page 3-36 for information about Tie classes.

`_example_Hello`	The `_example_<interfaceName>` class gives you an example of how you should implement the interface on the server. It provides the framework for the implementation code, leaving just the method implementation body blank. You can copy the example code to the directory where you will implement the `Hello` server object, rename it following your naming conventions (`HelloImpl.java` is used in the examples in this Guide), and just add the Java code to implement the methods.

IDL Interface Body

An IDL interface body contains the following kinds of declarations:

constants	The constant values that the interface exports.
types	Type definitions.
exceptions	Exception structures that the interface exports.
attributes	Any associated attributes exported by the interface.
operations	Operations are the methods that the interface supports.

IDL Types

This section gives a brief description of IDL datatypes and their mapping to Java datatypes. For more information about IDL types not covered here, see the CORBA specifications and the books cited in "For More Information" on page 3-42.

Basic Types

Mapping between IDL basic types and Java primitive types is straightforward. Table 3–1 shows the mappings, as well as possible CORBA exceptions that can be raised on conversion.

Table 3–1 IDL to Java Datatype Mappings

CORBA IDL Datatype	Java Datatype	Exception
`boolean`	`boolean`	
`char`	`char`	`CORBA::DATA_CONVERSION`
`wchar`	`char`	

Table 3–1 IDL to Java Datatype Mappings (Cont.)

CORBA IDL Datatype	Java Datatype	Exception
octet	byte	
string	java.lang.String	CORBA::MARSHAL
		CORBA::DATA_CONVERSION
wstring	java.lang.String	CORBA::MARSHAL
short	short	
unsigned short	short	
long	int	
unsigned long	int	
long long	long	
unsigned long long	long	
float	float	
double	double	

The IDL character type `char` is an 8-bit type, representing an ISO Latin-1 character that maps to the Java `char` type, which is a 16-bit unsigned element representing a Unicode character. On parameter marshalling, if a Java `char` cannot be mapped to an IDL `char`, a CORBA DATA_CONVERSION exception is thrown.

The IDL `string` type contains IDL chars. On conversion between Java `String`, and IDL `string`, a CORBA DATA_CONVERSION can be thrown. Conversions between Java strings and bounded IDL `string` and `wstring` can throw a CORBA MARSHALS exception if the Java `String` is too large to fit in the IDL string.

Constructed Types

Perhaps the most useful IDL constructed (aggregate) type for the Java developer is the struct. The IDL compiler converts IDL structs to Java classes. For example, the IDL specification:

```
module employee {
  struct EmployeeInfo {
    long empno;
    wstring ename;
    double sal;
  };
  ...
```

causes the IDL compiler to generate a separate Java source file for an `EmployeeInfo` class. It looks like this:

```
package employee;
final public class EmployeeInfo {
  public int empno;
  public java.lang.String ename;
  public double sal;
  public EmployeeInfo() {
  }
  public EmployeeInfo(
    int empno,
    java.lang.String ename,
    double sal
  ) {
    this.empno = empno;
    this.ename = ename;
    this.sal = sal;
  }
 ...
```

The class contains a public constructor with parameters for each of the fields in the struct. The field values are saved in instance variables when the object is constructed. Typically, these are passed by value to CORBA objects.

Collections

The two types of ordered collections in CORBA are *sequences* and *arrays*. An IDL sequence maps to a Java array with the same name. An IDL array is a multidimensional aggregate whose size in each dimension must be established at compile time.

The ORB throws a CORBA MARSHAL exception at runtime if sequence or array bounds are exceeded when Java data is converted to sequences or arrays.

IDL also generates a holder class for a sequence. The holder class name is the sequence's mapped Java class name with `Holder` appended to it.

The following IDL code shows how you can use a sequence of structs to represent information about employees within a department:

```
module employee {
  struct EmployeeInfo {
    long empno;
    wstring ename;
    double sal;
  };

  typedef sequence <EmployeeInfo> employeeInfos;

  struct DepartmentInfo {
    long deptno;
    wstring dname;
    wstring loc;
    EmployeeInfos employees;
  };
```

The Java class code that the IDL compiler generates for the `DepartmentInfo` class is:

```
package employee;
final public class DepartmentInfo {
  public int deptno;
  public java.lang.String dname;
  public java.lang.String loc;
  public employee.EmployeeInfo[] employees;
  public DepartmentInfo() {
  }
  public DepartmentInfo(
    int deptno,
    java.lang.String dname,
    java.lang.String loc,
    employee.EmployeeInfo[] employees
  ) {
    this.deptno = deptno;
    this.dname = dname;
    this.loc = loc;
    this.employees = employees;
  }
```

Notice that the sequence `employeeInfos` is generated as a Java array `EmployeeInfo[]`.

Specify an array in IDL as follows:

```
const long ArrayBound = 12;
typedef long larray[ArrayBound];
```

The IDL compiler generates this as:

```
public int[] larray;
```

When you use IDL constructed and aggregate types in your application, you must make sure to compile the generated `.java` files and load them into the Oracle8*i* database when the class is a server object. You should scan the generated `.java` files, and make sure that all required files are compiled and loaded. Study the `Makefile` (UNIX) or the `makeit.bat` batch file (Windows NT) of CORBA examples that define these types to see how the set of IDL-generated classes is compiled and loaded into the data server. A good example is "lookup" on page A-24.

Exceptions

You can create new user exception classes in IDL with the exception key word. For example:

```
  exception SQLError {
    wstring message;
  };
```

The IDL can declare that operations raise user-defined exceptions. For example:

```
interface employee {
  attribute name;
  exception invalidID {
    wstring reason;
  };
  ...
  wstring getEmp(long ID)
    raises(invalidID);
  };
};
```

CORBA System Exceptions

Mapping between OMG CORBA system exceptions and their Java form is also quite straightforward. These mappings are shown in Table 3–2.

Table 3–2 CORBA and Java Exceptions

OMG CORBA Exception	Java Exception
`CORBA::PERSIST_STORE`	`org.omg.CORBA.PERSIST_STORE`
`CORBA::BAD_INV_ORDER`	`org.omg.CORBA.BAD_INV_ORDER`
`CORBA::TRANSIENT`	`org.omg.CORBA.TRANSIENT`
`CORBA::FREE_MEM`	`org.omg.CORBA.FREE_MEM`
`CORBA::INV_IDENT`	`org.omg.CORBA.INV_IDENT`
`CORBA::INV_FLAG`	`org.omg.CORBA.INV_FLAG`
`CORBA::INTF_REPOS`	`org.omg.CORBA.INTF_REPOS`
`CORBA::BAD_CONTEXT`	`org.omg.CORBA.BAD_CONTEXT`
`CORBA::OBJ_ADAPTER`	`org.omg.CORBA.OBJ_ADAPTER`
`CORBA::DATA_CONVERSION`	`org.omg.CORBA.DATA_CONVERSION`
`CORBA::OBJECT_NOT_EXIST`	`org.omg.CORBA.OBJECT_NOT_EXIST`
`CORBA::TRANSACTIONREQUIRED`	`org.omg.CORBA.TRANSACTIONREQUIRED`
`CORBA::TRANSACTIONROLLEDBACK`	`org.omg.CORBA.TRANSACTIONROLLEDBACK`
`CORBA::INVALIDTRANSACTION`	`org.omg.CORBA.INVALIDTRANSACTION`

Getting by Without IDL

The Oracle8*i* JVM development environment offers the Inprise Caffeine tools, which enable development of pure Java distributed applications that follow the CORBA model. You can write your interface specifications in Java and use the `java2iiop` tool to generate CORBA-compatible Java stubs and skeletons.

Developers can also use the `java2idl` tool to code in pure Java, yet still have IDL available that can be shipped to customers who are using a CORBA server that does not support Java. This tool generates IDL from Java interface specifications. See Chapter 6, "Tools", for more information about `java2iiop` and `java2idl`.

A First CORBA Application

This section introduces the JServer CORBA application development process. It tells you how to write a simple but useful program that runs on a client system, connects to Oracle using IIOP, and invokes a method on a CORBA server object that is activated and runs inside an Oracle8*i* Java VM.

This section addresses the purely mechanical aspects of the development process. See "For More Information" on page 3-42 for references to documents on CORBA design.

The CORBA application development process has seven phases:

1. Design and write the object interfaces.
2. Generate stubs and skeletons, and other required support classes.
3. Write the server object implementations.
4. Use the client-side Java compiler to compile both the Java code that you have written, and the Java classes that were generated by the IDL compiler. Generate a JAR file to contain the classes and any other resource files that are needed.
5. Publish a name for the directly-accessible objects with the CosNaming service, so you can access them from the client program.
6. Write the client side of the application. This is the code that will run outside of the Oracle8*i* data server, on a workstation or PC.
7. Compile the client code using the JDK Java compiler.
8. Load the compiled classes into the Oracle8*i* database, using the `loadjava` tool and specifying the JAR file as its argument. Make sure to include all generated classes, such as stubs and skeletons. (Stubs are required in the server when the server object acts as a client to another CORBA object.)

The remainder of this section describes these steps in more detail, with IDL and Java code examples to illustrate the coding steps.

The first sample application asks the user for an employee number in the EMP table and returns the employee’s last name and current salary, or throws an exception if there is no employee in the database with that ID number.

Writing the IDL Code

The application requires only a single server-side object: some code that takes an ID number and queries the database for the other information about the employee.

The interface requires three things:

- an operation to query the database and return the information
- a data structure to hold the name and salary information
- an exception to be thrown back to the client if the employee is not found

The example defines an operation called `query` to get the information, uses an IDL `struct` to return the information, and defines an exception called `SQLError` to signal that no employee was found. Here is the IDL code:

```
module employee {

  struct EmployeeInfo {
    wstring name;
    long number;
    double salary;
  };

  exception SQLError {
    wstring message;
  };

  interface Employee {
    EmployeeInfo getEmployee (in long ID) raises (SQLError);
  };
};
```

This code specifies the three things listed above: a struct named `EmployeeInfo`, an operation or method named `getEmployee()`, and the `SQLError` exception.

Generate Stubs and Skeletons

Use the `idl2java` compiler to compile the interface description. Because there is no use of the Tie mechanism in this example, you can invoke the compiler with the `-no_tie` option. This means that two fewer classes are generated. The compiler generates the interface, helper, and holder classes for the three objects in the IDL file, as well as a stub and skeleton class for the `Employee` interface. (The 12th class is the example for the interface. See "Using IDL" on page 3-6 for more information about these classes.)

Compile the IDL as follows:

```
% idl2java -no_tie -no_comments employee.idl
```

Note: This section shows separate for each step of the process. Because developing a CORBA application includes many compilation, loading, and publishing steps, Oracle recommends that if you are working in a command-line oriented environment, you always use a makefile or a batch file to control the process. Or, you can use IDE products such as Oracle's JDeveloper to control the process.

Study the make or batch files that come with the CORBA programs on the CD for good examples.

Write the Server Object Implementation

For this example, you must implement the `Employee` interface. The `_example_Employee.java` file that the IDL compiler generates can provide a basis for the implementation. Here is the complete code that implements the interface:

```
package employeeServer;

import employee.*;
import java.sql.*;

public class EmployeeImpl extends _EmployeeImplBase {

  public EmployeeImpl() {
  }

  public EmployeeInfo getEmployee (int ID) throws SQLError {
    try {
      Connection conn =
          new oracle.jdbc.driver.OracleDriver().defaultConnection ();
      PreparedStatement ps =
          conn.prepareStatement ("select ename, sal from emp where empno = ?");
      try {
        ps.setInt (1, ID);
        ResultSet rset = ps.executeQuery ();
        if (!rset.next ())
          throw new SQLError ("no employee with ID " + ID);
        return new EmployeeInfo (rset.getString (1), ID, rset.getFloat (2));
      } finally {
        ps.close ();
      }
    } catch (SQLException e) {
      throw new SQLError (e.getMessage ());
    }
  }
}
```

This code uses the JDBC API to perform the query. Notice the use of a prepared statement to accommodate the variable in the WHERE clause of the query. See the *Oracle8i JDBC Developer's Guide and Reference* for more about Oracle8*i* JDBC. Also notice that when a JDBC `SQLException` is caught, the IDL-defined `SQLError` is thrown back to the client.

Write the Client Code

To access the server object you must be able to refer to it by name. In step 7 of this process, you publish the server object in the Oracle8*i* database. The client code looks up the published name, and activates the server object as a by-product of the look up. There are a number of other operations that go on when code such as that listed below looks up a published object. For example, the ORB on the server side is started and the client is authenticated using the environment properties supplied when the initial context object is created. See "IIOP Security" on page 4-40.

After getting parameters such as the name of the object to look up, an IIOP service name, and some authentication information such as the database username and password, the client code performs the following four steps:

1. Instantiates and populates a JNDI `InitialContext` object with the required connect properties. See "About JNDI" on page 4-6.
2. Invokes the `lookup()` method on the initial context, with a URL as a parameter that specifies the service name and the name of the object to be found. `lookup()` returns an object reference to the `Employee` CORBA server object. See "Looking Up an Object" on page 3-25 for more information.
3. Using the object reference returned by the `lookup()` method invokes the `getEmployee()` method on the object in the server. This method returns an `EmployeeInfo` class (derived from the IDL `EmployeeInfo` struct). For simplicity, an employee ID number is hard-coded as a parameter of this method invocation.
4. Prints the values returned by `getEmployee()` in the `EmployeeInfo` class.

```
import employee.*;
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";

// Step 1:
    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);

// Step 2:
    Employee employee = (Employee)ic.lookup (serviceURL + objectName);

// Step 3 (using SCOTT's employee ID number):
    EmployeeInfo info = employee.getEmployee (7788);

// Step 4:
    System.out.println (info.name + " " + info.number + " " + info.salary);
  }
}
```

When the client code runs, it should print the line

```
SCOTT 7788 3000.0
```

on the client system console.

Compiling the Java Source

You run the client-side Java byte code compiler to compile all the Java source that you have created, including the client and server object implementation that you wrote, as well as the Java sources for the classes that were generated by the IDL compiler.

For the preceding example, you must compile the following files:

- `employee/Employee.java`
- `employee/EmployeeHolder.java`
- `employee/EmployeeInfoHolder.java`
- `employee/EmployeeHelper.java`
- `employee/SQLErrorHolder.java`
- `employee/_EmployeeImplBase.java`
- `EmployeeImpl.java`
- `Client.java`

Other generated Java files are compiled following the dependencies that the Java compiler uses.

Oracle8*i* JServer supports the Java JDK compiler, release 1.1.6. You might be able to use other Java compilers, such as a compiler incorporated in an IDE, but only JDK 1.1.6 is supported for this release.

Load the Classes into the Database

CORBA server objects, such as the EmployeeImpl object created for this example, execute inside the Oracle8*i* database server. You must load them into the server so that they can be activated by the ORB as required. You must also load all dependent classes, such as IDL-generated Holder and Helper classes, and classes the server object uses, such as the `EmployeeInfo` class of this example.

Use the `loadjava` tool to load each of the server classes into the Oracle8*i* database. For the example in this section, issue the `loadjava` command in the following way:

```
% loadjava -oracleresolver -resolve -user scott/tiger
   employee/Employee.class employee/EmployeeHolder.class
   employee/EmployeeHelper.class employee/EmployeeInfo.class
   employee/EmployeeInfoHolder.class employee/EmployeeInfoHelper.class
   employee/SQLError.class employee/SQLErrorHolder.class
   employee/SQLErrorHelper.class employee/_st_Employee.class
   employee/_EmployeeImplBase.class employeeServer/EmployeeImpl.class
```

You do not load any client implementation classes or any other classes not used on the server side.

It is sometimes more convenient to combine the server classes into a JAR file, and simply use that file as the argument to the `loadjava` command. In this example, you could issue the command:

```
% jar -cf0 myJar.jar employee/Employee.class employee/EmployeeHolder.class \
   employee/EmployeeHelper.class employee/EmployeeInfo.class \
   employee/EmployeeInfoHolder.class employee/EmployeeInfoHelper.class \
   employee/SQLError.class employee/SQLErrorHolder.class \
   employee/SQLErrorHelper.class employee/_st_Employee.class \
   employee/_EmployeeImplBase.class employeeServer/EmployeeImpl.class
```

and then give the `loadjava` command as simply:

```
% loadjava -oracleresolver -resolve -user scott/tiger myJar.jar
```

Publish the Object Name

The final step in preparing the application is to publish the name of the CORBA server object implementation in the Oracle8*i* database. See "The Name Space" on page 3-24 for information about publishing and published objects.

For the example in this section, you can publish the server object using the `publish` command as follows:

```
% publish -republish -user scott -password tiger -schema scott
    -service sess_iiop://localhost:2481:ORCL
    /test/myEmployee employeeServer.EmployeeImpl employee.EmployeeHelper
```

This command specifies the following:

- `publish`—run the publish command
- `-republish`—overwrite any published object of the same name
- `-user scott`—scott is the username for the schema doing the publishing
- `-password tiger`—Scott's password
- `-schema scott`—the name of the schema in which to resolve classes
- `-service sess_iiop://localhost:2481:ORCL`—establishes the service name (see also "The Service Context Class" on page 4-29)
- `/test/myEmployee`—the name for the published object
- `employeeServer.EmployeeImpl`—the name of the class, loaded in the database, that implements the server object
- `employee.EmployeeHelper`—the name of the helper class

See "publish" on page 6-3 for more information about the `publish` command and its arguments.

Run the Example

To run this example, execute the client class using the client-side Java VM. 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 file `aurora_client.jar`

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

The following invocation of the JDK `java` command runs this example. The UNIX shell variable ORACLE_HOME might be represented as %ORACLE_HOME% on Windows NT and that JDK_HOME is the installation location of the Java Development Kit (JDK), version 1.1.6:

```
% java -classpath
.:$(ORACLE_HOME)/lib/aurora_client.jar:$(ORACLE_HOME)/jdbc/lib/classes111.zip:
$(ORACLE_HOME)/sqlj/lib/translator.zip:$(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
```

This example assumes that you invoke the client with four arguments on the command line:

- service name
- name of the published object to activate
- username
- password

From the `java` command you can see why it is almost always better to use a makefile or a batch file to build CORBA applications.

Locating Objects

One of the fundamental tasks that a CORBA programmer faces is discovering how to get a reference to a server object. The CORBA specifications permit a great deal of freedom to the implementer in this area.

As you saw in the example in the previous section, the Oracle8*i* solution is to publish non-transient objects in a Oracle8*i* database instance, using a CORBA *CosNaming* service. JServer provides a URL-based JNDI interface to CosNaming, to make it easy for clients written in Java to locate and activate published objects.

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 "sess_sh" on page 6-8 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 the place where most objects are published for the example programs in this guide. You can also 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. They are owned by SYS; therefore, to delete them, you would have to be connected in the session shell as SYS. The objects contained in `/etc` are:

```
deployejb    execute    loadjava    login    transactionFactory
```

The entries in the name space are actually represented by objects that are instances of the classes `oracle.aurora.AuroraServices.PublishingContext` and `oracle.aurora.AuroraServices.PublishedObject`. A publishing context represents a class that can contain other objects (a directory), and the `PublishedObject` class is 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, maintained in a separate table in the system tablespace. Each class or resource file can have a combination (union) of the following permissions:

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

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

You can set these permissions when the objects are loaded into the database, and can use the session shell tool to view and change object permissions. See "sess_sh" on page 6-8 for information about this tool.

Publishing means registering the object name in the database name service. The steps involved are as follows:

- Insert the name in the session namespace.
- Associate the name with the implementation class that was loaded.
- Provide the name of a helper class for the object.
- Assign permissions to the published name that determine who can modify, access, and execute the object.

Looking Up an Object

The JNDI `lookup()` method is the normal way that a client looks up an object whose name is published in the name space. When you invoke the `lookup()` method, you normally pass it a String parameter that specifies a URL containing the following:

- the service name
- the path name of the published object to look up

Service Name

The service name specifies a service that an IIOP presentation manages, and it represents a database instance. The format of the service URL is explained in "URL Syntax" on page 4-27. Briefly, the service name specifies the following components:

- a service
- the name of the host that manages the service presentation
- the port number of the listener for the target database instance on that host
- the system identifier (SID) for the database instance on the host

A typical example of a service name is `sess_iiop://localhost:2481:ORCL`, where `sess_iiop` is the service, `localhost` defaults to the host of the local database, 2481 is the default listener port for IIOP connections, and `ORCL` is the SID.

For more information about the service name, see "URL Syntax" on page 4-27.

Object name

The object name specifies the complete path name of the published object that you want to look up. For example: `/test/myServer`.

See "The JNDI InitialContext Class" on page 4-9 for further information about the `lookup()` method.

Activating ORBs and Server Objects

A CORBA application requires that an ORB be active on both the client system and the system running the server. In looking at the examples shown so far in this chapter, it is not obvious how the ORB is activated, either on the client or the server. This section presents more information about that topic.

Client Side

The client-side ORB is normally initialized as part of the processing that goes on when the client invokes the `lookup()` method on the JNDI `InitialContext` object that it instantiates.

If you need to get a reference to the client ORB, use the `init()` method on the ORB pseudo-object to get it, as shown in this statement:

```
org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init ();
```

The `init()` method invoked on the client with no parameters always returns a reference to the existing client ORB.

Server Side

The presentation that manages IIOP requests starts the ORB on the server when the session is created.

About Object Activation

Objects are activated on demand. When a client looks up an object, the ORB loads the object into memory and caches it. To activate the object, the ORB looks up the class by the fully-qualified class name under which the object was published. The class name is resolved in the schema defined at publication time, rather than the caller's schema. See the description of the command-line tool "publish" on page 6-3 for more information.

When the class is located, the ORB creates a new instance of the class, using `newInstance()`. For this reason, the no-argument constructor of a persistent object class must be **public**. If the class implements the `oracle.aurora.AuroraServices.ActivatableObject` interface (as determined by reflection), then the `_initializeAuroraObject()` message is sent to the instance. (See "Using the CORBA Tie Mechanism" on page 3-36 for an example that requires `_initializeAuroraObject()`).

There is no need for the server implementation to register persistent objects with the object adapter using a `boa.obj_is_ready()` call—the JServer ORB performs this automatically.

You register transient objects generated by other objects, such as persistent published objects, with the BOA using `obj_is_ready()`. For an example, see the `factory` demo in the `examples/corba/basic/factory` directory of the product CD.

CORBA Interceptors

Visibroker enables you to implement interceptors. The Visibroker documentation provides details for how to create them. When you want server interceptors in Oracle8*i*, you must create a PUBLIC synonym for the interceptor `init` class that implements `ServiceInit`.

Using SQLJ

You can often simplify the implementation of a CORBA server object by using Oracle8*i* SQLJ to perform static SQL operations. Using SQLJ statements results in less code than the equivalent JDBC calls and makes the implementation easier to understand and debug. This section describes a version of the example first shown in "A First CORBA Application" on page 3-16, but uses SQLJ rather than JDBC for the database access. Refer to the *Oracle8i SQLJ Developer's Guide and Reference* for complete information about SQLJ.

The only code that changes for this SQLJ implementation is in the `EmployeeImpl.java` file, which implements the `Employee` object. The SQLJ implementation, which can be called `EmployeeImpl.sqlj`, is listed below. You can contrast that with the JDBC implementation of the same object in "Write the Server Object Implementation" on page 3-18.

```
package employeeServer;

import employee.*;
import java.sql.*;

public class EmployeeImpl extends _EmployeeImplBase {
  public EmployeeInfo getEmployee (int ID) throws SQLError {
    try {
      String name = null;
      double salary = 0.0;
      #sql { select ename, sal into :name, :salary from emp
             where empno = :ID };
      return new EmployeeInfo (name, empno, (float)salary);
    } catch (SQLException e) {
      throw new SQLError (e.getMessage ());
    }
  }
}
```

The SQLJ version of this implementation is considerably shorter than the JDBC version. In general, Oracle recommends that you use SQLJ where you have static SQL commands to process, and use JDBC, or a combination of JDBC and SQLJ, in applications where dynamic SQL statements are required.

Running the SQLJ Translator

To compile the `EmployeeImpl.sqlj` file, you issue the following SQLJ command:

```
% sqlj -J-classpath
.:$(ORACLE_HOME)/lib/aurora_client.jar:$(ORACLE_HOME)/jdbc/lib/classes111.zip:
$(ORACLE_HOME)/sqlj/lib/translator.zip:$(ORACLE_HOME)/lib/vbjorb.jar:
$(ORACLE_HOME)/lib/vbjapp.jar:$(JDK_HOME)/lib/classes.zip -ser2class
  employeeServer/EmployeeImpl.sqlj
```

This command does the following:

- translates the SQLJ code into a pure Java file
- compiles the resulting `.java` source to get a `.class` file
- the `-ser2class` option translates SER files to `.class` files

The SQLJ translation generates two additional class files:

```
employeeServer/EmployeeImpl_SJProfile0
employeeServer/EmployeeImpl_SJProfileKeys
```

which you must also load into the database when you execute the `loadjava` command.

A Complete SQLJ Example

This example is available in complete form in the examples/corba/basic example directory, complete with a Makefile or Windows NT batch file so you can see how the example is compiled and loaded. See also "sqljimpl" on page A-9.

Migrating from JDK 1.1 to Java 2

Oracle8*i* JServer updated its ORB implementation to Visibroker 3.4, which is compatible with both JDK 1.1 and Java 2.

Note: All existing CORBA applications must regenerate their stubs and skeletons to work with 8.1.6. You must use the 8.1.6 tools when regenerating code from an IDL file.

Sun Microsystems's Java 2 contains an OMG CORBA implementation; JDK 1.1 did not contain an OMG CORBA implementation. Thus, when you imported the Inprise libraries and invoked the CORBA methods, it always invoked the Visibroker implementation. With the implementation being contained in Java 2, if you invoke the CORBA methods without any modifications—as discussed below—you will invoke the Sun Microsystems CORBA implementation, which can cause unexpected results.

The following lists the three methods for initializing the ORB on the client-side and recommendations for bypassing the Sun Microsystems CORBA implementation:

- JNDI Lookup—The setup for the lookup method is the same for both JDK 1.1 and Java 2. However, you must regenerate the stubs and skeletons.
- Aurora ORB Interface—The Aurora ORB provides an interface for initializing the ORB. If you do not use JNDI, your client initializes an ORB on its node to communicate with the ORB in the database. You can use an Aurora ORB on your client through this class.
- CORBA ORB Interface—If you want to use OMG's CORBA ORB interface, you must set a few properties to ensure you are accessing the correct implementation. If you do not wish to use the Aurora ORB on your client, you can use the pure CORBA interfaces. However, you must set up your environment to direct your calls to the correct implementation.

JNDI Lookup

If you are using JNDI on the client to access CORBA objects that reside in the server, no code changes are necessary. However, you must regenerate your CORBA stubs and skeletons.

Aurora ORB Interface

If your client environment uses JDK 1.1, you do not need to change your existing code. However, you must regenerate your stubs and skeletons.

If your client environment has been upgraded to Java 2, you can initialize the ORB through the `oracle.aurora.jndi.orb_dep.Orb.init` method. This method guarantees that when you initialize the ORB, it will initialize only a single ORB instance. That is, if you use the Java 2 ORB interface, it returns a new ORB instance each time you invoke the `init` method. Aurora's `init` method initializes a singleton ORB instance. Each successive call to `init` returns an object reference to the existing ORB instance.

In addition, the Aurora ORB interface manages the session-based IIOP connection.

oracle.aurora.jndi.orb_dep.Orb Class There are several `init` methods, each with a different parameter list. The following describes the syntax and parameters for each `init` method.

Note: The returned class for each `init` method is different. You can safely cast the `org.omg.CORBA.ORB` class to `com.visigenic.vbroker.orb.ORB`.

No Parameters

If you execute the `ORB.init` method that takes no parameters, it does the following:

- If no ORB instance exists, it creates an ORB instance and returns its reference to you.
- If an ORB instance exists, it returns the ORB reference to you.

Syntax

```
public com.visigenic.vbroker.orb.ORB init();
```

Providing ORB Properties

If you execute the `ORB.init` method that takes the ORB properties as the only parameter, it does the following:

- If no ORB instance exists, it creates an ORB instance, taking into account the properties argument, and returns its reference to you.
- If an ORB instance exists, it returns the ORB reference to you.

Syntax

```
public org.omg.CORBA.ORB init(Properties props);
```

Providing Input Arguments and ORB Properties

If you execute the `ORB.init` method that takes the ORB properties and ORB command-line arguments, it always creates an ORB instance and returns the reference to you.

Syntax

```
public org.omg.CORBA.ORB init(String[] args, Properties props);
```

Parameter	Description
Properties props	ORB system properties.

Parameter	Description
String[] args	Arguments that are passed to the ORB instance.

Example 3–1 Using Aurora ORB init method

The following example shows a client instantiating an ORB using the Aurora Orb class.

```
// Create the client object and publish it to the orb in the client
// Substitute Aurora's Orb.init for OMG ORB.init call
// old way: org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init ();
com.visigenic.vbroker.orb.ORB orb = oracle.aurora.jndi.orb_dep.Orb.init();
```

Providing ORB Properties with Username, Password, and Role

If you execute the `ORB.init` method that provides the ORB properties, username, password, and role as parameters, it does the following:

- If no ORB instance exists, it creates an ORB instance and returns its reference to you.
- If an ORB instance exists, it returns the ORB reference to you.

You would use this method when your client chooses to not use JNDI for ORB initialization and it receives a reference to an existing object from another client. To access an active object within a session, the new client must authenticate itself to the database in one of two ways:

- If SSL_CREDENTIALS is requested, provide the username, password, and role in the `init` method parameters. Then, when you invoke a method on the supplied object reference, the username, password, and role are passed implicitly on the first message to authenticate the client to the database.
- If the login protocol is requested, through either SSL_LOGIN or NON_SSL_LOGIN, the first client must pass object references to both the login object and the destination object. The second client authenticates itself by providing the username, password, and role on the `authenticate` method of the login object. Then, it executes any method on the object.

This method is how a second client invokes an active object in an established session.

Syntax

```
public org.omg.CORBA.ORB init(String un, String pw, String role,
                         boolean ssl, java.util.Properties props);
```

Parameter	Description
String un	The username for client-side authentication.
String pw	The password for client-side authentication.
String role	The role to use after logging on.
Boolean ssl	If true, SSL is enabled for the connection. If false, a NON-SSL connection is used.
Properties props	Properties that are used by the ORB.

CORBA ORB Interface

If you have implemented a pure CORBA client—that is, you do not use JNDI—you must set the following properties before the ORB initialization call. These properties direct the call to the Aurora implementation, rather than the Java 2 implementation. This ensures the behavior that you expect. The behavior expected from Visibroker is as follows:

- Even if you invoke `ORB.init` more than once, JServer creates only a single ORB instance. If you do not set these properties, be aware that each invocation of `ORB.init` will create a new ORB instance.
- The session IIOP connection is managed correctly.
- Callbacks from the server are managed correctly.

Property	Assign Value
org.omg.corba.ORBClass	com.visigenic.vbroker.orb
org.omg.corba.ORBSingletonClass	com.visigenic.vbroker.orb

Example 3–2 Assigning Visibroker values to OMG properties

The following example shows how to set up the OMG properties for directing the OMG CORBA `init` method to the Visibroker implementation.

```
System.getProperties().put("org.omg.CORBA.ORBClass",
                           "com.visigenic.vbroker.orb.ORB");
System.getProperties().put("org.omg.CORBA.ORBSingletonClass",
                           "com.visigenic.vbroker.orb.ORB");
```

Or you can set the properties on the command line, as follows:

```
java -Dorg.omg.CORBA.ORBClass=com.visigenic.vbroker.orb.ORB
     -Dorg.omg.CORBA.ORBSingletonClass=com.visigenic.vbroker.orb.ORB
```

Backwards Compatibility with 8.1.5

The tools provided with Oracle8*i*, such as `publish`, have been modified to work with either a JDK 1.1 or Java 2 environment. However, any code that has been generated or loaded with the 8.1.5 version of any tool, will not succeed. Make sure that you always use the 8.1.6 version of all tools. This rule applies to your CORBA stubs and skeletons. You must regenerate all stubs and skeletons with the 8.1.6 IDL compiler.

CORBA Callbacks

This section describes how a CORBA server object can call back to a client. The basic technique that is shown in this example is the following:

- Write a client object that runs on the client side and contains the methods the called-back-to object performs.
- Implement a server object that has a method that takes a reference to the client callback object as a parameter.
- In the client code:
  - Instantiate the client callback object.
  - Register it with the BOA.
  - Pass its reference to the server object that calls it.
- In the server object implementation, perform the callback to the client.

Note: Oracle8*i* currently does not support SSL security for callouts and callbacks.

IDL

The IDL for this example is shown below. There are two separate IDL files: `client.idl` and `server.idl`:

```
/* client.idl */
module client {
  interface Client {
    wstring helloBack ();
  };
};

/* server.idl */
#include <client.idl>

module server {
  interface Server {
    wstring hello (in client::Client object);
  };
};
```

Note that the server interface includes the interface defined in `client.idl`.

Client Code

The client code for this example must instantiate the client-side callback object and register it with the BOA so that it can be accessed by the server. The code performs the following steps to do this:

- Invokes the `init()` method, with no parameters, on the ORB pseudo-object. This returns a reference to the existing client-side ORB.
- Uses the ORB reference to initialize the BOA.
- Instantiates a new client object.
- Registers the client object with the client-side BOA.

The code to perform these steps is as follows:

```
org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init ();
org.omg.CORBA.BOA boa = orb.BOA_init ();
ClientImpl client = new ClientImpl ();
boa.obj_is_ready (client);
```

Finally, the client code calls the server object, passes it a reference to the registered client-side callback object, and prints its return value, as follows:

```
System.out.println (server.hello (client));
```

Callback Server Implementation

The implementation of the server-side object is very simple:

```
package serverServer;

import server.*;
import client.*;

public class ServerImpl extends _ServerImplBase {
  public String hello (Client client) {
    return "I Called back and got: " + client.helloBack ();
  }
}
```

The server simply returns a string that includes the string return value from the callback.

Callback Client-Server Implementation

The client-side callback server is implemented like this:

```
package clientServer;

import client.*;

public class ClientImpl extends _ClientImplBase {
  public String helloBack () {
    return "Hello Client World!";
  }
}
```

The client-side object is just like any other server object. But in this callback example it is running in the client ORB, which can be running on a client system, not necessarily running inside an Oracle8*i* database server.

Printback Example

Among the CORBA examples shipped on the CD there is a very interesting variant of the callback example called `printback`. This example shows how a server object can call back to a client to print strings from the server on the client's console. You can use code like this for debugging a running server object.

Using the CORBA Tie Mechanism

There is only one special consideration when you use the CORBA Tie, or delegation, mechanism rather than the inheritance mechanism for server object implementations. In the Tie case, you must implement the `oracle.aurora.AuroraServices.ActivatableObject` interface. This interface has a single method: `_initializeAuroraObject().`

(Note that earlier releases of the Oracle8*i* ORB required you to implement this method for all server objects. For this release, its implementation is required only for Tie objects.)

The implementation of `_initializeAuroraObject()` for a tie class is typically:

```
import oracle.aurora.AuroraServices.ActivatableObject;
...
public org.omg.CORBA.Object _initializeAuroraObject () {
  return new _tie_Hello (this);
...
```

where `_tie_<interface_name>` is the tie class generated by the IDL compiler.

You must also always include a public, parameterless constructor for the implementation object.

See the `tieimpl` example in the CORBA examples set for a complete example that shows how to use the Tie mechanism. See also "tieimpl" on page A-45 for the code.

Interoperability with non-Oracle ORBs

You can interoperate with Oracle8*i* from a client that uses another vendor's ORB. To do so, the vendor must provide the functionality that Oracle8*i* uses by being part of the database: functions such as session-based connections, extended CosNaming functions, and the login protocol. To provide this functionality, your ORB vendor must work with Oracle's Product Management to provide libraries for you.

All client-side functionality has been packaged into aurora_client.jar. This JAR file has been broken into two JAR files for interoperating with your ORB vendor:

- `aurora_orbindep.jar`—includes ORB independent features, such as JNDI
- `aurora_orbdep.jar`—includes Oracle ORB dependent functionality, such as session-based communication, the login protocol, and security context

Your ORB vendor needs to provide you the `aurora_orbdep.jar` file. Thus, you include their `aurora_orbdep.jar` file and the Oracle-provided `aurora_orbindep.jar` file to replace `aurora_client.jar`.

Note: If you do not remove `aurora_client.jar` file from your CLASSPATH, you will be using Oracle's classes, instead of your ORB vendor's classes.

The aurora_orbdep.jar includes the following functionality:

Function	Description
login	The login protocol performs the challenge / response protocol for authenticating the client to the database. See "IIOP Security" on page 4-40 for more information.
bootstrap	The boot service obtains key services, such as CosNaming.
extended CosNaming	The Aurora ORB extended CosNaming to automatically instantiate an object upon first lookup.
Session IIOP	Session IIOP is implemented to allow one client connect to more than a single IIOP session at the same time. See "Configuring CORBA and EJB in JServer" on page 4-11 for more information.
Credentials	The security context interceptor for the credential type of authentication.

Java Client using Oracle ORB

You perform the following if you choose to use the Oracle-provided ORB on your client:

1. Put `aurora_client.jar` in a directory that exists in the CLASSPATH.
2. Compile and run your CORBA application.

Java Client using non-Oracle ORB

You perform the following if you choose to use another vendor's ORB on your client:

1. Put `aurora_orbindep.jar` in a directory that exists in the CLASSPATH.
2. Contact your ORB vendor to receive their `aurora_orbdep.jar`.
3. Put their `aurora_orbdep.jar` in a directory that exists in the CLASSPATH.
4. Compile and run your CORBA application.

Note: If you do not remove `aurora_client.jar` file from your CLASSPATH, you will be using Oracle's classes, instead of your ORB vendor's classes.

C++ Client Interoperability

With C++ clients, the ORB vendor must provide the `aurora_client.jar` file functionality in shared libraries. The vendor will make use of Oracle-provided C++ login protocol for authentication. All clients are required to authenticate themselves to the database. One of the methods for authenticating is through the login protocol.

The login protocol is an Oracle-specific design, used for logging in to a database by providing a username and password to authenticate the client. The following example shows how to write a sample C++ CORBA client to Oracle8*i*. This example uses the Visigenics C++ ORB for its client-side ORB.

Example 3–3 C++ client using login protocol to authenticate

The following C++ client uses the Visigenics C++ ORB for the client-side ORB. Your implementation can be different, depending on the type of ORB you use.

```
#include <Login.h>
#include <oracle_orbdep.h>

// set up host, port, and SID
char *sid = NULL;
char *host = argv[1];
int port = atol(argv[2]);
if(argc == 4) sid = argv[3];

// set up username, password, and role
wchar_t *username = new wchar_t[6];
username[0] = 's';
username[1] = 'c';
username[2] = 'o';
username[3] = 't';
username[4] = 't';
username[5] = '\0';

wchar_t *password = new wchar_t[6];
password[0] = 't';
password[1] = 'i';
password[2] = 'g';
```

```
password[3] = 'e';
password[4] = 'r';
password[5] = '\0';

wchar_t *role = new wchar_t[1];
role[0] = '\0';

// Get the Name service Object reference
AuroraServices::PublishingContext_ptr rootCtx = NULL;

// Contact Visibroker's boot service for initializing
rootCtx = VisiCppBootstrap::getNameService (host, port, sid);

// Get the pre-published login object reference
AuroraServices::PublishedObject_ptr loginPubObj = NULL;
AuroraServices::LoginServer_ptr serv = NULL;
CosNaming::NameComponent *nameComponent = new CosNaming::NameComponent[2];

nameComponent[0].id = (const char *)"etc";
nameComponent[0].kind = (const char *)"";
nameComponent[1].id = (const char *)"login";
nameComponent[1].kind = (const char *)"";

CosNaming::Name *name1 = new CosNaming::Name(2, 2, nameComponent, 0);

// Lookup this object in the Name service
CORBA::Object_ptr loginCorbaObj = rootCtx->resolve (*name1);

// Make sure it is a published object
loginPubObj = AuroraServices::PublishedObject::_narrow (loginCorbaObj);

// create and activate this object (non-standard call)
loginCorbaObj = loginPubObj->activate_no_helper ();
serv = AuroraServices::LoginServer::_narrow (loginCorbaObj);

// Create a client login proxy object and authenticate to the DB
oracle_orbdep *_visi = new oracle_orbdep(serv);
Login login(_visi);
boolean res = login.authenticate(username, password, role);
```

Debugging Techniques

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

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

   Debug in a single address space, on a client system. Use of an IDE for client or server debugging is optional, though highly desirable.

2. Use Oracle8i trace files.

   The output of `System.out.println()` in the Oracle8*i* ORB goes to the server trace files. The directory for trace files is a parameter specified in the INITSID.ORA 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.

3. 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.

4. Use "printback" to redirect System.out.

   You can use the technique demonstrated in the example program "printback" on page A-37 to redirect `System.out` and `System.err println` to the client system console.

Perhaps the best way to develop and debug Java / CORBA code is to use either the second or third technique described above, then deploy into the Oracle8*i* ORB.

For More Information

This section lists some resources that you can access to get more information about CORBA and about CORBA application development using Java.

Books

The ORB and some of the CORBA services supplied with Oracle8*i* JServer are based on VisiBroker for Java code licensed from Inprise. *Programming with VisiBroker*, by D. Pedrick et al. (John Wiley and Sons, 1998), provides both an introduction to CORBA development from the VisiBroker point of view and an in-depth look at the VisiBroker CORBA environment.

Client/Server Programming with Java and CORBA, by R. Orfali and D. Harkey (John Wiley and Sons, 1998), covers CORBA development in Java. This book also uses the VisiBroker implementation for its examples.

You should be aware that the examples published in both of these books require some modification to run in the Oracle8*i* ORB. It is better to start off using the examples in the Appendices to this Guide, which are more extensive than the examples in the books cited and demonstrate all the features of Oracle8*i* CORBA. See also Appendix C, "Comparing the Oracle8i JServer and VisiBroker VBJ ORBs" for a discussion of the major differences between VisiBroker for Java and the Oracle8*i* implementation.

URLs

You can download specifications for CORBA 2.0 and for CORBA services from links available at the following web site:

```
http://www.omg.org/library/downinst.html
```

Documentation on Inprise’s VisiBroker for Java product is available at:

```
http://www.inprise.com/techpubs/visibroker/visibroker33/
```

4 Connections and Security

This chapter describes in detail how both CORBA and EJB 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:

Connection Basics

The examples in Chapter 3, "Developing CORBA Applications" and Chapter 2, "Enterprise JavaBeans" showed how to connect to Oracle, start a database server session, and activate a CORBA server object or an EJB, using a single URL specification. In the client examples, connection and object look up was done as follows:

```
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. myHello hello =
    (myHello) ic.lookup("sess_iiop://localhost:2481:ORCL/test/myHello");
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 4-27 for a discussion of the URL syntax.)
- Line 8 invokes a method on the object.

In line 7 above, when a client looks up an object, the client and server are doing a lot of things automatically:

- On the `lookup()` invocation, a session IIOP connection is made 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.
- Using the CosNaming service, the client locates the published object `/test/myHelloServer` in the session name space.
- On a client method invocation, the server activates the object and registers it with the basic object adapter (BOA).

- The client-side ORB narrows the object to the correct type, using the helper class published along with `/test/myHello`.

When the object reference is returned, the client can invoke a method such as `helloWorld()` on the activated, narrowed object, as in line 8 above. This example shows a CORBA server object being looked up and activated, but a similar set of steps, including the narrowing, occurs when an EJB is activated through its home interface.

The remainder of this chapter describes the connection, service and session context establishment, and authentication steps. Code examples are provided to demonstrate how to control session invocation in a much finer-grained way than in the basic example above.

This chapter also describes different types of client authentication in addition to the NON_SSL_LOGIN method that the basic examples use. This chapter also discusses other aspects of security.

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 in two ways:

* Using the `-grant` option with the `publish` tool.
* 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 "explicit" on page A-62 for an example that access the `Login` object explicitly.

Services

In networking, the *presentation layer* is responsible for making sure that data is represented in a format that the application and session layers can accommodate. In Oracle, *presentation* can refer to a service protocol that accepts incoming network requests and activates routines in the database kernel layer or in the Java VM to process the requests.

Earlier versions of the Oracle database server had a single service—the two-task common (TTC) layer. This is the service that processes incoming Net8 requests for database SQL services from Oracle tools (such as SQL*Plus), and customer-written applications (using Forms, Pro*C, or the OCI).

In addition to TTC support, Oracle8*i* JServer supplies session IIOP service, implemented by the class `oracle.aurora.server.SGiopServer`, which handles TCP/IP requests routed to the service entrypoint by the listener and dispatcher. The IIOP service is capable of starting, controlling, and terminating

Oracle8*i* database *sessions*, in the same way that an incoming TTC request from a tool such as SQL*Plus is capable of starting and terminating a database session.

When using the Oracle8*i* JServer tools, especially when developing EJB and CORBA applications, it is very important to distinguish the two service types: TTC and IIOP.

Tools such as `publish`, `deployejb`, and the session shell access CORBA objects, and so must connect using an IIOP port. In addition, EJB and CORBA clients, or distributed objects acting in a client role, must use an IIOP port when sending requests to Oracle.

On the other hand, tools such as `loadjava` and `dropjava` connect using a TTC port.

Figure 4–1 TTC and IIOP Services

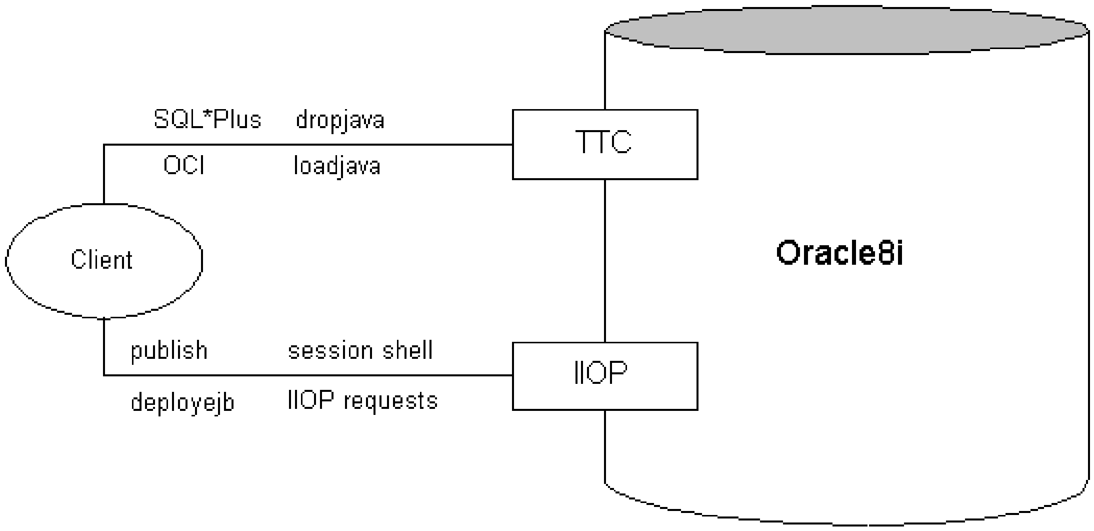


Figure 4–1 shows which tools and requests use TTC and which use IIOP database ports. 1521 is the default port number for TTC, and 2481 is the default for IIOP.

The session IIOP service embeds a session identifier in object references. This allows a single client to access multiple sessions, which would be impossible if there were no concept of a session identifier in object references.

The session IIOP service uses the foundation provided by the Oracle8*i* multi-threaded server to provide very high application scalability. See the *Oracle8i*

Java Developer's Guide for introductory information about application design and scalability. IIOP is discussed in greater detail in "About the Session IIOP Protocol" on page 4-10.

About JNDI

Clients use the *Java Naming and Directory Interface* (JNDI) interface to look up published objects in the session name space. JNDI is an interface supplied by Sun Microsystems that gives the Java application developer a way to access name and directory services. In addition to the API used by the application developer, some of whose classes and methods are described in this section, there is also a JNDI *Service Provider Interface* (SPI). Oracle8*i* JServer has implemented a SPI to the OMG CosNaming service, which provides the access to the published object name space.

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`.

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
```

It is also possible to access the session name space without using JNDI. See "Using Certificates for Client Authentication" on page 4-50 for a Java example that does not use JNDI.

The JNDI Context Interface

`Context` is an interface in the `javax.naming` package. All Oracle8*i* EJB and CORBA clients that use JNDI methods to look up and activate server objects must import this interface.

The `javax.naming.Context` interface forms the basis for the JNDI operations that you use to manage services and sessions in the Oracle8*i* ORB. This class is fully documented in the standard JNDI javadoc from Sun Microsystems. See `http://java.sun.com/products/jndi/index.htm` to obtain this documentation.

This section documents only the `Context` variables and methods that are most frequently used in Oracle8*i* CORBA and EJB application development.

Connecting Using JNDI

Before you can use JNDI to connect your client program to an Oracle8*i* server, you must set up an environment for the JNDI context. You can use a hash table or a properties list for the environment. 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. The remaining properties that you can set are for authentication. They are:

- `javax.naming.Context.SECURITY_PRINCIPAL`
- `javax.naming.Context.SECURITY_CREDENTIALS`
- `javax.naming.Context.SECURITY_ROLE`
- `javax.naming.Context.SECURITY_AUTHENTICATION`

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 4-48 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`.

See "Configuring CORBA and EJB in JServer" on page 4-11 for more information about configuration.

Context Methods

The Context interface contains a number of methods that the CORBA and EJB application developer will use. The methods required have been implemented in the `ServiceCtx` and `SessionCtx` classes that implement methods in the `Context` interface.

The JNDI InitialContext Class

`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)`

passing it a hashtable that has the environment information described in "Connecting Using JNDI" 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);
```

Method

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 create a new service context, specifying in the URL the service identifier. 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");
```

IIOP Services and Sessions

This section describes the Oracle8*i* IIOP services, session IIOP, and the use of the Oracle8*i* session for EJB and CORBA applications.

About the Session IIOP Protocol

The industry-standard CORBA does not support the concept of sessions, but the session concept is fundamental to the Oracle8*i* database and the JServer ORB. Because objects are activated within a session, the ORB and the database need a means to be able to distinguish objects within the same server process based on the sessions in which they are activated. You can activate objects within the existing session or another session. Session IIOP service provides a way for applications and server objects to distinguish sessions as well as services.

The session IIOP service introduces a new component tag, `SessionIIOP`, inside the IIOP profile (TAG_INTERNET_IOP—the OMG component tag for Oracle session IIOP is 0x4f524100). The session IIOP component tag has 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, and discussions are underway for other ORB vendors to support it as well.

Session Routing

When a client makes an IIOP connection to the database, the server code must decide 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.

With session IIOP the server is able to decide, on the basis of the object being activated, whether to activate a new session or route to an existing session, depending on the absence or presence of a session ID component tag inside the IIOP profile.

Configuring CORBA and EJB in JServer

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.

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 4-42 for information on SSL.
- See "Configuring SSL for EJB and CORBA" on page 4-21 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:

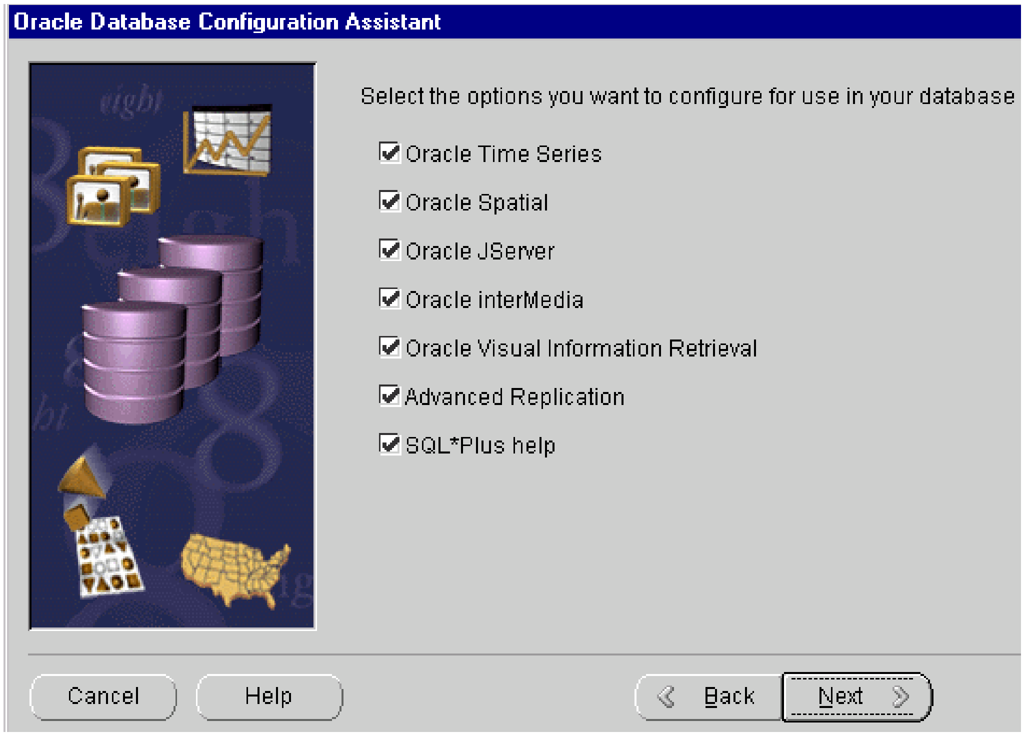
```
session_iiop://<hostname>/:<portnumber>/:<SID | service_name>
```

Oracle8*i* Custom Installation

If, within a Custom install, you choose the JServer option (as shown in Figure 4–2), 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 4-12.

Figure 4–2 Choosing the JServer option


Oracle Database Configuration Assistant

Select the options you want to configure for use in your database

- [✓] Oracle Time Series
- [✓] Oracle Spatial
- [✓] Oracle JServer
- [✓] Oracle interMedia
- [✓] Oracle Visual Information Retrieval
- [✓] Advanced Replication
- [✓] SQL*Plus help

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.

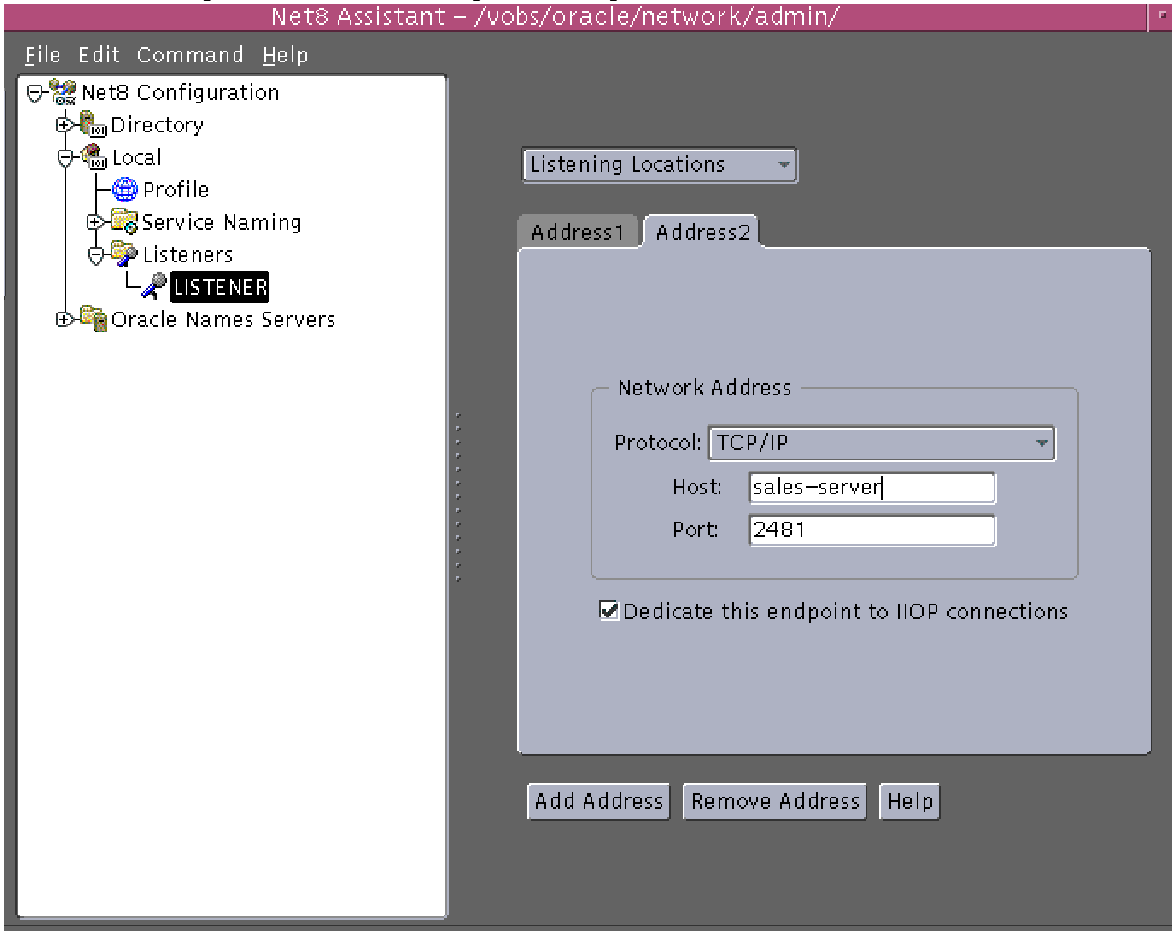
Net8 Assistant

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 4–3.

Figure 4–3 IIOP Listening Port Configuration



Network Address

Protocol: TCP/IP

Host: sales–server

Port: 2481

☑ Dedicate this endpoint to IIOP connections

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 4-13

2. Configure the listener for IIOP through the Net8 Assistant. These steps are described in "Net8 Assistant" on page 4-15.

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 4-15 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

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.
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)"
```

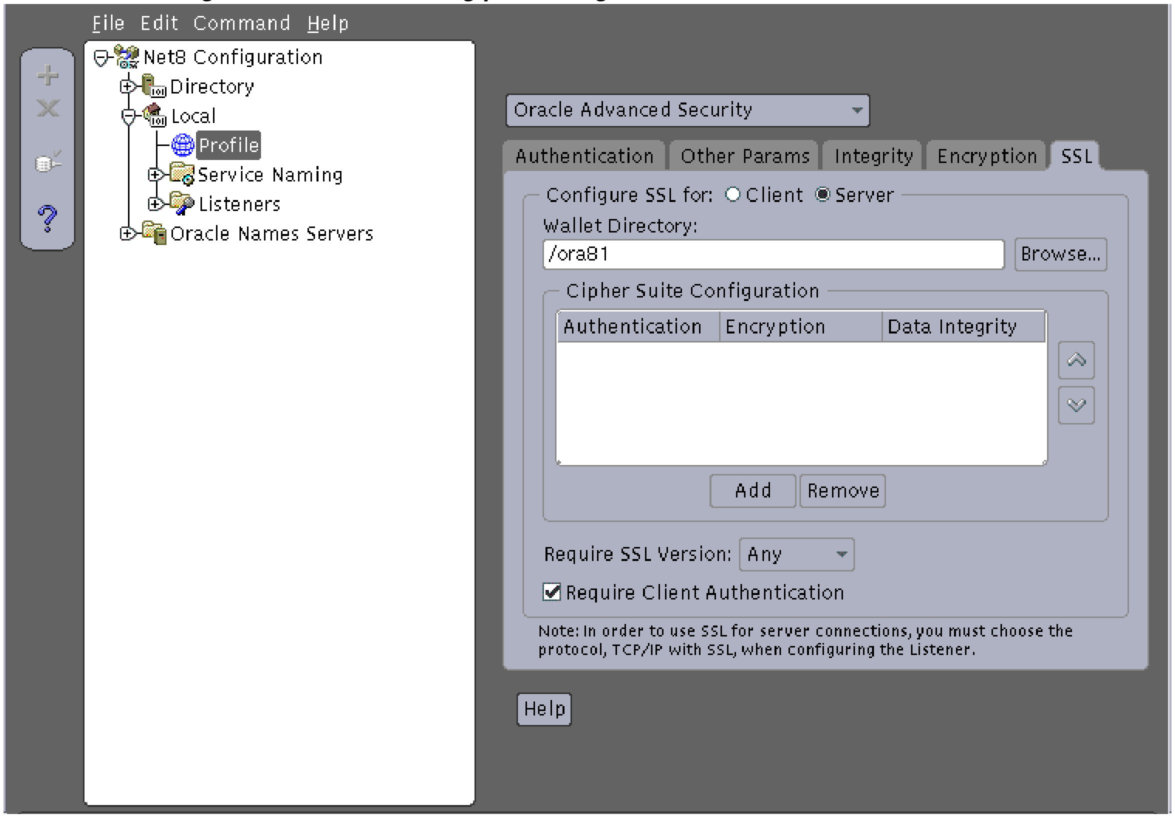
Configure the Wallet Location through Net8 Assistant

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 4–4.

Figure 4–4 IIOP listening port configuration



Configure SSL for: ○ Client ● Server

Wallet Directory: /ora81

Require SSL Version: Any

☑ Require Client Authentication

Note: In order to use SSL for server connections, you must choose the protocol, TCP/IP with SSL, when configuring the Listener.

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)))
```

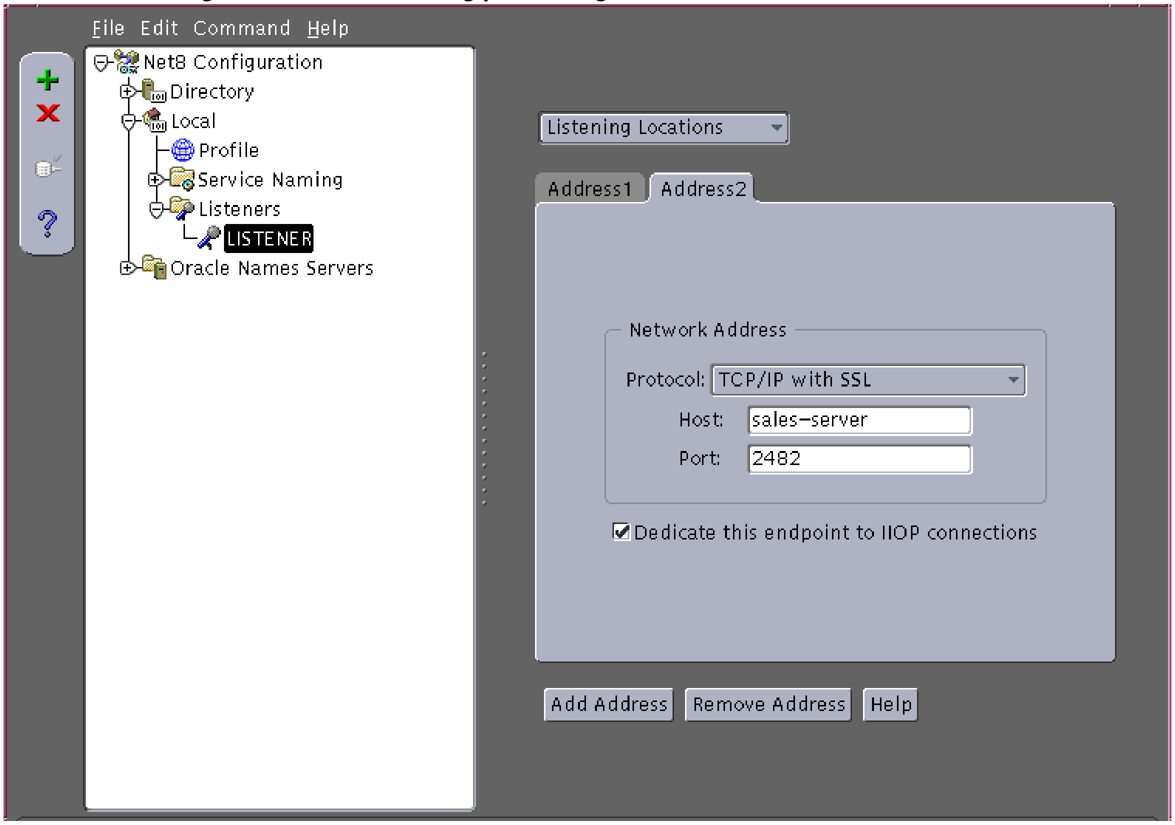
The *Oracle Advanced Security Administrator's Guide* describes how to set up the SSL wallet with the appropriate certificates.

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 4–3.

Figure 4–5   IIOP listening port configuration



Protocol: TCP/IP with SSL
Host: sales–server
Port: 2482
☑ Dedicate this endpoint to IIOP connections

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.

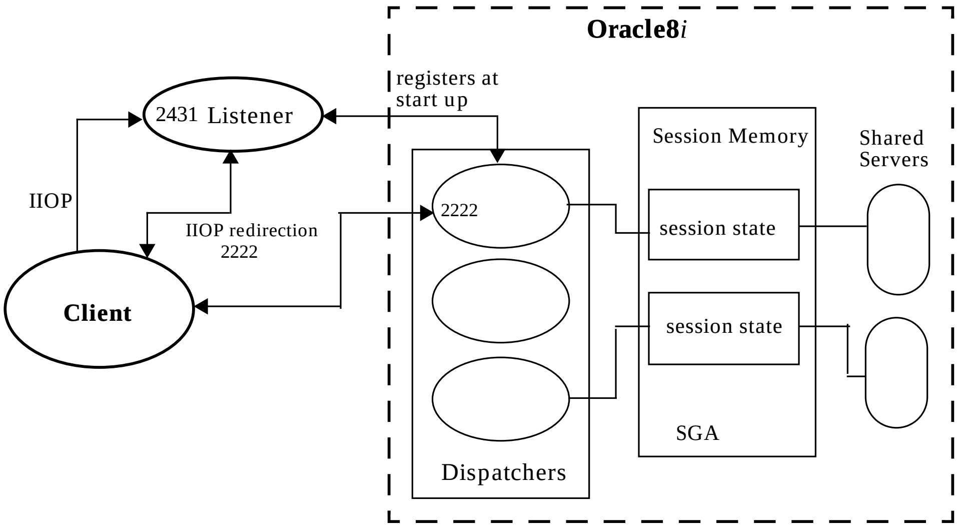
This places the following in the `listener.ora` file: it modifies the listener to specify TCPS —instead of TCP—as the protocol with port number 2482. The following shows an example of an SSL-enabled listener on the `sales-server` host.

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

Database Listeners and Dispatchers

When the listener receives a request for an IIOP connection from a client, it assigns an IIOP dispatcher to the client request, and sends an IIOP reply to ask the client to reconnect to the dispatcher. Figure 4–6 shows the interaction between the listener and the dispatchers, and also illustrates how an Oracle8*i* ORB session is activated.

Figure 4–6 Listener/Dispatcher Interaction



When a shared server services a new IIOP connection, it first creates a new database session for it 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.

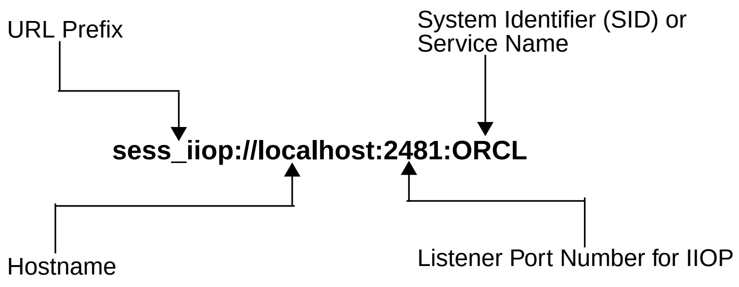
Further IIOP messages from the same client are routed directly to the existing session and handled similarly by the ORB.

When you configure a listener to accept both Net8 and IIOP connections, there is no need to distinguish between session IIOP and "standard" IIOP. The listener handles both on the same port. However, you do need a separate port for secure socket layer (SSL) connections. See "Using the Secure Socket Layer" on page 4-42 for more information about connecting using IIOP and SSL.

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 4–7.

Figure 4–7 Service URL


URL Prefix
System Identifier (SID) or Service Name
sess_iiop://localhost:2481:ORCL
Hostname
Listener Port Number for IIOP

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 name space, 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.

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 the ORB JavaDoc on the distribution CD for detailed documentation of these classes. The most relevant methods and variables are also described below.)

The Service Context Class

Oracle provides a service context class that 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 4–1 shows these.

Table 4–1 ServiceCtx Public Variables

String Name	Value
DEFAULT_SESSION	":default"
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"
THIS_SERVER	":thisServer"
THIS_SESSION	":thisSession"
Integer Name	**Integer Constructor**
SESS_IIOP	new Integer(2)

Table 4–1 ServiceCtx Public Variables (Cont.)

String Name	Value
IIOP	new Integer(1)

Methods

This section documents the public methods in this class that CORBA and EJB application developers can use.

`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" on page A-79 for a usage example.

`public synchronized SessionCtx login()`

`login()` authenticates the caller using the properties in the initial context environment and then activates a new session and returns the session context. The returned object is narrowed to the appropriate type.

Throws `javax.naming.NamingException`.

`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.`

`public Object _lookup(String string)`

`_lookup()` looks up a published object in the database instance associated with the service context and either returns the object, or throws `javax.naming.NamingException.` Unlike with `lookup()`, the object is not activated.

The Session Context Class

Oracle provides a session context class, `SessionCtx`, that extends the JNDI `Context` class. 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.

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

In the simple cases, a client starts a new server session implicitly when it activates a server object, such as an EJB or a CORBA server object. But Oracle8*i* also gives you the ability to control session start-up explicitly, either from the client or from a server object.

Starting a New Session

In general, when you look up a published object using the URL notation and you specify a hostname and port, then the object is activated in a new session. For example when an activated CORBA server object or an EJB looks up a second server object, using the same series of statements as the first client would use:

```
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);
SomeObject myObj =
   (SomeObject) ic.lookup("sess_iiop://localhost:5521:ORCL/test/someobject");
```

then the object `myObj` is activated in a separate session from the session in which the server object that did the lookup is running.

Using thisServer

If the server object must look up and activate a new published object in the *same session* in which it is running, then the server object should use the *thisServer/:thisSession* notation in place of the `hostname:port:SID` in the URL. 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/someobject");
```

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.

Starting a Named Session From a Client

In the simple case, you let the JNDI initial context `lookup()` method also start the session and authenticate the client. The session then becomes the default session (and has the name `:default`).

If you then create additional objects in the client and activate them, the new objects run in the same session. Even if you create a new JNDI initial context and look up the same or a new object using that context, the object is instantiated in the same session as the first object.

There are cases, however, when a client needs to activate an object in a separate session from any current objects. Do this as follows:

- Create a new service context. For example:

  ```
  ServiceCtx service = (ServiceCtx) ic.lookup(
                 "sess_iiop://localhost:2481:ORCL");
  ```

- Create a new session context by invoking `createSubcontext()` on the service context.

  ```
  SessionCtx new_session = (SessionCtx) service.createSubcontext(
                                       ":session1");
  ```

  Name the new session in the parameter to `createSubcontext()`, for example ":session1". The name must start with a colon (':') and cannot contain a slash ('/').

- Authenticate the client by invoking the `login()` method on the new session:

  ```
  new_session.login("scott", "tiger", null);
  ```

The following is a more detailed code example that demonstrates this technique. The complete example is in "twosessions" on page A-86.

```
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
Hello hello1 = (Hello)session1.activate (objectName);
Hello hello2 = (Hello)session2.activate (objectName);

// Verify that the objects are indeed different
hello1.setMessage ("Hello from Session1");
hello2.setMessage ("Hello from Session2");

System.out.println (hello1.helloWorld ());
System.out.println (hello2.helloWorld ());
```

Closing a Session

You can release objects that you have instantiated in a session by sending a `_release()` message to them. For example, if you previously activated an object named `bank`, then invoking `bank._release()` deactivates the `bank` object.

When the last object activated in a session is released, the connection is closed.

You can also invoke `orb.shutdown()` from a client program. This method releases all objects and connections from the client.

Example: Activating Services and Sessions

This section describes in greater detail how you can explicitly activate a session IIOP service and then activate one or more Oracle8*i* sessions in the context of the service. The simplest way to activate services and sessions is to use the JNDI methods provided in the `ServiceCtx` and `SessionCtx` classes.

This section demonstrates service and session activation, as well as explicit login authentication, by way of a useful example: `lister.java`. This program recursively lists the names of all published objects in the session name space, along with the creation dates and owners.

Unlike most of the other example programs in this guide, the lister program does not start by activating a published object. In the other example programs, the service and session are usually started automatically, as a by-product of the published object look up. In this example the service and session must be specifically activated by the client program.

The example starts by instantiating 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 that only the `URL_PKG_PREFIXES` `Context` variable is filled in—the other information will be provided in the `login.authenticate()` method parameters.

Next, create a new JNDI Context. This is the necessary first step in all programs that will use JNDI methods. Pass in the hashtable, as usual.

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

Then 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, hostname, listener port, and SID:

```
ServiceCtx service =
    (ServiceCtx) ic.lookup("sess_iiop://localhost:2481:ORCL");
```

The next step is to initiate a session. Do this by invoking the `createSubcontext()` method on the service context object, as follows:

```
SessionCtx session = (SessionCtx) service.createSubcontext(":session1");
```

Note that 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.

The final step before you can access the published object tables is to authenticate the client program to the database. Do this by calling the `login()` method on the session context object:

```
session.login("scott", "tiger", null);    // role is null
```

Finally, the example starts listing by calling the `listOneDirectory()` static method, which recursively lists all directories (`PublishingContexts`) and leafs (`PublishedObjects`) in the published names hierarchy:

```
listOneDirectory ("/", session);
```

The complete code for the example is reproduced in the following section. The code includes some minimal formatting to align the printed output. Follow the same procedures as for the sample applications in Appendix A, "Example Code: CORBA" to compile and run this example.

Lister.java

```
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingEnumeration;
import javax.naming.Binding;
import javax.naming.NamingException;
import javax.naming.CommunicationException;

import oracle.aurora.jndi.sess_iiop.ServiceCtx;
import oracle.aurora.jndi.sess_iiop.SessionCtx;
import oracle.aurora.jndi.sess_iiop.ActivationException;
import oracle.aurora.AuroraServices.PublishedObject;
import oracle.aurora.AuroraServices.objAttribsHolder;
import oracle.aurora.AuroraServices.objAttribs;
import oracle.aurora.AuroraServices.ctxAttribs;
import oracle.aurora.jts.client.AuroraTransactionService;
import oracle.aurora.AuroraServices.LoginServer;
import oracle.aurora.client.Login;

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

public class Lister {

  public static void main (String[] args) throws Exception {
    if (args.length != 3) {
      System.out.println("usage: Lister serviceURL user password");
      System.exit(1);
    }
    String serviceURL = args [0];
    String username = args [1];
    String password = args [2];

    // 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 a session in the instance.
    // The session name must start with a colon(:).
    SessionCtx session = (SessionCtx) service.createSubcontext(":session1");
    session.login(username, password, null);

    // Print a header line.
    System.out.println
      ("\n\nName                          Create Date              Owner");
    listOneDirectory ("/", session);
  }


  public static void listOneDirectory (String name, SessionCtx ctx)
       throws Exception  {
    System.out.print(name);
    for (int i = name.length(); i < 30; i++)
      System.out.print(" ");
    ctxAttribs attribs = null;
    try {
      attribs = ctx.getAttributes();
    } catch (org.omg.CORBA.NO_PERMISSION e) {
      return;
    }

    System.out.print(attribs.creation_ts);
    for (int i = 30 + attribs.creation_ts.length(); i < 55; i++)
      System.out.print(" ");
    System.out.print(attribs.owner);
    /*
     * You could also add output for the access permissions:
     *  attribs.read
     *  attribs.write
     *  attribs.execute
     */

    System.out.println();

    // Show the sub entries
    listEntries(ctx, name);
  }

  public static void listEntries (Context context, String prefix)
       throws Exception  {
    NamingEnumeration bindings = context.list("");
    while (bindings.hasMore()){
      Binding binding = (Binding) bindings.next();
      String name = binding.getName();
      Object object = context.lookup(name);
      if (object instanceof SessionCtx)
        listOneDirectory(prefix + name + "/", (SessionCtx) object);
      else if (object instanceof PublishedObject)
        listOneObject(prefix + name, (PublishedObject) object);
      else
        // We should never get here.
        System.out.println(prefix + name + ": " + object.getClass());
    }
  }


  public static void listOneObject (String name, PublishedObject obj)
       throws Exception  {
    objAttribsHolder holder = new objAttribsHolder();
    try {
      obj.get_attributes(holder);
    } catch (org.omg.CORBA.NO_PERMISSION e) {
      return;
    }

    objAttribs attribs = holder.value;
    System.out.print(name);
    for (int i = name.length(); i < 30; i++)
      System.out.print(" ");
    System.out.print(attribs.creation_ts);
    for (int i = 30 + attribs.creation_ts.length(); i < 55; i++)
      System.out.print(" ");
    System.out.print(attribs.owner);

    /*
     * You could also add output for:
     *  attribs.class_name
     *  attribs.schema
     *  attribs.helper
     * and the access permissions:
     *  attribs.read
     *  attribs.write
     *  attribs.execute
     */

    System.out.println();
  }
}
```

Starting a New Session From a Server Object

Starting a new session from a CORBA server object or from an EJB is identical to starting a session from an application client. You can start the session implicitly by using `lookup()` on an initial context to look up and activate another published object, or you can start a new service context, and from that a new session, just as "Starting a Named Session From a Client" on page 4-33 shows.

```
Hashtable env = new Hashtable ();
env.put (Context.URL_PKG_PREFIXES, "oracle.aurora.jndi");
Context ic = new InitialContext (env);
employee =
  (Employee)ic.lookup ("sess_iiop://thisServer/test/myEmployee");
```

Any new session connection must always authenticate itself to the database server. See "clientserverserver" on page A-67 for an example that starts a new session from a CORBA server object.

If you need to activate a new object in the same session from another server object, use the `thisServer` indicator. See "Using thisServer" on page 4-32 for more information.

Controlling Session Duration

A session normally ends when the last client connection terminates. However, a server object can control the session duration by using the `oracle.aurora.net.Presentation.sessionTimeout()` method. The method takes one parameter; the session timeout value in seconds. The session timeout clock starts ticking when the last client request completes. 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);
```

See the example "timeout" on page A-72 for an example that sets session timeout on the server side.

Note: When you use the `sessionTimeout()` method, you must add `$(ORACLE_HOME)/javavm/lib/aurora.zip to` your CLASSPATH.

Ending a Session

To terminate a database session, use the `exitSession()` method. For example,

```
oracle.aurora.vm.OracleRuntime.exitSession(1);
```

The **`int`** parameter for `exitSession(int x)` is an exit value, similar to the value supplied for `System.exit()`.

Note: `System.exit()` does not terminate a database session.

IIOP Security

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 "Access Control" on page 2-27 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.

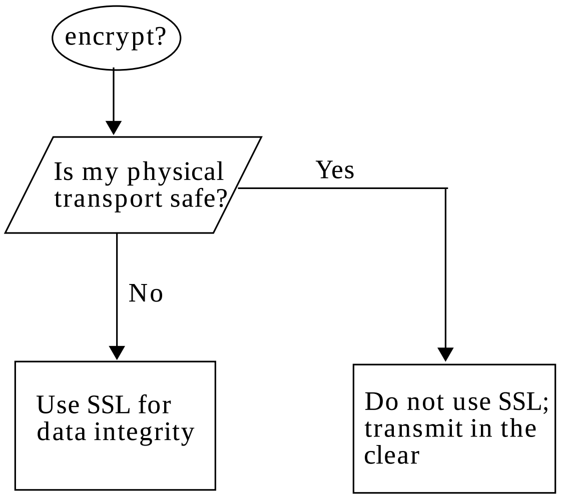
    - User authorization—The client or server can perform authorization on a provided certificate. This type of authorization can be performed only when the client or server authenticates itself by providing a certificate.

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 4–8 Data Integrity Decision Tree


encrypt?
Is my physical transport safe?
Yes
No
Use SSL for data integrity
Do not use SSL; transmit in the clear

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 4-57 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 4–2 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 4–2 SSL Version Numbers

	Server Setting			
Client 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.

Note: The current version of Oracle8*i* does not support SSL security for callouts and callbacks from the server.

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 "sharedsession" on page A-79 for an example. 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.

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.

```
final private static String testCert_base64 =
  "MIICejCCAeOgAwIBAgICAmowDQYJKoZIhvcNAQEEBQAwazELMAkGA1UEBhMCVVMx" +
  "DzANBgNVBAoTBk9yYWNsZTEoMCYGA1UECxMfRW50ZXJwcmlzZSBBcHBsaWNhdGlv" +
  "biBTZXJ2aWNlczEhMB8GA1UEAxMYRUFTUUEgQ2VydGlmaWNhdGUgU2VydmVyMB4X" +
  "DTk5MDgxNzE2MjIxMloXDTAwMDIxMzE2MjIxMlowgYUxCzAJBgNVBAYTAlVTMRsw" +
  "GQYDVQQKExJPcmFjbGUgQ29ycG9yYXRpb24xPDA6BgNVBAsUMyoqIFNlY3VyaXR5" +
  "IFRFU1RJTkcgQU5EIEVWQUxVQVRJT04gT05MWSB2ZXJzaW9uMiAqKjEbMBkGA1UE" +
  "AxQSdGVzdEB1cy5vcmFjbGUuY29tMHwwDQYJKoZIhvcNAQEBBQADawAwaAJhANG1" +
  "Kk2K7uOOtI/UBYrmTe89LVRrG83Eb0/wY3xWGelkBeEUTwW57a26u2M9LZAfmT91" +
  "e8Afksqc4qQW23Sjxyo4ObQK3Kth6y1NJgovBgfMu1YGtDHaSn2VEg8p58g+nwID" +
  "AQABozYwNDARBglghkgBhvhCAQEEBAMCAMAwHwYDVR0jBBgwFoAUDCHwEuJfIFXD" +
  "a7tuYNO8bOw1EYwwDQYJKoZIhvcNAQEEBQADgYEARC5rWKge5trqgZ18onldinCg" +
  "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. 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.

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 4–1 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 4–1 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

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 4-46 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.

Note: If the client wants to verify the server against trustpoints or authorize the server, it is up to the client to set up its trustpoints and parse the server's certificates for authorization. See "Authorization" on page 4-57 for more information.

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 4–2 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 4–2 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 "Configuring CORBA and EJB in JServer" on page 4-11.

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 4–3 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 4–3 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.

```
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`.

Non-JNDI Clients

It is possible for clients to access server objects without using the JNDI classes shown in the other sections of this chapter. These clients can connect to an Oracle server by using straight CosNaming methods. The following example shows how to do this.

```
import org.omg.CORBA.Object;
import org.omg.CosNaming.*;
import oracle.aurora.AuroraServices.*;
import oracle.aurora.client.Login;

public class Client {

  public static void main(String args[]) throws Exception {
    // Parse the args
    if (args.length != 4) {
      System.out.println("Must supply host/port, username/ password");
      System.exit(1);
    }
    String host = "sess_iiop://localhost:2481:ORCL";
    String port = "2481";
    String username = "scott";
    String password = "tiger";

    // access the Aurora Names Service

    Bank.Account account = null;
    Bank.AccountManager manager = null;

    try {

      // Get the Name service Object reference (Only ORB specific thing)
      PublishingContext rootCtx = null;
      // See the README file with this demo for more about VisiAurora.
      rootCtx = VisiAurora.getNameService(host, Integer.parseInt(port));

      // Get the pre-published login object reference
      PublishedObject loginObj = null;
      LoginServer   serv = null;
      NameComponent[] name = new NameComponent[2];
      name[0] = new NameComponent("etc", "");
      name[1] = new NameComponent("login", "");

      // look up this object in the name service
      Object lo = rootCtx.resolve(name);

      // Make sure it is a published object
      loginObj = PublishedObjectHelper.narrow(lo);

      // create and activate this object (non- standard call)
      lo = loginObj.activate_no_helper();
      serv = LoginServerHelper.narrow(lo);

      // Create a client login proxy object and authenticate to the DB
      Login login = new Login(serv);
      login.authenticate(username, password, null);

      // Now create and get the bank object reference
      PublishedObject bankObj = null;
      name[0] = new NameComponent("test", "");
      name[1] = new NameComponent("bank", "");

      // look up this object in the name service
      Object bo = rootCtx.resolve(name);

      // Make sure it is a published object
      bankObj = PublishedObjectHelper.narrow(bo);

      // create and activate this object (non- standard call)
      bo = bankObj.activate_no_helper();
      manager = Bank.AccountManagerHelper.narrow(bo);

      account = manager.open("Jack.B.Quick");

      float balance = account.balance();
      System.out.println
          ("The balance in Jack.B.Quick's account is $" + balance);
    } catch (org.omg.CORBA.SystemException ex) {
      System.out.println("Caught System Ex: " + ex);
      ex.printStackTrace();
    } catch(java.lang.Exception ex) {
      System.out.println("Caught Unknown Ex: " + ex);
      ex.printStackTrace();
```

```
    }
  }
}
```

You can obtain documentation and other collateral information about JNDI from the following web site:

`http://java.sun.com/products/jndi/index.html`

5 Transaction Handling

This chapter covers transaction management for both CORBA and EJB applications. Transaction handling in the two distributed component development models has some fundamental similarities, but there are also some differences. For example, the application developer who is using EJBs can elect to have the EJB container manage all transactions in a way that is transparent to the client application and to the bean developer. The developer does not have to write any transaction code at all—the transactional properties of the application can be declared at bean deployment time. In this sense, EJBs are said to have *declarative transactional* capability.

The CORBA developer, on the other hand, must use the transactional APIs provided—usually a mapping of a subset of the OMG Object Transaction Service (OTS) API, such as the Java Transaction Service (JTS) that is supplied with Oracle8*i* JServer. The CORBA developer must code calls to a transaction service to enable transactional properties for distributed objects, where this is required.

But the EJB developer might require finer-grained control of the application's transactional properties than that offered by the declarative transactional capabilities built-in to the EJB container. In this case, the developer can use explicit calls to transaction API methods, either on the client side or in the bean implementations themselves.

This chapter discusses the following topics:

Transaction Overview

A transaction is a unit of work, usually associated with a database management system. Transactions are described in terms of the so-called ACID properties. A transaction is:

- *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.

Most of the transactional features that are part of the Oracle8*i* database server are available to the CORBA or EJB distributed application developer.

Oracle8*i* JServer supports two transaction APIs for use in CORBA and EJB applications:

- the Java Transaction Service (JTS) API
- the UserTransaction interface

The JTS is a Java binding to the OMG Object Transaction Service (OTS). It is used for client-side demarcated transactions, and for transaction management in CORBA server objects.

The UserTransaction interface is used in EJBs, where the bean is running using the transaction attribute TX_BEAN_MANAGED.

Limitations

The implementations of JTS that is supplied for this Oracle8*i* release is intended mostly to support client-side transaction demarcation. As such it has some limitations that you should be aware of when designing your application.

No Distributed Transactions

This implementation of JTS does not manage distributed transactions. Transaction control distributed among multiple database servers, with support for the required two-phase commit protocol, will be available in an upcoming release of Oracle8*i* JServer.

Resources

The JTS transaction API supplied with Oracle8*i* JServer manages only one resource: an Oracle8*i* database session. A transaction cannot span multiple servers **or** multiple database sessions in a single service.

Transaction contexts are never propagated outside a server. If a server object calls out to another server, the transaction context is not carried along.

However, a transaction can involve one or many objects. The transaction can encompass one or many methods of these objects. The scope of a transaction is defined by a *transaction context* that is shared by the participating objects.

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 `SubtransactionsUnavailable` exception.

Timeouts

Methods of the JST that support transaction timeout, such as `setTimeout()`, do not work in this release. You can invoke them from your code, and no exception is thrown, but they have no effect.

Interoperability

The transaction services supplied with this release do not interoperate with other OTS implementations.

Transaction Demarcation

A transaction is said to be *demarcated*. This simply means that it has a definite beginning and definite end 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.

In a distributed object application, transactions are often described as *client-side demarcated* (or sometimes just *client demarcated*), or *server-side demarcated* (equivalently *server demarcated*).

In client-side demarcation, a transactional client explicitly encloses one or more method invocations on a server object with demarcation methods that begin and end transactions. The begin and end demarcaters are method calls on the client-side transaction service. See "Client-Side Demarcation" on page 5-10 for specific examples.

Server-side transaction demarcation implies that the server-side object begins and either commits or rolls back a transaction. Note that a transaction can span several objects, any one of which can suspend, resume, or end the transaction.

Transaction Context

The *transaction context* is a pseudo-object that is passed to the server object from the client, or from one server object to another, in the case where one server object is invoking methods on another, and hence acting as its client. The transaction context carries the state of the transaction.

After a client-side transaction service is initialized, and a begin transaction method is invoked, the transaction service implicitly creates a transaction context, and assigns a transaction ID number to the context. The client transaction service then propagates the transaction context to each participant in the transaction, that is, to each object that the client calls.

Propagation of the transaction context on each method invocation is normally transparent to the client program. The transaction context is maintained by the transaction service for each client. Transaction contexts are propagated transparently from the transaction initiator to the server object. On the client side, an interceptor is engaged to submit the transaction context on any method call to a server object. A server-side interceptor extracts the transaction context information, and makes it available to the server object.

As stated in "Limitations" on page 5-2, a transaction context cannot span multiple sessions. Each new session connection requires a new transaction context.

Transaction Service Interfaces

Oracle8*i* supports a version of the JTS. The JTS is a Java mapping of the OMG Object Transaction Service (OTS). There are two classes that the application developer can use:

- `TransactionService`
- `UserTransaction`, implemented by `oracle.aurora.jts.client.AuroraTransactionService`

The section below describes the `TransactionService` interface. Because it is used with EJBs, the `UserTransaction` class is described in "AuroraUserTransaction" on page 5-17.

TransactionService

Use the `TransactionService` to initialize a transaction context on the client. Include the `AuroraTransactionService` package in your Java client source with the following **import** statements:

```
import oracle.aurora.jts.client.AuroraTransactionService;
import javax.jts.*;
import oracle.aurora.jts.util.*;
```

These classes are included in the library file `aurora_client.jar`, which must be in the classpath when compiling and executing all source files that use the JTS.

There is only one method in this package that you can call:

```
public synchronized static void initialize(Context initialContext,
                                           String serviceName)
```

This method initializes the transaction context on a client. The parameters are:

`initialContext`	The `context` object returned by a JNDI `Context` constructor.
`serviceName`	The complete service name. For example `sess_iiop://localhost:2481:ORCL`

An example of using `initialize()` is:

```
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 initialContext = new InitialContext(env);
AuroraTransactionService.initialize
          (initialContext, "sess_iiop://localhost:2481:ORCL");
```

See also the complete example in "clientside" on page A-96.

Using The Java Transaction Service

The JTS package itself contains methods that a client-side or server-side object uses to begin transactions, commit or roll back a transaction, and perform utility functions such as setting the transaction timeout. The JTS methods should be used in CORBA or EJB clients, or in CORBA server objects. Developers implementing EJBs, and who need fine-grained transaction control within beans should use the `UserTransaction` interface in a bean-managed state. See "Transaction Management for EJBs" on page 5-11 for more information.

To use the JTS methods, code the following import statements in your source:

```
import oracle.aurora.jts.util.TS;
import javax.jts.util.*;
import org.omg.CosTransactions.*;
```

The `oracle.aurora.jts.util` package is included in the library file `aurora_client.jar`, which must be in the classpath for all Java sources that use the JTS.

You use the static methods in the `TS` class to get the transaction service.

Java Transaction Service Methods

The JTS includes the following methods:

`public static synchronized TransactionService getTS()`

`getTS()` returns a transaction service object. Once a transaction service has been obtained, you can invoke the static method `getCurrent()` on it to return a `Current` pseudo-object, the transaction context. Then you can invoke methods to begin, suspend, resume, commit, or roll back the current transaction on the `Current` pseudo-object.

Here is an example that begins a new transaction on a client, starting with getting the JNDI initial context:

```
import oracle.aurora.jndi.sess_iiop.ServiceCtx;
import oracle.aurora.jts.client.AuroraTransactionService;
import javax.naming.Context;
import javax.naming.InitialContext;
import java.util.Hashtable;
...
Context ic = new InitialContext(env);
...
AuroraTransactionService.initialize(ic, serviceURL);
```

```
...
Employee employee = (Employee)ic.lookup (serviceURL + objectName);
EmployeeInfo info;
oracle.aurora.jts.util.TS.getTS().getCurrent().begin();
```

If there is no transaction service available, then `getTS()` throws a `NoTransactionService` exception.

Current Transaction Methods

The methods that you can call to control transactions on the current transaction context are the following:

`public void begin()`

Begins a new transaction.

Can throw these exceptions:

- `NoTransactionService`—if you have not initialized a transaction context.
- `SubtransactionsUnavailable`—if you invoke a `begin()` before the current transaction has been committed or rolled back.

See the section "TransactionService" on page 5-5 for information about initialization.

`public Control suspend()`

Suspends the current transaction in the session. Returns a `Control` transaction context pseudo-object. You must save this object reference for use in any subsequent `resume()` invocations. Invoke `suspend()` in this way:

```
org.omg.CosTransactions.Control c =
          oracle.aurora.jts.util.TS.getTS().getCurrent().suspend();
```

`suspend()` can throw these exceptions:

- `NoTransactionService`—if you have not initialized a transaction context.
- `TransactionDoesNotExist`—if not in an active transaction context. This can occur if a `suspend()` follows a previous `suspend()`, with no intervening `resume()`.

If `suspend()` is invoked outside of a transaction context, then a `NoTransactionService` exception is thrown. If `suspend()` is invoked before `begin()` has been invoked, or after a `suspend()`, the a exception is thrown.

public void resume(Control which)

Resumes a suspended transaction. Invoke this method after a `suspend()`, in order to resume the specified transaction context. The `which` parameter must be the transaction `Control` object that was returned by the previous matching `suspend()` invocation in the same session. For example:

```
org.omg.CosTransactions.Control c =
             oracle.aurora.jts.util.TS.getTS().getCurrent().suspend();
...   // do some non-transactional work
oracle.aurora.jts.util.TS.getTS().getCurrent().resume(c);
```

`resume()` can throw:

- `InvalidControl`—if the `which` parameter is not valid, or is null.

public void commit(boolean report_heuristics)

Commits the current transaction. Set the `report_heuristics` parameter to **`false`**.

(The `report_heuristics` parameter is set to **`true`** for extra information on two-phase commits. Because this release of JServer does not support the two-phase commit protocol for distributed objects, use of the `report_heuristics` parameter is not meaningful. It is included for compatibility with future releases.)

`commit()` can throw:

- `HeuristicMixed`—if `report_heuristics` was set true, and a two-phase commit is in progress.
- `HeuristicHazard`—if `report_heuristics` was set true, and a two-phase commit is in progress.

The `HeuristicMixed` and `HeuristicHazard` exceptions are documented in the OTS specification. See "For More Information" on page 5-21 for the location of the OTS specification.

If there is no active transaction, `commit()` throws a `NoTransaction` exception.

public void rollback()

Rolls back the effects of the current transaction.

Invoking `rollback()` has the effect of ending the transaction, so invoking any JTS method except `begin()` after a `rollback()` throws a `NoTransaction` exception.

If not in a transaction context, `rollback()` throws the `NoTransaction` exception.

`public void rollback_only() throws NoTransaction {`

`rollback_only()` modifies the transaction associated with the current thread so that the only possible outcome is to roll back the transaction. If not in a transaction context, `rollback_only()` throws the `NoTransaction` exception.

`public void set_timeout(int seconds)`

This method is not supported, and has no effect if invoked. The default timeout value is 60 seconds in all cases.

`public Status get_status()`

You can call `get_status()` at any time to discover the status of the current transaction. Possible return values are:

- `javax.transaction.Status.StatusActive`
- `javax.transaction.Status.StatusMarkedRollback`
- `javax.transaction.Status.StatusNoTransaction`

The complete set of status **int**s is defined in `javax.transaction.Status`.

`public String get_transaction_name() {`

Invoke `get_transaction_name()` to see the name of the transaction, returned as a String. If this method is invoked before a `begin()`, after a `rollback()`, or outside of a transaction context, it returns a null string.

CORBA Examples

This section shows some examples that use the JTS interface for CORBA client code and CORBA server objects. See "Transaction Examples" on page A-96 for a set of complete examples that you can run and modify.

Client-Side Demarcation

Follow these steps to use JTS methods in your CORBA client code:

- Import the following packages:
  - `oracle.aurora.jts.client.AuroraTransactionService`
  - `oracle.aurora.jts.util.TS`
  - `org.omg.CosTransactions`
- Invoke `AuroraTransactionService.initialize()`, passing to it the service URL for the application (for example, `sess_iiop://localhost:2481:ORCL`) and the JNDI initial context.
- Begin a transaction by invoking `oracle.aurora.jts.util.TS.getTS().getCurrent().begin()`.

For example:

```
import oracle.aurora.jndi.sess_iiop.ServiceCtx;
import oracle.aurora.jts.client.AuroraTransactionService;
import oracle.aurora.jts.util.TS;
import org.omg.CosTransactions.*;
// Include normal startup code...
// Initialize a transaction context...
AuroraTransactionService.initialize(ic, serviceURL);

// Begin a transaction...
oracle.aurora.jts.util.TS.getTS().getCurrent().begin();

// Call methods that involve SQL DML...
employee.updateEmployee(info);
...
// Commit (or roll back) the SQL statements...
oracle.aurora.jts.util.TS.getTS().getCurrent().commit(false);
```

For a complete example that uses these techniques for client-side transaction demarcation, see "clientside" on page A-96.

Server-Side JTS

Follow these steps to use JTS methods in your CORBA server object code:

- Import the following packages:

  ```
  oracle.aurora.jts.util.TS
  org.omg.CosTransactions
  ```

- You do not need to invoke `AuroraTransactionService.initialize()` on the server, as the server does this for you.
- Begin a transaction by invoking `TS.getTS().getCurrent().begin()`. You can do this in a separate method, or as part of a method that does the first SQL DML using JDBC or SQLJ. A transaction spans methods—its scope is within the session where it is begun.
- End a transaction by invoking

  ```
  TS.getTS().getCurrent().commit(false), or
  TS.getTS().getCurrent().rollback().
  ```

You can also invoke other JTS methods, such as `set_timeout()`, from within a server object.

See the complete example at "serversideJTS" on page A-107 for a demonstration of CORBA server-side transaction demarcation.

Transactions in Multiple Sessions

See the complete example at "multiSessions" on page A-120 for an example that establishes multiple server sessions, each with its own transaction context.

Transaction Management for EJBs

The previous sections focused on general aspects of transaction management for distributed objects, and on transaction management for CORBA applications using the JTS.

An EJB application can also use JTS—*on the client side only*—to manage transactions. More typically an EJB application uses declarative transaction management, letting the EJB container provide the transaction control. You do this by specifying the appropriate value for the TransactionAttribute of the EJB deployment descriptor, either for the whole EJB, or on a method-by-method basis, where applicable.

For example, if the deployment descriptor for a bean declares that the bean has the transaction attribute TX_REQUIRES_NEW, then the bean container starts a transaction before each invocation of bean method, and attempts to commit the transaction when the method completes.

The following sections describe the values that you can set for the EJB transaction attribute.

Declarative Transactions

The bean deployer declares the transaction handling characteristics of a bean in the deployment descriptor. This is specified in the transaction attribute, which has six possible values:

- TX_NOT_SUPPORTED
- TX_REQUIRED
- TX_SUPPORTS
- TX_REQUIRES_NEW
- TX_MANADATORY
- TX_BEAN_MANAGED

The semantics of these attribute values are described in this section. See "Programming Restrictions" on page 2-31 for more information about the EJB deployment descriptor itself.

TX_NOT_SUPPORTED

When TX_NOT_SUPPORTED is declared for the bean itself, it means that Oracle8*i* does not invoke transaction support for the bean methods. However, a method declaration in the deployment descriptor can over-ride this declaration. If the client is in a transaction (has established an active transaction context), then the bean container suspends transaction support during delegation of method calls on the bean, and resumes the transaction context when the method call completes.

The suspended transaction context is not propagated to other objects that are invoked from within the bean code.

A bean that is running under TX_NOT_SUPPORTED cannot perform any SQL operations. An exception is thrown by the EJB server if this is attempted.

TX_REQUIRED

If the client invokes a bean method with the TX_REQUIRED attribute, there are two possibilities:

The client had not started a transaction	If the client has not established a transaction context, the bean starts a new transaction for each method call. The transaction is committed, if possible, after each call completes. The commit protocol is completed before the bean results are sent to the client. Oracle8*i* sends the transaction context for the transaction that it has established to other resources or EJBs that are invoked from the current bean.
The client had started a transaction	The bean container delegates calls to the bean methods using the client transaction context.

The transaction context is passed to other Enterprise JavaBean objects that are invoked from the enterprise Bean object, as long as they are in the same session.

TX_SUPPORTS

If the client has established a transaction context, then the bean container uses that context. If the client has no established transaction context, then the EJB methods are invoked with no transaction support.

TX_REQUIRES_NEW

The container always invokes the bean methods with a new transaction. The container commits the transaction, if possible, before sending the method results to the client.

If the client has established a transaction content, the client transaction is suspended before the bean transaction is started, and is resumed when the bean transaction completes (at the end of each method call).

If the client has established a transaction context, the association is suspended before the new transaction is started and is resumed when the new transaction has completed.

The container-managed transaction context is passed to the resources or other EJB objects that the bean invokes.

An enterprise Bean that has the TX_REQUIRES_NEW transaction attribute is always invoked in the scope of a new transaction. The container starts a new transaction before delegating a method call to the enterprise Bean object, and attempts to commit the transaction when the method call on the enterprise Bean object has completed. The container performs the commit protocol before the method result is sent to the client.

The new transaction context is passed to the resources or other enterprise Bean objects that are invoked from the enterprise Bean object.

TX_MANDATORY

If an EJB method is invoked with the TX_MANADATORY attribute, the client transaction context is always used. If the client has not established a transaction context, the container throws the `TransactionRequired` exception to the client.

The client transaction context is propagated to the resources or other enterprise Bean objects that are invoked from the enterprise Bean object.

TX_BEAN_MANAGED

The bean-managed attribute value is the one that lets the bean get access to the transaction service on its own behalf. Session beans get access to the transaction service through the session context that is supplied to the bean at initialization, as a parameter in the `setSessionContext()` call. The `SessionContext` interface subclasses `EJBContext.`

The bean implementation must use the `javax.jts.UserTransaction` interface methods to manage transactions on its own. See "Using The Java Transaction Service" on page 5-6 for a description of these methods.

The TX_BEAN_MANAGED attribute value cannot be mixed with other transaction attribute values. For example, if the bean-level descriptor, or one of the method-level descriptors, specifies TX_BEAN_MANAGED, then all method-level descriptors present must specify TX_BEAN_MANAGED. When using bean-managed transactions, the transaction boundaries span bean methods. You can begin a transaction in one method, and the transaction can be rolled back or committed in a separate method, called later.

The container makes the `javax.jts.UserTransaction` interface available to the enterprise Bean though the `EJBContext.getUserTransaction()` method, as illustrated in the following example.

```
import javax.jts.UserTransaction;
...
EJBContext ic = ...;
...
UserTransaction tx = ic.getUserTransaction();
tx.begin();
... // do work
tx.commit();
```

The container must manage transactions on a TX_BEAN_MANAGED Bean as follows.

When a client invokes a stateful TX_BEAN_MANAGED Bean, the container suspends any incoming transaction. The container allows the session instance to initiate a transaction using the `javax.jts.UserTransaction` interface. The instance becomes associated with the transaction and remains associated until the transaction terminates.

When a Bean-initiated transaction is associated with the instance, methods on the instances run under that transaction.

It is possible that a business method that initiated the transaction completes without committing or rolling back the transaction. The container must retain the association between the transaction and the instance across multiple client calls until the transaction terminates.

Table 5–1 Effect of declarative transaction attribute (Cont.)

Transaction Attribute Value	Client Transaction	Transaction of EJB Method
TX_NOT_SUPPORTED	none	none
	T1	none
TX_REQUIRED	none	new transaction - T2
	T1	T1
TX_SUPPORTS	none	none
	T1	T1
TX_REQUIRES_NEW	none	new transaction - T2
	T1	T2
TX_MANDATORY	none	TransactionRequired exception thrown

Transaction Attribute Value	Client Transaction	Transaction of EJB Method
	T1	T1

Session Synchronization Interface

An EJB 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.

JDBC

If you are using JDBC calls in your bean to update a database, you should *not* also use JDBC to perform transaction services, by calling methods on the JDBC connection. Do *not* code JDBC calls on a connection, for example:

```
Connection conn = ...
...
conn.commit();  // DO NOT DO THIS!!
```

You also avoid doing direct SQL commits or rollbacks through JDBC. Code the bean to either handle transactions directly using the `javax.jts.UserTransactions` interface (if the `TransactionAttribute` value is TX_BEAN_MANAGED), or let the bean container manage the bean transactions.

AuroraUserTransaction

You use the `UserTransaction` interface to manage transactions in Enterprise JavaBeans. The `UserTransaction` interface is a higher-level interface than the raw JTS, although its functionality is almost identical. However, EJB developers *must* use the `UserTransaction` interface for EJB bean-managed transaction support. The `UserTransaction` interface is used only for bean-managed EJBs.

See "serversideJTS" on page B-51 for a complete example of bean-managed transaction control.

To incorporate `UserTransaction` methods in your bean implementation code, follow these steps:

- Import the `javax.jts.UserTransaction` package.
- Be sure to get the session context, using the `setSessionContext()` `SessionBean` method. See the example.
- Invoke the `UserTransaction` methods on the transaction context. See the example.

Methods

`public void begin()`

`begin()` creates a new transaction and associates it with the current bean.

Throws `IllegalStateException` if you attempt to invoke it in the context of an existing transaction.

`public void commit()`

`commit()` commits the transaction results, and completes the transaction associated with the current bean. When the `commit()` method completes, the bean is no longer associated with a transaction.

`commit()` can throw any of the following exceptions:

- `TransactionRolledbackException`
- `HeuristicMixedException`
- `HeuristicRollbackException`
- `SecurityException`
- `IllegalStateException`

`public int getStatus()`

Returns the status of the current transaction. See "Java Transaction Service Methods" on page 5-6 for more information about the status values that can be returned.

`public void resume()`

Resumes a suspended transaction.

`public void rollback()`

Rolls back the effects of the current transaction.

`rollback()` can throw the following exceptions:

- IllegalStateException
- SecurityException

`public void setRollbackOnly()`

The effect of a `setRollbackOnly()` invocation is that the only possible conclusion to the current transaction is a roll back operation. Any attempt to perform a `commit()` after `setRollbackOnly()` is invoked results in a exception.

`setRollBackOnly()` throws an `IllegalStateException`, if not in a transaction.

`public void setTransactionTimeout(int arg1)`

This method is implemented, but has no effect in this release. The timeout value is always 60 seconds.

Session Synchronization

An EJB can optionally implement the session synchronization interface, to be notified by the server of the state of the transaction. 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.

EJB Transaction Examples

This section shows a few abbreviated examples of transaction management for EJB applications. For a set of complete programs, see "Transaction Examples" on page B-41.

Client-Side Demarcated

If your EJB application requires client-side transaction demarcation, you use the JTS interface, as explained in "Using The Java Transaction Service" on page 5-6. See the section "clientside" on page B-41 for a complete example of EJB client-side transaction demarcation.

Transaction Management in an EJB

Use the `UserTransaction` interface to set up a transaction context within an EJB. In the bean implementation, make sure to import the `javax.jts.UserTransaction` package. Unlike the `TransactionService` when used on the client side, you do not need to initialize the `UserTransaction` interface from within an EJB. The container does that for you.

Getting the Session Context

In the EJB, use the `setSessionContext()` session bean method to obtain the session context, and save it in an instance variable. For example, code this implementation of the `setSessionContext()` method:

```
public class XBean implements SessionBean {
  SessionCtx ctx;
  ...
  public void setSessionContext(SessionContext ctx) {
    this.ctx = ctx;
  }
```

You can then use the session context `ctx` to invoke `UserTransaction` methods.

Beginning a Transaction

Invoke the `begin()` method as follows:

```
ctx.getUserTransaction.begin();
```

to start a transaction.

Committing a Transaction

Invoke the `commit()` method as follows:

```
ctx.getUserTransaction().commit();
```

to end the transaction with a commit.

Other UserTransaction Methods

Invoke other methods of the `UserTransaction` interface in the same way that you do a `begin()` or a `commit()`—invoke them on a `UserTransaction` object of the session context.

See the section "serversideJTS" on page B-51 for a complete example that uses the `UserTransaction` interface in an EJB.

JDBC

If you are using JDBC calls in your CORBA server object or EJB 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 SQL exception to be thrown.

You must also avoid doing direct SQL commits or rollbacks through JDBC. Code the bean to either handle transactions directly using the `javax.jts.UserTransactions` interface (if the `TransactionAttribute` value is TX_BEAN_MANAGED), or let the bean container manage the bean transactions.

For More Information

Information on the Java Transaction Service is available at:

`http://java.sun.com:/products/jts/index.html`

The Sun JTS specification is available at:

`http://java.sun.com/products/jts/index.html`

The OTS specification is part of the CORBA services specification. Chapter 10 (individually downloadable) contains the OTS specification. Get it at:

`http://www.omg.org/library/csindx.html`

6 Tools

This chapter describes the tools you use to deploy CORBA implementations and Enterprise JavaBeans in the Oracle8*i* Java environment. You run these tools from a Unix shell or the Windows NT DOS prompt.

The tools described in this chapter as follows:

- Session Namespace Tools
- Enterprise JavaBean Tools
- VisiBroker™ for Java Tools
- Miscellaneous Tools

See the *Oracle8i Java Developer's Guide* for information on `loadjava` and `dropjava`.

Session Namespace Tools

Each database instance running the Oracle8*i* JServer software has a session namespace, which the Oracle8*i* ORB uses to activate CORBA and EJB objects. A *session namespace* is a hierarchical collection of objects known as PublishedObjects and PublishingContexts. PublishedObjects are the leaves of the hierarchy and PublishingContexts are the nodes, analogous to UNIX file system files and directories. Each PublishedObject is associated with a class schema object that represents a CORBA or EJB implementation. To activate a CORBA or EJB object, a client refers to a PublishedObject's name. From the PublishedObject, the Oracle8*i* ORB obtains the information necessary to find and launch the corresponding class schema object.

Creating a PublishedObject is known as *publishing* and can be done with the command-line `publish` tool or the interactive session shell, both of which this section describes. CORBA server developers create PublishedObjects explicitly after loading the implementation of an object with `loadjava`. EJB developers do not explicitly load or publish their implementations; the `deployejb` tool (see "deployejb" on page 6-22) implicitly does both.

A *PublishedObject* has the following *attributes*:

- Schema Object Name: the name of the Java class schema object associated with the PublishedObject.
- Schema: the name of the schema containing the corresponding class schema object.
- Helper Schema Object Name: the name of the helper class the Oracle8*i* ORB uses to automatically narrow a reference to an instance of the CORBA object or EJB.

PublishedObjects and PublishingContexts, as with their file and directory counterparts, have owners and rights (privileges). An owner can be a user name or a role name; only the owner can change the ownership or rights of a PublishedObject or PublishingContext. Table 6–1 describes session namespace rights.

Table 6–1 PublishingContext and PublishedObject Rights

Right	Meaning for PublishingContext	Meaning for PublishedObject
read	List contents and attributes (type, rights and creation time).	List object attributes (type, schema object, schema, helper, rights, and creation time).

Table 6–1 PublishingContext and PublishedObject Rights (Cont.)

Right	Meaning for PublishingContext	Meaning for PublishedObject
write	Create a PublishedObject or PublishingContext in the PublishingContext.	Republish object.
execute	Use contents to resolve a name.	Activate associated class.

Oracle8*i* creates a session namespace automatically when the Oracle8*i* ORB is configured. The PublishingContexts contained in Table 6–2 are present in all session namespaces:

Table 6–2 Initial PublishingContexts and Rights

Name	Owner	Read	Write	Execute
`/`	`SYS`	`PUBLIC`	`SYS`	`PUBLIC`
`/bin`	`SYS`	`PUBLIC`	`SYS`	`PUBLIC`
`/etc`	`SYS`	`PUBLIC`	`SYS`	`PUBLIC`
`/test`	`SYS`	`PUBLIC`	`PUBLIC`	`PUBLIC`

Because by default only `/test` is writable by PUBLIC, you will normally create PublishingContexts and PublishedObjects subordinate to `/test`.

publish

The `publish` tool creates or replaces (republishes) a PublishedObject in a PublishingContext. It is not necessary to republish when you update a Java class schema object; republishing is required only to change a PublishedObject's attributes. To publish, you must have write permission (the write right) for the destination PublishingContext; by default only the PublishingContext `/test` is writable by PUBLIC. To republish you must additionally have the write right for the PublishedObject.

Syntax

```
publish [options]
<name> <class> [<helper>] -user <username> -password <password>
-service <serviceURL>
```

where options are:

```
[-describe]
  [{-g | -grant} {<user> | <role>}[,{<user> | <role>}]...]
  [{-h | -help}]
  [-iiop]
  [-role <role>]
  [-republish]
  [-schema <schema>]
  [-ssl]
  [-useServiceName]
  [-version]
```

Argument Summary

Table 6–3 summarizes the `publish` tool arguments.

Table 6–3 publish Tool Argument Summary

Option	Description
`<name>`	Name of the PublishedObject being created or republished; PublishingContexts are created if necessary.
`<class>`	Name of the class schema object that corresponds to <name>.
`<helper>`	Name of the Java class schema object that implements the `narrow()` method for `<class>`.
`-user`	Specifies identity with which to log into the database instance named in `-service`.
`-password`	Specifies authenticating password for the username specified with `-user`.

Table 6–3 publish Tool Argument Summary (Cont.)

Option	Description
`-service`	URL identifying database whose session namespace is to be "opened" by `sess_sh`. The serviceURL has the form: `sess_iiop://`*`<host>:<lport>:<sid>`*. `<host>` is the computer that hosts the target database; `<lport>` is the listener port that has been configured to listen for session IIOP; `<sid>` is the database instance identifier. Example: `sess_iiop://localhost:2481:orcl` which matches the default installation on the invoker's machine.
`-describe`	Summarizes the tool's operation, then exits.
`-grant`	After creating or republishing the PublishedObject, grants read and execute rights to the sequence of `<user>` and `<role>` names. When republishing, replace the existing users / roles that have read / execute rights with the `<user>` and `<role>` names. To selectively change the rights of a PublishedObject, use the `sess_sh`'s `chmod` command. Note that to activate a CORBA object or EJB, a user must have the execute right for both the PublishedObject and the corresponding class schema object. The sequence of user and role names must be a comma-separated list, containing no internal spaces.
`-help`	Summarizes the tool's syntax, then exits.
`-iiop`	Connects to the target database with IIOP instead of the default session IIOP. Use this option when publishing to a database server that has been configured without session IIOP.
`-role`	Role to assume for the publish; no default.
`-republish`	Directs `publish` to replace an existing PublishedObject; without this option, the `publish` tool rejects an attempt to publish an existing name. If the PublishedObject does not exist, `publish` creates it. Republishing deletes non-owner rights; use the `-grant` option to add read / execute rights when republishing.
`-schema`	The schema containing the Java <class> schema object. If you do not specify, the `publish` tool uses the invoker's schema.
`-ssl`	Connects to the database with SSL server authentication. You must have configured the database for SSL to use this option, and you must specify an SSL listener port in `-service`.

Table 6–3 publish Tool Argument Summary (Cont.)

Option	Description
`-useServiceName`	If you are using a service name instead of an SID in the URL, you must specify this flag. Otherwise, the tool assumes the last string in the URL is the SID.
`-version`	Shows the tool's version, then exist.

Here is a `publish` example.

Publish the CORBA server implementation `vbjBankTestbank.AccountManagerImpl` and its helper class as `/test/bankMgr` in the tool invoker's schema:

```
publish /test/bankMgr vbjBankTestServer.AccountManagerImpl \
vbjBankTestServer.AccountManagerHelper \
-user SCOTT -password TIGER \
-service sess_iiop://dlsun164:2481:orcl
```

remove

The `remove` tool removes a PublishedObject or PublishingContext from a session namespace. It does not remove the Java class schema object associated with a PublishedObject; use `dropjava` to do that.

Syntax

```
remove <name> -user <username> -password <password> -service <serviceURL>
[options]
  [{-d | -describe}]
  [{-h | -help}]
  [-iiop]
  [{-r | -recurse}]
  [-role role]
  [-ssl]
  [-useServiceName]
  [-version]
```

Argument Summary

Table 6–4 describes the `remove` arguments.

Table 6–4 remove Argument Summary

Option	Description
`<name>`	Name of PublishingContext or PublishedObject to be removed.
`-user`	Specifies identity with which to log into the instance named in `-service`.
`-password`	Specifies authenticating password for the `<username>` you specified with `-user`.
`-service`	URL identifying database whose session namespace is to be “opened” by `sess_sh`. The serviceURL has the form: `sess_iiop://<host>:<lport>:<sid>`. `<host>` is the computer that hosts the target database; `<lport>` is the listener port that has been configured to listen for session IIOP; `<sid>` is the database instance identifier. Example: `sess_iiop://localhost:2481:orcl` which matches the default installation on the invoker’s machine.
`-describe`	Summarizes the tool’s operation, then exits.
`-help`	Summarizes the tool’s syntax, then exits.
`-iiop`	Connects to the target database with IIOP instead of the default session IIOP. Use this option when removing from a database server that has been configured without session IIOP.
`-recurse`	Recursively removes `<name>` and all subordinate PublishingContexts; required to remove a PublishingContext.
`-role`	Role to assume for the remove; no default.
`-ssl`	Connects to the database with SSL server authentication. You must have configured the database for SSL to use this option.
`-useServiceName`	If you are using a service name instead of an SID in the URL, you must specify this flag. Otherwise, the tool assumes the last string in the URL is the SID.
`-version`	Shows the tool’s version, then exits.

Here are examples of `remove` tool usage.

- Remove a PublishedObject named `/test/testhello`:

```
remove /test/testhello -user SCOTT -password TIGER \
-service sess_iiop://dlsun164:2481:orcl
```

- Remove a PublishingContext named `/test/etrader`:

  ```
  remove -r /test/etrader -user SCOTT -password TIGER \
  -service sess_iiop://dlsun164:2481:orcl
  ```

sess_sh

The `sess_sh` (session shell) tool is an interactive interface to a database instance's session namespace. You specify database connection arguments when you start `sess_sh`. It then presents you with a prompt to indicate that it is ready for commands.

The `sess_sh` gives a session namespace much of the "look and feel" of a Unix file system you access through a shell, such as the C shell. For example, the session shell command:

```
ls /alpha/beta/gamma
```

means "List the PublishedObjects and PublishingContexts in the PublishingContext known as `/alpha/beta/gamma`". (NT users note: `/alpha/beta/gamma`, not `\alpha\beta\gamma`.) Indeed, many session shell command names that operate on PublishingContexts have the same names as their Unix shell counterparts that operate on directories. For example: `mkdir` (create a PublishingContext) and `cd` (change the working PublishingContext).

In addition to Unix-style manipulation of PublishingContexts and PublishedObjects, the session shell can launch an *executable*, which is analogous to a Java standalone application, that is, a class with a static `main()` method. Executables must have been loaded with `loadjava`, but not published—publishing is for CORBA and EJB objects only.

Syntax

```
sess_sh [options] -user <user> -password <password> -service <serviceURL>
  [-d | -describe]
  [-h | -help]
  [-iiop]
  [-role <rolename>]
  [-ssl]
  [-useServiceName]
  [-version]
```

Argument Summary

Table 6–5 summarizes the `sess_sh` command line arguments.

Table 6–5 sess_sh Argument Summary

Option	Description
`-user`	Specifies user's name for connecting to the database.
`-password`	Specifies user's password for connecting to the database.
`-service`	URL identifying database whose session namespace is to be "opened" by `sess_sh`. The serviceURL has the form: `sess_iiop://<host>:<lport>:<sid>`. `<host>` is the computer that hosts the target database; `<lport>` is the listener port configured to listen for session IIOP; `<sid>` is the database instance identifier. Example: `sess_iiop://localhost:2481:orcl` which matches the default database installation on the invoker's machine.
`-describe`	Summarizes the tool's operation, then exits.
`-help`	Summarizes the tool's syntax, then exits.
`-iiop`	Connects to the target database with plain IIOP instead of the default session IIOP. Use this option for a database server configured without session IIOP.
`-role`	Role to pass to database; there is no default.
`-ssl`	Connect to the database with SSL server authentication. You must have configured the database for SSL and specify an SSL port to use this option.
`-useServiceName`	If you are using a service name instead of an SID in the URL, you must specify this flag. Otherwise, the tool assumes the last string in the URL is the SID.
`-version`	Shows the command's version, then exits.

Here is a `sess_sh` example.

Open a session shell on the session namespace of the database `orcl` on listener port 2481 on host `dbserver`.

```
sess_sh -user scott -password tiger -service sess_iiop://dbserver:2481:orcl
```

cd Command

The `cd` command is analogous to a Unix shell's `cd` command; it changes the working PublishingContext.

Syntax

```
cd [path]
```

Here is an example.

Change to root PublishingContext:

```
$ cd /
```

chmod Command

The `chmod` command is analogous to a Unix shell's `chmod` command; it changes the users or roles that have rights for a PublishingContext or PublishedObject. See Table 6–1 on page 6-2 for descriptions of the read, write, and execute rights. Only the object's owner can change its rights.

Syntax

```
chmod  [options] {+|-}{r|w|e} {<user> | <role>} [, {<user> | <role>} ...] \
<objectname>
  [-h | -help]
  [-version]
```

Argument Summary

Table 6–6 summarizes the `chmod` arguments.

Table 6–6 chmod Argument Summary

Option	Description
`+/-rwe`	Specifies the right (read, write, or execute) to be added (+) or removed (-) for `<user>` or `<role>`.
`<user> | <role>`	Specifies the user or role whose rights are to be increased or decreased.
`<objectname>`	Specifies the name of the PublishingContext or PublishedObject whose rights are to be changed.
`-help`	Summarizes the command's syntax, then exits.
`-version`	Shows the command's version, then exits.

Here are some chmod examples.

- Give execute rights for `/alpha/beta/gamma` to Scott and Nancy:

  ```
  $ chmod +x scott nancy /alpha/beta/gamma
  ```

- Remove Scott's write rights for the same object:

  ```
  $ chmod -w scott /alpha/beta/gamma
  ```

chown Command

The chown command is analogous to the Unix chown command; it changes the ownership of a PublishingContext or PublishedObject. The owner of a newly created PublishingContext or PublishedObject is the user who publishes it. To change a PublishingContext's or PublishedObject's ownership you must be SYS.

Syntax

```
chown [options] {<user> | <role>} <objectname>
  [-h | -help]
  [-version]
```

Argument Summary

Table 6–7 summarizes the `chown` arguments.

Table 6–7 chown Argument Summary

Option	Description
`<user> | <role>`	Specifies the user or role to be the new owner.
`<objectname>`	Specifies the name of the PublishingContext or PublishedObject whose owner is to be changed.
`-help`	Summarizes the command's syntax, then exits.
`-version`	Shows the command's version, then exits.

Here is a `chown` example.

Make Scott the owner of `/alpha/beta/gamma`:

```
$ chown scott /alpha/beta/gamma
```

exit Command

The `exit` command terminates `sess_sh`.

Syntax

```
exit
```

Here is an example:

Leave the session shell:

```
$ exit
%
```

help Command

The `help` command summarizes the syntax of the session shell commands.

Syntax

```
help
```

Here is a `help` example.

```
$ help
Commands are of the format <command> [arg1, ar2...]
Intrinsic Commands:
    exit           exit the shell
    help           prints this message
    version        print version inforamtion
    pwd            print working directory
    cd             change working directory
    ls             list directory
    ln             link name
    chmod          change read, write or execute permissions on an object
    chown          change an objects owner
    mkdir          create a directory
    mv             move an object or directory to another location
    rm             remove an object or directory
    lls            list directory on local file system
    lpwd           print local file system working directory
    lcd            change the local file systems working directory
    loadjar        load java classes, source, resources from jar files into the
server
    loadfile       load java classes, source, resources from files into the
server
    publish        publish an object
    republish      republish an object
    java           execute the "main" method on a java class
```

java Command

The `java` command is analogous to the JDK `java` command; it invokes a class's static `main()` method. The class must have been loaded with `loadjava`. (There is no point to publishing a class that will be invoked with the `java` command.) The `java` command provides a convenient way to test Java code that runs in the database. In particular, the command catches exceptions and redirects the class's standard output and standard error to the session shell, which displays them as with any other command output. (The usual destination of standard out and standard error for Java classes executed in the database is one or more database server process trace files, which are inconvenient and may require DBA priviliges to read.)

Syntax

```
java class [arg1 ... argn] [options]
  [{-h | -help}]
  [-schema <schema>]
  [-version]
```

Argument Summary

Table 6–8 summarizes the `java` arguments.

Table 6–8 java Argument Summary

Option	Description
`class`	Names the Java class schema object that is to be executed.
`arg1 ... argn`	Arguments to the class's `main()` method.
`-help`	Summarizes the command's syntax, then exits.
`-schema`	Names the schema containing the class to be executed; the default is the invoker's schema.
`-version`	Shows the command's version, then exits.

Here is a `java` command example.

Say hello and display arguments:

```
package hello;
public class World {
    public World() {
        super();
    }
    public static void main(String[] argv) {
        System.out.println("Hello from the JServer/ORB");
        if (argv.length != 0)
            System.out.println("You supplied " + argv.length + " arguments: ");
            for (int i = 0; i < argv.length; i++)
                System.out.println(" arg[" + i + "] : " + argv[i]);
    }
}
```

Compile, load, publish, and run the executable as follows, substituting your userid, host, and port information as appropriate:

```
% javac hello/World.java
% loadjava -r -user scott/tiger@localhost:2481:orcl hello/World.class
% sess_sh -user scott -password tiger -service sess_iiop://localhost:2481:orcl
$ java testhello alpha beta
Hello from the JServer/ORB
You supplied 2 arguments:
arg[0] : alpha
arg[1] : beta
$
```

lcd Command

The `lcd` (local `cd`) command changes the local working directory just as executing `cd` outside of the session shell would.

Syntax

```
lcd [path]
```

Here is an example of the `lcd` command.

Change the file system directory to `alpha/beta`:

```
$ lcd alpha/beta
```

lls Command

The `lls` (local `ls`) command lists the contents of the working directory, just as executing `ls` outside of the session shell would.

Syntax

```
lls
  [-l]
  [<path>]
```

Argument Summary

Table 6–9 summarizes the `lls` command's arguments.

Table 6–9 lls Argument Summary

Option	Description
`-l`	Lists the directory in long format.
`<path>`	Lists the directory named in `<path>`.

Here is an `lls` command example.

List the working file system directory in long format:

```
$ lls -l
```

ln Command

The `ln` (link) command is analogous to the Unix `ln` command. A link is a synonym for a PublishingContext or PublishedObject. A link can prevent a reference to a PublishingContext or PublishedObject from becoming invalid when you move a PublishingContext or PublishedObject (see "mv Command" on page 6-18); creating a link with the old name makes the object accessible by both its old and new names.

Syntax

```
ln <object> <link>
```

Argument Summary

Table 6–10 summarizes the `ln` arguments.

Table 6–10 ln Argument Summary

Option	Description
`<object>`	The name of the PublishingContext or PublishedObject for which a link is to be created.
`<link>`	The synonym by which `<object>` is also to be known.

Here is an `ln` command example.

Preserve access via `old` although the object's name is changed to `new`:

```
$ mv old new
$ ln new old
```

lpwd Command

The `lpwd` (local print working directory) command displays the name of the working directory, just as executing `pwd` outside of the session shell would.

Syntax

```
lpwd
```

Here is an example of the `lpwd` command that shows the working directory:

```
$ lpwd
/home/usr/billc
```

ls Command

The `ls` (list) command shows the contents of PublishingContexts as the Unix `ls` command shows the contents of directories.

Syntax

```
ls [options] [{<pubcon> | <pubobj} [{<pubcon> | <pubobj}] ...]
  [-dir]
  [-h | -help]
  [-l]
  [-ld | ldir]
  [-R]
  [-version]
```

Argument Summary

Table 6–11 describes the `ls` arguments.

Table 6–11 ls Argument Summary

Option	Description
`<pubcon> | <pubobj>`	Name of PublishingContext(s) and / or PublishingObject(s) to be listed; the default is the working PublishingContext.
`-dir`	Shows only PublishingContexts; analogous to the Unix `ls -d` command.
`-help`	Summarizes the command's syntax, then exits.

Table 6–11 ls Argument Summary (Cont.)

Option	Description
`-l`	Shows contents in long (detailed) format. The long format includes name, creation time, owner, and rights. For PublishedObjects, the option also shows class, schema, and helper.
`-ldir`	Lists PublishingContexts in long format, ignoring PublishingObjects; analogous to Unix `ls -ld` command.
`-R`	Lists recursively.
`-version`	Shows the command's version, then exits.

Here are examples of the `ls` command.

Show contents of the root PublishingContext in short format:

```
$ ls /
bin/
etc/
test/
```

Show contents of the root PublishingContext in long format:

```
$ ls -l /
Read    Write   Exec      Owner   Date   Time   Name      Schema    Class     Helper
PUBLIC  SYS     PUBLIC    SYS     Dec 14 14:59  bin/
PUBLIC  SYS     PUBLIC    SYS     Dec 14 14:59  etc/
PUBLIC  PUBLIC  PUBLIC    SYS     Dec 14 14:59  test/
```

Show contents of the `/test` PublishingContext in long format:

```
$ ls -l test
Read  Write Exec  Owner Date   Time  Name Schema Class                  Helper
SCOTT SCOTT SCOTT SCOTT Dec 14 16:32 bank SCOTT  Bank.AccountManagerImpl Bank.AccountManagerHelper
```

mkdir Command

The `mkdir` command is analogous to the Unix shell `mkdir` command; it creates a PublishingContext. You must have the write right for the target PublishingContext to use `mkdir` in it.

Syntax

```
mkdir [options] <name>
 [-path]
```

Argument Summary

Table 6–12 describes the `mkdir` arguments.

Table 6–12 mkdir Argument Summary

Option	Description
`<name>`	Name of PublishingContext to create.
`-path`	Creates intermediate PublishingContexts if they do not exist.

Here are examples of the `mkdir` command.

Create a PublishingContext called `/test/alpha` (`/test` exists):

```
mkdir /test/alpha
```

Create a PublishingContext called `/test/alpha/beta/gamma` (`/test/alpha/beta` does not exist):

```
$ mkdir -path /test/alpha/beta/gamma
```

mv Command

The `mv` command is analogous to the Unix shell `mv` command.

Syntax

```
mv <old> <new>
```

Here is an example of the `mv` command.

Change the name of `/test/foo` to `/test/bar`:

```
$ mv /test/foo /test/bar
```

publish Command

The `publish` command creates or replaces (republishes) a PublishedObject in a PublishingContext. It is not necessary to republish when you update a Java class schema object that has been published; republish only to change a PublishedObject's attributes. To publish, you must have the write right for the destination PublishingContext; to republish you must also have the write right for the PublishedObject.

Syntax

```
publish <name> <class> <helper> [options]
  [{-e | -executable}]
  [{-g | -grant} {<user> | <role>}[,{<user> | <role>} ... ]]
  [{-h | -help}]
  [-republish]
  [-schema <schema>]
  [-version]
```

Argument Summary

Table 6–13 summarizes the `publish` command arguments.

Table 6–13 publish Command Argument Summary

Option	Description
`<name>`	Name of the PublishedObject being created or republished; PublishingContexts are created if necessary.
`<class>`	Name of the class schema object that corresponds to <name>.
`<helper>`	Name of the Java class schema object that implements the `narrow()` method for `<class>`.
`-grant`	After creating or republishing the PublishedObject, grants read and execute rights to the sequence of `<user>` and `<role>` names. When republishing, replaces the existing users/roles that have read/execute rights with the `<user>` and `<role>` names. To selectively change the rights of a PublishedObject, use the `session shell`'s `chmod` command. Note that to activate a CORBA object or EJB, a user must have the execute right for both the PublishedObject and the corresponding class schema object.
`-help`	Summarizes the command's syntax, then exits.
`-republish`	Directs `publish` to replace an existing PublishedObject; without this option, the `publish` command rejects an attempt to publish an existing name. If the PublishedObject does not exist, it is created. Republishing deletes non-owner rights; use the `-grant` option to add read/execute rights when republishing.
`-schema`	The schema containing the Java `<class>` schema object; if you do not specify, the command uses the invoker's schema.
`-version`	Shows the command's version, then exits.

Here is an example of the `publish` command.

Publish the CORBA server implementation `Bank.AccountManagerImpl` and its helper class as `/test/bank` in the command invoker's schema:

```
$ ls -l /test
$ publish /test/bank Bank.AccountManagerImpl Bank.AccountManagerHelper
$ ls -l /test
Read  Write Exec  Owner Date   Time  Name  Schema Class                   Helper
SCOTT SCOTT SCOTT SCOTT Dec 14 16:32 bank  SCOTT  Bank.AccountManagerImpl Bank.AccountManagerHelper
```

pwd Command

The pwd command displays the name of the current working PublishingContext. It is analogous to the Unix `pwd` command.

Syntax

```
pwd
```

Here is an example of the `pwd` command.

```
$ pwd
/test/alpha
```

rm Command

The `rm` (remove) command is analogous to the `rm -r` Unix shell commands; it removes a PublishedObject or a PublishingContext, including its contents. To remove an object, you must have the write right for the containing PublishingContext.

Syntax

```
rm [options] <object> ... <object>
  [{-h | -help}]
  [-r]
  [-version]
```

Argument Summary

Table 6–14 describes the `rm` arguments.

Table 6–14 rm Argument Summary

Option	Description
`<object>`	Name of PublishedObject or PublishingContext to be removed.
`-help`	Summarizes the command’s syntax, then exits.
`-r`	Interprets `<object>` as a PublishingContext; removes it and its contents recursively.
`-version`	Shows the command’s version, then exits.

Here is an example of the `rm` command.

Remove the PublishedObject `/test/bank`:

```
rm /test/bank
```

Remove the PublishingContext `/test/release3` and everything it contains:

```
rm -r /test/release3
```

version Command

The `version` command shows the version of the `sess_sh` tool.

Syntax

```
version
```

Here is an example of the `version` command.

Display the session shell’s version:

```
$ version
1.0
```

Enterprise JavaBean Tools

Instead of `loadjava` and `publish`, Enterprise JavaBean developers use the `deployejb` tool, which performs equivalent operations, as well as generating and compiling infrastructure code for the EJB. The `ejbdescriptor` tool is a utility for translating between the text and serialized object forms of EJB deployment descriptors.

deployejb

From a deployment descriptor and a JAR containing interfaces and classes, the `deployejb` tool makes an EJB implementation ready for test or production clients to invoke. `deployejb` converts the text descriptor to a serialized object, generates and compiles classes that effect client-bean communication, loads compiled classes into the database, and publishes the bean's home interface name in the session namespace so clients can look it up with JNDI. The BeanHomeName must refer to a PublishingContext for which the `deployejb` invoker has the write right; see "publish" on page 6-3 for the rights required to publish.

To invoke a deployed bean, the client's CLASSPATH must include the remote and home interface files and the JAR generated by `deployejb`.

Syntax

```
deployejb -user <username> -password <password> -service <serviceURL>
-descriptor <file> -temp <dir> <beanjar>
  [-addclasspath <dirlist>]
  [-describe]
  [-generated <clientjar>]
  [-help]
  [-iiop]
  [-keep]
  [-republish]
  [-role <role>]
  [-ssl]
  [-useServiceName]
  [-verbose]
  [-version]
```

Argument Summary

Table 6–15 summarizes the `deployejb` arguments.

Table 6–15 deployejb Argument Summary

Argument	Description and Values
`-user`	Specifies the schema into which the EJB classes will be loaded.
`-password`	Specifies the password for `<username>`.

Table 6–15 deployejb Argument Summary (Cont.)

Argument	Description and Values
`-service`	URL identifying database in whose session namespace the EJB is to be published. The serviceURL has the form: `sess_iiop://<host>:<lport>:<sid>` `<host>` is the computer that hosts the target database; `<lport>` is the listener port configured to listen for session IIOP; `<sid>` is the database instance identifier. Example: `sess_iiop://localhost:2481:orcl` which matches the default installation on the invoker's machine.
`-descriptor`	Specifies the text file containing the EJB deployment descriptor.
`-temp`	Specifies a temporary directory to hold intermediate files `deployejb` creates. Unless you specify `-keep`, `deployejb` removes the files and the directory when it completes.
`<beanjar>`	Specifies the name of the JAR containing the bean interface and implementation files.
`-addclasspath`	Specifies directories containing interface and/or implementation dependency classes not contained in `<beanjar>`. Format of `<dirlist>` is the same as `javac`'s `CLASSPATH` argument. Required for `-beanonly`.
`-beanonly`	Skips generation of interface files. This is useful if you change only the bean implementation.
`-describe`	Summarizes the tool's operation, then exits.
`-generated`	Specifies the name of the output (generated) JAR file, which contains communication files bean clients need. If you do not specify, the output JAR file has the name of the input JAR file with `_generated` appended.
`-help`	Summarizes the tool's syntax, then exits.
`-iiop`	Connects to the target database with IIOP instead of the default session IIOP. Use this option when deploying to a database server that has been configured without session IIOP.
`-keep`	Do not remove the temporary files generated by the tool. This option may be useful for debugging because it provides access to the source files `deployejb` generates.
`-republish`	Replaces the published BeanHomeName attributes if the BeanHomeName has already been published, otherwise publishes it.

Table 6–15 deployejb Argument Summary (Cont.)

Argument	Description and Values
`-role`	Specifies role to assume when connecting to the database; no default.
`-ssl`	Connects to the database with SSL authentication and encryption.
`-useServiceName`	If you are using a service name instead of an SID in the URL, you must specify this flag. Otherwise, the tool assumes the last string in the URL is the SID.
`-verbose`	Emits detailed status information while running.
`-version`	Shows the tool's version, then exits.

Argument Details

addclasspath

`deployejb` needs the classes the home and remote interfaces depend on and the classes the bean implementation depends on. These dependency classes can either be included in the `<beanjar>` file or directories containing them or can be specified in the `-addclasspath` argument. The first approach is less prone to error, the second can substantially reduce `deployejb`'s run time. If you use `-addclasspath`, then you must ensure that the classes have been loaded before you run a client that activates the EJB.

Here is a `deployejb` example.

Basic invocation specifying the name of the generated client JAR file:

```
deployejb -user scott -password tiger -service sess_iiop://dbserver:2481:orcl \
  -descriptor myBeanDescriptor.txt -temp /tmp/ejb \
  -generated myBeanClient.jar myBean.jar
```

ejbdescriptor

Each EJB implementation includes a serialized Java object known as a deployment descriptor. The values in a deployment descriptor are not readable by people, yet people must create them and might sometimes have to read them. The `ejbdescriptor` tool transforms a serialized deployment descriptor to text and converse. Developers are most likely to use `ejbdescriptor` to extract the deployment descriptor data from an EJB developed for a non-Oracle environment. The `deployejb` tool calls `ejbdescriptor` to build a deployment descriptor from the text file you specify in the `-descriptor` argument.

Syntax

```
ejbdescriptor
  {-parse | -dump}
  <infile> <outfile>
```

Argument Summary

Table 6–16 describes the `ejbdescriptor` arguments.

Table 6–16 ejbdescriptor Argument Summary

Option	Description
`-parse`	Creates serialized deployment descriptor `<outfile>` from `<infile>`.
`-dump`	Creates text file `<outfile>` from serialized deployment descriptor `<infile>`.
`infile`	Name of text file (`-parse`) or serialized deployment descriptor (`-dump`) to read. The default is standard in. The conventional suffix for a descriptor text file is `.ejb`; for a serialized descriptor it is `.ser`.
`outfile`	Name of text file (`-dump`) or serialized deployment descriptor (`-parse`) to write. The default is standard out. The conventional suffix for a descriptor text file is `.ejb`; for a serialized descriptor it is `.ser`.

Here are examples of the `ejbdescriptor` tool.

Create a text file representation of a descriptor:

```
ejbdescriptor -dump beandescriptor.ser beandescriptor.ejb
```

Create a serialized deployment descriptor from a text file:

```
ejbdescriptor -parse beandescriptor.ejb beandescriptor.ser
```

Display the contents of a deployment descriptor:

```
ejbdescriptor -dump beandescriptor.ser
```

VisiBroker™ for Java Tools

The `idl2java`, `java2idl`, and `java2iiop` tools developed by Inprise for their VisiBroker for Java product (release 3.2) are distributed with Oracle8*i*. The Oracle8*i* JServer CD contains the documentation for these tools; the documentation can also be viewed or downloaded from `http://www.inprise.com`. Because the Oracle8*i* run-time environment differs somewhat from the VisiBroker environment, some VisiBroker tool options might not work in Oracle8*i* JServer as they are described in the VisiBroker documentation. In particular, do not specify the `-portable` option to `idl2java` or `java2iiop,` because because the current Oracle8*i* ORB does not support DII.

Miscellaneous Tools

This section describes special-purpose tools.

java2rmi_iiop

In the current JServer Enterprise JavaBeans implementation, EJBs communicate with clients by RMI-over-IIOP. This presents a difficulty for a CORBA client that wants to pass an object to an EJB for the EJB to invoke (call back) because the CORBA transport is IIOP, not RMI-over-IIOP. The CORBA client needs to pass the EJB an object the EJB can invoke with RMI-over-IIOP. The `java2rmi_iiop` tool generates the stubs, skeletons, and other classes a client or server needs to make an object remotely invocable by an EJB. (`java2rmi_iiop` is the analog of the VisiBroker for Java `java2iiop` tool, except that it expects interfaces that extend `java.rmi.Remote` rather than `org.omg.CORBA.Object`)

The Java interface definitions must follow the RMI spec:

- Interfaces must extend `java.rmi.Remote.`
- All remote methods must throw at least `java.rmi.RemoteException.`
- All arguments and return values of the remote methods must be valid RMI types.

Syntax

```
java2rmi_iiop [options] <file>.java ...
  [-no_bind]
  [-no_comments]
  [-no_examples]
  [-no_tie]
  [-root_dir <directory>]
  [-verbose]
  [-version]
  [-W <number>]
  [-wide]
```

Argument Summary

Table 6–17 summarizes the `java2rmi_iiop` arguments.

Table 6–17 java2rmi_iiop Argument Summary

Argument	Description
`-nobind`	Suppresses the generation of `bind()` methods.
`-no_comments`	Suppresses comments in generated code.
`-no_examples`	Suppresses the generation of example code.
`-no_tie`	Suppresses the generation of tie code.
`-root_dir`	Places all generated files in the specified directory instead of in the current directory.
`-verbose`	Emits extra messages.
`-version`	Displays the version of VisiBroker for Java that you are currently running.
`-W`	Setting this option to 0 (zero) suppresses all warnings from the compiler.
`-wide`	Maps Java `String`/`char` to IDL `wstring`/`wchar`.

Example

Generate RMI-over-IIOP class files for an RMI interface:

```
java2rmi_iiop Dictionary.java
```

modifyprops

Some aspects of the Oracle8*i* ORB are governed by properties it reads when a new session running the ORB starts. You can change these properties with the `modifyprops` tool. Developers should change ORB properties only when Oracle technical support provides instructions to do so.

Syntax

```
modifyprops {-u | -user} <user/password@<database> [options]
{<key> <value> [,<key> <value>] ... | <key> -delete}
  [{-o | -oci8}]
  [{-t | -thin}]
```

Argument Summary

Table 6–18 summarizes the `modifyprops` arguments.

Table 6–18 modifyprops Argument Summary

Argument	Description
`-user`	Specifies a user, password, and optional database connect string. See "user" on page 6-28 for details.
`-oci8`	Directs `modifyprops` to connect with the database using the oci8 JDBC driver. `-oci8` and `-thin` are mutually exclusive; if neither is specified, then `-oci8` is used by default. Choosing `-oci8` implies the form of the database connect string. See "user" on page 6-28 for details.
`-thin`	Directs `modifyprops` to communicate with the database using the thin JDBC driver. `-oci8` and `-thin` are mutually exclusive; if neither is specified, then `-oci8` is used by default. Choosing `-thin` implies the form of database connect string See "user" on page 6-28 for details.
`<key> <value>`	Oracle technical support will advise you of the values to enter for `<key>` and `<value>`.

Argument Details

user

```
{-user | -u} <user>/<password>[@<database>]
```

The permissible forms of `@<database>` depend on whether you specify `-oci8` or `-thin`; `-oci8` is the default.

- `-oci8`: `@<database>` is optional. If you do not specify, then `modifyprops` uses the user's default database. If specified, then `<database>` can be a TNS name or a Net8 name-value list.
- `-thin`: `@<database>` is required. The format is `<host>:<lport>:<SID>`.
  - `<host>` is the name of the machine running the database.
  - `<lport>` is the listener port that has been configured to listen for Net8 connections. In a default installation, it is 5521.
  - `<SID>` is the database instance identifier. In a default installation it is `ORCL`.

A
Example Code: CORBA

This chapter contains all the CORBA example code included on the product CD. See the EJB/CORBA README for the locations of the examples.

Basic Examples

The following are the basic CORBA examples:

- helloworld
- sqljimpl
- jdbcimpl
- factory
- lookup
- callback
- printback
- tieimpl
- bank
- pureCorba

Here is the README for the basic examples:

```
The examples in the basic/ directories demonstrate various CORBA
programming techniques that you can use to write CORBA server
objects, as well as the client code that calls the server object.

The examples are short, and each example shows just one or two aspects
of Oracle8i CORBA programming. The examples come with either a
standard Makefile (UNIX) or a batch file (Windows NT) that will
perform all the steps required to compile, load, and run the example.

To run an example, you must have access to an Oracle8i database
server that hosts the Oracle8i server-side Java VM, and that has
the standard SCOTT demo schema installed. Some of the examples
use the EMP and DEPT demo tables in the SCOTT schema.

The SCOTT schema must also have write access to the CORBA name space
starting at the 'test' directory, which is true of the install database.
The tables that support the publishing directories are established when
your Oracle8i system with the Java option is built. You can use the
Session Shell to verify the presence of the test directory. See the
Oracle8i EJB and CORBA Developer's Guide for information about the
Session Shell.

You must also have the INIT.ORA, tnsnames.ora, and listener.ora files
```

```
configured properly to accept both standard listener and IIOP incoming
connections which is done for you in the install database. See the
Oracle8i Net8 Administrator's Guide for information about setting up
these files.

Each example publishes one or more objects in the database. To lookup
and activate the published object, the client uses the Oracle8i JNDI
interface to the CosNaming implementation. The examples all connect
using the SCOTT as the username, TIGER as the password, and for
simplicity, NON_SSL_LOGIN as the connection protocol.

The makefiles/batch files provided with the examples expect that you
have the java and javac programs from the Sun JDK 1.1.3 (beta) or JDK
1.1.6 (production) in your PATH.  They also expect that your CLASSPATH
contains the Java runtime classes (classes.zip) corresponding to your
java interpreter.  The UNIX makefiles abd NT batch files take care of
adding the ORACLE specific jar and zip files to your CLASSPATH.

For your reference here is a list of jar and zip files that the
makefiles/batch files use:

ORACLE_HOME/lib/aurora_client.jar       # Oracle 8i ORB runtime
ORACLE_HOME/lib/aurora.jar              # Oracle 8i in-the-database runtime
#use classes12.zip if using Java 2
ORACLE_HOME/jdbc/lib/classes111.zip     # for JDBC examples
ORACLE_HOME/sqlj/lib/translator.zip     # for SQLJ examples
ORACLE_HOME/lib/vbjapp.jar              # Inprise VisiBroker library
ORACLE_HOME/lib/vbjorb.jar              # VisiBroker library
ORACLE_HOME/lib/vbj30ssl.jar            # required if you modify any
                                        # client code to use SSL
The example programs are:

helloworld - The CORBA version of {printf("Hello world!");}. Look at
             this example first

bank       - an Oracle8i-compatible version of the VisiBroker Bank
             example.

sqljimpl   - Uses server-side JDBC to retrieve data from a database
             server.  Uses the SQLJ preprocessor.  Demonstrates CORBA
             structs, sequences and exceptions.

jdbcimpl   - Like sqljimpl, but uses the more verbose JDBC syntax to retrieve
             the data.
```

```
factory    - Oracle8i implementation of the factory design pattern.

lookup     - Demonstrates one CORBA server object activating and
             calling an other CORBA object in its own session.
             Also demonstrates CORBA structs and sequences.

callback   - Shows how to call a client from a server object.

printback  - Shows how to print data from a server object
             on the client console or screen.

tieimpl    - Demonstrates using the CORBA TIE (delegation) method
             instead of inheritance to code a CORBA object. This
             is the helloworld example done with TIE rather than
             inheritance.


The code in the examples is not always commented, but each of the
examples has its own readme file. The readme explains what the code
does, and points out any special features used in the example.

Each of these examples has been tested on Solaris 2.6 and Windows
NT 4.0. If you have problems compiling or running the examples on
these or on another supported platform, please inform your Oracle
support representative.
```

helloworld

readme.txt

```
Overview
========

This is a very simple CORBA example. The helloWorld server object merely
returns a greeting plus the Java VM version number to the client.

The purpose of the example is to show the minimum code needed to
lookup a published object, activate it by invoking a method on it, and
use the value that the method returns on the client side.

Note that the name of the object as published in the database is 'myHello',
and not the class name 'HelloImpl'. The name of the published object is
completely independent of its class name. In this and other examples, the only
place that the published object name is visible is in the Makefile or the
```

```
runit.bat batch file, in the publish and run targets.

Note also that the publish command passes in the name of the CORBA helper
class. The ORB on the server side uses the helper object to narrow the object
that it looks up to the appropriate type.


Source files
============

hello.idl
------------

The CORBA IDL for the example.  Defines a single interface Hello with a single
method helloWorld(). The interface is defined in the Module named 'hello',
which determines the name of the directory in which the idl2java compiler
places the generated files.

The helloWorld() method returns a CORBA wstring, which maps to a Java String
type:

module hello
  interface Hello
    wstring helloWorld()


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
```

```
#use classes12.zip if using Java 2
$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 CORBA server object to find and activate it
   - 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.



helloServer/HelloImpl.java
--------------------------

Implements the IDL-specified Hello interface. The interface has one
method, helloWorld(), that returns a String to the caller.

helloWorld() invokes System.getProperty("oracle.server.version") to get the
version number of the Java VM.

This object performs no database access.



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.
```

hello.idl

```
module hello {
  interface Hello {
    wstring helloWorld ();
  };
};
```

Client.java

```
import hello.Hello;
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);

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

helloServer/HelloImpl.java

```
package helloServer;

import hello.*;

public class HelloImpl extends _HelloImplBase {
  public String helloWorld() {
    String v = System.getProperty("oracle.server.version");
    return "Hello client, your javavm version is " + v + ".";
  }
}
```

sqljimpl

readme.txt

```
Overview
========

The example shows:

    - how to use the SQLJ translator on the server side to query data from the
        EMP table.
    - returning complex data to the client using an IDL struct/Java class

This example is a SQLJ version of the jdbcimpl example. It is useful to
compare the two examples.


Source files
============

employee.idl
------------

See the employee.idl description in ../jdbcimpl/readme.txt.


Client.java
-----------

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
#use classes12.zip if using Java 2
```

```
$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 Employee CORBA server object to find and activate it
   - invokes the getEmployee() method, with the parameter "SCOTT", to
       return Scott's employee ID and salary
   - prints the result
   - tries to use getEmployee("bogus") to return information about
       employee named bogus. This will fail, and return the SQLError
       exception, which is printed.

The printed output is:

SCOTT 7788 3000.0
Error retrieving employee "bogus": no rows found for select into statement


employeeServer/EmployeeImpl.java
--------------------------------

This class implements the Employee interface. The getEmployee() method
simply declares two variables to hold the empno and sal information
from the EMP table. The method then defines and calls a SQLJ statement
that selects information about the employee named in the input
parameter into the variables, constructs a new EmployeeInfo object
using the query information, and returns it to the invoker.

It is instructive to contrast this example with the jdbcimpl example,
which uses JDBC rather than SQLJ to query the database.

You can also contrast this example with the lookup example, which uses a SQLJ
iterator to retrieve a multi-row result set from the database.


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.
```

employee.idl

```
module employee {
  struct EmployeeInfo {
    wstring name;
    long number;
    double salary;
  };

  exception SQLError {
    wstring message;
  };

  interface Employee {
    EmployeeInfo getEmployee (in wstring name) raises (SQLError);
  };
};
```

Client.java

```
import employee.*;

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);

    try {
      Employee employee = (Employee)ic.lookup (serviceURL + objectName);
      EmployeeInfo info = employee.getEmployee ("SCOTT");
      System.out.println (info.name + " " + info.number + " " + info.salary);
      // This one will fail and raise a SQLError exception
      EmployeeInfo info2 = employee.getEmployee ("bogus");
      System.out.println (info.name + " " + info.number + " " + info.salary);
    } catch (SQLError e) {
      System.out.println ("Error retrieving employee \"bogus\": " + e.message);
    }
  }
}
```

employeeServer/employeeImpl.sqlj

```
package employeeServer;

import employee.*;
import oracle.aurora.AuroraServices.ActivatableObject;
import java.sql.*;

public class EmployeeImpl
       extends _EmployeeImplBase
       implements ActivatableObject
{
  public EmployeeInfo getEmployee (String name) throws SQLError {
    try {
      int empno = 0;
      double salary = 0.0;
      #sql { select empno, sal into :empno, :salary from emp
            where ename = :name };
      return new EmployeeInfo (name, empno, (float)salary);
    } catch (SQLException e) {
      throw new SQLError (e.getMessage ());
    }
  }

  public org.omg.CORBA.Object _initializeAuroraObject () {
    return this;
  }
}
```

jdbcimpl

readme.txt

```
Overview
========
This example demonstrates:

   - how to use JDBC calls on the server side to query
       data from the EMP table
   - how to return complex data to the client using an IDL struct/Java class.
   - handling SQLException exceptions on the server side and returning
       them as CORBA exceptions.


Source files
============

employee.idl
------------

The CORBA IDL for this example defines a struct, an exception, and one
interface.

module employee
  struct EmployeeInfo
    wstring name
    long number
    double salary

  exception SQLError
    wstring message

  interface Employee
    EmployeeInfo getEmployee (in wstring name) raises (SQLError)

The EmployeeInfo struct is defined to consist of a string for the
employee name, and two numerics for employee number and salary.

The SQLError exception returns SQL exceptions to the client
invoker.

The Employee interface defines a method that returns an EmployeeInfo
struct, and takes an employee name as its input parameter.


Client.java
-----------

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
#use classes12.zip if using Java 2
$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 Employee CORBA server object to find and activate it
   - invokes the getEmployee() method, with the parameter "SCOTT", to
       return Scott's employee ID and salary
   - prints the result
   - tries to use getEmployee("bogus") to return information about
       employee named bogus. This will fail, and return the SQLError
       exception, which is printed.

The printed output is:



employeeServer/EmployeeImpl.java
--------------------------------

This class implements the Employee interface. The getEmployee() method
gets access to the defatul server-side JDBC connection, then uses a
PreparedStatement to construct a query for EMPNO and SAL on the EMP
table. The query WHERE clause is constructed from the in parameter
ENAME.
```

```
The prepared statement is executed, and the information for the
(first) employee of that name is extracted from the result set, and
inserted into a new EmployeeInfo object, which is then returned to the
invoker.

Note the use of the finally {} clause to close the prepared statement,
which also closes the result set.



Client application output
=========================

The client application prints:

SCOTT 7788 3000.0
Error retrieving employee "bogus": no employee named bogus


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.

employee.idl

```
module employee {
  struct EmployeeInfo {
    wstring name;
    long number;
    double salary;
  };

  exception SQLError {
    wstring message;
  };

  interface Employee {
    EmployeeInfo getEmployee (in wstring name) raises (SQLError);
  };
};
```

Client.java

```
import employee.*;

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);

    try {
      Employee employee = (Employee)ic.lookup (serviceURL + objectName);
      EmployeeInfo info = employee.getEmployee ("SCOTT");
      System.out.println (info.name + " " + info.number + " " + info.salary);
      // This one will fail and raise a SQLError exception
      EmployeeInfo info2 = employee.getEmployee ("bogus");
    } catch (SQLError e) {
      System.out.println ("Error retrieving employee \"bogus\": " + e.message);
    }
  }
}
```

employeeServer/EmployeeImpl.java

```
package employeeServer;

import employee.*;
import oracle.aurora.AuroraServices.ActivatableObject;
import java.sql.*;

public class EmployeeImpl
       extends _EmployeeImplBase
       implements ActivatableObject
{
  public EmployeeInfo getEmployee (String name) throws SQLError {
    try {
      Connection conn =
new oracle.jdbc.driver.OracleDriver().defaultConnection ();
      PreparedStatement ps =
conn.prepareStatement ("select empno, sal from emp where ename = ?");
      try {
ps.setString (1, name);
ResultSet rset = ps.executeQuery ();
if (!rset.next ())
  throw new SQLError ("no employee named " + name);
return new EmployeeInfo (name, rset.getInt (1), rset.getFloat (2));
      } finally {
ps.close ();
      }
    } catch (SQLException e) {
      throw new SQLError (e.getMessage ());
    }
  }

  public org.omg.CORBA.Object _initializeAuroraObject () {
    return this;
  }
}
```

factory

readme.txt

```
Overview
========

This example demonstrates a CORBA factory design pattern for a simple object.
It uses the orb.connect() method to register the transient (i.e. unnamed)
object created by the factory.


Source files
============

factory.idl
----------

The CORBA IDL that defines the server-side objects. It defines two interfaces:
```

```
interface Hello
  wstring helloWorld ()

interface HelloFactory {
  Hello create (in wstring message)

HelloFactory is used to create new Hello objects. The Hello object is
just the simple object, as in the helloworld example in this set, that
returns a greeting String to the client invoker. In this example, the
factory creates the object with a specified content.


Client.java
-----------

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
#use classes12.zip if using Java 2
$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 HelloFactory CORBA server object to find and
activate it
   - invokes the factory create() method twice to create two separate objects
       in the session. The create() method sets the greeting that is returned
   - on each object, invokes the helloWorld() method
   - prints the result

The printed output is:

Hello World!
Goodbye World!



factoryServer/HelloFactoryImpl.java
-----------------------------------

This class implements the HelloFactory interface. It creates a new
Hello object (compare the Hello interface), and registers the new
object with the server-side Basic Object Adapter (BOA) using the
connect() method. connect() is the portable version of obj_is_ready().

The created object reference is then returned to the invoker.


factoryServer/HelloImpl.java
----------------------------

This class implements the Hello interface. It contains a public
constructor that saves the message, and one method, helloWorld(),
that returns the message passed in the constructor to the invoker.


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.
```

factory.idl

```
module factory {
  interface Hello {
    wstring helloWorld ();
  };
  interface HelloFactory {
    Hello create (in wstring message);
  };
};
```

Client.java

```
import factory.*;

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);

    HelloFactory factory = (HelloFactory)ic.lookup (serviceURL + objectName);
    Hello hello = factory.create ("Hello World!");
    Hello hello2 = factory.create ("Goodbye World!");
    System.out.println (hello.helloWorld ());
    System.out.println (hello2.helloWorld ());
  }
}
```

factoryServer/HelloImpl.java

```
package factoryServer;

import factory.*;

public class HelloImpl extends _HelloImplBase
{
  String message;

  public HelloImpl (String message) {
    this.message = message;
  }

  public String helloWorld () {
    return message;
  }
}
```

factoryServer/HelloFactoryImpl.java

```
package factoryServer;

import factory.*;
import oracle.aurora.AuroraServices.ActivatableObject;

public class HelloFactoryImpl
       extends _HelloFactoryImplBase
       implements ActivatableObject
{
  public Hello create (String message) {
    HelloImpl hello = new HelloImpl (message);
    _orb().connect (hello);
    return hello;
  }

  public org.omg.CORBA.Object _initializeAuroraObject () {
    return this;
  }
}
```

lookup

readme.txt

```
Overview
========

This example demonstrates:

    - using CORBA structs and sequences
    - one CORBA object invoking and calling another in the same session,
        using 'thisSession' in the URL.
    - using the SQLJ translator for ease in implementing static SQL DML
        statements.
    - exception handling.


Source files
```

```
============

employee.idl
------------

The CORBA IDL for the example.  Defines:

EmployeeInfo struct
sequence of EmployeeInfo
DepartmentInfo struct, containing the sequence
SQLError CORBA exception
Employee interface
  getEmployees()
Department interface
  getDepartment()

The SQLError exception is used so that SQLException messages can
be passed back to the client.


Client.java
-----------

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
#use classes12.zip if using Java 2
$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 Employee CORBA server object to find and activate it
   - looks up the published Department CORBA server object to find and activate
it
   - invokes the getDepartment() method on it, passing in a department
       number. This method returns a DepartmentInfo struct (class), which
       contains information about the department plus a Java vector of
       employee names, ID numbers, and salaries for each employee in
       the specified department.
   - prints the returned information in a for(;;) loop, one iteration
       for each employee in the department

For the standard demo EMP and DEPT tables, the client prints:

RESEARCH 20 DALLAS
  SMITH 800.0
  JONES 2975.0
  SCOTT 3000.0
  ADAMS 1100.0
  FORD 3000.0


employeeServer/DepartmentImpl.sqlj
----------------------------------

Implements the IDL-specified Department interface.  The interface
has one method, getDepartment(), that returns the information
about the department and each of the employees in it.  The most
interesting thing to note about this method is that it looks up
and activates a second CORBA server object, that was published in
the database as /test/myEmployee, and calls a method on it.

Note that the employee object is activated *in the same session*
through the use of the thisServer literal in the URL.

This method returns a DepartmentInfo struct.


employeeServer/EmployeeImpl.sqlj
--------------------------------

Implements the Employee interface.  There is one method --
```

```
getEmployees(). This method queries the EMP table, using a
SQLJ named iterator, and returns an array of EmployeeInfo structs.
The caller (getDepartment()) combines the array returned by
getEmployees() with the results of its own query for the
department attributes, and returns all the information to the
client program.

If the SQLJ code throws a SQLException, it is caught, and a
CORBA-defined SQLError is thrown. This in turn would be
propagated back to the client, where it is handled.



Compiling and Running the Example
=================================

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.
```

employee.idl

```
module employee {
  struct EmployeeInfo {
    long empno;
    wstring ename;
    double sal;
  };

  typedef sequence <EmployeeInfo> employeeInfos;

  struct DepartmentInfo {
    long deptno;
    wstring dname;
    wstring loc;
    EmployeeInfos employees;
  };

  exception SQLError {
    wstring message;
  };

  interface Employee {
    EmployeeInfos getEmployees (in long deptno) raises (SQLError);
  };

  interface Department {
    DepartmentInfo getDepartment (in long deptno) raises (SQLError);
  };
};
```

Client.java

```
import employee.*;

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);

    Department department = (Department) ic.lookup (serviceURL + objectName);
    DepartmentInfo info = department.getDepartment (20);
    System.out.println (info.dname + " " + info.deptno + " " + info.loc);

    EmployeeInfo[] infos = info.employees;
    int i;
    for (i = 0; i < infos.length; i++)
      System.out.println ("  " + infos[i].ename + " " + infos[i].sal);
  }
}
```

employeeServer/DepartmentImpl.sqlj

```
package employeeServer;

import employee.*;
import oracle.aurora.jndi.sess_iiop.ServiceCtx;
import oracle.aurora.AuroraServices.ActivatableObject;
import javax.naming.*;
import java.sql.*;
import java.util.*;

public class DepartmentImpl
```

```
       extends _DepartmentImplBase
       implements ActivatableObject
{
  Employee employee = null;

  public DepartmentInfo getDepartment (int deptno) throws SQLError {
    try {
      if (employee == null) {
Hashtable env = new Hashtable ();
env.put (Context.URL_PKG_PREFIXES, "oracle.aurora.jndi");
Context ic = new InitialContext (env);
employee =
  (Employee)ic.lookup ("sess_iiop://thisServer/test/myEmployee");
      }

      EmployeeInfo[] employees = employee.getEmployees (deptno);
      String dname;
      String loc;
      #sql { select dname, loc into :dname, :loc from dept
                    where deptno = :deptno };

      return new DepartmentInfo (deptno, dname, loc, employees);
    } catch (SQLException e) {
      throw new SQLError (e.getMessage ());
    } catch (NamingException e) {
      throw new SQLError ("Naming Exception: " + e.getMessage ());
    }
  }

  public org.omg.CORBA.Object _initializeAuroraObject () {
    return this;
  }
}
```

employeeServer/EmployeeImpl.sqlj

```
package employeeServer;

import employee.*;
import oracle.aurora.AuroraServices.ActivatableObject;
import java.sql.*;
import java.util.Vector;

#sql iterator EmpIterator (int empno, String ename, double sal);
```

```
public class EmployeeImpl extends _EmployeeImplBase {
  public EmployeeInfo[] getEmployees (int deptno) throws SQLError {
    try {
      Vector vector = new Vector ();
      EmpIterator empit;
      #sql empit = { select empno, ename, sal from emp
            where deptno = :deptno };
      while (empit.next ())
           vector.addElement (new EmployeeInfo (empit.empno(), empit.ename(),
     empit.sal()));
      empit.close ();
      EmployeeInfo[] result = new EmployeeInfo[vector.size ()];
      vector.copyInto (result);
      return result;
    } catch (SQLException e) {
      throw new SQLError (e.getMessage ());
    }
  }
}
```

callback

readme.txt

```
Overview
========

callback shows a CORBA server object that calls back to the client-side
object. It works by activating a new object in the client-side ORB, using the
Basic Object Adapter (BOA), and boa.obj_is_ready(), and sending a reference to
that object to the CORBA server object.



Source files
============

client.idl
----------

The CORBA IDL that defines the client-side object, that will be called
from the server.

interface Client
```

```
  wstring helloBack()


server.idl
----------

The CORBA IDL that defines the server-side object, that will be called
from the client, and that will in turn call back to the client.

interface Server
  wstring hello (in client::Client object)

Since the object is registered on the client side, and is not
published in the database, to perform a callback the server object
must have a reference to the client-side object. In this example, the
server is called with a reference to the object that has been
registered with the client-side Basic Object Adapter (BOA) as a
parameter.


Client.java
-----------

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
#use classes12.zip if using Java 2
$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 CORBA 'Server' object to find and activate it
   - starts up the ORB on the client system (ORB.init())
   - gets the basic object adapter object (BOA)
   - instantiates a new client callback object (new ClientImpl()), and
       registers it with the object adapater (boa.obj_is_ready(client))
   - invokes the hello() method on the server object, passing it the
       reference to the client callback object
```

It is important to do the lookup() before initializing the ORB on the Client side: The lookup call initializes the ORB in a way that's compatible with Oracle 8i. The following org.omg.CORBA.ORB.init() call does not initialize a new ORB instance but just returns the orb that was initialized by the lookup call.

The client prints:

```
I Called back and got: Hello Client World!
```

which is the concatenation of the strings returned by the server object, and the called-back client-side object.

serverServer/ServerImpl.java

This class implements the server interface. The code has one method, hello(), which returns its own String ("I called back and got: ") plus the String that it gets as the return from the callback to the client.

clientServer/ClientImpl.java

This class implements the client interface. It has a public constructor, which is required, and a single method, helloBack(), which simply returns the String "Hello Client World!" to the client that called it (the server object 'server' in this case).

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.idl

```
module client {
  interface Client {
    wstring helloBack ();
  };
};
```

Client.java

```
import server.*;
import client.*;
import clientServer.*;

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);

    // Get the server object before preparing the client object
    // You have to do it in that order to get the ORB initialized correctly
    Server server = (Server)ic.lookup (serviceURL + objectName);

    // Create the client object and publish it to the orb in the client
    //org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init ();
    com.visigenic.vbroker.orb.ORB orb = oracle.aurora.jndi.orb_dep.Orb.init();
    org.omg.CORBA.BOA boa = orb.BOA_init ();
    ClientImpl client = new ClientImpl ();
    boa.obj_is_ready (client);

    // Pass the client to the server that will call us back
    System.out.println (server.hello (client));
  }
}
```

server.idl

```
#include <client.idl>

module server {
  interface Server {
    wstring hello (in client::Client object);
  };
};
```

clientServer/ClientImpl.java

```
package clientServer;

import client.*;
import oracle.aurora.AuroraServices.ActivatableObject;

public class ClientImpl extends _ClientImplBase implements ActivatableObject
{
  public String helloBack () {
    return "Hello Client World!";
  }

  public org.omg.CORBA.Object _initializeAuroraObject () {
    return this;
  }
}
```

serverServer/ServerImpl.java

```
package serverServer;

import server.*;
import client.*;
import oracle.aurora.AuroraServices.ActivatableObject;

public class ServerImpl extends _ServerImplBase implements ActivatableObject
{
  public String hello (Client client) {
    return "I Called back and got: " + client.helloBack ();
  }

  public org.omg.CORBA.Object _initializeAuroraObject () {
    return this;
  }
}
```

printback

readme.txt

```
Overview
========

This example demonstrates how to write output to a file descriptor on the
client side from a CORBA server object.

This is a very handy technique for making output from a server object appear
on the console of the client. You can use it for debugging as well as other
informational purposes.


Source files
============

printer.idl
-----------

The IDL defines a byteArray as a sequence of octets, and one
interface, ByteStream, with write(), flush(), and close() methods.


interface ByteStream
  oneway void write(in byteArray bytes)
  oneway void flush()
  oneway void close()

Note that the methods are oneway, that is non-blocking.


hello.idl
---------
```

```
IDL to define the Hello interface.

interface Hello
  oneway void setup (in printer::ByteStream stream)
  void helloWorld ();


Client.java
-----------

The client code looks up and activates the CORBA server object
(hello), then initializes the client-side ORB and BOA, and registers a
new ByteStreamImpl object with the BOA.

The parameter for the ByteStreamImpl constructor is a FileOutputStream
object, with the out handle as its target.

The client then invokes the hello.setup() method, with the
BOA-registered ByteStream object as the parameter. This essentially
resets 'out' to point to the RemoteOutputStream class, which overrides
the write() and close() methods of the standard PrintStream that is
normally attached to 'out'. Now, when the a server object writes to
'out', the output is redirected to the client-side ByteStream object,
where it can be printed on the client console.

To test this, the client then invokes hello.helloWorld(), which prints
back to the client-side ByteStream.


printerServer/ByteStreamImpl.java
---------------------------------

This class implements the client-side printer service. It implements
write() method that gets invoked by the server-side CORBA object, and
writes to the client console.


helloServer/HelloImpl.java
--------------------------

This class implements the methods directly called by the client:
setup() and helloWorld().


printerClient/RemoteOutputStream.java
```

```
-------------------------------------

This class implements methods that override the standard PrintStream
write(), flush(), and close() methods, for use by the
HelloImpl.setup() method.


Client application output
=========================

The client application prints:

Hello World!
counting 0
counting 1
counting 2
counting 3
counting 4
counting 5
counting 6
counting 7
counting 8
counting 9
counting 10
counting 11
counting 12
counting 13
counting 14
counting 15
counting 16
counting 17
counting 18
counting 19
Thats all
...(repeated 3 times, once for each client call to hello.helloWorld().

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.
```

printer.idl

```
module printer {
  typedef sequence<octet> byteArray;
  interface ByteStream {
    oneway void write (in byteArray bytes);
    oneway void flush ();
    oneway void close ();
  };
};
```

hello.idl

```
#include <printer.idl>
```

```
module hello {
  interface Hello {
    oneway void setup (in printer::ByteStream stream);
    void helloWorld ();
  };
};
```

Client.java

```
import hello.*;
import printerServer.*;

import oracle.aurora.jndi.sess_iiop.ServiceCtx;

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

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);

    // Get the server object before preparing the client object
    // You have to do it in that order to get the ORB initialized correctly
    Hello hello = (Hello)ic.lookup (serviceURL + objectName);

    // Create the client object and publish it to the orb in the client
    //org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init ();
    com.visigenic.vbroker.orb.ORB orb = oracle.aurora.jndi.orb_dep.Orb.init();
    org.omg.CORBA.BOA boa = orb.BOA_init ();
    ByteStreamImpl byte_stream =
      new ByteStreamImpl (new FileOutputStream (FileDescriptor.out));
    boa.obj_is_ready (byte_stream);

    // Pass the client to the server that will call us back
    hello.setup (byte_stream);
    hello.helloWorld ();
    hello.helloWorld ();
    hello.helloWorld ();
    Thread.sleep(3000);
    System.out.println("Thats all");
  }
}
```

printerClient/RemoteOutputStream.java

```
package printerClient;

import printer.ByteStream;

import java.io.OutputStream;
import java.io.PrintStream;
import java.io.IOException;

public class RemoteOutputStream extends OutputStream
{
  ByteStream remote;

  // Static entrypoint to make System.out and System.err use the
  // remote stream.
  public static void setStreams (ByteStream remote) {
    OutputStream os = new RemoteOutputStream (remote);
    PrintStream p = new PrintStream (os, true);
    System.setOut (p);
    System.setErr (p);
  }

  public RemoteOutputStream (ByteStream remote) {
    this.remote = remote;
  }

  public void write (int b) {
    byte[] buf = { (byte)b };
    write (buf);
  }

  public void write (byte b[]) {
    remote.write (b);
  }

  public void write (byte buf[], int off, int count) {
    if (off == 0 && count == buf.length)
      write (buf);
    else if (off >= 0 && off < buf.length && count > 0) {
      byte[] temp = new byte [count];
      System.arraycopy (buf, off, temp, 0, count);
      write (temp);
    }
  }

  public void flush () {
    //    remote.flush ();
  }

  public void close () {
    remote.close ();
  }
}
```

helloServer/HelloImpl.java

```
package helloServer;

import hello.*;
import printer.*;
import printerClient.*;
import java.io.PrintStream;

import oracle.aurora.AuroraServices.ActivatableObject;

public class HelloImpl extends _HelloImplBase implements ActivatableObject
{
  PrintStream out;
  ByteStream remote;

  public HelloImpl () {
    super ();
    out = null;
  }

  public void setup (ByteStream remote) {
    this.remote = remote;
    out = new PrintStream (new RemoteOutputStream (remote));
  }

  public void helloWorld () {
    if (out != null){
      out.println ("Hello World!");
      int i;
      for (i = 0; i < 20; i++)
out.println ("counting " + i);
    }
  }

  public org.omg.CORBA.Object _initializeAuroraObject () {
    return this;
  }
}
```

printerServer/ByteStreamImpl.java

```
package printerServer;

import java.io.OutputStream;
import java.io.IOException;

public class ByteStreamImpl extends printer._ByteStreamImplBase
{
  OutputStream stream;

  public ByteStreamImpl (OutputStream stream) {
    super ();
    this.stream = stream;
  }

  public void write (byte[] bytes) {
    try {
      stream.write (bytes);
    } catch (IOException e) {}
  }

  public void flush () {
    try {
      stream.flush ();
    } catch (IOException e) {}
  }

  public void close () {
    try {
      stream.close ();
    } catch (IOException e) {}
  }
}
```

tieimpl

readme.txt

```
Overview
========

This is a CORBA TIE (delegation) implementation of the helloworld example. See
the readme for that example for more information. It uses the
_initializeAuroraObject() method to return a class delegate, rather than the
object itself.


Source files
============

hello.idl
------------

(See the helloworld example readme file.)


Client.java
-----------

(See the helloworld example readme file.)


helloServer/HelloImpl.java
--------------------------

Implements the IDL-specified Hello interface. The interface has one
method, helloWorld(), that returns a String to the caller.
```

```
Note that the class definition *implements* the IDL-generated
HelloOperations interface, rather than extending _HelloImplBase, as in
the helloworld example.

The class also implements the Aurora ActivateableObject
interface. ActivatableObject has only one method:
_initializeAuroraObject(), which returns the class to be activated by
the BOA.

This class performs no database access.


Client-side output
==================

The client prints the returned String "Hello World!" and then exits
immediately.


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.

hello.idl

```
module hello {
  interface Hello {
    wstring helloWorld ();
  };
};
```

Client.java

```
import hello.Hello;

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);

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

helloServer/HelloImpl.java

```
package helloServer;

import hello.*;
import oracle.aurora.AuroraServices.ActivatableObject;

public class HelloImpl implements HelloOperations, ActivatableObject
{
  public String helloWorld () {
    return "Hello World!";
  }

  public org.omg.CORBA.Object _initializeAuroraObject () {
    return new _tie_Hello (this);
  }
}
```

bank

readme.txt

```
bank demonstrates:

This is an Oracle8i-compatible version of the VisiBroker Bank
example. The major differences from the Vb example are:

(1) There is no server main loop. For Oracle8i the
"wait-for-activation" loop is part of the IIOP presentation (MTS
server).

(2) _boa.connect(object) is used instead of the less portable
 _boa_obj_is_ready(object) in the server object implementation to
register the new Account objects.

(3) The client program contains the code necessary to lookup the
AccountManager object (published under /test/myBank) and activate it,
and to authenticate the client to the server. (Note that object
activation and authentication, via NON_SSL_LOGIN, happen "under the
covers" so to speak on the lookup() method invocation.)

(4) There is also a tie implementation of this example, with the
server being AccountManagerImplTie.java.
```

Bank.idl

```
// Bank.idl

module Bank {
  interface Account {
    float balance();
  };
  interface AccountManager {
    Account open(in string name);
  };
};
```

Client.java

```
// Client.java

import bankServer.*;
import Bank.*;

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 objectName user password "
 + "accountName");
      System.exit(1);
    }
    String serviceURL = args [0];
    String objectName = args [1];
    String user = args [2];
    String password = args [3];
    String name = args [4];

    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);

    AccountManager manager =
      (AccountManager)ic.lookup (serviceURL + objectName);

    // Request the account manager to open a named account.
    Bank.Account account = manager.open(name);

    // Get the balance of the account.
    float balance = account.balance();

    // Print out the balance.
    System.out.println
      ("The balance in " + name + "'s account is $" + balance);
  }
}
```

bankServer/AccountImpl.java

```
// AccountImpl.java
package bankServer;

public class AccountImpl extends Bank._AccountImplBase {
  public AccountImpl(float balance) {
    _balance = balance;
  }
  public float balance() {
    return _balance;
  }
  private float _balance;
}
```

bankServer/AccountManagerImpl.java

```
package bankServer;

import java.util.*;

public class AccountManagerImpl
  extends Bank._AccountManagerImplBase {

  public synchronized Bank.Account open(String name) {

    // Lookup the account in the account dictionary.
    Bank.Account account = (Bank.Account) _accounts.get(name);

    // If there was no account in the dictionary, create one.
    if(account == null) {

      // Make up the account's balance, between 0 and 1000 dollars.
      float balance = Math.abs(_random.nextInt()) % 100000 / 100f;

      // Create the account implementation, given the balance.
      account = new AccountImpl(balance);

      _orb().connect(account);

      // Print out the new account.
      // This just goes to the system trace file for Oracle 8i.
      System.out.println("Created " + name + "'s account: " + account);

      // Save the account in the account dictionary.
      _accounts.put(name, account);
    }
    // Return the account.
    return account;
  }

  private Dictionary _accounts = new Hashtable();
  private Random _random = new Random();

}
```

bankServer/AccountManagerImplTie.java

```
package bankServer;

import java.util.*;
import oracle.aurora.AuroraServices.ActivatableObject;


public class AccountManagerImplTie
  implements Bank.AccountManagerOperations,
  ActivatableObject {


  public synchronized Bank.Account open(String name) {

    // Lookup the account in the account dictionary.
    Bank.Account account = (Bank.Account) _accounts.get(name);

    // If there was no account in the dictionary, create one.
    if(account == null) {

      // Make up the account's balance, between 0 and 1000 dollars.
      float balance = Math.abs(_random.nextInt()) % 100000 / 100f;

      // Create the account implementation, given the balance.
      account = new AccountImpl(balance);

      org.omg.CORBA.ORB.init().BOA_init().obj_is_ready(account);

      // Print out the new account.
      // This just goes to the system trace file for Oracle 8i.
      System.out.println("Created " + name + "'s account: " + account);

      // Save the account in the account dictionary.
      _accounts.put(name, account);
    }
    // Return the account.
    return account;
  }

  public org.omg.CORBA.Object _initializeAuroraObject () {
    return new Bank._tie_AccountManager(this);
  }

  private Dictionary _accounts = new Hashtable();
  private Random _random = new Random();
}
```

pureCorba

Bank.idl

```
// Bank.idl

module Bank {
  interface Account { float balance(); };
  interface AccountManager { Account open(in string name); };
};
```

Client.java

```
import java.lang.Exception;

import org.omg.CORBA.Object;
import org.omg.CORBA.SystemException;
import org.omg.CosNaming.NameComponent;

import oracle.aurora.client.Login;
import oracle.aurora.AuroraServices.LoginServer;
import oracle.aurora.AuroraServices.LoginServerHelper;
import oracle.aurora.AuroraServices.PublishedObject;
import oracle.aurora.AuroraServices.PublishingContext;
import oracle.aurora.AuroraServices.PublishedObjectHelper;

import Bank.Account;
import Bank.AccountManager;
import Bank.AccountManagerHelper;

public class Client {
  public static void main(String args[]) throws Exception {
    // Parse the args
    if (args.length != 5) {
      System.out.println ("usage: Client host port sid username password");
      System.exit(1);
    }
    String host = args[0];
    int    port = Integer.parseInt (args[1]);
    String sid  = args[2];
    String username = args[3];
    String password = args[4];

    // Declarations for an account and manager
    Account account = null;
    AccountManager manager = null;

    // access the Aurora Names Service
    try {
      // Get the Name service Object reference (Only ORB specific thing)
      PublishingContext rootCtx = null;
      rootCtx = VisiAurora.getNameService (host, port, sid);

      // Get the pre-published login object reference
      PublishedObject loginPubObj = null;
      LoginServer serv = null;
      NameComponent[] nameComponent = new NameComponent[2];
      nameComponent[0] = new NameComponent ("etc", "");
      nameComponent[1] = new NameComponent ("login", "");

      // Lookup this object in the Name service
      Object loginCorbaObj = rootCtx.resolve (nameComponent);

      // Make sure it is a published object
      loginPubObj = PublishedObjectHelper.narrow (loginCorbaObj);

      // create and activate this object (non-standard call)
      loginCorbaObj = loginPubObj.activate_no_helper ();
      serv = LoginServerHelper.narrow (loginCorbaObj);

      // Create a client login proxy object and authenticate to the DB
      Login login = new Login (serv);
      login.authenticate (username, password, null);

      // Now create and get the bank object reference
      PublishedObject bankPubObj = null;
      nameComponent[0] = new NameComponent ("test", "");
      nameComponent[1] = new NameComponent ("bank", "");

      // Lookup this object in the name service
      Object bankCorbaObj = rootCtx.resolve (nameComponent);

      // Make sure it is a published object
      bankPubObj = PublishedObjectHelper.narrow (bankCorbaObj);

      // create and activate this object (non-standard call)
```

```
      bankCorbaObj = bankPubObj.activate_no_helper ();
      manager = AccountManagerHelper.narrow (bankCorbaObj);

      account = manager.open ("Jack.B.Quick");

      float balance = account.balance ();
      System.out.println ("The balance in Jack.B.Quick's account is $"
                          + balance);
    } catch (SystemException e) {
      System.out.println ("Caught System Exception: " + e);
      e.printStackTrace ();
    } catch (Exception e) {
      System.out.println ("Caught Unknown Exception: " + e);
      e.printStackTrace ();
    }
  }
}
```

VisiAurora.java

```
import java.lang.Exception;
import java.net.UnknownHostException;
import java.net.InetAddress;
import java.util.Properties;

// CORBA specific imports
import org.omg.CORBA.Object;
import org.omg.CORBA.InitialReferences;
import org.omg.CORBA.InitialReferencesHelper;
import org.omg.CORBA.SystemException;

// Visigenic specific imports
import com.visigenic.vbroker.orb.ORB;
import com.visigenic.vbroker.orb.GiopOutputStream;
import com.visigenic.vbroker.GIOP.Version;
import com.visigenic.vbroker.IOP.IOR;
import com.visigenic.vbroker.IOP.TaggedComponent;
import com.visigenic.vbroker.IOP.TaggedProfile;
import com.visigenic.vbroker.IOP.TAG_INTERNET_IOP;
import com.visigenic.vbroker.IIOP_1_1.ProfileBody;
import com.visigenic.vbroker.IIOP_1_1.ProfileBodyHelper;

// Oracle specific imports
import oracle.aurora.AuroraServices.PublishingContext;
import oracle.aurora.AuroraServices.PublishingContextHelper;
import oracle.aurora.sess_iiop.orb_dep.TAG_SESSION_IOP;
import oracle.aurora.sess_iiop.orb_dep.ComponentBody;
import oracle.aurora.sess_iiop.orb_dep.ComponentBodyHelper;

public class VisiAurora {
  public static PublishingContext getNameService (String host, int port,
  String sid)
  {
    PublishingContext nameServiceCtx = null;
    try {
      // Get the Boot service object reference
      Object initRefObj = getBootIOR (host, port, sid);
      InitialReferences initRef = InitialReferencesHelper.narrow (initRefObj);

      // get the oracle CosName service reference
      Object nsObj = initRef.get ("NameService");
      nameServiceCtx = PublishingContextHelper.narrow (nsObj);
    } catch (SystemException e) {
      System.out.println ("Caught System Exception: " + e);
      e.printStackTrace ();
    } catch (Exception e) {
      System.out.println ("Caught Unknown Exception: " + e);
      e.printStackTrace ();
    }
    return nameServiceCtx;
  }

  public static Object getBootIOR (String host, int port, String sid)
       throws UnknownHostException
  {
    // NOTE: 1. if you wish to use sess_iiop then comment-out pt.#2,
    //          and #4 below and initialize the ORB using the following:
    Properties props = new Properties ();
    props.put ("ORBServices", "oracle.aurora.sess_iiop.orb_dep");
    //ORB visiORB = (ORB) org.omg.CORBA.ORB.init ((String[]) null, props);
    ORB visiORB =
      (ORB)oracle.aurora.jndi.orb_dep.Orb.init (props);

    // NOTE: 2. if you wish to use iiop then comment-out pt.#1 and #3
    //          and initialize the ORB using the following line:
    // ORB visiORB = (ORB) oracle.aurora.jndi.orb_dep.Orb.init ();

    // common to both (sess_iiop and iiop)
    String ipAddr = InetAddress.getByName (host).getHostAddress ();
    Version version = new Version ((byte)1, (byte)1);
    ComponentBody sessionBody =
      new ComponentBody ("ORCL", 0, visiORB.getLocalHost (),
new byte[] {});
    GiopOutputStream output = visiORB.newGiopOutputStream ();

    output.byteOrder (visiORB.JAVA_ENDIAN);
    output.write_boolean (visiORB.JAVA_ENDIAN);
    ComponentBodyHelper.write (output, sessionBody);

    TaggedComponent component =
      new TaggedComponent (TAG_SESSION_IOP.value, output.toByteArray ());
    TaggedComponent[] taggedComponents = { component };

    byte[] objectKey = getObjectKey (sid);

    ProfileBody profileBody = new
      ProfileBody (version, ipAddr, (short)port, objectKey,
  taggedComponents);

    output.offset (0);
    output.byteOrder (visiORB.JAVA_ENDIAN);
    output.write_boolean (visiORB.JAVA_ENDIAN);

    ProfileBodyHelper.write (output, profileBody);
    byte[] profileData = output.toByteArray ();
    TaggedProfile  profile =
      new TaggedProfile (TAG_INTERNET_IOP.value, profileData);
    TaggedProfile[] taggedProfiles = { profile };

    return visiORB.iorToObject (new IOR ("IDL:CORBA/InitialReferences:1.0",
taggedProfiles));
 }

 public static byte [] getObjectKey (String sid)
 {
    String preSID = new String ("ORCL(CONNECT_DATA=(SID=");

    // NOTE: 3. if you wish to use sess_iiop then comment out pt.#1
    //          and #4, and use the following postSID:
    String postSID = new String (")(SESSION_ID=0))");
    // NOTE: 4. if you wish to use iiop then comment out pt.#1 and #3
    //          above and use the following postSID:
    // String postSID = new String ("))");
    // common to both (sess_iiop and iiop)
    String preINIT = new String (preSID + sid + postSID);

    byte[] b1 = new byte [preINIT.length () + 1];

    System.arraycopy (preINIT.getBytes (), 0, b1, 0, preINIT.length ());
    b1 [preINIT.length ()] = 0;

    String initString = new String ("INIT");
    byte[] objectKey = new byte [b1.length + initString.length ()];
    System.arraycopy (b1, 0, objectKey, 0, b1.length);
    System.arraycopy (initString.getBytes (), 0, objectKey, b1.length,
      initString.length ());

    return objectKey;
  }
}
```

bankServer/AccountImpl.java

```
package bankServer;

import Bank.*;

public class AccountImpl extends _AccountImplBase {
  private float _balance;

  public AccountImpl () { _balance = (float) 100000.00; }
  public AccountImpl (float balance) { _balance = balance; }
  public float balance () { return _balance; }
}
```

bankServer/AccountManagerImpl.java

```
package bankServer;

// import the idl-generated classes
import Bank.*;

import java.util.Dictionary;
import java.util.Random;
import java.util.Hashtable;

// Corba specific imports
import org.omg.CORBA.Object;
```

```
// Aurora-orb specific imports
import oracle.aurora.AuroraServices.ActivatableObject;

public class AccountManagerImpl
       extends _AccountManagerImplBase
       implements ActivatableObject
{
  private Dictionary _accounts = new Hashtable ();
  private Random _random = new Random ();

  // Constructors
  public AccountManagerImpl () { super (); }
  public AccountManagerImpl (String name) { super (name); }

  public Object  _initializeAuroraObject () {
    return new AccountManagerImpl ("BankManager");
  }

  public synchronized Account open (String name) {
    // Lookup the account in the account dictionary.
    Account account = (Account) _accounts.get (name);

    // If there was no account in the dictionary, create one.
    if (account == null) {
      // Make up the account's balance, between 0 and 1000 dollars.
      float balance = Math.abs (_random.nextInt ()) % 100000 / 100f;

      // Create the account implementation, given the balance.
      account = new AccountImpl (balance);

      // Make the object available to the ORB.
      _orb ().connect (account);

      // Print out the new account.
      System.out.println ("Created " + name + "'s account: " + account);

      // Save the account in the account dictionary.
      _accounts.put (name, account);
    }

    // Return the account.
    return account;
  }
}
```

Session Examples

The following are the CORBA session examples:

- explicit
- clientserverserver
- timeout
- sharedsession
- twosessions
- twosessionsbyname

Here is the README file for the session examples:

```
The examples in the session/ directories demonstrate various CORBA
programming techniques that you can use to create and manage sessions
in Oracle8i.

The examples are short, and each example shows just one, or at the
most a few aspects of Oracle8i CORBA session handling. The examples
are mostly slight variants on the basic helloworld example. None of
these examples do any database access.

You should first study the 'explicit' example. This example shows you
how to use JNDI to connect and activate a CORBA object by doing each
step of the process explicitly. In the other, basic/ examples, things
such as authentication are done automatically for you, for example
when you specify NON_SSL_LOGIN as the authentication method in the
Initial Context.


Running the Examples
====================

To run the examples, you must have access to an Oracle8i database
server that hosts the Oracle8i server-side Java VM.

The SCOTT schema must have write access to the CORBA name space
starting at the 'test' directory, which is true of the install
database.  The tables that support the publishing directories are
established when your Oracle8i system with the Java server option is
built. You can use the Session Shell to verify the presence of the
test directory. See the Oracle8i EJB and CORBA Developer's Guide for
information about the Session Shell.
```

```
You must also have the INIT.ORA, tnsnames.ora, and listener.ora files
configured properly to accept both standard TTC as well as IIOP
incoming connections. This is done for you in the install test
database. See the Net8 Administrator's Guide for information about
setting up these files.

For simplicity. most of these examples connect directly to the
dispatcher port. Your production code should use the listener for
better scalability.

Each example publishes one or more objects in the database. To
lookup and activate the published object, the client uses the
Oracle8i JNDI interface to the CosNaming implementation.

The Makefiles or batch files provided with the examples expect that you
have the java and javac commands from the Sun JDK 1.1.x (with x >= 3)
in your PATH.  They also expect that your CLASSPATH contains the Java
runtime classes (classes.zip) corresponding to your java interpreter.
The makefiles/batch files take care of adding the ORACLE specific jar
and zip files to your CLASSPATH.

For reference here is a list of jar and zip files that the
makefiles/batch files use:

ORACLE_HOME/lib/aurora_client.jar      # Oracle 8i ORB runtime
ORACLE_HOME/lib/aurora.jar             # Oracle 8i in-the-database runtime
#use classes12.zip if using Java 2
ORACLE_HOME/jdbc/lib/classes111.zip    # for JDBC examples
ORACLE_HOME/sqlj/lib/translator.zip    # for SQLJ examples
ORACLE_HOME/lib/vbjapp.jar             # Inprise VisiBroker library
ORACLE_HOME/lib/vbjorb.jar             # VisiBroker library
ORACLE_HOME/lib/vbj30ssl.jar           # required if you modify any
                                       # client code to use SSL


The example programs are:

explicit -     shows how to get the JNDI initial context, authenticate
               the client explicitly using a login server objct and a
               client proxy login object (and stub), create a session
               "by hand", and so on. Study this example carefully.

clientserverserver -   create a new session from within a server
               object.
```

```
timeout -      client sets the session timeout value from the server object.

sharedsession - client writes an object reference to a file, and a
               second client reads the ref, and uses it to invoke a
               method on the object in the session started by the first
               client.

twosessions -  client creates two separate sessions explicitly, and
               invokes a method on an object in each session.

twosessionsbyname - client creates two separate named sessions, and
               activates a separate object in each session. This example
               uses the SessionCtx login method to authenticate the client,
               rather than the fully explicit login object activation used
               in the twosessions example.



The code in the examples is not always commented, but each of the
examples has its own readme file. The readme explains what the code
does, and points out any special features used in the example.

Each of these examples has been tested on Solaris 2.6 and Windows
NT 4.0. If you have problems compiling or running the examples on
these or on another supported platform, please inform your Oracle
support representative.
```

explicit

readme.txt

```
Overview
========

Demonstrates how a client can activate a CORBA server object
explicitly, and the use of the login object for client authentication.


Compare this example to the ../examples/corba/basic/helloworld case.
In the basic example, only three client-side call are made to lookup and
activate a server object, and then invoke one of its methods:

  Context ic = new InitialContext(env);
  Hello hello = (Hello) ic.lookup(serviceURL + objectName);
  System.out.println(hello.helloWorld ());

This example makes explicit much that is handled "under the covers" in
the simple helloworld example.

You should study this example before going on to the other examples in
this CORBA sessions directory.


Source files
============

hello.idl
---------

The CORBA IDL for the example. The IDL for the Hello interface simply
defines one method:

interface Hello
  wstring helloWorld()

which must be implemented by the helloServer.HelloImpl.java code.


Client.java
-----------

You invoke the client program from a command line prompt, and pass it
four arguments, the

   - service URL (service ID, hostname, and port)
   - name of the published object to lookup and instantiate
   - username
   - password that authenticate 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
#use classes12.zip if using Java 2
```

```
$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 a JNDI Context (InitialContext())
   - looks up the service URL to get a ServiceCtx (service context)  object
   - creates a session context. This activates a new session in the server.
   - activates a login server object
   - creates a new client-side login object
   - authenticates the client (login.authenticate())
   - activates a Hello object
   - invokes the helloWorld() method on the Hello object, and print the results

The printed output is:

Hello World!


helloServer/HelloImpl.java
--------------------------

This file implements the method specified in the hello.idl
file: helloWorld(). It simple returns the greeting to the client.


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.
```

hello.idl

```
module hello {
  interface Hello {
    wstring helloWorld ();
  };
};
```

Client.java

```
import hello.Hello;

import oracle.aurora.jndi.sess_iiop.ServiceCtx;
import oracle.aurora.jndi.sess_iiop.SessionCtx;
import oracle.aurora.AuroraServices.LoginServer;
import oracle.aurora.client.Login;

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];

    // 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 a session in the instance.  The session name must start by a :
    SessionCtx session = (SessionCtx)service.createSubcontext (":session1");

    // Activate the LoginServer object at the well known name etc/login
    LoginServer login_server = (LoginServer)session.activate ("etc/login");

    // Create the login client and authenticate with the login protocol
    Login login = new Login(login_server);
    login.authenticate (user, password, null);

    // Activate the Hello object and call its helloWorld method
    Hello hello = (Hello)session.activate (objectName);
    System.out.println (hello.helloWorld ());
  }
}
```

helloServer/HelloImpl.java

```
package helloServer;

import hello.*;
import oracle.aurora.AuroraServices.ActivatableObject;
```

```
public class HelloImpl extends _HelloImplBase implements ActivatableObject
{
  public String helloWorld () {
    return "Hello World!";
  }

  public org.omg.CORBA.Object _initializeAuroraObject () {
    return this;
  }
}
```

clientserverserver

readme.txt

```
clientserverserver demonstrates:

(1) A CORBA server object that instantiates a second session in the
same server, and calls methods on it.

The basic structure of this example is a client program that
instantiates a server object, then invokes a method on it that sets a
String to "Hello World!". The client then invokes the getOtherHello()
method on the server object. This method takes the authentication and
service identifier information from the client, and creates a second
server object *in a different session*.


Source files
============

hello.idl
---------

The CORBA IDL for the example.  Defines an interface, Hello, with 4
methods:

interface Hello
  wstring helloWorld();
  void setMessage (
           in wstring message);
  void getOtherHello (
           in wstring user,
          in wstring password,
          in wstring objectURL) raises (AccessError);
  wstring otherHelloWorld()

and one exception: AccessError.


Client.java
-----------

The client looks up and instantiates a Hello CORBA server object.
The client then invokes setMessage() on this object to set its message
variable. Next the client invokes getOtherHello(), to have the first
CORBA server object create a second Hello object. The first server
Hello object will set a different message in the message instance
variable. The client finally calls otherHelloWorld() on the first
object, which indirectly returns the message set in the second object.

The result of all this is that client prints:

Hello World!
Hello from the Other Hello Object

on its console.


helloServer/HelloImpl.java
--------------------------

This server class implements the four methods specified in hello.idl:

setMessage() simply sets the class variable message to the
input parameter.

helloWorld() returns to the client whatever String setMessage
set.

getOtherHello() takes three parameters: a username, password, and a
service URL (e.g. "sess_iiop://<hostname>:<dispatcher_port>"). It then
instantiates a second Hello server object, and sets its message
variable to "Hello from the Other Hello Object".

otherHelloWorld() invokes the helloWorld() method on the second
object, and returns its message string to the client.
```

```
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
$


On UNIX, enter the command 'make all' or just simply 'make' in the
shell to compile, load, and publish the objects, and run the
client program.  Other targets are 'make compile', 'make load',
'make publish', and 'make run'.

On Windows NT, use the batch file to compile, load, publish and
run.
```

hello.idl

```
module hello {
  exception AccessError {
    wstring message;
  };

  interface Hello {
    wstring helloWorld ();
    void setMessage (in wstring message);
    void getOtherHello (in wstring user, in wstring password,
                this wstring objectURL)
                 raises (AccessError);
    wstring otherHelloWorld ();
  };
};
```

Client.java

```
import hello.Hello;

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
    Hello hello = (Hello)ic.lookup (serviceURL + objectName);
    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 ());
  }
}
```

helloServer/HelloImpl.java

```
package helloServer;

import hello.*;
import oracle.aurora.AuroraServices.ActivatableObject;
import oracle.aurora.jndi.sess_iiop.ServiceCtx;
import oracle.aurora.AuroraServices.ActivatableObject;
import javax.naming.*;
import java.util.*;
```

```
public class HelloImpl extends _HelloImplBase implements ActivatableObject
{
  String message;
  Hello otherHello;

  public String helloWorld () {
    return message;
  }

  public void setMessage (String message) {
    this.message = message;
  }

  public void getOtherHello (String user, String password, String URL)
       throws AccessError
  {
    try {
      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);

      otherHello = (Hello)ic.lookup (URL);
      otherHello.setMessage ("Hello from the Other Hello Object");
    } catch (Exception e) {
      e.printStackTrace ();
      throw new AccessError (e.toString ());
    }
  }

  public String otherHelloWorld () {
    if (otherHello != null)
      return otherHello.helloWorld ();
    else
      return "otherHello not accessed yet";
  }

  public org.omg.CORBA.Object _initializeAuroraObject () {
    return this;
  }
}
```

timeout

readme.txt

```
Overview
========

Timeout shows you how to set the session timeout from a server object.
For testing the timeout, a second client is provided. The second client
is authenticated using a login IOR that the first client writes to a file..

The basic structure of this example is a client program that
instantiates two server objects in separate sessions.

Compare this example with the ..corba/session/clientserverserver
example, in which the client instantiates a server object, and that
server object then instantiates a second server object in a different
session.


Source files
============

hello.idl
---------

The CORBA IDL

interface Hello
  wstring helloWorld ()
  void setMessage (in wstring message)
  void setTimeOut (in long seconds)


Client.java
-----------

You invoke the client program from a command line prompt, and pass it
seven arguments:

   - the service URL (service ID, hostname, and port)
   - the name of the published object to lookup and instantiate
   - a username
   - a password that authenticates the client to the Oracle8i database
   - the name of a file that the client writes the hello IOR into
   - the name of a file that the client writes the login IOR into
   - the session timeout value in seconds

For example: % java -classpath LIBs Client sess_iiop://localhost:2222
scott tiger hello.ior login.ior 30

where LIBs is the classpath that must include

$ORACLE_HOME/lib/aurora_client.jar
#use classes12.zip if using Java 2
$ORACLE_HOME/jdbc/lib/classes111.zip
$ORACLE_HOME/lib/vbjorb.jar
$ORACLE_HOME/lib/vbjapp.jar
$JAVA_HOME/lib/classes.zip


This first client gets a reference to a Hello object, and sets its
message instance variable to "As created by Client1.java". It then sets
the session timeout to the number of seconds passed as the sixth parameter.

Next, the client writes the stringified hello IOR and login IOR to the
file named in the fifth and sixth parameter, then exits. The session remains
alive, on account of the timeout parameter.

This client program prints

Client1: As created by Client1
Set session timeout to 30 seconds
Client1: wrote the login IOR
Client1: exiting...


Client2.java
------------

The Client2 program reads the IOR for the hello object, and the IOR
for the login object. These were written to files by Client1.

The login IOR is required because the client uses NON_SSL_LOGIN as the
authentication mechanism. This requires that the client2 program get a
reference to a login server object, and then instantiate a client-side
proxy object to communicate with the server-side login object, in
order to authenticate.

This client program is then run twice, generating the following output:
```

```
Client2: Got the hello IOR
Client2: got the login IOR.
Client2: authenticated.
Client2: As created by Client1
Client2: exiting...
sleep 40
Client2: Got the hello IOR
Client2: got the login IOR.
Session Timed out as Expected


helloServer/HelloImpl.java
--------------------------

Implements the methods specified in hello.idl:

String helloWorld()
void setMessage(String message)
void setTimeOut(int seconds)


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.

Client1.java

```
import hello.Hello;

import java.io.FileOutputStream;
import java.io.ObjectOutputStream;
import java.io.OutputStream;
import java.util.Hashtable;
import javax.naming.Context;
import javax.naming.InitialContext;

import oracle.aurora.jndi.sess_iiop.ServiceCtx;
import oracle.aurora.client.*;
import oracle.aurora.AuroraServices.LoginServer;

import org.omg.CORBA.ORB;

public class Client1
{
  public static void main (String[] args) throws Exception {
    if (args.length != 7) {
      System.out.println ("usage: Client serviceURL objectName user" +
  " password iorfile loginfile timeout");
      System.exit(1);
    }
    String serviceURL = args [0];
    String objectName = args [1];
    String user = args [2];
    String password = args [3];
    String iorfile = args [4];
    String loginfile = args [5];
    int timeout = Integer.parseInt (args [6]);

    // 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);

    // get the object from the 8i server
    Hello hello = (Hello)ic.lookup (serviceURL + objectName);
    hello.setMessage ("As created by Client1");
    System.out.println ("Client1: " + hello.helloWorld());

    // Make the session survive timeout seconds after its last connection
    // is dropped.
    hello.setTimeOut (timeout);
    System.out.println ("Set session timeout to " + timeout + " seconds");

    // create and write loginIOR for Client2 to access our session
    LoginServer lserver =
      (LoginServer) (ic.lookup (serviceURL + "/etc/login"));
    ORB orb = ORB.init ();
    String loginior = orb.object_to_string (lserver);
    OutputStream ls = new FileOutputStream (loginfile);
    ls.write (loginior.getBytes ());
    ls.close ();
    System.out.println ("Client1: wrote the login IOR");

    // Write the IOR to a file for Client2.java to access our session
    OutputStream os = new FileOutputStream (iorfile);
    String ior = orb.object_to_string (hello);
    os.write (ior.getBytes ());
    os.close ();

    // Client1 exits...
    // Note: session still continues on the server (till time-out)
    System.out.println ("Client1: exiting...");
  }
```

```
}
```

Client2.java

```
import hello.Hello;
import hello.HelloHelper;

import java.io.*;
import javax.naming.*;
import java.util.Hashtable;

import oracle.aurora.jndi.sess_iiop.ServiceCtx;
import oracle.aurora.client.*;
import oracle.aurora.AuroraServices.*;

import org.omg.CORBA.ORB;
import org.omg.CORBA.Object;
import org.omg.CORBA.BindOptions;
import org.omg.CORBA.COMM_FAILURE;

public class Client2
{
  public static void main (String[] args) throws Exception {
    if (args.length != 4) {
      System.out.println ("usage: Client2 user password iorfile loginfile");
      System.exit (1);
    }
    String user = args [0];
    String password = args [1];
    String iorfile = args [2];
    String loginfile = args [3];

    // Initialize the ORB for accessing objects in 8i
    // You have to initialize the ORB that way.
    // You will be authenticated using the login IOR read
    // from the file.
    ORB orb = ServiceCtx.init (null, null, null, false, null);

    // Read the object-ior from iorfile
    InputStream is = new FileInputStream (iorfile);
    byte[] iorbytes = new byte [is.available ()];
    is.read (iorbytes);
    is.close ();
    String ior = new String (iorbytes);
    System.out.println ("Client2: Got the hello IOR");
    // Read the login IOR from the loginfile.
    FileInputStream ls = new FileInputStream (loginfile);
    byte[] loginbytes = new byte [ls.available ()];
    ls.read (loginbytes);
    ls.close ();
    String loginior = new String (loginbytes);
    System.out.println ("Client2: got the login IOR.");

    // Try to authenticate the Client2 to Client1's session
    try {
      Object lobj = orb.string_to_object (loginior);
      LoginServer lserver = LoginServerHelper.narrow (lobj);
      lserver._bind_options (new BindOptions (false, false));

      Login login = new Login (lserver);
      boolean result = login.authenticate (user, password, null);
    } catch (COMM_FAILURE e1) {
      System.out.println ("Session Timed out as Expected");
      System.exit (0);
    } catch (Exception e) {
      System.out.println ("Login failed: " + e.getMessage ());
      System.exit (1);
    }

    // login succeeded.... print it out
    System.out.println ("Client2: authenticated.");

    // Access the object from the ior and print its message
    Hello hello = HelloHelper.narrow (orb.string_to_object (ior));
    System.out.println ("Client2: " + hello.helloWorld ());

    // Disconnect from the object by exiting
    System.out.println ("Client2: exiting...");
  }
}
```

helloServer/HelloImpl.java

```
package helloServer;

import hello.*;
import oracle.aurora.AuroraServices.ActivatableObject;
import oracle.aurora.net.Presentation;
```

```
public class HelloImpl extends _HelloImplBase implements ActivatableObject
{
  String message;

  public String helloWorld () {
    return message;
  }

  public void setMessage (String message) {
    this.message = message;
  }

  public void setTimeOut (int seconds) {
    Presentation.sessionTimeout (seconds);
  }

  public org.omg.CORBA.Object _initializeAuroraObject () {
    return this;
  }
}
```

sharedsession

readme.txt

```
Overview
========

Sharedsession client1 writes an object reference out to a file. The second
client reads the IOR, and uses it to access an object in the same session
started by the first client.


Sources
=======

hello.idl
---------


The CORBA IDL for the example. Specifies one interface with two
methods:

interface Hello
```

```
  wstring helloWorld()
  void setMessage(in wstring message)


Client1.java
------------

There are two client programs in this example. You invoke the first
client program (Client1.class) from a command line prompt, and pass it
six arguments:

   - the service URL (service ID, hostname, and port)
   - the name of a published object to lookup and instantiate
   - a username (e.g. SCOTT)
   - a password (e.g. TIGER)
   - a filename in which to save the hello IOR from this client
   - a filename in which to save the login IOR

This client should be run in the background. Use & in a UNIX shell, or
START in NT.

For example:
% java -classpath LIBs Client sess_iiop://localhost:2222 scott tiger
hello.ior login.ior &

where LIBs is the classpath that must include

$ORACLE_HOME/lib/aurora_client.jar
#use classes12.zip if using Java 2
$ORACLE_HOME/jdbc/lib/classes111.zip
$ORACLE_HOME/lib/vbjorb.jar
$ORACLE_HOME/lib/vbjapp.jar
$JAVA_HOME/lib/classes.zip

The client looks up and activates a Hello object, then sets its
message instance variable to "As created by Client1". The client
then writes the stringified IOR to the file specified on the command
line. (Note that a client-side ORB has to be specifically activated (
org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init();) to get access to
the object_to_string() ORB method.)

Then the client loops invoking helloWorld() on its Hello object. At
some point, the second client will have changed the message in the
object, and that will be visible in the first client's output.
```

```
The first client then sleeps for 20 seconds, before exiting.


Client2.java
------------

You invoke the second client program (Client2.class) from a command
line prompt, and pass it four arguments:

   - a username (e.g. SCOTT)
   - a password (e.g. TIGER)
   - a filename from which to read the hello IOR from client1
   - a filename from which to read the login IOR from client1


This client sleeps for 5 seconds, then tries to read the hello IOR
from the file written by client1. When read, client2 then reads the
login IOR, and authenticates itself to the session.

The client then sets the message instance variable to "Client2 was
here and modified the message". The first client, still running, will
print this new message out.


helloServer/HelloImpl.java
--------------------------

This source file implements the two methods specified in the hello.idl
file: setMessage() to set the instance variable message, and
helloWorld() to return the value set in message.


Compiling and Running the Example
=================================

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.
```

hello.idl

```
module hello {
  interface Hello {
    wstring helloWorld ();
    void setMessage (in wstring message);
  };
};
```

Client1.java

```
import hello.Hello;

import java.io.*;
import javax.naming.*;
import java.util.Hashtable;

import oracle.aurora.jndi.sess_iiop.ServiceCtx;
import oracle.aurora.client.*;
import oracle.aurora.AuroraServices.*;

public class Client1
{
  public static void main (String[] args) throws Exception {
    if (args.length != 6) {
      System.out.println
        ("usage: Client serviceURL objectName user password " +
 "loginfile iorfile");
      System.exit(1);
    }
    String serviceURL = args [0];
    String objectName = args [1];
    String user = args [2];
    String password = args [3];
    String loginIORFile = args [4];
    String helloIORFile = args [5];

    Hashtable env = new Hashtable();
    env.put (Context.URL_PKG_PREFIXES, "oracle.aurora.jndi");
    Context ic = new InitialContext (env);

    LoginServer lserver = (LoginServer)ic.lookup (serviceURL + "/etc/login");
    new Login (lserver).authenticate (user, password, null);

    Hello hello = (Hello)ic.lookup (serviceURL + objectName);
    hello.setMessage ("As created by Client1");

    writeIOR (lserver, loginIORFile);
    writeIOR (hello, helloIORFile);

    int i;
    for (i = 0; i < 10; i++) {
      System.out.println ("Client1: " + i + ": " + hello.helloWorld ());
      Thread.sleep (4000);
    }

    System.out.println("Client1: exiting...");
  }

  static public void writeIOR (org.omg.CORBA.Object object, String iorFile)
       throws Exception
  {
    org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init ();
    String ior = orb.object_to_string (object);
    OutputStream os = new FileOutputStream (iorFile);
    os.write (ior.getBytes ());
    os.close ();
  }
}
```

Client2.java

```
import hello.Hello;
import hello.HelloHelper;

import oracle.aurora.jndi.sess_iiop.ServiceCtx;

import javax.naming.*;
import java.util.Hashtable;

import java.io.*;

import oracle.aurora.client.*;
import oracle.aurora.AuroraServices.*;


public class Client2 {
  public static void main (String[] args) throws Exception {
    if (args.length != 4) {
      System.out.println("usage: Client2 user password loginfile hellofile");
      System.exit(1);
    }
    String user = args [0];
    String password = args [1];
    String loginIORfile = args [2];
    String helloIORfile = args [3];

    // Initialize the ORB for accessing objects in 8i
    // You have to initialize the ORB that way.
    // You will be authenticated using the login object IOR retrieved
    // from the loginfile, so the parameters are null.
    org.omg.CORBA.ORB orb = ServiceCtx.init (null, null, null, false, null);

    // Read the IORs from the IOR files
    String loginIOR = getIOR (loginIORfile);
    String helloIOR = getIOR (helloIORfile);

    // Authenticate with the login Object
    LoginServer lserver =
      LoginServerHelper.narrow (orb.string_to_object (loginIOR));
    lserver._bind_options (new org.omg.CORBA.BindOptions (false, false));

    Login login = new Login (lserver);
    login.authenticate (user, password, null);
    System.out.println("Client2: authenticated.");

    // Access the Hello object from its ior and change its message
    Hello hello = HelloHelper.narrow (orb.string_to_object (helloIOR));
    hello.setMessage ("Client2 was here and modified the message");

    System.out.println ("Client2: " + hello.helloWorld());

    System.out.println("Client2: exiting...");
  }

  // Read an IOR from an IOR file.
  static String getIOR (String iorFile)
       throws Exception
  {
    // Loop until the ior file is available
    InputStream is = null;
    int i;
    for (i = 0; i < 10; i++) {
      try {
is = new FileInputStream(iorFile);
      } catch (FileNotFoundException e) {}
      Thread.sleep(1000);
    }

    if (is == null){
      System.out.println("Client2 timed out before finding " + iorFile);
      System.exit(1);
    }

    byte[] iorbytes = new byte [is.available ()];
    is.read (iorbytes);
    is.close ();
    String ior = new String (iorbytes);
    System.out.println("Client2: got the IOR from " + iorFile);
    return ior;
  }
}
```

helloServer/HelloImpl.java

```
package helloServer;

import hello.*;
import oracle.aurora.AuroraServices.ActivatableObject;

public class HelloImpl extends _HelloImplBase implements ActivatableObject
{
  String message;

  public String helloWorld () {
    return message;
  }

  public void setMessage (String message) {
    this.message = message;
  }

  public org.omg.CORBA.Object _initializeAuroraObject () {
    return this;
  }
}
```

twosessions

readme.txt

```
Overview
========

Twosessions demostrates a client that instantiates two separate sessions in
the server, and calls methods on objects in each session. It also demos use of
the login object for client authentication.


Compare this example to the ../examples/corba/session/clientserverserver
example, in which the client instantiates a server object, and that server
object then instantiates a second server object in a different session.


Source files
============

hello.idl
```

```
---------

The CORBA IDL for the example. The IDL for the Hello object simply
defines two methods:

interface Hello
  wstring helloWorld ();
  void setMessage (in wstring message);

which must be implemented by the helloServer.HelloImpl.java code.


Client.java
-----------

You invoke the client program from a command line prompt, and pass it
four arguments: the service URL (service ID, hostname, and port), the
name of the published object to lookup and instantiate, and a username
and password that authenticate the client to the Oracle8i database
server.

For example:
% java -classpath LIBs Client sess_iiop://localhost:2222 scott tiger

where LIBs is the classpath that must include

$ORACLE_HOME/lib/aurora_client.jar
#use classes12.zip if using Java 2
$ORACLE_HOME/jdbc/lib/classes111.zip
$ORACLE_HOME/lib/vbjorb.jar
$ORACLE_HOME/lib/vbjapp.jar
$JAVA_HOME/lib/classes.zip


The client first obtains a service context in the normal way, by
getting a JNDI Context object, and looking up the service context on
it, using the service URL (e.g., sess_iiop://localhost:2222).
The service context is then used to create new named sessions,
:session1 and :session2. On each session, a login server object is
instantiated, then a login client is obtained, and the authenticate()
method on the login client is used to authenticate the client.

Note that this form of authentication is what happens automatically
when a server object is instantiated, and the JNDI context is obtained
by passing in the username, password, optional database role, and the
```

```
value NON_SSL_LOGIN in the environmentg hashtable.

In this example, because the sessions are instantiated overtly, it is
necessary to also do the authentication overtly.

After session instantiation and authentication, a Hello object is
instantiated in each session, the helloWorld() method is invoked
on each, and the returned String is printed on the console.

The printed output is:

Hello from Session1
Hello from Session2


helloServer/HelloImpl.java
--------------------------

This source file implements the two methods specified in the hello.idl
file: setMessage() to set the instance variable message, and
helloWorld() to return the value set in message.

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.

hello.idl

```
module hello {
  interface Hello {
    wstring helloWorld ();
    void setMessage (in wstring message);
  };
};
```

Client.java

```
import hello.Hello;

import oracle.aurora.jndi.sess_iiop.ServiceCtx;
import oracle.aurora.jndi.sess_iiop.SessionCtx;
import oracle.aurora.AuroraServices.LoginServer;
import oracle.aurora.client.Login;

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];

    // 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);

    // Verify that the objects are indeed different
    hello1.setMessage ("Hello from Session1");
    hello2.setMessage ("Hello from Session2");

    System.out.println (hello1.helloWorld ());
    System.out.println (hello2.helloWorld ());
  }
}
```

helloServer/HelloImpl.java

```
package helloServer;

import hello.*;
import oracle.aurora.AuroraServices.ActivatableObject;
```

```
public class HelloImpl extends _HelloImplBase implements ActivatableObject
{
  String message;

  public String helloWorld () {
    return message;
  }

  public void setMessage (String message) {
    this.message = message;
  }

  public org.omg.CORBA.Object _initializeAuroraObject () {
    return this;
  }
}
```

twosessionsbyname

readme.txt

```
Overview
========

Twosessionbyname shows a client that creates two separate server sessions by
name, and then does a JNDI lookup() on the sessions, using the names.

Compare this example to ../examples/corba/session/twosessions/*. In the
twosessionsbyname example, the session name is used to do a short-hand lookup
and instantiation of server object by using the session name in the URL
parameter of the lookup() method. In the twosessions example, two sessions are
created by name, but the names are not used.


Sources
=======

Client.java
-----------

You invoke the client program from a command line prompt, and pass it
four arguments: the service URL (service ID, hostname, and port), the
name of the published object to lookup and instantiate, and a username
```

```
and password that authenticate the client to the Oracle8i database
server.

For example:
% java -classpath LIBs Client sess_iiop://localhost:2222 scott tiger

where LIBs is the classpath that must include

$ORACLE_HOME/lib/aurora_client.jar
#use classes12.zip if using Java 2
$ORACLE_HOME/jdbc/lib/classes111.zip
$ORACLE_HOME/lib/vbjorb.jar
$ORACLE_HOME/lib/vbjapp.jar
$JAVA_HOME/lib/classes.zip

The client instantiates two sessions by name, then two Hello objects,
one in each session, then verifies that the object are different by
setting the message instance variable in each object to a different
value, and calling helloWorld() on each object, and printing the
result.

The output of the client program is:

Hello from Session1
Hello from Session2


helloServer/HelloImpl.java
--------------------------

This source file implements the two methods specified in the hello.idl
file: setMessage() to set the instance variable message, and
helloWorld() to return the value set in message.

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.
```

hello.idl

```
module hello {
  interface Hello {
    wstring helloWorld ();
    void setMessage (in wstring message);
  };
};
```

Client.java

```
import hello.Hello;

import oracle.aurora.jndi.sess_iiop.ServiceCtx;
import oracle.aurora.jndi.sess_iiop.SessionCtx;
import oracle.aurora.AuroraServices.LoginServer;
import oracle.aurora.client.Login;
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];

    // 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 the 2 sessions
    SessionCtx session1 = (SessionCtx)service.createSubcontext (":session1");
    SessionCtx session2 = (SessionCtx)service.createSubcontext (":session2");

    // Login the sessions using the shortcut login method
    session1.login (user, password, null);
    session2.login (user, password, null);

    // Activate the objects by usign the fully specified URL that contains
    // the session name
    Hello hello1 = (Hello)ic.lookup (serviceURL + "/:session1" + objectName);
    Hello hello2 = (Hello)ic.lookup (serviceURL + "/:session2" + objectName);

    // Verify that the objects are indeed different
    hello1.setMessage ("Hello from Session1");
    hello2.setMessage ("Hello from Session2");

    System.out.println (hello1.helloWorld ());
    System.out.println (hello2.helloWorld ());
  }
}
```

helloServer/HelloImpl.java

```
package helloServer;

import hello.*;
import oracle.aurora.AuroraServices.ActivatableObject;

public class HelloImpl extends _HelloImplBase implements ActivatableObject
{
  String message;

  public String helloWorld () {
    return message;
  }

  public void setMessage (String message) {
    this.message = message;
  }

  public org.omg.CORBA.Object _initializeAuroraObject () {
    return this;
  }
}
```

Transaction Examples

The following are the CORBA transaction examples:

- clientside
- serversideJDBC
- serversideJTS
- serversideLogging
- multiSessions

clientside

readme.txt

```
Overview
========


The clientside example shows how to do transaction management for
CORBA server objects from the client application, using the XA JTS
methods.

This example also shows a server object that uses SQLJ in its methods.

Source files
============

employee.idl
------------

The CORBA IDL for the example.  Defines:

An EmployeeInfo struct
A SQLError exception
An Employee interface, with
    EmployeeInfo getEmployee ()
    void updateEmployee ()

The SQLError exception is used so that SQLException messages can
be passed back to the client.

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 server object 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/myEmployee scott tiger
where LIBs is the classpath that must include

$ORACLE_HOME/lib/aurora_client.jar
#use classes12.zip if using Java 2
$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())
   - initializes the Aurora transaction service
   - looks up the myEmployee CORBA published object on the server
       (this step also authenticates the client using NON_SSL_LOGIN and
        activates the server object)
   - starts a new transaction: TS.getTS().getCurrent().begin();
   - gets and prints information about the employee SCOTT
   - increases SCOTT's salary by 10%
   - updates the EMP table with the new salary by calling the updateEmployee()
        method on the employee object
   - gets and prints the new information
   - commits the update: TS.getTS().getCurrent().commit(false);

The printed output is:

SCOTT 3000.0
Increase by 10%
```

```
SCOTT 3300.0

Note that the starting value is taken from the EMP table when the
example starts to run, so you may see a different salary amount. The new
salary amount is written back to the database, and will be used as the
new starting amount if you run this example again.

employeeServer/EmployeeImpl.sqlj
--------------------------------
Implements the Employee interface.  This file implements the two
methods specified in the IDL: getEmployee() and updateEmployee(),
using SQLJ for ease of DML coding.

If the SQLJ code throws a SQLException, it is caught, and a
CORBA-defined SQLError is thrown. This in turn would be
propagated back to the client, where it is handled.

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.
```

employee.idl

```
module employee {
  struct EmployeeInfo {
    wstring name;
    long number;
    double salary;
  };

  exception SQLError {
    wstring message;
  };

  interface Employee {
    EmployeeInfo getEmployee (in wstring name) raises (SQLError);
    void updateEmployee (in EmployeeInfo name) raises (SQLError);
  };
};
```

Client.java

```
import employee.*;

import oracle.aurora.jndi.sess_iiop.ServiceCtx;

import oracle.aurora.jts.client.AuroraTransactionService;

import oracle.aurora.jts.util.*;

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);

    AuroraTransactionService.initialize (ic, serviceURL);

    Employee employee = (Employee)ic.lookup (serviceURL + objectName);
    EmployeeInfo info;

    TS.getTS ().getCurrent ().begin ();

    info = employee.getEmployee ("SCOTT");
    System.out.println (info.name + " "  + " " + info.salary);
    System.out.println ("Increase by 10%");
    info.salary += (info.salary * 10) / 100;
    employee.updateEmployee (info);
    info = employee.getEmployee ("SCOTT");
    System.out.println (info.name + " "  + " " + info.salary);

    TS.getTS ().getCurrent ().commit (true);
  }
}
```

employeeServer/EmployeeImpl.sqlj

```
package employeeServer;

import employee.*;
import java.sql.*;
```

```
public class EmployeeImpl
       extends _EmployeeImplBase {

  public EmployeeInfo getEmployee (String name) throws SQLError {
    try {
      int empno = 0;
      double salary = 0.0;
      #sql { select empno, sal into :empno, :salary from emp
            where ename = :name };
      return new EmployeeInfo (name, empno, (float)salary);
    } catch (SQLException e) {
      throw new SQLError (e.getMessage ());
    }
  }

  public void updateEmployee (EmployeeInfo employee) throws SQLError {
    try {
      #sql { update emp set ename = :(employee.name), sal = :(employee.salary)
                    where empno = :(employee.number) };
    } catch (SQLException e) {
      throw new SQLError (e.getMessage ());
    }
  }
}
```

serversideJDBC

readme.txt

```
Overview
========

The serversideJDBC example shows how to do transaction management for
CORBA server objects from objects themselves, using SQL transaction
control statements in the JDBC calls.


Source files
============

employee.idl
------------

The CORBA IDL for the example.  Defines:
```

```
An EmployeeInfo struct
A SQLError exception
An Employee interface, with
  EmployeeInfo getEmployee (in wstring name)
  void updateEmployee (in EmployeeInfo name)
  void commit()

The SQLError exception is used so that SQLException messages can
be passed back to the client.


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 server object 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/myEmployee scott tiger


where LIBs is the classpath that must include

$ORACLE_HOME/lib/aurora_client.jar
#use classes12.zip if using Java 2
$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 myEmployee CORBA published object on the server
       (this step also authenticates the client using NON_SSL_LOGIN and
        activates the server object)
   - gets and prints information about the employee SCOTT
   - increases SCOTT's salary by 10%
   - updates the EMP table with the new salary by calling the updateEmployee()
      method on the employee object
   - commits the update by invoking employee.commit()

In other words, this client does everything that the ../clientside/Client.java
program did, but does the transaction handling (a commit only) on the server.

The printed output is:

Beginning salary = 3000.0
Increase by 10%
Final Salary = 3300.0

Note that the starting value is taken from the EMP table when the
example starts to run, so you may see a different salary amount. The new
salary amount is written back to the database, and will be used as the
new starting amount if you run this example again.


employeeServer/EmployeeImpl.sqlj
--------------------------------

Implements the Employee interface.  This file implements the two
methods specified in the IDL: getEmployee() and updateEmployee(),
using SQLJ for ease of DML coding.

EmployeeImpl.sqlj also implements a commit() method, that uses JDBC to issue a
SQL COMMIT statement.

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.
```

employee.idl

```
module employee {
  struct EmployeeInfo {
    wstring name;
    long number;
    double salary;
  };

  exception SQLError {
    wstring message;
  };

  interface Employee {
    EmployeeInfo getEmployee (in wstring name) raises (SQLError);
    void updateEmployee (in EmployeeInfo name) raises (SQLError);
    void commit () raises (SQLError);
  };
};
```

Client.java

```
import employee.*;

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];

    // get the handle to the InitialContext
    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);

    // This is using Server-side TX services, specifically, JDBC TX:

    // Now, get the handle to the object and it's info
    Employee employee = (Employee)ic.lookup (serviceURL + objectName);
    EmployeeInfo info = employee.getEmployee ("SCOTT");
    System.out.println ("Beginning salary = " + info.salary);
    System.out.println ("Increase by 10%");

    // do work on the object or it's info
    info.salary += (info.salary * 10) / 100;

    // call update on the server-side
    employee.updateEmployee (info);

    // call commit on the server-side
    employee.commit ();

    System.out.println ("Final Salary = " + info.salary);
  }
}
```

employeeServer/EmployeeImpl.sqlj

```
package employeeServer;

import employee.*;
import java.sql.*;

import oracle.aurora.jts.util.*;
import org.omg.CosTransactions.*;

public class EmployeeImpl
       extends _EmployeeImplBase
{

  public EmployeeInfo getEmployee (String name) throws SQLError {
    try {
      int empno = 0;
      double salary = 0.0;
      #sql { select empno, sal into :empno, :salary from emp
            where ename = :name };
      return new EmployeeInfo (name, empno, (float)salary);
    } catch (SQLException e) {
      throw new SQLError (e.getMessage ());
    }
  }

  public void updateEmployee (EmployeeInfo employee) throws SQLError {
    try {
      #sql { update emp set ename = :(employee.name), sal = :(employee.salary)
                    where empno = :(employee.number) };
    } catch (SQLException e) {
      throw new SQLError (e.getMessage ());
    }
  }

  public void commit () throws SQLError {
    try {
      #sql { commit };
    } catch (SQLException e) {
      throw new SQLError (e.getMessage ());
    }
  }
}
```

serversideJTS

readme.txt

```
Overview
========

The serversideJTS example shows how to do transaction management for
CORBA server objects from the server object, using the XA JTS
methods.

Compare this example with the clientside example, in which all
transaction management is done on the client.

This example also shows a server object that uses SQLJ in its methods.

Source files
============

employee.idl
------------
The CORBA IDL for the example.  Defines:

An EmployeeInfo struct
A SQLError exception
An Employee interface, with
    EmployeeInfo getEmployee(in wstring name)
    EmployeeInfo getEmployeeForUpdate(in wstring name)
    void updateEmployee(in EmployeeInfo name)

The SQLError exception is used so that SQLException messages can
be passed back to the client.

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 server object 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/myEmployee scott tiger

where LIBs is the classpath that must include

$ORACLE_HOME/lib/aurora_client.jar
#use classes12.zip if using Java 2
$ORACLE_HOME/jdbc/lib/classes111.zip
$ORACLE_HOME/lib/vbjorb.jar
$ORACLE_HOME/lib/vbjapp.jar
$JAVA_HOME/lib/classes.zip

The client code is almost exactly the same as the code in
../clientside/Client.java, but without the JTS transaction calls.

The client code performs the following steps:

   - gets the arguments passed on the command line
   - creates a new JNDI Context (InitialContext())
   - initializes the Aurora transaction service
   - looks up the myEmployee CORBA published object on the server
       (this step also authenticates the client using NON_SSL_LOGIN and
        activates the server object)
   - gets and prints information about the employee SCOTT
   - decreases SCOTT's salary by 10%
   - updates the EMP table with the new salary by calling the updateEmployee()
        method on the employee object
   - gets and prints the new information

The printed output is:

Beginning salary = 3000.0
Decrease by 10%
Final Salary = 2700.0

Note that the starting value is taken from the EMP table when the
example starts to run, so you may see a different salary amount. The new
salary amount is written back to the database, and will be used as the
new starting amount if you run this example again.

employeeServer/EmployeeImpl.sqlj
```

```
--------------------------------
Implements the Employee interface.  This file implements the three
methods specified in the IDL: getEmployee(), getEmployeeForUpdate(),
and updateEmployee(), using SQLJ for ease of DML coding.

EmployeeImpl also adds two private methods, commitTrans() and
startTrans(), that perform XA JTS transaction management from the
server.

Note that on the server there is no need to call
AuroraTransactionService.initialize() to initialize the transaction
manager. This is done automatically by the server ORB.

If the SQLJ code throws a SQLException, it is caught, and a
CORBA-defined SQLError is thrown. This in turn would be
propagated back to the client, where it is handled.

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.
```

employee.idl

```
module employee {
  struct EmployeeInfo {
    wstring name;
    long number;
    double salary;
  };

  exception SQLError {
    wstring message;
  };

  interface Employee {
    EmployeeInfo getEmployee (in wstring name) raises (SQLError);
    EmployeeInfo getEmployeeForUpdate (in wstring name) raises (SQLError);
    void updateEmployee (in EmployeeInfo name) raises (SQLError);
  };
};
```

Client.java

```
import employee.*;

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];

    // get the handle to the InitialContext
    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);

    // This is using Server-side TX services, specifically, JTS/XA TX:

    // get handle to the object and it's info
    Employee employee = (Employee)ic.lookup (serviceURL + objectName);

    // get the info about a specific employee
    EmployeeInfo info = employee.getEmployee ("SCOTT");
    System.out.println ("Beginning salary = " + info.salary);
    System.out.println ("Decrease by 10%");
    // do work on the object or it's info
    info.salary -= (info.salary * 10) / 100;

    // call update on the server-side
    employee.updateEmployee (info);

    System.out.println ("Final Salary = " + info.salary);
  }
}
```

employeeServer/EmployeeImpl.sqlj

```
package employeeServer;

import employee.*;
import java.sql.*;

import oracle.aurora.jts.util.*;
import org.omg.CosTransactions.*;

public class EmployeeImpl extends _EmployeeImplBase
{
```

```
Control c;

private void startTrans () throws SQLError {
  try {
    TS.getTS ().getCurrent ().begin ();
  } catch (Exception e) {
    throw new SQLError ("begin failed:" + e);
  }
}

private void commitTrans () throws SQLError {
  try {
    TS.getTS ().getCurrent ().commit (true);
  } catch (Exception e) {
    throw new SQLError ("commit failed:" + e);
  }
}

public EmployeeInfo getEmployee (String name) throws SQLError {
  try {
    startTrans ();

    int empno = 0;
    double salary = 0.0;
    #sql { select empno, sal into :empno, :salary from emp
          where ename = :name };
    c = TS.getTS().getCurrent().suspend();
    return new EmployeeInfo (name, empno, (float)salary);
  } catch (SQLException e) {
    throw new SQLError (e.getMessage ());
  } catch (Exception e) {
    throw new SQLError (e.getMessage());
  }
}

public EmployeeInfo getEmployeeForUpdate (String name) throws SQLError {
  try {
    startTrans ();

    int empno = 0;
    double salary = 0.0;
    #sql { select empno, sal into :empno, :salary from emp
          where ename = :name for update };
    return new EmployeeInfo (name, empno, (float)salary);
  } catch (SQLException e) {
      throw new SQLError (e.getMessage ());
    }
  }

  public void updateEmployee (EmployeeInfo employee) throws SQLError {
    try {
      TS.getTS().getCurrent().resume(c);

      #sql { update emp set ename = :(employee.name), sal = :(employee.salary)
                    where empno = :(employee.number) };
      commitTrans ();
    } catch (SQLException e) {
      throw new SQLError (e.getMessage ());
    } catch (Exception e) {
      throw new SQLError (e.getMessage ());
    }
  }
}
```

serversideLogging

readme.txt

```
Overview
========

The serversideLoggin example shows how to do transaction management
for CORBA server objects both directly from the client application, as
in the clientside example, but also adds a method in the server object
that suspends the current transaction, starts a new transaction, and
writes some data out to a table. The second transaction is then
committed and the first transaction is resumed.

Finally, in the original transaction context on the client the update
that happened in a server object method is committed, to end the
transaction.

Source files
============

employee.idl
------------
The CORBA IDL for the example.  Defines:
```

```
An EmployeeInfo struct
A SQLError exception
An Employee interface, with
    EmployeeInfo getEmployee(in wstring name)
    EmployeeInfo getEmployeeForUpdate(in wstring name)
    void updateEmployee(in EmployeeInfo name)

The SQLError exception is used so that SQLException messages can
be passed back to the client.

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 server object 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/myEmployee scott tiger

where LIBs is the classpath that must include
$ORACLE_HOME/lib/aurora_client.jar
#use classes12.zip if using Java 2
$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())
   - initializes the Aurora transaction service
   - looks up the myEmployee CORBA published object on the server
       (this step also authenticates the client using NON_SSL_LOGIN and
        activates the server object)
   - starts a new transaction: TS.getTS().getCurrent().begin();
   - gets and prints information about the employee SCOTT
   - increases SCOTT's salary by 10%
   - updates the EMP table with the new salary by calling the updateEmployee()
        method on the employee object
   - gets and prints the new information
   - commits the update: TS.getTS().getCurrent().commit(false);

The client application prints:

Beginning salary = 3000.0
Increase by 10%
End salary = 3300.0

Note that the starting value is taken from the EMP table when the
example starts to run, so you may see a different salary amount. The new
salary amount is written back to the database, and will be used as the
new starting amount if you run this example again.

log.sql
-------
This SQL script creates the log_table table that is used by the
EmployeeImpl class to log database updates.

employeeServer/EmployeeImpl.sqlj
--------------------------------
Implements the Employee interface.  This file implements the three
methods specified in the IDL: getEmployee(), getEmployeeForUpdate(),
and updateEmployee() These methods use SQLJ for ease of DML coding.

The class also implements a private method, log(), that is invoked by
the getEmployee() and getEmployeeForUpdate() methods. The log() method
suspends the current transaction, begins a new transaction, and
updates the log_table with information on who did what.

If the SQLJ code throws a SQLException, it is caught, and a
CORBA-defined SQLError is thrown. This in turn is propagated back to
the client, where it is handled.

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.
```

employee.idl

```
module employee {
  struct EmployeeInfo {
    wstring name;
    long number;
    double salary;
  };

  exception SQLError {
    wstring message;
  };

  interface Employee {
    EmployeeInfo getEmployee (in wstring name) raises (SQLError);
    EmployeeInfo getEmployeeForUpdate (in wstring name) raises (SQLError);
    void updateEmployee (in EmployeeInfo name) raises (SQLError);
  };
};
```

Client.java

```
import employee.*;

import oracle.aurora.jndi.sess_iiop.ServiceCtx;

import oracle.aurora.jts.client.AuroraTransactionService;
import oracle.aurora.jts.util.TS;

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];

    // get an handle to the InitialContext
    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);

    // get handle to the TX-Factory
    AuroraTransactionService.initialize (ic, serviceURL);

    // create an instance of an object to be modified in the TX
    Employee employee = (Employee)ic.lookup (serviceURL + objectName);
    EmployeeInfo info;

    // start the TX
    TS.getTS ().getCurrent ().begin ();

    // get employee-info filled up in the TX from the server
    info = employee.getEmployeeForUpdate ("SCOTT");
    System.out.println ("Beginning salary = " + info.salary);
    System.out.println ("Increase by 10%");

    // do work on the object in the TX; e.g. change the info
    info.salary += (info.salary * 10) / 100;

    // update the info in the TX
    employee.updateEmployee (info);

    // get and print the employee and it's info
    info = employee.getEmployee ("SCOTT");
    System.out.println ("End salary = " + info.salary);

    // commit the TX
    TS.getTS ().getCurrent ().commit (true);
  }
}
```

log.sql

```
create table log_table (when date, which number, who number,
               what varchar2(2000));
exit
```

employeeServer/EmployeeImpl.sqlj

```
package employeeServer;

import employee.*;
import oracle.aurora.AuroraServices.ActivatableObject;
import java.sql.*;

import oracle.aurora.rdbms.DbmsJava;
import oracle.aurora.rdbms.Schema;

import oracle.aurora.jts.util.*;
import org.omg.CosTransactions.*;

public class EmployeeImpl
       extends _EmployeeImplBase
       implements ActivatableObject
{
  public EmployeeInfo getEmployee (String name) throws SQLError {
    try {
      int empno = 0;
      double salary = 0.0;
      log ("getEmployee (" + name + ")");
      #sql { select empno, sal into :empno, :salary from emp
            where ename = :name };
      return new EmployeeInfo (name, empno, (float)salary);
    } catch (SQLException e) {
      throw new SQLError (e.getMessage ());
    }
  }

  public EmployeeInfo getEmployeeForUpdate (String name) throws SQLError {
    try {
      int empno = 0;
      double salary = 0.0;
      log ("getEmployeeForUpdate (" + name + ")");
      #sql { select empno, sal into :empno, :salary from emp
            where ename = :name for update };
      return new EmployeeInfo (name, empno, (float)salary);
    } catch (SQLException e) {
      throw new SQLError (e.getMessage ());
    }
  }

  public void updateEmployee (EmployeeInfo employee) throws SQLError {
    log ("updateEmployee (" + employee + ")");
    try {
      #sql { update emp set ename = :(employee.name), sal = :(employee.salary)
                    where empno = :(employee.number) };
    } catch (SQLException e) {
      throw new SQLError (e.getMessage ());
    }
  }

  private void log (String message) throws SQLError {
    try {
      // Get the current TX and suspendTxn it
      Control c = TS.getTS ().getCurrent ().suspend ();

      // Start a new transaction
      TS.getTS ().getCurrent ().begin ();

      // Get the current user name
      int ownerNumber = Schema.currentSchema ().ownerNumber ();

      // Get the session-id
```

```
      int sessID = DbmsJava.sessionID (DbmsJava.USER_SESSION);

      // Insert the information in the log table
      #sql { insert into log_table (who, which, when, what)
                    values (:ownerNumber, :sessID, sysdate, :message) };

      // Commit the TX started for logging the info
      TS.getTS ().getCurrent ().commit (true);

      // Resume the suspended TX
      TS.getTS ().getCurrent ().resume (c);
    } catch (Exception e) {
      throw new SQLError (e.toString ());
    }
  }

  public org.omg.CORBA.Object _initializeAuroraObject () {
    return this;
  }
}
```

multiSessions

readme.txt

```
Overview
========

Source files
============

employee.idl
------------

Client.java
-----------
You invoke the client program from a command prompt, and pass it five
arguments, the

   - service URL (service ID, hostname, port, and SID if port is a listener)
   - name of the published server object to lookup and instantiate
   - username
   - password that authenticates the client to the Oracle8i database server
   - number of new threads/sessions to create
```

```
For example:
% java -classpath LIBs Client sess_iiop://localhost:2481:ORCL \
        /test/myEmployee scott tiger 3

where LIBs is the classpath that must include
$ORACLE_HOME/lib/aurora_client.jar
#use classes12.zip if using Java 2
$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())
   - in a for-loop, creates different sessions using the ClientThread
     class, and prints out information about the session ID

The printed output from Client should be something like this:

Starting ClientThread (:session0)
Starting ClientThread (:session1)
Beginning salary = 3000.0  in :session0
Increase by 100:session0
End salary = 3100.0  in :session0
Beginning salary = 3100.0  in :session1
Increase by 200:session1
End salary = 3300.0 in :session1
Starting ClientThread (:session2)
Beginning salary = 3300.0  in :session2
Decrease by 300:session2
End salary = 3000.0  in :session2

The starting value is taken from the EMP table when the example starts to run.

ClientThread.java
-----------------
The ClientThread constructor creates a new named session in the server, and
authenticates the client with NON_SSL_LOGIN, using the Context, service URL,
published object name, username, and password passed as parameters.
(NON_SSL_LOGIN is specified in the Context passed from Client.java.)

The implementation of run() first yields to any other running threads. When
run, it then initializes its transaction context, activates an Employee object
```

```
in its session, and starts a new transaction.

It then selects for update the SCOTT row in the EMP table, by calling a method
on the employee object, and updates SCOTT's salary in a way dependent on the
name of the session (this is a Dilbert world).

Finally, it prints the new salary information, and commits the update (thus
unlocking the EMP table row).

employeeServer/EmployeeImpl.sqlj
--------------------------------
Implements the Employee interface.  This file implements the two methods
specified in the IDL: getEmployee(), getEmployeeForUpdate(), and
updateEmployee(), using SQLJ for ease of DML coding.

See the description of this file in ../clientside/employeeServer for more
information.

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.
```

employee.idl

```
module employee {
  struct EmployeeInfo {
    wstring name;
    long number;
    double salary;
  };

  exception SQLError {
    wstring message;
  };

  interface Employee {
    EmployeeInfo getEmployee (in wstring name) raises (SQLError);
    EmployeeInfo getEmployeeForUpdate (in wstring name) raises (SQLError);
    void updateEmployee (in EmployeeInfo name) raises (SQLError);
  };
};
```

Client.java

```
import employee.*;

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 objectName user password "
  + "sessionsCount");
      System.exit (1);
    }
    String serviceURL = args [0];
    String objectName = args [1];
    String user = args [2];
    String password = args [3];
    int sessionCount = Integer.parseInt (args[4]);

    // get the handle to InitialContext
    // Note: authentication is done per session in ClientThread
    Hashtable env = new Hashtable ();
    env.put (Context.URL_PKG_PREFIXES, "oracle.aurora.jndi");
    Context ic = new InitialContext (env);

    // invoke different sessions using ClientThread
    for (int i = 0; i < sessionCount; i++) {
      String sessionName = new String (":session" + i);
      ClientThread ct =
                 new ClientThread (ic, serviceURL, objectName, sessionName,
                   user, password);
      System.out.println ("Starting ClientThread (" + sessionName + ")");
      ct.start ();
    }
  }
}
```

ClientThread.java

```
import employee.*;

import oracle.aurora.jts.client.AuroraTransactionService;
import oracle.aurora.jndi.sess_iiop.ServiceCtx;
import oracle.aurora.jndi.sess_iiop.SessionCtx;
import oracle.aurora.AuroraServices.LoginServer;
import oracle.aurora.client.Login;
import oracle.aurora.jts.util.TS;

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

public class ClientThread extends Thread
{
  private Context ic = null;
  private String serviceURL = null;
  private String objectName = null;
  private String sessionName = null;
  private SessionCtx session = null;

  public ClientThread () {}

  public ClientThread (Context ic, String serviceURL, String objectName,
       String sessionName, String user, String password)
  {
    try {
      this.ic = ic;
      ServiceCtx service = (ServiceCtx)ic.lookup (serviceURL);
      this.session = (SessionCtx)service.createSubcontext (sessionName);

      LoginServer login_server = (LoginServer)session.activate ("etc/login");
      Login login = new Login (login_server);
      login.authenticate (user, password, null);

      this.serviceURL = serviceURL;
      this.sessionName = sessionName;
      this.objectName = objectName;
    } catch (Exception e) {
      e.printStackTrace ();
    }
  }

  public void run () {
    try {
      this.yield ();

      // Get handle to the TX-Factory
      AuroraTransactionService.initialize (ic, serviceURL + "/" + sessionName);

      // create an instance of an employee object in the session
      Employee employee = (Employee)session.activate (objectName);
      EmployeeInfo info;

      // start the transaction
      TS.getTS ().getCurrent ().begin ();

      // Get the info about an employee
      // Note: lock is set on the row using 'for update' clause
      // while select operation
      info = employee.getEmployeeForUpdate ("SCOTT");
      System.out.println ("Beginning salary = " + info.salary +
                  "  in " + sessionName);
```

```
        // arbitrarily change the value of the salary,
        // e.g. depending on sessionName
        if (sessionName.endsWith("0")) {
                       System.out.println ("Increase by 100" + sessionName);
                       info.salary += 100;
        } else if (sessionName.endsWith("1")) {
                       System.out.println ("Increase by 200" + sessionName);
                       info.salary += 200;
        } else {
                       System.out.println ("Decrease by 300" + sessionName);
                       info.salary -= 300;
        }

        // Try sleeping this thread for a while before updating the info
        // Note: the other threads MUST wait
        // (since selected with 'for update' clause)
        this.sleep (2000);

        // update the infomation in the transaction
        employee.updateEmployee (info);

        // Get and print the info in the transaction
        // Note: do NOT use 'for update' here
        info = employee.getEmployee ("SCOTT");
        System.out.println ("End salary = " + info.salary + "  in " +
                        sessionName);

        // commit the changes
        TS.getTS ().getCurrent ().commit (true);

      } catch (Exception e) {
        e.printStackTrace ();
      }
    }
  }
```

employeeServer/EmployeeImpl.sqlj

```
 package employeeServer;

import employee.*;
import oracle.aurora.AuroraServices.ActivatableObject;
import java.sql.*;
import oracle.aurora.jts.util.*;
import org.omg.CosTransactions.*;

public class EmployeeImpl
       extends _EmployeeImplBase
       implements ActivatableObject
{
  public EmployeeInfo getEmployee (String name) throws SQLError {
    try {
      int empno = 0;
      double salary = 0.0;
      #sql { select empno, sal into :empno, :salary from emp
              where ename = :name };
      return new EmployeeInfo (name, empno, (float)salary);
    } catch (SQLException e) {
      throw new SQLError (e.getMessage ());
    }
  }

  public EmployeeInfo getEmployeeForUpdate (String name) throws SQLError {
    try {
      int empno = 0;
      double salary = 0.0;
      #sql { select empno, sal into :empno, :salary from emp
            where ename = :name for update };
      return new EmployeeInfo (name, empno, (float)salary);
    } catch (SQLException e) {
      throw new SQLError (e.getMessage ());
    }
  }

  public void updateEmployee (EmployeeInfo employee) throws SQLError {
    try {
      #sql { update emp set ename = :(employee.name), sal = :(employee.salary)
                    where empno = :(employee.number) };
    } catch (SQLException e) {
      throw new SQLError (e.getMessage ());
    }
  }

  public org.omg.CORBA.Object _initializeAuroraObject () {
    return this;
  }
}
```

RMI Examples

The following are the RMI examples:

- helloworld
- callouts
- callback

helloworld

readme.txt

```
Overview
========

The CORBA/RMI helloworld example is the basic example that shows you how
to do RMI calls using the Oracle8i IIOP/RMI transport.

The hello directory contains the interface file Hello.java. This file is
compiled by the java2rmi_iiop compiler to produce the stub and helper files
that are needed to access the remote object that is defined in:
helloServer/HelloImpl.java

Note that hello/Hello.java imports both java.rmi.Remote and
java.rmi.RemoteException, which is required for RMI interfaces.

This example uses the java2rmi_iiop command line tool to generate the required
support classes for the remore object. See the "Tools" chapter of the Oracle8i
EJB and CORBA Developer's Guide for information about this tool.


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
#use classes12.zip if using Java 2
$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 CORBA server object to find and activate it
   - invokes the hello() method on the server object
   - prints the return from hello()

The printed output is the unsurprising:

Hello World!

helloServer/HelloImpl.java
--------------------------
HelloImpl.java defines the hello() method.

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.java

```
import hello.Hello;

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);

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

hello/Hello.java

```
package hello;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface Hello extends Remote {
  public String helloWorld () throws RemoteException;
}
```

helloServer/HelloImpl.java

```
package helloServer;

import hello.*;
import oracle.aurora.AuroraServices.ActivatableObject;

public class HelloImpl extends _HelloImplBase implements ActivatableObject
{
  public String helloWorld () throws java.rmi.RemoteException {
    return "Hello World!";
  }

  public org.omg.CORBA.Object _initializeAuroraObject () {
    return this;
  }
}
```

callouts

Client.java

```
import hello.Hello;

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);

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

HelloRMIClient.java

```
import hello.*;
import oracle.aurora.AuroraServices.ActivatableObject;

import java.rmi.*;
import java.rmi.registry.*;
import java.rmi.server.*;
```

```
public class HelloRMIClient
{
  static public void main (String args[]) throws Exception {
    Hello hello = (Hello) Naming.lookup ("rmi://localhost/subHello");
    System.out.println (hello.helloWorld ());
  }
}
```

HelloRMIServer.java

```
import hello.Hello;
import helloServer.HelloRMIImpl;
import java.rmi.*;
import java.rmi.server.*;

public class HelloRMIServer {
  public static void main (String args[]) throws Exception {
    //    System.setSecurityManager (new RMISecurityManager ());
    HelloRMIImpl hello = new HelloRMIImpl ();
    System.out.println("Hello RMI Server ready.");
  }
}
```

hello/Hello.java

```
package hello;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface Hello extends Remote {
  public String helloWorld () throws RemoteException;
}
```

helloServer/HelloImpl.java

```
package helloServer;

import hello.*;
import oracle.aurora.AuroraServices.ActivatableObject;

import java.rmi.*;
import java.rmi.registry.*;
import java.rmi.server.*;
```

```
public class HelloImpl extends _HelloImplBase implements ActivatableObject
{
  public String helloWorld () throws RemoteException {
    try {
      Hello hello = (Hello) Naming.lookup ("rmi://localhost/subHello");
      return hello.helloWorld ();
    } catch (Exception e) {
      return (e.toString ());
    }
  }

  public org.omg.CORBA.Object _initializeAuroraObject () {
    return this;
  }
}
```

helloServer/HelloRMIImpl.java

```
package helloServer;
import hello.*;

import java.util.*;
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;

public class HelloRMIImpl extends UnicastRemoteObject implements Hello
{
  public HelloRMIImpl () throws RemoteException {
    super ();
    try {
      Naming.rebind ("subHello", this);
    } catch (Exception e) {
      System.out.println("Caught exception: " + e.getMessage());
      e.printStackTrace();
    }
  }

  public String helloWorld () throws java.rmi.RemoteException {
    return "Hello from the RMI server!";
  }
}
```

callback

readme.txt

```
Overview
========

The CORBA/RMI callback example shows how you can do a callback from a CORBA
server object to a client system using RMI for the callback. There is no IDL
for this example. Rather, the sources server/Server.java and
client/Client.java are used by the java2rmi_iiop compiler to generate the
required stub and helper classes.

Compare this example with the corba/basic/callback example, which uses CORBA
IDL, and CORBA callback mechanisms on the client.


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
#use classes12.zip if using Java 2
$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 CORBA server object to find and activate it
   - instantiates a Client callback object
   - invokes the hello() method on the server object, passing it the callback
     object reference (clientImpl)
   - prints the return from hello(clientImpl)

The printed output is:

I Called back and got: Hello Client World!


server/Server.java
------------------

Server.java defines the hello() method.


serverServer/ServerImpl.java
----------------------------

Implements the hello() method defined in server/Server.java.


client/Client.java
------------------

Defines the helloback() method.


clientServer/ClientImpl.java
----------------------------

Implements the helloback() method.



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.java

```
import server.Server;
import clientServer.ClientImpl;

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);

    ClientImpl clientImpl = new ClientImpl ();
    Server server = (Server)ic.lookup (serviceURL + objectName);
    System.out.println (server.hello (clientImpl));
  }
}
```

client/Client.java

```
package client;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface Client extends Remote {
  public String helloBack () throws RemoteException;
}
```

clientServer/ClientImpl.java

```
package clientServer;

import client.*;
import oracle.aurora.AuroraServices.ActivatableObject;
```

```
public class ClientImpl extends _ClientImplBase implements ActivatableObject
{
  public String helloBack () throws java.rmi.RemoteException {
    return "Hello Client World!";
  }

  public org.omg.CORBA.Object _initializeAuroraObject () {
    return this;
  }
}
```

server/Server.java

```
package server;

import client.Client;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface Server extends Remote {
  public String hello (Client client) throws RemoteException;
}
```

serverServer/ServerImpl.java

```
package serverServer;

import server.*;
import client.*;
import oracle.aurora.AuroraServices.ActivatableObject;

public class ServerImpl extends _ServerImplBase implements ActivatableObject
{
  public String hello (Client client) throws java.rmi.RemoteException {
    return "I Called back and got: " + client.helloBack ();
  }

  public org.omg.CORBA.Object _initializeAuroraObject () {
    return this;
  }
}
```

Applet Examples

The following are the Applet examples:

- innetscape
- inappletviewer

innetscape

hello.idl

```
module hello {
  interface Hello {
    wstring helloWorld ();
    void setMessage (in wstring message);
  };
};
```

ClientApplet.htm

```
<!-- /* Adapted from Visigenic's ClientApplet example */ -->
<h1>Oracle 8i ORB Client Applet</h1>
<hr>
<center>
  <applet codebase="." code="ClientApplet" archive="applet.jar"
          width=200 height=80>
  <param name="serviceURL"
value="sess_iiop://<host-name>:<port-num>:<oracle-sid>">
  <param name="objectName" value="/test/myHello">
  <param name="user" value="scott">
  <param name="password" value="tiger">

  <h2>You are probably not running a Java enabled browser.
  Please use a Java enabled browser (or enable your browser for Java)
  to view this applet...</h2>

  </applet>
</center>
<hr>
```

ClientApplet.java

```
/* Adapted from Visigenic's ClientApplet example */
import hello.Hello;
import hello.HelloHelper;

import netscape.security.PrivilegeManager;

import oracle.aurora.jndi.sess_iiop.ServiceCtx;
import oracle.aurora.jndi.sess_iiop.SessionCtx;
import oracle.aurora.client.Login;
import oracle.aurora.AuroraServices.PublishedObject;
import oracle.aurora.AuroraServices.PublishedObjectHelper;
import oracle.aurora.AuroraServices.LoginServer;
import oracle.aurora.AuroraServices.LoginServerHelper;

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

public class ClientApplet extends java.applet.Applet
{
  private TextField _messageField;
  private TextField _outputField;
  private Button _helloButton;

  private Hello hello;

  public boolean action (Event ev, Object arg) {
    if (ev.target == _helloButton) {
      if (hello != null) {

                  // We need these privileges again as the ORB may connect
                  PrivilegeManager.enablePrivilege ("UniversalConnect");
                  PrivilegeManager.enablePrivilege ("UniversalPropertyRead");
                  PrivilegeManager.enablePrivilege ("UniversalPropertyWrite");

                  hello.setMessage (_messageField.getText ());
                  _outputField.setText (hello.helloWorld ());
      }
      return true;
    }
    return false;
  }

  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("Message"));
add(_messageField = new TextField ("Hello World!"));
add(_helloButton = new Button ("Hello"));
add(_outputField = new TextField ());
_outputField.setEditable (false);

String serviceURL = getParameter ("serviceURL");
String objectName = getParameter ("objectName");
String user = getParameter ("user");
String password = getParameter ("password");


try {
  PrivilegeManager.enablePrivilege ("UniversalConnect");
  PrivilegeManager.enablePrivilege ("UniversalPropertyRead");
  PrivilegeManager.enablePrivilege ("UniversalPropertyWrite");

  Hashtable env = new Hashtable ();
  env.put (Context.URL_PKG_PREFIXES, "oracle.aurora.jndi");
  Context ic = new InitialContext (env);

  ServiceCtx service = (ServiceCtx)ic.lookup(serviceURL);
  SessionCtx session1 = (SessionCtx)service.createSubcontext (":session1");

  // Because the Netscape security mechanism prevents usage
  // of the reflection apis deep down in the JNDI runtime
  // we have to activate the objects by hand.

  // Manually activate the login object
  PublishedObject po1 =
              PublishedObjectHelper.narrow ((org.omg.CORBA.Object)
                    (session1.lookup ("/etc/login")));
  LoginServer login_server =
              LoginServerHelper.narrow (po1.activate_no_helper ());

  // Log in the database
  Login login1 = new Login (login_server);
  login1.authenticate (user, password, null);

  // Manually activate the hello object
  PublishedObject po2 =
              PublishedObjectHelper.narrow ((org.omg.CORBA.Object)
                    (session1.lookup (objectName)));
  hello = HelloHelper.narrow (po2.activate_no_helper ());
```

```
    } catch (Exception e) {
      _outputField.setText (e.toString ());
      hello = null;
    }
  }
}
```

helloServer/HelloImpl.java

```
package helloServer;

import hello.*;
import oracle.aurora.AuroraServices.ActivatableObject;

public class HelloImpl extends _HelloImplBase implements ActivatableObject
{
  String message;

  public String helloWorld () {
    return message;
  }

  public void setMessage (String message) {
    this.message = message;
  }

  public org.omg.CORBA.Object _initializeAuroraObject () {
    return this;
  }
}
```

inappletviewer

hello.idl

```
module hello {
  interface Hello {
    wstring helloWorld ();
    void setMessage (in wstring message);
  };
};
```

Clientapplet.htm

```
<!-- /* Adapted from Visigenic's ClientApplet example */ -->
<h1>Oracle 8i ORB Client Applet</h1>
<hr>
<center>
  <applet code=ClientApplet.class width=200 height=80>
  <param name="service" value="sess_iiop://localhost:2481:javavm5">
  <param name="objectName" value="/test/myHello">
  <param name="user" value="scott">
  <param name="password" value="tiger">

  <h2>You are probably not running a Java enabled browser.
  Please use a Java enabled browser (or enable your browser for Java)
  to view this applet...</h2>

  </applet>
</center>
<hr>
```

ClientApplet.java

```
/* Adapted from Visigenic's ClientApplet example */
import hello.Hello;

import oracle.aurora.jndi.sess_iiop.ServiceCtx;

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

public class ClientApplet extends java.applet.Applet
{
  private TextField _messageField;
  private TextField _outputField;
  private Button _helloButton;

  private Hello hello;

  public boolean action (Event ev, Object arg) {
    if (ev.target == _helloButton) {
      if (hello != null) {
              hello.setMessage (_messageField.getText ());
              _outputField.setText (hello.helloWorld ());
      }
      return true;
    }
    return false;
  }

  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("Message"));
    add(_messageField = new TextField ("Hello World!"));
    add(_helloButton = new Button ("Hello"));
    add(_outputField = new TextField ());
    _outputField.setEditable (false);

    String serviceURL = getParameter ("service");
    String objectName = getParameter ("objectName");
    String user = getParameter ("user");
    String password = getParameter ("password");

    try {
      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);

      hello = (Hello)ic.lookup (serviceURL + objectName);
    } catch (Exception e) {
      _outputField.setText (e.toString ());
      hello = null;
    }
  }
}
```

helloServer/HelloImpl.java

```
package helloServer;

import hello.*;
import oracle.aurora.AuroraServices.ActivatableObject;

public class HelloImpl extends _HelloImplBase implements ActivatableObject
{
  String message;

  public String helloWorld () {
    return message;
  }

  public void setMessage (String message) {
    this.message = message;
  }

  public org.omg.CORBA.Object _initializeAuroraObject () {
    return this;
  }
}
```

JNDI Example

The lister example demonstrates JNDI.

lister

readme.txt

```
Lister demonstrates

(1) Using Service Context createSubcontext() method to create a new
session.

(2) Authentication using the session context login() method.

(3) Recursively listing the instance published object tree.


Source files
============


Lister.java
-----------

Invoke the Lister client program from the command line by doing:

% Lister serviceURL username password
```

```
where the serviceURL is a session IIOP service, such as

sess_iiop://<hostname>:<dispatcher_port>

for example:

% Lister sess_iiop://localhost:2222 scott tiger

The lister client first gets a JNDI Initial Context object, ic. Note
that environment passed to the InitialContext() method has only the
Context.URL_PKG_PREFIXES value ("oracle.aurora.jndi"), and not the
username, password, and authentication type, as do many of the other
examples. This is because Lister will authenticate by getting a server
login object, after first instantiating a new session.

The next call in Lister is look up the service on the Context object,
passing in the service identifier string.

Once the service is obtained, a new named session is created. Note
that the session name must start with a colon (:).

The session context is then used to activate the login server at the
standard published location /etc/login. This server object is
preconfigured for you when the database is built. If it is not there,
see your DBA or system administrator.

The Lister client then creates a login client, and invokes its
authenticate() method, passing in the username and password (with a null
role).

The client then uses the SessionCtx object to walk the published
object directory hierarchy, starting from the root ("/"). The name of
the file, its creation date, and the file owner are printed as each
object is encountered.

This example could be expanded to list other attributes of each
published object, such as the access permissions. See the Session
Shell examples for a complete listing (ls -l) of all published object
attributes and associated files.


Compiling and Running the Example
=================================

On UNIX, enter the command 'make' in the shell to compile and run
```

```
the Lister client program.

On Windows NT, use the batch file to compile and run.
```

Lister.java

```
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingEnumeration;
import javax.naming.Binding;
import javax.naming.NamingException;
import javax.naming.CommunicationException;

import oracle.aurora.jndi.sess_iiop.ServiceCtx;
import oracle.aurora.jndi.sess_iiop.SessionCtx;
import oracle.aurora.jndi.sess_iiop.ActivationException;
import oracle.aurora.AuroraServices.PublishedObject;
import oracle.aurora.AuroraServices.objAttribsHolder;
import oracle.aurora.AuroraServices.objAttribs;
import oracle.aurora.AuroraServices.ctxAttribs;
import oracle.aurora.jts.client.AuroraTransactionService;
import oracle.aurora.AuroraServices.LoginServer;
import oracle.aurora.client.Login;

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

import java.util.Hashtable;

public class Lister {

  public static void main (String[] args) throws Exception {
    if (args.length != 3) {
      System.out.println("usage: Lister serviceURL user password");
      System.exit(1);
    }
    String serviceURL = args [0];
    String username = args [1];
    String password = args [2];

    // 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 a session in the instance.
    // The session name must start with a colon(:).
    SessionCtx session = (SessionCtx) service.createSubcontext(":session1");
    session.login(username, password, null);

    // Print a header line.
    System.out.println
      ("\n\nName                         Create Date              Owner");
    listOneDirectory ("/", session);
  }


  public static void listOneDirectory (String name, SessionCtx ctx)
       throws Exception  {
    System.out.print(name);
    for (int i = name.length(); i < 30; i++)
      System.out.print(" ");
    ctxAttribs attribs = null;
    try {
      attribs = ctx.getAttributes();
    } catch (org.omg.CORBA.NO_PERMISSION e) {
      return;
    }

    System.out.print(attribs.creation_ts);
    for (int i = 30 + attribs.creation_ts.length(); i < 55; i++)
      System.out.print(" ");
    System.out.print(attribs.owner);

    /*
     * You could also add output for the access permissions:
     *  attribs.read
     *  attribs.write
     *  attribs.execute
     */

    System.out.println();

    // Show the sub entries
    listEntries(ctx, name);
```

```
}

public static void listEntries (Context context, String prefix)
     throws Exception  {
  NamingEnumeration bindings = context.list("");
  while (bindings.hasMore()){
    Binding binding = (Binding) bindings.next();
    String name = binding.getName();
    Object object = context.lookup(name);
    if (object instanceof SessionCtx)
      listOneDirectory(prefix + name + "/", (SessionCtx) object);
    else if (object instanceof PublishedObject)
      listOneObject(prefix + name, (PublishedObject) object);
    else
      // We should never get here.
      System.out.println(prefix + name + ": " + object.getClass());
  }
}


public static void listOneObject (String name, PublishedObject obj)
     throws Exception  {
  objAttribsHolder holder = new objAttribsHolder();
  try {
    obj.get_attributes(holder);
  } catch (org.omg.CORBA.NO_PERMISSION e) {
    return;
  }

  objAttribs attribs = holder.value;
  System.out.print(name);
  for (int i = name.length(); i < 30; i++)
    System.out.print(" ");

  System.out.print(attribs.creation_ts);
  for (int i = 30 + attribs.creation_ts.length(); i < 55; i++)
    System.out.print(" ");
  System.out.print(attribs.owner);

  /*
   * You could also add output for:
   *  attribs.class_name
   *  attribs.schema
   *  attribs.helper
   * and the access permissions:
      *  attribs.read
      *  attribs.write
      *  attribs.execute
      */

     System.out.println();
   }
 }
```

B

Example Code: EJB

This chapter contains all the EJB example code included on the product CD. See the EJB/CORBA README for the locations of the examples.

Basic Examples

helloworld

readme.txt

```
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
$ORACLE_HOME/jdbc/lib/classes111.zip
#If using Java 2, use classes12.zip instead of classes111.zip
$ORACLE_HOME/jlib/jssl-1_1.jar        # required if you modify any
$ORACLE_HOME/jlib/javax-ssl-1_1.jar   # client code to use SSL
$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.
```

hello.ejb

```
SessionBean helloServer.HelloBean
{
  BeanHomeName = "test/myHello";
  RemoteInterfaceClassName = hello.Hello;
  HomeInterfaceClassName = hello.HelloHome;

  AllowedIdentities = { PUBLIC };
  RunAsMode = CLIENT_IDENTITY;
//  TransactionAttribute = TX_SUPPORTS;
}
```

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);

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

helloServer/HelloBean.java

```
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 () {}
}
```

hello/Hello.java

```
package hello;

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

public interface Hello extends EJBObject
{
  public String helloWorld () throws RemoteException;
}
```

hello/HelloHome.java

```
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;
}
```

saveHandle

readme.txt

```
Overview
========

This example shows how a client program can get a bean handle, using
getHandle(), and write it out to a file. A second client then reads the bean
handle, and accesses the first client's bean.

For simplicity, the example simply writes the bean handle out to a file. In a
'real' program, you would use some other less kludgy but more complicated
means to pass the bean handle.

This example uses SSL_CREDENTIAL authentication for both clients, so the
```

```
Oracle server must have access to a cwallet.sso SSL credential for the example
to run.

Also, the session that the first client creates) must still be alive when
Client2 runs, so you have 60 seconds to run Client2 after Client1 prints its
message. (60 seconds is the timeout value set in the deployment descriptor.)

(See the timeout example in the ejb session directory for a way to keep a
session alive programmatically after the client terminates. You can also set a
high value in the SessionTimeout attribute in the deployment descriptor.)

Source Files
============
Client1.java
-----------
You invoke the first client program from a command prompt, and pass it five
arguments, the
   - service URL (service ID, hostname, port, and SID if port is a listener)
   - name of the published bean to lookup and instantiate
   - database username
   - password that authenticates the client to the Oracle8i database server
   - the name of a file to hold the bean handle

For example:
% java -classpath LIBs Client1 sess_iiop://localhost:2481:ORCL \
    /test/saveHandle scott tiger handlefile.dat

where LIBs is the classpath that must include
$ORACLE_HOME/lib/aurora_client.jar
$ORACLE_HOME/jdbc/lib/classes111.zip
#If using Java 2, use classes12.zip instead of classes111.zip
$ORACLE_HOME/jlib/jssl-1_1.jar       # required if you modify any
$ORACLE_HOME/jlib/javax-ssl-1_1.jar  # client code to use SSL
$ORACLE_HOME/lib/vbjorb.jar
$ORACLE_HOME/lib/vbjapp.jar
$ORACLE_HOME/lib/vbj30ssl.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, testBean
   - sets up an object output stream, using the file name supplied
   - writes the bean handle to the output as an object
   - invokes the query method on the test bean, and prints the results
   - updates the employee's salary

The printed output from Client1 is:

Client1: 7499 (ALLEN) has salary 2600.0

Client2.java
------------
Client2 is called with four arguments. They are:
   - the service URL
   - the username
   - the password
   - the name of the file from which to read the bean handle

Client2 reads the bean handle from the file, and invokes the query() method on
the bean that that gets.

The printed output from Client2 is:

Client2: read the bean handle from the file.
Client2: 7499 (ALLEN) now has salary 3100.0

saveHandle.ejb
--------------
The deployment descriptor for the bean.  If the SessionTimeout attribute is
commented out, that is a work-around for an 8.1.4 bug.

save/saveHandle.java
--------------------
The bean remote interface. Specifies the query() and update() methods that are
implemented in saveHandleServer/saveHandleBean.java.

save/saveHandleHome.java
------------------------
The bean home interface. Specifies the query() and update() methods that are
implemented in saveHandleServer/saveHandleBean.java.

saveHandleServer/saveHandleBean.sqlj
------------------------------------
The bean implementation.

saveHandleServer/EmpRecord.java
-------------------------------
```

```
The class that the update() method of the bean returns.

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.
```

saveHandle.ejb

```
// saveHandle EJB deployment descriptor.

SessionBean saveHandleServer.saveHandleBean {
  BeanHomeName = "test/saveHandle";
  RemoteInterfaceClassName = save.saveHandle;
  HomeInterfaceClassName = save.saveHandleHome;
  AllowedIdentities = {SCOTT};

  SessionTimeout = 60;
  StateManagementType = STATEFUL_SESSION;

  RunAsMode = CLIENT_IDENTITY;

  public save.EmpRecord query (int e) throws SQLException {
    TransactionAttribute = TX_REQUIRED;
    RunAsMode = CLIENT_IDENTITY;
    AllowedIdentities = { SCOTT };
  }

  public void update (int e, double s) throws SQLException {
    TransactionAttribute = TX_REQUIRED;
    RunAsMode = CLIENT_IDENTITY;
    AllowedIdentities = { SCOTT };
  }

  public String getMessage() throws RemoteException {
    RunAsMode = CLIENT_IDENTITY;
    AllowedIdentities = { SCOTT };
  }

  public void setMessage(String message) throws RemoteException {
    RunAsMode = CLIENT_IDENTITY;
    AllowedIdentities = { SCOTT };
  }
}
```

Client1.java

```
import save.saveHandle;
import save.saveHandleHome;
import save.EmpRecord;

import java.io.*;

import oracle.aurora.jndi.sess_iiop.ServiceCtx;

import javax.naming.Context;
import javax.naming.InitialContext;
import java.util.Hashtable;
import java.sql.SQLException;
```

```
public class Client1 {
  public static void main (String [] args) throws Exception {
    int empNumber = 7499;  // ALLEN

    if (args.length != 5) {
      System.out.println("usage: Client serviceURL objectName user password"
 + " handlefile");
      System.exit(1);
    }
    String serviceURL = args [0];
    String objectName = args [1];
    String user = args [2];
    String password = args [3];
    String handlefile = args [4];

    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);

    // Access the Bean
    saveHandleHome home = (saveHandleHome)ic.lookup (serviceURL + objectName);
    saveHandle testBean = home.create ();

    // Save the bean handle to a file.
    FileOutputStream fostream = new FileOutputStream (handlefile);
    ObjectOutputStream ostream = new ObjectOutputStream (fostream);
    ostream.writeObject (testBean.getHandle ());
    ostream.flush ();
    fostream.close ();

    // Get name and current salary.
    EmpRecord empRec = testBean.query(empNumber);
    System.out.print("Client1: ");
    System.out.println(empRec.empno + " (" + empRec.ename
       + ") has salary " + empRec.sal);

    // Increase ALLEN's salary.
    testBean.update (empNumber, empRec.sal + 500.00);
    testBean.setMessage("Client1 updated 7499's salary/");
    // Sleep 30 seconds to let Client2 connect to the SessionBean
    //    Thread.sleep (30000);
  }
}
```

Client2.java

```
import java.io.FileInputStream;
import java.io.ObjectInputStream;
import oracle.aurora.jndi.sess_iiop.ServiceCtx;

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

import save.saveHandle;
import save.saveHandleHome;
import save.EmpRecord;


public class Client2 {
  public static void main (String [] args) throws Exception {
    int empNumber = 7499;     // ALLEN

    if (args.length != 4) {
      System.out.println("usage: Client serviceURL username password"
           + " handlefile");
      System.exit(1);
    }
    String serviceURL = args [0];
    String username = args [1];
    String password = args [2];
    String handlefile = args [3];

    Hashtable env = new Hashtable();
    env.put (Context.URL_PKG_PREFIXES, "oracle.aurora.jndi");
    Context ic = new InitialContext (env);
    ServiceCtx service = (ServiceCtx)ic.lookup (serviceURL);

    // Initialize the service context to authenticate. Role and props
    // are null. Use SSL credential authentication.
    service.init (username, password, null, true, null);

    // Get a ref to the bean, by reading the file.
    FileInputStream finstream = new FileInputStream (handlefile);
    ObjectInputStream istream = new ObjectInputStream (finstream);
    javax.ejb.Handle handle  = (javax.ejb.Handle)istream.readObject ();
    finstream.close ();
    saveHandle bean = (saveHandle)handle.getEJBObject ();
    System.out.println ("Client2: read the bean handle from the file.");

    // Run the query on the bean handle.
    EmpRecord empRec = bean.query (empNumber);
    System.out.println("Client2: " + bean.getMessage());
    System.out.println("Client2: " +
        empRec.empno + " (" + empRec.ename +
        ") now has salary " + empRec.sal);

  }
}
```

save/saveHandle.java

```
package save;

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

public interface saveHandle extends EJBObject  {

  public EmpRecord query (int empNumber)
       throws java.sql.SQLException, RemoteException;

  public void update (int empNumber, double newSalary)
       throws java.sql.SQLException, RemoteException;

  public String getMessage()
       throws RemoteException;

  public void setMessage(String message)
       throws RemoteException;
}
```

save/saveHandleHome.java

```
package save;

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

public interface saveHandleHome extends EJBHome {
  public saveHandle create()
       throws CreateException, RemoteException;
}
```

save/EmpRecord.java

```
package save;

import java.rmi.*;

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;
  }
}
```

saveHandleServer/saveHandleBean.sqlj

```
package saveHandleServer;

import save.EmpRecord;

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

#sql iterator EmpIter (int empno, String ename, double sal);

public class saveHandleBean implements SessionBean {
  String message = "No message";
  SessionContext ctx;

  public void update(int empNumber, double newSalary)
       throws SQLException, RemoteException
  {
    #sql {update emp set sal = :newSalary where empno = :empNumber};
  }

  public EmpRecord query (int empNumber) throws SQLException, RemoteException
  {
    String ename;
    double sal;

    #sql { select ename, sal into :ename, :sal from emp
                  where empno = :empNumber };

    return new EmpRecord (ename, empNumber, sal);
  }

  public String getMessage() throws RemoteException {
    return message;
  }

  public void setMessage(String message) throws RemoteException {
    this.message = message;
  }

  public void ejbCreate() throws CreateException, RemoteException {
  }

  public void ejbActivate() {
  }

  public void ejbPassivate() {
  }

  public void ejbRemove() {
  }

  public void setSessionContext(SessionContext ctx) {
    this.ctx = ctx;
  }
}
```

sqljimpl

readme.txt

```
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.
```

employee.ejb

```
// sqljimpl EJB deployment descriptor.

SessionBean employeeServer.EmployeeBean {
  BeanHomeName = "test/employeeBean";
  RemoteInterfaceClassName = employee.Employee;
  HomeInterfaceClassName = employee.EmployeeHome;

  AllowedIdentities = {SCOTT};

  //  SessionTimeout = 20;
  StateManagementType = STATEFUL_SESSION;

  RunAsMode = CLIENT_IDENTITY;

  TransactionAttribute = TX_REQUIRED;
}
```

Client.java

```
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);
  }
}
```

employee/Employee.java

```
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;
}
```

employee/EmployeeHome.java

```
package employee;

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

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

employee/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;
  }
}
```

employeeServer/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);
  }
}
```

jdbcimpl

readme.txt

```
Overview
========

This example demonstrates using JDBC in an EJB to do a database query.
This example does a simple query of the database EMP table, using JDBC
methods.

Compare this example with the sqljimpl basic EJB example, which uses
SQLJ instead of JDBC 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:2481:ORCL \
         /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:
```

```
Employee name is KING
Employee sal is  5000.0

employeeServer/employeeBean.java
--------------------------------
This class is the bean implementation. A JDBC prepared statement is used
to formulate the query, which contains a WHERE clause.

The result set 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.
```

employee.ejb

```
// jdbcimpl EJB deployment descriptor

SessionBean employeeServer.EmployeeBean {
  BeanHomeName = "test/employeeJDBCBean";
  RemoteInterfaceClassName = employee.Employee;
  HomeInterfaceClassName = employee.EmployeeHome;

  AllowedIdentities = {SCOTT};

  //SessionTimeout = 20;
  StateManagementType = STATEFUL_SESSION;

  RunAsMode = CLIENT_IDENTITY;

  TransactionAttribute = TX_REQUIRED;
}
```

Client.java

```
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 = testBean.query (7839);
    System.out.println ("Employee name is " + empRec.ename);
    System.out.println ("Employee sal is  " + empRec.sal);
  }
}
```

employee/Employee.java

```
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;
}
```

employee/Employeehome.java

```
package employee;

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

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

employee/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;
  }
}
```

employeeServer/EmployeeBean.java

```
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
  {
    Connection conn =
      new oracle.jdbc.driver.OracleDriver().defaultConnection ();
    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();
    }
    //    return null;
  }
}
```

callback

readme.txt

```
Overview
========

This example shows how an EJB can do callbacks to the client system. The
callback mechanism uses RMI over IIOP, and the Caffeine tool java2rmi_iiop is
used to generate the required classes for the RMI mechanisms.

The EJB is called with a reference to a client-side callback object
```

```
(clientImpl), and the bean itself returns a message plus the message that it
gets when it calls back to the client.

That is, the EJB returns "I called back and got: " plus the return value that
it gets when it invokes the client-side callback object method helloBack(),
which in this example is "Hello Client World!".

The UNIX makefile or the makeit.bat NT batch file shows how to invoke
the java2rmi_iiop compiler to generate the required stub and other classes for
the RMI callback mechanism.

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/myServerBean scott tiger

where LIBs is the classpath that must include

$ORACLE_HOME/lib/aurora_client.jar
$ORACLE_HOME/jdbc/lib/classes111.zip
#If using Java 2, use classes12.zip instead of classes111.zip
$ORACLE_HOME/jlib/jssl-1_1.jar       # required if you modify any
$ORACLE_HOME/jlib/javax-ssl-1_1.jar   # client code to use SSL
$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, server
   - invokes the hello() method on the server object, passing it the
```

```
        client-side callback object (clientImpl), and prints the results

The printed output from the client is:

I Called back and got: Hello Client World!

server.ejb
----------
The ServerBean deployment descriptor.

server/ServerHome.java
----------------------
The ServerBean home interface.

server/Server.java
------------------
The ServerBean remote interface.

serverServer/ServerBean.java
----------------------------
The ServerBean implementation. It calls the client-side callback object.

client/Client.java
------------------
The remote interface for the client callback class.

clientServer/ClientImpl.java
----------------------------
The implementation of the client callback class. Note the use of
ActivatableObject in this class.

Compiling and Running the Example
=================================
UNIX
----
Enter the command '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 server.Server;
import server.ServerHome;
import clientServer.ClientImpl;

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);

    // now, create the ClientBean.
    ClientImpl clientImpl = new ClientImpl ();

    // now, create the Server Bean object
    ServerHome server_home = (ServerHome)ic.lookup (serviceURL + objectName);
    Server server = server_home.create ();
    System.out.println (server.hello (clientImpl));
  }
}
```

server.ejb

```
// This the generic database work bean template

SessionBean serverServer.ServerBean
{
  BeanHomeName = "test/myServerBean";
  RemoteInterfaceClassName = server.Server;
  HomeInterfaceClassName = server.ServerHome;

  AllowedIdentities = { PUBLIC };

  // SessionTimeout = 0;
  // StateManagementType = STATEFUL_SESSION;

  RunAsMode = CLIENT_IDENTITY;

  TransactionAttribute = TX_NOT_SUPPORTED;
}
```

client/Client.java

```
package client;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface Client extends Remote {
  public String helloBack () throws RemoteException;
}
```

clientServer/ClientImpl.java

```
package clientServer;

import client.Client;

import java.rmi.RemoteException;
import org.omg.CORBA.Object;

import oracle.aurora.AuroraServices.ActivatableObject;

public class ClientImpl extends client._ClientImplBase implements
ActivatableObject
{
  public String helloBack () throws RemoteException {
    return "Hello Client World!";
  }

  public Object _initializeAuroraObject () {
    return this;
  }
}
```

server/Server.java

```
package server;

import client.Client;

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

public interface Server extends EJBObject
{
  public String hello (Client client) throws RemoteException;
}
```

server/ServerHome.java

```
package server;

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

public interface ServerHome extends EJBHome
{
  public Server create () throws RemoteException, CreateException;
}
```

serverServer/ServerBean.java

```
package serverServer;

import server.Server;
import server.ServerHome;
import client.Client;

import javax.ejb.SessionBean;
import javax.ejb.SessionContext;
import javax.ejb.CreateException;
import java.rmi.RemoteException;

public class ServerBean implements SessionBean
{
  // Methods of the Hello interface
  public String hello (Client client) throws RemoteException
  {
    return "I Called back and got: " + client.helloBack ();
  }

  // Methods of the SessionBean
  public void ejbCreate () throws RemoteException, CreateException {}
  public void ejbRemove() {}
  public void setSessionContext (SessionContext ctx) {}
  public void ejbActivate () {}
  public void ejbPassivate () {}
}
```

beanInheritance

readme.txt

```
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
$ORACLE_HOME/jdbc/lib/classes111.zip
#If using Java 2, use classes12.zip instead of classes111.zip
$ORACLE_HOME/jlib/jssl-1_1.jar       # required if you modify any
$ORACLE_HOME/jlib/javax-ssl-1_1.jar   # client code to use SSL
$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.java

```
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());
  }
}
```

foo.ejb

```
SessionBean inheritanceServer.FooBean
{
  BeanHomeName = "/test/foo";
  RemoteInterfaceClassName = inheritance.Foo;
  HomeInterfaceClassName = inheritance.FooHome;

  AllowedIdentities = { PUBLIC };
  RunAsMode = CLIENT_IDENTITY;
}
```

bar.ejb

```
SessionBean inheritanceServer.BarBean
{
  BeanHomeName = "/test/bar";
  RemoteInterfaceClassName = inheritance.Bar;
  HomeInterfaceClassName = inheritance.BarHome;

  AllowedIdentities = { PUBLIC };
  RunAsMode = CLIENT_IDENTITY;
}
```

inheritance/Foo.java

```
package inheritance;

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

public interface Foo extends EJBObject
{
  public String hello () throws RemoteException;
}
```

inheritance/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;
}
```

inheritance/Bar.java

```
package inheritance;

import java.rmi.RemoteException;

public interface Bar extends inheritance.Foo
{
  public String hello2 () throws RemoteException;
}
```

inheritance/BarHome.java

```
package inheritance;

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

public interface BarHome extends EJBHome {
  public Bar create () throws RemoteException, CreateException;
}
```

inheritanceServer/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 () {
  }
}
```

inheritanceServer/BarBean.java

```
package inheritanceServer;

import java.rmi.RemoteException;
import javax.ejb.SessionBean;
import javax.ejb.SessionContext;
import javax.ejb.CreateException;

public class BarBean extends FooBean
{
  // Methods of the SessionBean are all from ancestor
  public void ejbCreate () throws RemoteException, CreateException {
    super.ejbCreate();
  }

  public String hello () throws RemoteException {
    return "Hello World from bar";
  }

  public String hello2 () throws RemoteException {
    return "Hello World 2 from bar";
  }
}
```

Transaction Examples

clientside

employee.ejb

```
SessionBean employeeServer.EmployeeBean
{
  BeanHomeName = "test/myEmployee";
  RemoteInterfaceClassName = employee.Employee;
  HomeInterfaceClassName = employee.EmployeeHome;

  AllowedIdentities = { PUBLIC };
  RunAsMode = CLIENT_IDENTITY;
  TransactionAttribute = TX_SUPPORTS;
}
```

Client.java

```
import employee.Employee;
import employee.EmployeeHome;
import employee.EmployeeInfo;

import oracle.aurora.jndi.sess_iiop.ServiceCtx;

import oracle.aurora.jts.client.AuroraTransactionService;
import oracle.aurora.jts.util.TS;

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];

    // create InitialContext
    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);

    // initialize the transaction service
    AuroraTransactionService.initialize (ic, serviceURL);

    // 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 ();

    // get Control to the transaction
    TS.getTS ().getCurrent ().begin ();

    // 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);

    // commit the updated value
    TS.getTS ().getCurrent ().commit (true);

    // start another transaction to get the info again
    // Note: you can also do this before the previous-TX is committed
    //       without starting a new TX (see ../serversideJTS/Client.java)
    TS.getTS ().getCurrent ().begin ();
```

```
    EmployeeInfo infoAfterUpdate = employee.getEmployee ("SCOTT");
    System.out.println ("End salary = " + infoAfterUpdate.salary);

    // commit the updated value
    TS.getTS ().getCurrent ().commit (false);
  }
}
```

employee/EmployeeInfo.java

```
package employee;

public class EmployeeInfo implements java.io.Serializable {
  public String name = null;
  public int number = 0;
  public double salary = 0;

  public EmployeeInfo (String name, int number, double salary) {
    this.name = name;
    this.number = number;
    this.salary = salary;
  }
}
```

employee/Employee.java

```
package employee;

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

import java.sql.SQLException;

public interface Employee extends EJBObject
{
  public EmployeeInfo getEmployee (String name)
       throws RemoteException, SQLException;

  public void updateEmployee (EmployeeInfo employee)
       throws RemoteException, SQLException;
}
```

employee/EmployeeHome.java

```
package employee;
```

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

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

employeeServer/EmployeeBean.sqlj

```
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 () {}
}
```

multiSessions

employee.ejb

```
SessionBean employeeServer.EmployeeBean
{
  BeanHomeName = "test/myEmployee";
  RemoteInterfaceClassName = employee.Employee;
  HomeInterfaceClassName = employee.EmployeeHome;

  AllowedIdentities = { PUBLIC };
  RunAsMode = CLIENT_IDENTITY;
  TransactionAttribute = TX_SUPPORTS;
}
```

Client.java

```
import employee.*;

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 objectName " +
  "user password sessionsCount");
      System.exit (1);
    }
    String serviceURL = args [0];
    String objectName = args [1];
    String user = args [2];
    String password = args [3];
    int sessionCount = Integer.parseInt (args[4]);

    // create InitialContext
    // Note: authentication is done per session in ClientThread
    Hashtable env = new Hashtable ();
    env.put (Context.URL_PKG_PREFIXES, "oracle.aurora.jndi");
    Context ic = new InitialContext (env);

    // invoke different sessions using ClientThread
    for (int i = 0; i < sessionCount; i++) {
      String sessionName = new String (":session" + i);
      ClientThread ct = new ClientThread (ic, serviceURL, objectName,
  sessionName, user, password);
      System.out.println ("Starting ClientThread (" + sessionName + ")");
      ct.start ();
    }
  }
}
```

ClientThread.java

```
import employee.*;

import oracle.aurora.jts.client.AuroraTransactionService;
import oracle.aurora.jndi.sess_iiop.ServiceCtx;
import oracle.aurora.jndi.sess_iiop.SessionCtx;
import oracle.aurora.AuroraServices.LoginServer;
import oracle.aurora.client.Login;
import oracle.aurora.jts.util.TS;

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

public class ClientThread extends Thread
{
  private Context ic = null;
  private String serviceURL = null;
  private String objectName = null;
  private String sessionName = null;
  private SessionCtx session = null;

  public ClientThread () {}

  public ClientThread (Context ic, String serviceURL, String objectName,
       String sessionName, String user, String password)
  {
    try {
      this.ic = ic;
      ServiceCtx service = (ServiceCtx)ic.lookup (serviceURL);
      this.session = (SessionCtx)service.createSubcontext (sessionName);
      System.out.println ("activating the " + sessionName + " in " +
                   serviceURL);

      LoginServer login_server = (LoginServer)session.activate ("etc/login");
      Login login = new Login (login_server);
      login.authenticate (user, password, null);

      this.serviceURL = serviceURL;
      this.sessionName = sessionName;
      this.objectName = objectName;
    } catch (Exception e) {
      e.printStackTrace ();
    }
  }

  public void run () {
    try {
      this.yield ();

      // Get handle to the TX-Factory
      AuroraTransactionService.initialize (ic, serviceURL + "/" + sessionName);

      // create an instance of an employee object in the session
      EmployeeHome employee_home =
                 (EmployeeHome)ic.lookup (serviceURL + "/" + sessionName +
                   objectName);

      Employee employee = employee_home.create ();

      System.out.println ("employee_home.create () DONE in " + sessionName);

      EmployeeInfo info = null;

      // start the transaction
      TS.getTS ().getCurrent ().begin ();

      // get the info about an employee
      // Note: lock is set on the row using 'for update' clause
      //       while select operation
```

```
        info = employee.getEmployeeForUpdate ("SCOTT");
        System.out.println ("Beginning salary = " + info.salary + "  in " +
                     sessionName);

        // arbitrarily change the value of the salary,
        // for e.g. depending on sessionName
        if (sessionName.endsWith ("0")) {
                   System.out.println ("10% Increase" + sessionName);
                   info.salary += (info.salary * 10) / 100;
        } else if (sessionName.endsWith ("1")) {
                   System.out.println ("20% Increase" + sessionName);
                   info.salary += (info.salary * 20) / 100;
        } else {
                   System.out.println ("30% Increase" + sessionName);
                   info.salary += (info.salary * 30) / 100;
        }

        // try sleeping this-thread for a while before updating the info
        // Note: the other threads MUST wait (since selected with
        //                                    'for update' clause)
        this.sleep (2000);

        // update the infomation in the transaction
        employee.updateEmployee (info);

        // get and print the info in the transaction
        // Note: overwrite info structure
        //       doNOT use 'for update' here (unnecessary locking)
        info = employee.getEmployee ("SCOTT");
        System.out.println ("End salary = " + info.salary + "  in " +
                 sessionName);

        // commit the changes
        TS.getTS ().getCurrent ().commit (true);
      } catch (Exception e) {
        e.printStackTrace ();
      }
    }
  }
```

employee/Employee.java

```
package employee;

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

import java.sql.SQLException;

public interface Employee extends EJBObject
{
  public EmployeeInfo getEmployee (String name)
       throws RemoteException, SQLException;

  public EmployeeInfo getEmployeeForUpdate (String name)
       throws RemoteException, SQLException;

  public void updateEmployee (EmployeeInfo employee)
       throws RemoteException, SQLException;
}
```

employee/EmployeeHome.java

```
package employee;

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

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

employee/EmployeeInfo.java

```
package employee;

public class EmployeeInfo implements java.io.Serializable {
  public String name = null;
  public int number = 0;
  public double salary = 0;

  public EmployeeInfo (String name, int number, double salary) {
    this.name = name;
    this.number = number;
    this.salary = salary;
  }
}
```

employeeServer/EmployeeBean.sqlj

```
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 remote 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 EmployeeInfo getEmployeeForUpdate (String name)
       throws RemoteException, SQLException
  {
    int empno = 0;
    double salary = 0.0;
    #sql { select empno, sal into :empno, :salary from emp
                  where ename = :name for update };

    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 () {}
}
```

serversideJTS

Client.java

```
import employee.Employee;
import employee.EmployeeHome;
import employee.EmployeeInfo;

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];

    // create InitialContext
    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);

    // get handle to the employee object
    EmployeeHome employee_home = (EmployeeHome)ic.lookup (serviceURL +
             objectName);

    Employee employee = employee_home.create ();

    EmployeeInfo info = employee.getEmployeeForUpdate ("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);

    info = employee.getEmployee ("SCOTT");
    System.out.println ("End salary = " + info.salary);
  }
}
```

employee.ejb

```
SessionBean employeeServer.EmployeeBean
{
  BeanHomeName = "test/myEmployee";
  RemoteInterfaceClassName = employee.Employee;
  HomeInterfaceClassName = employee.EmployeeHome;

  AllowedIdentities = { PUBLIC };
  RunAsMode = CLIENT_IDENTITY;
  TransactionAttribute = TX_BEAN_MANAGED;
}
```

employee/Employee.java

```
package employee;

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

import java.sql.SQLException;

public interface Employee extends EJBObject
{
  public EmployeeInfo getEmployee (String name)
       throws RemoteException, SQLException;
  public EmployeeInfo getEmployeeForUpdate (String name)
       throws RemoteException, SQLException;

  public void updateEmployee (EmployeeInfo employee)
       throws RemoteException, SQLException;
}
```

employee/EmployeeHome.java

```
package employee;

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

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

employee/EmployeeInfo.java

```
package employee;

public class EmployeeInfo implements java.io.Serializable {
  public String name = null;
  public int number = 0;
  public double salary = 0;

  public EmployeeInfo (String name, int number, double salary) {
    this.name = name;
    this.number = number;
    this.salary = salary;
  }
}
```

employeeServer/EmployeeBean.sqlj

```
package employeeServer;

import employee.*;

import javax.ejb.SessionBean;
import javax.ejb.CreateException;
import javax.ejb.SessionContext;
import javax.jts.UserTransaction;

import java.rmi.RemoteException;
import java.sql.SQLException;

public class EmployeeBean implements SessionBean
{
  SessionContext ctx;

  // Methods of the Employee interface
  public EmployeeInfo getEmployee (String name)
       throws RemoteException, SQLException
  {
    ctx.getUserTransaction ().begin ();

    int empno = 0;
    double salary = 0.0;
    #sql { select empno, sal into :empno, :salary from emp
                  where ename = :name };

    ctx.getUserTransaction ().commit ();

    return new EmployeeInfo (name, empno, salary);

  }

  public EmployeeInfo getEmployeeForUpdate (String name)
       throws RemoteException, SQLException
  {
    ctx.getUserTransaction ().begin ();

    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) };
    ctx.getUserTransaction ().commit ();
  }

  // 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 () {}
}
```

serversideLogging

Client.java

```
import employee.Employee;
import employee.EmployeeHome;
import employee.EmployeeInfo;

import oracle.aurora.jndi.sess_iiop.ServiceCtx;
import oracle.aurora.jts.client.AuroraTransactionService;
import oracle.aurora.jts.util.TS;

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];

    // create InitialContext
    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);

      // initialize the transaction service
      AuroraTransactionService.initialize (ic, serviceURL);

      // get Control to the transaction
      TS.getTS ().getCurrent ().begin ();

      // get handle to the employee object
      EmployeeHome employee_home = (EmployeeHome)ic.lookup
                                            (serviceURL + objectName);

      Employee employee = employee_home.create ();

      EmployeeInfo info = employee.getEmployeeForUpdate ("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);

      // commit the updated value
      TS.getTS ().getCurrent ().commit (true);

      // start another transaction to get the info again
      // Note: you can also do this before the previous-TX is committed
      //       without starting a new TX (see ../serversideJTS/Client.java)
      TS.getTS ().getCurrent ().begin ();

      // re-query for the info object
      EmployeeInfo newInfo = employee.getEmployee ("SCOTT");
      System.out.println ("End salary = " + newInfo.salary);

      // commit the updated value
      TS.getTS ().getCurrent ().commit (false);
    }
  }
```

employee.ejb

```
// This the generic database work bean template

SessionBean employeeServer.EmployeeBean {
  BeanHomeName = "test/myEmployee";
  RemoteInterfaceClassName = employee.Employee;
  HomeInterfaceClassName = employee.EmployeeHome;

  AllowedIdentities = { PUBLIC };
  RunAsMode = CLIENT_IDENTITY;
  TransactionAttribute = TX_SUPPORTS;
}
```

log.sql

```
drop table log_table cascade constraints;

create table log_table (when date, which number, who number, what
varchar2(2000));
exit
```

employee/Employee.java

```
package employee;

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

import java.sql.*;

public interface Employee extends EJBObject
{
  public EmployeeInfo getEmployee (String name)
       throws RemoteException, SQLException;

  public EmployeeInfo getEmployeeForUpdate (String name)
       throws RemoteException, SQLException;

  public void updateEmployee (EmployeeInfo employee)
       throws RemoteException, SQLException;
}
```

employee/EmployeeHome.java

```
package employee;
```

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

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

employee/EmployeeInfo.java

```
package employee;

import java.rmi.*;

public class EmployeeInfo implements java.io.Serializable {
  public String name = null;
  public int number = 0;
  public double salary = 0;

  public EmployeeInfo (String name, int number, double salary) {
    this.name = name;
    this.number = number;
    this.salary = salary;
  }
}
```

employeeServer/EmployeeBean.sqlj

```
package employeeServer;

import employee.*;
import loggingServer.Logging;
import loggingServer.LoggingHome;

import javax.ejb.SessionBean;
import javax.ejb.CreateException;
import javax.ejb.SessionContext;
import javax.jts.UserTransaction;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;

import java.rmi.RemoteException;
import java.sql.SQLException;
import java.util.Hashtable;

public class EmployeeBean implements SessionBean
{
  SessionContext ctx;
  Logging logServer = null;

  // 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 EmployeeInfo getEmployeeForUpdate (String name)
       throws RemoteException, SQLException
  {
    int empno = 0;
    double salary = 0.0;
    logServer.log ("EJB: getEmployeeForUpdate (" + name + ")");
    #sql { select empno, sal into :empno, :salary from emp
                  where ename = :name for update };
    return new EmployeeInfo (name, empno, salary);
  }

  public void updateEmployee (EmployeeInfo employee)
       throws RemoteException, SQLException
  {
    logServer.log ("EJB: updateEmployee (" + employee.name + ")");
    #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
  {
    try {
      // create InitialContext
      Hashtable env = new Hashtable ();
      env.put (Context.URL_PKG_PREFIXES, "oracle.aurora.jndi");
      Context ic = new InitialContext (env);

      // Now, to create the loggingBean
      String objectName = new String ("/test/loggingService");
      LoggingHome logBean_home =
          (LoggingHome)ic.lookup ("sess_iiop://thisServer" + objectName);

      logServer = logBean_home.create ();
    } catch (NamingException e) {
      e.printStackTrace ();
    }

    try {
      logServer.log ("EJB: Create Employee");
    } catch (SQLException e) {
      e.printStackTrace ();
    }
  }
  public void ejbRemove () {}
  public void setSessionContext (SessionContext ctx) {
    this.ctx = ctx;
  }
  public void ejbActivate () {}
  public void ejbPassivate () {}
}
```

loggingServer/Logging.java

```
package loggingServer;

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

import java.sql.*;

public interface Logging extends EJBObject {
  public void log (String message) throws RemoteException, SQLException;
}
```

loggingServer/LoggingBean.sqlj

```
package loggingServer;
```

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

import java.sql.*;

import oracle.aurora.rdbms.DbmsJava;
import oracle.aurora.rdbms.Schema;

public class LoggingBean implements SessionBean {
  SessionContext ctx;

  public void log (String message) throws RemoteException, SQLException {
    int ownerNumber = Schema.currentSchema ().ownerNumber ();
    // System.out.println ("ownerNumber = " + ownerNumber);

    // get the session-id
    int sessID = DbmsJava.sessionID (DbmsJava.USER_SESSION);

    #sql { insert into log_table (who, which, when, what) values
                        (:ownerNumber, :sessID, sysdate, :message) };
  }

  public void ejbCreate () throws RemoteException, CreateException {}
  public void ejbRemove () {}
  public void setSessionContext (SessionContext ctxArg) {
    ctx = ctxArg;
  }
  public void ejbActivate () {}
  public void ejbPassivate () {}
}
```

loggingServer/LoggingHome.Java

```
package loggingServer;

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

public interface LoggingHome extends EJBHome {
  public Logging create () throws RemoteException, CreateException;
}
```

loggingServer/logBean.ejb

```
// This the generic database work bean template

SessionBean loggingServer.LoggingBean {
  BeanHomeName = "test/loggingService";
  RemoteInterfaceClassName = loggingServer.Logging;
  HomeInterfaceClassName = loggingServer.LoggingHome;

  TransactionAttribute = TX_REQUIRES_NEW;
  RunAsMode = CLIENT_IDENTITY;
  AllowedIdentities = { PUBLIC };

  EnvironmentProperties {
    prop1 = value1;
    prop2 = "value two";
  }
}
```

Session Examples

- timeout
- clientserverserver

timeout

readme.txt

```
Overview
========

The timeout example shows how you can control session timeout from an EJB.  A
first client program invokes a bean method to set the session timeout value,
and a second client program tests the timeout, by first calling a method on
the bean in the session within the timeout interval, and then after the
timeout has expired. In the second case, the method invocation should fail.

In order for the second client to be able to invoke a method on the same bean
in the same session, the first client saves both the bean handle and a login
object reference to disk, to be read by the scond client.

Source Files
============

Client1.java
-----------
You invoke the first client program from a command prompt, and pass it seven
arguments, which are the
   - service URL (service ID, hostname, port, and SID if port is a listener)
   - name of the published bean home interface
   - username
   - password that authenticates the client to the Oracle8i database server
   - a file name to which to write the login IOR
   - a file name to which to write the object handle
   - a time out value in seconds

For example:
% java -classpath LIBs Client1 sess_iiop://localhost:2481:ORCL \
     /test/myHello scott tiger login.dat handle.dat 30

where LIBs is the classpath that must include
$ORACLE_HOME/lib/aurora_client.jar
```

```
$ORACLE_HOME/jdbc/lib/classes111.zip
#If using Java 2, use classes12.zip instead of classes111.zip
$ORACLE_HOME/jlib/jssl-1_1.jar        # required if you modify any
$ORACLE_HOME/jlib/javax-ssl-1_1.jar   # client code to use SSL
$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())
   - gets a login server and authenticates the client
   - looks up and activates a Hello object
   - sets the object's message to "As created by Client1"
   - prints "Client1: " plus the massage, the
       message got by invoking helloWorld() on the Hello object
   - sets the session timeout by invoking setTimeout() on hello
   - writes the login IOR and the bean handle to files

The printed output is:

Client2.java
-----------
You invoke the second client program from a command prompt, and pass it four
arguments, the
   - username
   - password
   - a file name from which to read the login IOR, which must be the
        same as passed to Client1
   - a file name from which to read the object handle, the same as that
        passed to Client1

For example:
% java -classpath LIBs Client2 sess_iiop://localhost:2481:ORCL \
      scott tiger login.dat handle.dat

The client code performs the following steps:
   - reads the login object from the disk
   - reads the bean handle from disk
   -

hello.ejb
---------
The bean deployment descriptor.
```

```
helloServer/HelloBean.java
--------------------------
The bean implementation. Implements the methods helloWorld(), setMessage(),
and setTimeout(). Note that the call to Presentation.sessionTimeout() requires
that following import statement:

import oracle.aurora.net.Presentation;

hello/Hello.java
----------------
The bean remote interface.

hello/HelloHome.java
--------------------
The bean's 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.
```

hello.ejb

```
SessionBean helloServer.HelloBean
{
  BeanHomeName = "test/myHello";
  RemoteInterfaceClassName = hello.Hello;
  HomeInterfaceClassName = hello.HelloHome;

  SessionTimeout = 30;

  AllowedIdentities = { PUBLIC };
  RunAsMode = CLIENT_IDENTITY;
  TransactionAttribute = TX_NOT_SUPPORTED;
  // TransactionAttribute = TX_REQIRES_NEW;
  // TransactionAttribute = TX_BEAN_SUPPORTED;
}
```

Client1.java

```
import hello.Hello;
import hello.HelloHome;

import java.io.FileOutputStream;
import java.io.ObjectOutputStream;
import java.io.OutputStream;
import java.util.Hashtable;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.ejb.Handle;

import oracle.aurora.jndi.sess_iiop.ServiceCtx;
import oracle.aurora.client.*;
import oracle.aurora.AuroraServices.LoginServer;

import org.omg.CORBA.ORB;
import org.omg.CORBA.Object;
```

```
public class Client1
{
  // For ejb the session Timeout is set through deployment descriptor.
  // hence, no tmeout is taken as an input parameter
  public static void main (String[] args) throws Exception {
    if (args.length != 6) {
      System.out.println
        ("usage: Client serviceURL objectName username password " +
                   "loginIORfile objHandlefile");
      System.exit(1);
    }
    String serviceURL = args [0];
    String objectName = args [1];
    String username = args [2];
    String password = args [3];
    String loginIORfile = args [4];
    String objHandlefile = args [5];

    // get and set the initial Context
    Hashtable env = new Hashtable();
    env.put (Context.URL_PKG_PREFIXES, "oracle.aurora.jndi");
    Context ic = new InitialContext (env);

    // 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 loginIOR file for Client2 to read and get into the
    // same session
    writeIOR (lserver, loginIORfile);

    // Also, save the bean handle to a file for Client2 to access
    // our object in the session
    FileOutputStream fostream = new FileOutputStream (objHandlefile);
    ObjectOutputStream ostream = new ObjectOutputStream (fostream);
    ostream.writeObject (hello.getHandle ());
    ostream.flush ();
    fostream.close ();
    // exit... note the session is still running on the server (till
    // it times-out)
    System.out.println ("Client1: exiting...");
  }

  static public void writeIOR (Object object, String iorFile)
      throws Exception
  {
    ORB orb = ORB.init ();
    String ior = orb.object_to_string (object);
    OutputStream os = new FileOutputStream (iorFile);
    os.write (ior.getBytes ());
    os.close ();
  }
}
```

Client2.java

```
import hello.Hello;
import hello.HelloHome;

// java specific
import java.io.InputStream;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.ObjectInputStream;

import javax.ejb.Handle;

// oracle specific
import oracle.aurora.jndi.sess_iiop.ServiceCtx;
import oracle.aurora.client.Login;
import oracle.aurora.AuroraServices.LoginServer;
import oracle.aurora.AuroraServices.LoginServerHelper;

// CORBA specific
import org.omg.CORBA.ORB;
import org.omg.CORBA.BindOptions;
import org.omg.CORBA.COMM_FAILURE;

public class Client2
{
  public static void main (String[] args) throws Exception {
    boolean ssl = true;
```

```
if (args.length != 4) {
  System.out.println
      ("usage: Client2 username password loginIORfile objHandlefile");
  System.exit (1);
}
String username = args [0];
String password = args [1];
String loginIORfile = args [2];
String objHandlefile = args [3];

// Initialize the ORB for accessing objects in 8i
// You have to initialize the ORB that way.
// You will be authenticated using the login IOR read
// from the file.
ORB orb = ServiceCtx.init (null, null, null, false, null);

// Read the IORs from the IOR files... start with loginIOR
String loginIOR = getIOR (loginIORfile);

// Get a ref to the bean, by reading the file.
FileInputStream finstream = new FileInputStream (objHandlefile);
ObjectInputStream istream = new ObjectInputStream (finstream);
Handle helloHandle  = (Handle)istream.readObject ();
finstream.close ();
Hello hello = (Hello)helloHandle.getEJBObject ();
System.out.println ("Client2: read the bean handle from " +
            objHandlefile);

// Authenticate with the login Object
LoginServer lserver =
  LoginServerHelper.narrow (orb.string_to_object (loginIOR));
lserver._bind_options (new BindOptions (false, false));

try {
  Login login = new Login (lserver);
  login.authenticate (username, password, null);
} catch (COMM_FAILURE e1) {
  // this is thrown when the session times-out
  System.out.println ("Session Timed out as Expected");
  System.exit(0);
} catch (Exception e) {
  System.out.println("Login failed: " + e.getMessage());
  System.exit(1);
}
```

```
    // if Login passes then print client is authenticated...
    System.out.println ("Client2: authenticated.");

    // Access the object from the ior and print its message
    System.out.println ("Client2: " + hello.helloWorld ());

    // Disconnect from the object by exiting
    System.out.println ("Client2: exiting...");
  }

  // Read an IOR from an IOR file.
  static String getIOR (String iorFile) throws Exception
  {
    // Loop until the ior file is available
    InputStream is = null;
    int i;
    for (i = 0; i < 10; i++) {
      try {
          is = new FileInputStream (iorFile);
      } catch (FileNotFoundException e) {}
      Thread.sleep (1000);
    }

    if (is == null){
      System.out.println ("Client2 timed out before finding " + iorFile);
      System.exit (1);
    }

    byte[] iorbytes = new byte [is.available ()];
    is.read (iorbytes);
    is.close ();
    String ior = new String (iorbytes);
    System.out.println ("Client2: got the IOR from " + iorFile);
    return ior;
  }
}
```

hello/Hello.java

```
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 setTimeout (int seconds) throws RemoteException;
}
```

hello/HelloHome.java

```
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;
}
```

helloServer/HelloBean.java

```
package helloServer;

import hello.*;

import javax.ejb.SessionBean;
import javax.ejb.SessionContext;
import javax.ejb.CreateException;
import java.rmi.RemoteException;

import oracle.aurora.net.Presentation;

public class HelloBean implements SessionBean
{
  String message;

  // Methods of the Hello interface
  public String helloWorld () throws RemoteException {
    return message;
  }
  public void setMessage (String message) throws RemoteException {
    this.message = message;
  }

  public void setTimeout (int seconds) throws RemoteException {
    Presentation.sessionTimeout (seconds);
  }

  // Methods of the SessionBean
  public void ejbCreate () throws RemoteException, CreateException {}
  public void ejbRemove () {}
  public void setSessionContext (SessionContext ctx) {}
  public void ejbActivate () {}
  public void ejbPassivate () {}
}
```

clientserverserver

readme.txt

```
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
$ORACLE_HOME/jdbc/lib/classes111.zip
#If using Java 2, use classes12.zip instead of classes111.zip
$ORACLE_HOME/jlib/jssl-1_1.jar        # required if you modify any
$ORACLE_HOME/jlib/javax-ssl-1_1.jar   # client code to use SSL
$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 ());
  }
}
```

hello.ejb

```
SessionBean helloServer.HelloBean
{
  BeanHomeName = "test/myHello";
  RemoteInterfaceClassName = hello.Hello;
  HomeInterfaceClassName = hello.HelloHome;

  AllowedIdentities = { PUBLIC };
  RunAsMode = CLIENT_IDENTITY;
  TransactionAttribute = TX_NOT_SUPPORTED;
  // TransactionAttribute = TX_REQIRES_NEW;
  // TransactionAttribute = TX_BEAN_SUPPORTED;
}
```

hello/Hello.java

```
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;
}
```

hello/HelloHome.java

```
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;
}
```

helloServer/HelloBean.java

```
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 () {}
  }
```

C

Comparing the Oracle8*i* JServer and VisiBroker VBJ ORBs

This appendix, which is for developers who are familiar with the VisiBroker VBJ ORB, summarizes the main differences between that ORB and the current version of the Oracle8*i* JServer ORB. Each ORB supports multiple styles of usage, but this appendix compares only the most commonly used styles. In particular, it assumes that VBJ clients use the helper `bind()` method to find objects by name, whereas Oracle8*i* clients use the JNDI `lookup()` method for the same purpose. It also assumes that Oracle8*i* clients use Oracle's session IIOP to communicate with server objects, though the JServer ORB also supports the standard IIOP used by the VBJ ORB.

The differences in the ORBs are summarized in these sections:

- Object References Have Session Lifetimes
- The Database Server is the Implementation Mainline
- Server Object Implementations are Deployed by Loading and Publishing
- Implementation by Inheritance is Nearly Identical
- Implementation by Delegation is Different
- Clients Look Up Object Names with JNDI
- No Interface or Implementation Repository

At the end of the appendix, equivalent client and server implementations of the same IDL for the VBJ and Aurora ORBs are provided for comparison.

Object References Have Session Lifetimes

The Aurora ORB creates object instances in database *sessions*. When a session disappears, references to objects created in that session become invalid; attempts to use them incur the “object does not exist” exception. A session disappears when the last client connection to the session is closed or the session’s timeout value is reached. An object in a session can set the session timeout value with `oracle.aurora.net.Presentation.sessionTimeout()` optionally providing a client interface to this method, which a client can call if it wants an object to persist after client connections to the session are closed.

The life of a typical Oracle8*i* CORBA object proceeds as follows:

- A client looks up an object implementation’s name with JNDI specifying the database where the implementation has been published.
- The Oracle ORB responds by instantiating an object of the type, and returning a reference to the client.
- The client calls methods on the object, and may pass the reference to other clients who may then call methods on the object.
- The object ceases to exist when its session is destroyed.

The Database Server is the Implementation Mainline

An Oracle8*i* server object implementation consists of a single class. Developers do not write a mainline server because the database server is the mainline. If the database is running, all implementations published in that database are available to clients. The database server dynamically assigns MTS threads to implementations. An implementation may multithread its own execution with Java threads.

Server Object Implementations are Deployed by Loading and Publishing

Loading an object implementation into a database with the `loadjava` tool makes that implementation accessible to the ORB running in that database. Publishing an loaded implementation's name to a database's session name space with the `publish` tool makes the implementation accessible to clients by name. Every CORBA object implementation must be loaded but only those whose names will be looked up by clients need to be published.

Implementation by Inheritance is Nearly Identical

To implement the hypothetical interface `Alpha` in Oracle8*i*, you write a class called `AlphaImpl` which extends `AlphaImplBase` and defines the Java methods that implement the IDL operations. You *may* also provide instance initialization code in an `_initializeAuroraObject()` method which the Oracle ORB will call when it creates a new instance.

Implementation by Delegation is Different

For an Oracle8*i* implementation by delegation (tie), the class you write extends a class you have defined and implements two Oracle-defined interfaces. The first interface, whose name is the IDL interface name concatenated with `Operations`, defines the methods corresponding to the IDL operations. The second interface, called `ActivatableObject`, defines a single method called `_initializeAuroraObject()`. To implement this method, create and return an instance. Here is a minimal example:

```
// IDL
module hello {
  interface Hello {
    wstring helloWorld ();
  };
};

// Aurora tie implementation
package helloServer;

import hello.*;
import oracle.aurora.AuroraServices.ActivatableObject;

public class HelloImpl implements HelloOperations, ActivatableObject
//, extends <YourClass>
{
  public String helloWorld () {
    return "Hello World!";
  }

  public org.omg.CORBA.Object _initializeAuroraObject () {
    // create and initialize an instance and return it, for example ...
    return new _tie_Hello (this);
  }
}
```

Clients Look Up Object Names with JNDI

An Oracle8*i* client can look up a published object by name with CORBA COSNaming or with the simpler JNDI (Java Naming and Directory Interface) which interacts with COSNaming in the client's behalf.

A client creates an initial JNDI context for a particular database with a Java constructor, for example:

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

The `env` parameter specifies user name and password under which the client is logging in. Because object implementations run in database servers, CORBA object users (via their clients) must identify and authenticate themselves to the database as they would for any database operation.

To obtain an instance of a published implementation, the client calls the JNDI context's `lookup()` method, passing a URL that names the target database and the published name of the desired object implementation. The lookup() call returns a reference to an instance in the target database. A client may pass the reference (perhaps in stringified form) to other clients, and the reference will remain valid as long as the session in which the associated object was created survives. Clients that use copies of the same object reference share the object's database session.

If a client executes `lookup()` twice in succession with the same parameters, the second object reference is identical to the first, that is, it refers to the instance created by the first `lookup()` call. However, if a client creates a second session and does the second `lookup()` in that session, a different instance is created and its reference returned.

No Interface or Implementation Repository

The current version of the Oracle8*i* ORB does not include an interface repository or an implementation repository.

The Bank Example in Aurora and VBJ

The following sections compare implementations of the bank example widely used in VBJ documentation. Both client and server are shown as they would be implemented in Oracle8*i* and VBJ. All implementations use inheritance.

The Bank IDL Module

```
// Bank.idl

module Bank {
  interface Account {
    float balance();
  };
  interface AccountManager {
    Account open(in string name);
  };
};
```

Aurora Client

```
// Client.java

import bankServer.*;
import Bank.*;

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:2222";
    String objectName = "/test/myBank";
    String username = "scott";
    String password = "tiger";

    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);

    AccountManager manager =
      (AccountManager) ic.lookup(serviceURL + objectName);

    // use args[0] as the account name, or a default.
    String name = args.length == 1 ? args[0] : "Jack B. Quick";

    // Request the account manager to open a named account.
    Bank.Account account = manager.open(name);

    // Get the balance of the account.
    float balance = account.balance();

    // Print out the balance.
    System.out.println
      ("The balance in " + name + "‘s account is $" + balance);
  }
}
```

VBJ Client

```
// Client.java

public class Client {

  public static void main(String[] args) {
    // Initialize the ORB.
    org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args,null);
    // Locate an account manager.
    Bank.AccountManager manager =
      Bank.AccountManagerHelper.bind(orb, "BankManager");
    // use args[0] as the account name, or a default.
    String name = args.length > 0 ? args[0] : "Jack B. Quick";
    // Request the account manager to open a named account.
    Bank.Account account = manager.open(name);
      // Get the balance of the account.
    float balance = account.balance();
    // Print out the balance.
    System.out.println
      ("The balance in " + name + "‘s account is $" + balance);
  }
}
```

Aurora Account Implementation

```
// AccountImpl.java
package bankServer;

public class AccountImpl extends Bank._AccountImplBase {
  public AccountImpl(float balance) {
    _balance = balance;
  }
  public float balance() {
    return _balance;
  }
  private float _balance;
}
```

VBJ Account Implementation

```
// AccountImpl.java

public class AccountImpl extends Bank._AccountImplBase {
  public AccountImpl(float balance) {
    _balance = balance;
  }
  public float balance() {
    return _balance;
  }
  private float _balance;
}
```

Aurora Account Manager Implementation

```
// AccountManagerImpl.java
package bankServer;
```

```
import java.util.*;

public class AccountManagerImpl extends Bank._AccountManagerImplBase {

  public AccountManagerImpl() {
    super();
  }

  public AccountManagerImpl(String name) {
    super(name);
  }

  public synchronized Bank.Account open(String name) {
    // Lookup the account in the account dictionary.
    Bank.Account account = (Bank.Account) _accounts.get(name);
    // If there was no account in the dictionary, create one.
    if(account == null) {

      // Make up the account’s balance, between 0 and 1000 dollars.
      float balance = Math.abs(_random.nextInt()) % 100000 / 100f;

      // Create the account implementation, given the balance.
      account = new AccountImpl(balance);

      _orb().connect (account);

      // Print out the new account.
      // This just goes to the system trace file for Aurora.
      System.out.println("Created " + name + "‘s account: " + account);

      // Save the account in the account dictionary.
      _accounts.put(name, account);
    }
    // Return the account.
    return account;
  }

  private Dictionary _accounts = new Hashtable();
  private Random _random = new Random();

}
```

VBJ Account Manager Implementation

```
// AccountManagerImpl.java

import java.util.*;

public class AccountManagerImpl extends Bank._AccountManagerImplBase {
  public AccountManagerImpl(String name) {
    super(name);
  }
  public synchronized Bank.Account open(String name) {
    // Lookup the account in the account dictionary.
    Bank.Account account = (Bank.Account) _accounts.get(name);
    // If there was no account in the dictionary, create one.
    if(account == null) {
      // Make up the account's balance, between 0 and 1000 dollars.
      float balance = Math.abs(_random.nextInt()) % 100000 / 100f;
      // Create the account implementation, given the balance.
      account = new AccountImpl(balance);
      // Make the object available to the ORB.
      _boa().obj_is_ready(account);
      // Print out the new account.
      System.out.println("Created " + name + "'s account: " + account);
      // Save the account in the account dictionary.
      _accounts.put(name, account);
    }
    // Return the account.
    return account;
  }
  private Dictionary _accounts = new Hashtable();
  private Random _random = new Random();
}
```

VBJ Server Mainline

```
// Server.java


public class Server {

  public static void main(String[] args) {
    // Initialize the ORB.
    org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args,null);
    // Initialize the BOA.
    org.omg.CORBA.BOA boa = orb.BOA_init();
    // Create the account manager object.
    Bank.AccountManager manager =
      new AccountManagerImpl("BankManager");
    // Export the newly created object.
    boa.obj_is_ready(manager);
    System.out.println(manager + " is ready.");
    // Wait for incoming requests
    boa.impl_is_ready();
  }

}
```

D 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