JBoss™ 3.2 Workbook
for
Enterprise JavaBeans™, 3rd Edition

About the Series

Each of the books in this series is a server-specific companion to the third edition of Richard Monson-Haefel's best-selling and award-winning *Enterprise JavaBeans* (O'Reilly 2001), available at *http://www.oreilly.com/* and at all major retail outlets. It guides the reader step by step through the exercises called out in that work, explains how to build and deploy working solutions in a particular application server, and provides useful hints, tips, and warnings.

These workbooks provide serious developers with the best possible foundation for success in EJB development on their chosen platforms.

Series Titles Available

WebLogic™ Server 6.1 Workbook for *Enterprise JavaBeans™ 3rd Edition*

WebSphere™ 4.0 AEs Workbook for *Enterprise JavaBeans™ 3rd Edition*

JBoss™ 3.0 Workbook for *Enterprise JavaBeans™ 3rd Edition*

JBoss™ 3.2 Workbook for *Enterprise JavaBeans™, 3rd Edition*

Bill Burke and Sacha Labourey

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Paris • Sebastopol • Taipei • Tokyo

JBoss 3.2 Workbook for *Enterprise JavaBeans, 3rd Edition*, by Bill Burke and Sacha Labourey

Printed in the United States of America.

Published by O'Reilly & Associates, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

Companion volume to *Enterprise JavaBeans, 3rd Edition*, by Richard Monson-Haefel, published by O'Reilly & Associates, Inc., 2001.

O'Reilly & Associates books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (safari.oreilly.com). For more information contact our corporate/institutional sales department: (800) 998-9938 or corporate@oreilly.com.

Series Editor: Brian Christeson

Printing History:

May 2003: First Edition

Why didn't I take the blue pill?

Table of Contents

Table of Figures

Preface

This workbook is designed to be a companion for O'Reilly's *Enterprise JavaBeans, Third Edition*, by Richard Monson-Haefel, for users of JBoss™, an open-source J2EE™ application server. It is one of a series of workbooks that is being published by O'Reilly & Associates as an informative companion to that best-selling work.

The goal of this workbook is to provide the reader with step-by-step instructions for installing, configuring, and using JBoss and for deploying and running the examples from *Enterprise JavaBeans*.

This book is based on the production release of JBoss **3.2.0** and includes all the EJB 2.0 examples from the *Enterprise JavaBeans* book. All the examples in this workbook will work properly with JBoss 3.0.3 and **3.2.0** and above, but not with earlier versions of JBoss.

Contents of This Book

This workbook is divided into three sections:

- **Server Installation and Configuration** – This section will walk you through downloading, installing, and configuring JBoss. It will also provide a brief overview of the structure of the JBoss installation.
- **Exercises** – This section contains step-by-step instructions for downloading, building, and running the example programs in *Enterprise JavaBeans, Third Edition* (which, for brevity, this workbook will refer to as "the EJB book"). The text will also walk through the various deployment descriptors and source code to point out JBoss features and concerns.
- **Appendix** – This section provides useful information that did not fit neatly in the other sections: a collection of XML snippets for configuring a few popular JDBC drivers from various database vendors.

Because JBoss 3.2 is an EJB 2.0-compliant J2EE implementation, the EJB 1.1 exercises referred to in the EJB book are not included in this workbook.

The workbook text for each exercise depends on the amount of configuration required for the example program, but will generally also include the following information:

- Compiling and building the example code
- Deploying the EJB components to the application server
- Running the example programs and evaluating the results

The exercises were designed to be built and executed in order. Every effort was made to remove any dependencies between exercises by including all components each one needs in the directory

for that exercise, but some dependencies still exist. The workbook text will guide you through these where they arise.

Also note that this workbook is not intended to be a course on database configuration or design. The exercises have been designed to work out-of-the-box with the open-source database Hypersonic SQL, which is shipped with JBoss, and the application server creates all database tables automatically, at run time.

On-Line Resources

This workbook is designed for use with the EJB book and with downloadable example code, both available from our web site:

http://www.oreilly.com/catalog/entjbeans3/workbooks/index.html

We will post any errata here, and any updates required to support changes in specifications or products. This site also contains links to many popular EJB-related sites on the Internet.

We hope you find this workbook useful in your study of Enterprise JavaBeans and the JBoss open-source J2EE implementation. Comments, suggestions, and error reports on the text of this workbook or the downloaded example files are welcome and appreciated. Please post on the JBoss Forum:

http://www.jboss.org/forum.jsp?forum=152

To obtain more information about JBoss or the JBoss project please visit the project's web site:

http://www.jboss.org/

There you will find links to detailed JBoss documentation, on-line forums, and events happening in the JBoss community. You will also be able to obtain detailed information on JBoss training, support, and consulting services.

The JBossGroup™ has also produced books on JBoss and other J2EE standards, among them *JBoss Administration and Development*, by Marc Fleury and Scott Stark, and *JMX: Managing J2EE with Java Management Extensions*, by Marc Fleury and Juha Lindfors.

Conventions Used in This Book

Italics are used for:

- Filenames and pathnames
- Names of hosts, domains, and applications
- URLs and email addresses
- New terms where they are defined

Boldface is used for:

- Emphasis
- Buttons, menu items, window and menu names, and other UI items you are asked to interact with

`Constant width` is used for:

- Code examples and fragments
- Sample program output
- Class, variable, and method names, and Java keywords used within the text
- SQL commands, table names, and column names
- XML elements and tags
- Commands you are to type at a prompt

`Constant width bold` is used for emphasis in some code examples.

`Constant width italic` is used to indicate text that is replaceable. For example, in *`BeanName`*`PK`, you would replace *`BeanName`* with a specific bean name.

Side comments appear in a sans-serif font, with a bullet symbol suggesting the kind of comment:

- ❖ Hints and side observations
- ➢ Tips and tricks
- 💣 Cautions and warnings

An Enterprise JavaBean consists of many parts; it's not a single object, but a collection of objects and interfaces. To refer to an Enterprise JavaBean as a whole, we use the name of its business name in Roman type followed by "bean" or the acronym "EJB." For example, we will refer to the Customer EJB when we want to talk about the enterprise bean in general. If we put the name in constant width font, we are referring explicitly to the bean's class name, and usually to its remote interface. Thus `CustomerRemote` is the remote interface that defines the business methods of the Customer bean.

Acknowledgements

We would like to thank Marc Fleury, the founder of JBoss, for recommending us for this book and Richard Monson-Haefel for accepting the recommendation. We would also like to thank Greg Nyberg, the author of the WebLogic edition in this series of workbooks. The example programs he provided in his workbook were a great starting place for us and made our lives much easier.

Special thanks also go out to those who reviewed and critiqued this work: Dain Sundstrom and the rest of JBoss Group, Daniel Ruflé, and Thomas Laresch. We would like to publicly recognize

the series editor, Brian Christeson, for his courage in digging so deeply in this book and relentlessly hunting down our anglish misthakes (especially Sacha's Franco-British dialect).

Finally, Bill would like to thank his wife for putting up with all his whining and complaining, and Sacha promises Sophie that he will no longer use the writing of this workbook as an excuse for being late for any of their rendezvous.

Server Installation and Configuration

This chapter will guide you through the steps required to install a fully working JBoss server.

Along the way you will learn about JBoss 3.2's micro-kernel architecture, and the last section will show you how to install the code for the forthcoming exercises.

If at any time you need more detailed information about JBoss configuration, visit the JBoss web site, *http://www.jboss.org/*, where you will find comprehensive on-line documentation.

About JBoss

JBoss is a collaborative effort of a worldwide group of developers to create an open-source application server based on the Java 2 Platform, Enterprise Edition (J2EE). With more than a million copies downloaded in less than 12 months. JBoss is the leading J2EE application server.

JBoss implements the full J2EE stack of services:

- EJB (Enterprise JavaBeans)
- JMS (Java Message Service)
- JTS/JTA (Java Transaction Service/Java Transaction API)
- Servlets and JSP (JavaServer Pages)
- JNDI (Java Naming and Directory Interface)

It also provides advanced features such as clustering, JMX, web services, and IIOP (Internet Inter-ORB Protocol) integration.

Because JBoss code is licensed under the LGPL (GNU Lesser General Public License, see *http://www.gnu.org/copyleft/lesser.txt*), you can use it freely, at no cost, in any commercial application, or redistribute it as is.

Installing JBoss Application Server

Before going any further, make sure you have the J2SE JDK 1.3 or higher installed and correctly configured.

To download the JBoss binaries, go to the JBoss web site at *http://www.jboss.org/* and follow the **Downloads** link. There you will find all current binaries in both *zip* and *tar.gz* archive formats. Download the package that best meets your needs.

Extract the downloaded archive in the directory of your choice. Under Windows, you can use the *WinZip* utility to extract the archive content. Under Unix, you can use the following commands:

```
$ gunzip jboss-3.2.0.tar.gz
$ tar xf jboss-3.2.0.tar
```

Then change to the *$JBOSS_HOME/bin* directory and launch the run script that matches your OS:

Unix:

```
$ run.sh
```

Windows:

```
C:\jboss-3.2.0\bin>run.bat
```

That's it! You now have a fully working JBoss server!

Discovering the JBoss Directory Structure

Installing JBoss creates the following directory structure:

Figure 1: JBoss directory structure

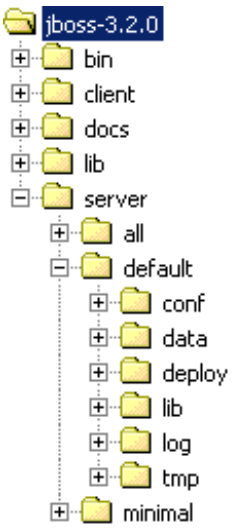


Table 1 discusses the purposes of the various directories:

Table 1: JBoss directories

Directory	Description
`bin`	Scripts to start and shut down JBoss.
`client`	Client-side Java libraries (JARs) required to communicate with JBoss.
`docs`	Sample configuration files (for database configuration, etc.)
`docs/dtd`	DTDs (Document Type Definitions) for the various XML files used in JBoss.
`lib`	JARs loaded at startup by JBoss and shared by all JBoss configurations. (You won't put your own libraries here.)
`server`	Various JBoss configurations. (Each configuration must be in a different sub-directory. The name of the sub-directory represents the name of the configuration. As distributed, JBoss contains three configurations: `minimal`, `default`, and `all`.)
`server/all`	JBoss's complete configuration; starts all services, including clustering and IIOP.
`server/minimal`	JBoss's minimal configuration; starts only very basic services; cannot be used to deploy EJBs.
`server/default`	JBoss's default configuration; used when no configuration name is specified on JBoss command line.
`server/default/conf`	JBoss's configuration files. (You will learn more about the content of this directory in the next section.)
`server/default/data`	JBoss's database files (embedded database or JBossMQ, for example).
`server/default/deploy`	JBoss's hot-deployment directory. (Any file or directory dropped in this directory is automatically deployed in JBoss: EJBs, WARs, EARs, and even services.)
`server/default/lib`	JARs that JBoss loads at startup when starting this particular configuration. (The `all` and `minimal` configurations also have this directory and the next two.)
`server/default/log`	JBoss's log files.
`server/default/tmp`	JBoss's temporary files.

If you want to define your own configuration, create a new sub-directory under the *server* directory containing the appropriate files. To start JBoss with a given configuration, use the `-c` parameter on the command line:

Windows:

```
C:\jboss-3.2.0\bin> run.bat -c config-name
```

Unix:

```
$ ./run.sh -c config-name
```

JBoss Configuration Files

As the previous section described, JBoss's *server* directory can contain any number of directories, each representing a different JBoss configuration.

The *server/config-name/conf* directory contains JBoss's configuration files. The purpose of the various files is discussed in Table 2.

Table 2: JBoss configuration files

File	Description
jacorb.properties	JBoss IIOP configuration
jbossmq-state.xml	JBossMQ (JMS implementation) user configuration
jboss-service.xml	Definition of JBoss's services launched at startup (class loaders, JNDI, deployers, etc.)
log4j.xml	Log4J logging configuration
login-config.xml	JBoss security configuration (JBossSX)
standardjaws.xml	Default configuration for JBoss's legacy CMP 1.1 engine; contains JDBC-to-SQL mapping information for various databases, default CMP settings, logging configuration, etc.
standardjboss.xml	Default container configuration
standardjbosscmp-jdbc.xml	Same as *standardjaws.xml* except that it is used for JBoss's CMP 2.0 engine.

Deployment in JBoss

The deployment process in JBoss is very straightforward. In each configuration, JBoss constantly scans a specific directory for changes: *$JBOSS_HOME/server/*config-name*/deploy*

This directory is generally referred to informally as "*the deploy directory*."

You can copy to this directory:

- Any JAR library (the classes it contains are automatically added to the JBoss classpath)
- An EJB JAR
- A WAR (Web Application aRrchive)
- An EAR (Enterprise Application aRchive)
- An XML file containing JBoss MBean definitions
- A directory ending in *.jar*, *.war*, or *.ear* and containing respectively the extracted content of an EJB JAR, a WAR, or an EAR.

To *redeploy* any of the above files (JAR, WAR, EAR, XML, etc.), simply overwrite it with a more recent version. JBoss will detect the change by comparing the files' timestamps, undeploy the previous files, and deploy their replacements. To redeploy a directory, update its modification timestamp by using a command-line utility such as *touch*.

To *undeploy* a file, just remove it from the deploy directory.

A Quick Look at JBoss Internals

Since version 3.0, JBoss has been built around a few very powerful concepts that allow users to customize and fine-tune their servers for very specific needs, not limited to J2EE. This flexibility allows JBoss to be used in very different environments, ranging from embedded systems to very large server clusters.

The next few sections will comment on some of these concepts briefly.

Micro-Kernel Architecture

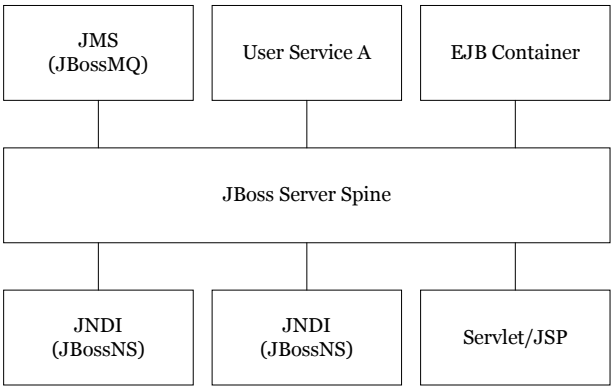
JBoss is based on a micro-kernel design in which components can be plugged at run time to extend its behavior.

This design fits particularly well with the J2EE platform, which is essentially a service-based platform. The platform contains services for persistence, transactions, security, naming, messaging, logging, and so on.

Other application servers are generally built as monolithic applications containing all services of the J2EE platform at all times. JBoss takes a radically different approach: Each of these services is hot-deployed as a component running on top of a very compact core, called the *JBoss Server Spine*. Furthermore, users are encouraged to implement their own services to run on top of JBoss.

- ❖ Consequently, the JBoss application server is not limited to J2EE applications, and indeed is frequently used to build any kind of application requiring a strong and reliable base. For this reason, the JBoss core is also known as the *WebOS*.

Figure 2: JBoss server spine with some hot-deployed services



JBoss Server Spine itself is based on Sun's *JMX* (*Java Management eXtensions*) specification, making any deployed component automatically manageable in a standard fashion. In the JMX terminology, a service deployed in JBoss is called an *MBean* (a *managed bean*).

- More information about the JMX specification can be found at the Sun web site *http://java.sun.com/products/JavaManagement/*

Hot Deployment

Since Release 2.0, JBoss has been famous for being the first J2EE-based application server to support hot deployment and redeployment of applications (EJB JAR, WAR, and EAR), while many application servers required a restart to update an application.

Thanks to its micro-kernel architecture and revolutionary Java class loader, JBoss 3.0 and later releases push this logic further. Not only can they hot-deploy and -redeploy applications, but they can hot-(re)deploy any service, and keep track of dependencies between services.

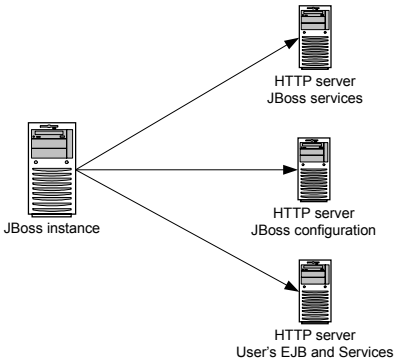
These features make JBoss usable in very demanding environments such as telecommunications systems.

Net Boot

JBoss is able to boot itself and your applications from any network location, just by pointing the JBoss Server Spine to a simple URL. This allows you to manage the entire configuration of a cluster of JBoss nodes from one central web server. This impressive flexibility makes deployment of new servers very easy.

- JBoss's bootstrap code is approximately 50K, which makes it suitable for many embedded systems.

Figure 3: A JBoss instance bootstrapping from three distinct netboot servers



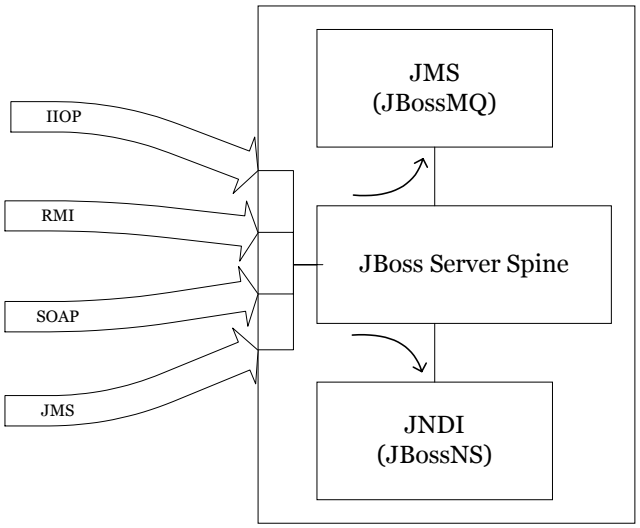
Detached Invokers

JBoss completely detaches the protocol handlers on which invocations are received from the target service that eventually serves the requests. Consequently, when a new handler (called an *invoker* in JBoss) for a given protocol is deployed in JBoss, all existing services and applications can automatically be reached through this new invocation transport.

JBoss 3.2 currently supports the following kinds of invokers:

- RMI
- RMI over HTTP
- IIOP
- JMS
- SOAP
- HA-RMI (Clustering over RMI)

Figure 4: Detached invokers



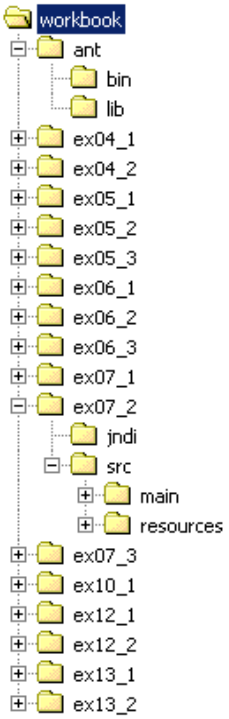
Exercise Code Setup and Configuration

You can download the example code for the exercises from http://www.oreilly.com/catalog/entjbeans3/workbooks/index.html. Exercises that require a database will use JBoss's default embedded database. Consequently, no additional database setup is required. Appendix A will show you how to configure JBoss to use a different database if you want to.

Exercises Directory Structure

The example code is organized as a set of directories, one for each exercise. You'll find the code of each exercise in the *src/main* sub-directory and the configuration files in *src/resources*.

Figure 5: Exercises directory structure



To build and run the exercises, you'll use the *Ant* tool. A *build.xml* is provided for each exercise. It contains the *Ant* configuration needed to compile the classes, build the EJB JAR, deploy it to JBoss, and run the client test applications. For this reason, the *Ant* tool is provided with the exercises and can be found in the *ant* directory.

- You can find out more about *Ant* at the Apache Jakarta web site http://jakarta.apache.org/ant/

Environment Setup

For the *Ant* scripts to work correctly, you first need to set some environment variables in the shells you will use to run the exercises:

- The `JAVA_HOME` environment variable must point to where your JDK is installed.
- The `JBOSS_HOME` environment variable must point to where JBoss is installed.
- The directory containing the *Ant* scripts must be in your path.

Depending on your platform, you'll have to execute commands like these:

Windows:

```
C:\workbook\ex04_1> set JAVA_HOME=C:\jdk1.3
C:\workbook\ex04_1> set JBOSS_HOME=C:\jboss-3.2.0
C:\workbook\ex04_1> set PATH=..\ant\bin;%PATH%
```

Unix:

```
$ export JAVA_HOME=/usr/local/jdk1.3
$ export JBOSS_HOME=/usr/local/jboss-3.2.0
$ export PATH=../ant/bin:$PATH
```

In each chapter you'll find detailed instructions on how to build, deploy, and run the exercises using *Ant*.

Exercises for Chapter 4

Exercise 4.1:
A Simple Entity Bean

The Cabin EJB demonstrates basic CMP 2.0 capability for a simple entity bean mapped to a single table. The following sections outline the steps necessary to build, deploy, and execute the Cabin EJB example. Please note that because you're using JBoss's default embedded database you don't need to configure the database or create tables. The code you'll see here mirrors the example code provided in Chapter 4 of the EJB book.

Start Up JBoss

Start up JBoss as described in the *Server Installation and Configuration* chapter at the beginning of this book.

Initialize the Database

The database table for this exercise will automatically be created in JBoss's default database, HypersonicSQL, when the EJB JAR is deployed.

Build and Deploy the Example Programs

Perform the following steps:

1. Open a command prompt or shell terminal and change to the *ex04_1* directory created by the extraction process.
2. Set the `JAVA_HOME` and `JBOSS_HOME` environment variables to point to where your JDK and JBoss 3.2 are installed. Examples:

Windows:

```
C:\workbook\ex04_1> set JAVA_HOME=C:\jdk1.3
C:\workbook\ex04_1> set JBOSS_HOME=C:\jboss-3.2.0
```

Unix:

```
$ export JAVA_HOME=/usr/local/jdk1.3
$ export JBOSS_HOME=/usr/local/jboss-3.2.0
```

3. Add `ant` to your execution path. *Ant* is the build utility

Windows:

```
C:\workbook\ex04_1> set PATH=..\ant\bin;%PATH%
```

Unix:

```
$ export PATH=../ant/bin:$PATH
```

4. Perform the build by typing `ant`. *Ant* uses *build.xml* to figure out what to compile and how to build your JARs.

If you need to learn more about the *Ant* utility, visit the Ant project at the Jakarta web site at http://jakarta.apache.org/ant/index.html.

Ant compiles the Java source code, builds the EJB JAR, and deploys the JAR simply by copying it to JBoss's *deploy* directory. If you are watching the JBoss console window, you will notice that JBoss automatically discovers the EJB JAR once it has been copied into the *deploy* directory, and automatically deploys the bean.

Another particularly interesting thing about building EJB JARs is that there is no special EJB compilation step. Unlike other servers, JBoss does not generate code for client stubs. Instead, it has a lightweight mechanism that creates client proxies when the EJB JAR is deployed, accelerating the development and deployment cycle.

Deconstructing build.xml

The *build.xml* file provided for each workbook exercise gives the *Ant* utility information about how to compile and deploy your Java programs and EJBs. The following build tasks can be executed by typing `ant taskname`:

- The default task (just typing `ant` without a task name) compiles the code, builds the EJB JAR, and deploys the JAR into JBoss. The deployment procedure is just a simple copy into the JBoss *deploy* directory.
- `ant compile` compiles all the Java source files.
- `ant clean` removes all *.class* and *.jar* files from the working directory, and undeploys the JAR from JBoss by deleting the file from JBoss's *deploy* directory.
- `ant clean.db` provides you with a clean copy of the HypersonicSQL database used throughout the exercises. This task works only with HypersonicSQL.

  `clean.db` can be used only when JBoss is not running.

- `run.client_xxx` runs a specific example program. Each exercise in this book will have a `run.client` rule for each example program.

Here's a breakdown of what is contained in *build.xml*.

```
<project name="JBoss" default="ejbjar" basedir=".">
```

The `default` attribute defines the default target that `ant` will run if you type only `ant` on the command line. The `basedir` attribute tells *Ant* what directory to run the build in.

```
<property environment="env"/>
<property name="src.dir" value="${basedir}/src/main"/>
<property name="src.resources" value="${basedir}/src/resources"/>
<property name="jboss.home" value="${env.JBOSS_HOME}"/>
<property name="build.dir" value="${basedir}/build"/>
<property name="build.classes.dir" value="${build.dir}/classes"/>
```

All the properties defined above are variables that *Ant* will use throughout the build process. You can see that the `JBOSS_HOME` environment variable is pulled from the system environment and other directory paths defined.

```
<path id="classpath">
      <fileset dir="${jboss.home}/client">
          <include name="**/*.jar"/>
      </fileset>
 <pathelement location="${build.classes.dir}"/>
 <pathelement location="${basedir}/jndi"/>
</path>
```

To compile and run the example applications in this workbook you need to add all the JARS in *$JBOSS_HOME/client* to the Java classpath. Also notice that *build.xml* inserts the *${basedir}/jndi* directory into the classpath. A *jndi.properties* file in this directory enables the example programs to find and connect to JBoss's JNDI server.

```
<property name="build.classpath" refid="classpath"/>

<target name="prepare" >
  <mkdir dir="${build.dir}"/>
  <mkdir dir="${build.classes.dir}"/>
</target>
```

The `prepare` target creates the directories where the Java compiler will place compiled classes.

```
<target name="compile" depends="prepare">
  <javac srcdir="${src.dir}"
         destdir="${build.classes.dir}"
         debug="on"
         deprecation="on"
         optimize="off"
         includes="**">
          <classpath refid="classpath"/>
  </javac>
</target>
```

The `compile` target will compile all the Java files under the *src/main* directory. Notice that it depends on the `prepare` target; `prepare` will run before the `compile` target is executed.

```
  <target name="ejbjar" depends="compile">
    <jar jarfile="build/titan.jar">
      <fileset dir="${build.classes.dir}">
           <include name="com/titan/cabin/*.class"/>
      </fileset>
      <fileset dir="${src.resources}/">
          <include name="**/*.xml"/>
      </fileset>
     </jar>
     <copy file="build/titan.jar"
           todir="${jboss.home}/server/default/deploy"/>
  </target>
```

The `ejbjar` target creates the EJB JAR file and deploys it to JBoss simply by copying it to JBoss's *deploy* directory.

```
  <target name="run.client_41a" depends="ejbjar">
    <java classname="com.titan.clients.Client_1" fork="yes" dir=".">
      <classpath refid="classpath"/>
    </java>
  </target>

  <target name="run.client_41b" depends="ejbjar">
    <java classname="com.titan.clients.Client_2" fork="yes" dir=".">
      <classpath refid="classpath"/>
    </java>
  </target>
```

The `run.client_xxx` targets are used to run the example programs in this chapter.

```
  <target name="clean.db">
    <delete dir="${jboss.home}/server/default/db/hypersonic"/>
  </target>
```

The `clean.db` target cleans the default database used by JBoss for the example programs in this book. Remember you can use it only when JBoss is not running.

```
  <target name="clean">
    <delete dir="${build.dir}"/>
    <delete file="${jboss.home}/server/default/deploy/titan.jar"/>
  </target>
</project>
```

The `clean` target removes compiled classes and undeploys the EJB JAR from JBoss by deleting the JAR file in the *deploy* directory.

Examine the JBoss-Specific Files

You do not need any JBoss-specific files to write a simple EJB. For an entity bean as simple as the Cabin EJB, JBoss creates the appropriate database tables within its embedded database Hypersonic SQL by examining the *ejb-jar.xml* deployment descriptor.

- In later chapters you will learn how to map entity beans to different data sources and pre-existing database tables using JBoss-specific CMP deployment descriptors.

By default, JBoss uses the `<ejb-name>` from the bean's *ejb-jar.xml* deployment descriptor for the JNDI binding of the bean's home interface. If you do not like this default, you can override it in a *jboss.xml* file. Clients use this name to look up an EJB's home interface. For this example, `CabinEJB` is bound to `CabinHomeRemote`.

jboss.xml

```
<jboss>
  <enterprise-beans>
    <entity>
      <ejb-name>CabinEJB</ejb-name>
      <jndi-name>CabinHomeRemote</jndi-name>
    </entity>
  </enterprise-beans>
</jboss>
```

Examine and Run the Client Applications

Two example programs implement the sample clients provided in the EJB book:

- *Client_1.java* creates a single Cabin bean, populates each of its attributes, then queries the created bean with the primary key.
- *Client _2.java* creates 99 additional Cabins with a variety of different data that will be used in subsequent exercises.

Client_1.java

```
package com.titan.clients;

import com.titan.cabin.CabinHomeRemote;
import com.titan.cabin.CabinRemote;

import javax.naming.InitialContext;
import javax.naming.Context;
import javax.naming.NamingException;
import javax.rmi.PortableRemoteObject;
import java.rmi.RemoteException;
```

```
public class Client_1
{
   public static void main(String [] args)
   {
      try
      {
         Context jndiContext = getInitialContext();
         Object ref = jndiContext.lookup("CabinHomeRemote");
         CabinHomeRemote home = (CabinHomeRemote)
            PortableRemoteObject.narrow(ref,CabinHomeRemote.class);
         CabinRemote cabin_1 = home.create(new Integer(1));
         cabin_1.setName("Master Suite");
         cabin_1.setDeckLevel(1);
         cabin_1.setShipId(1);
         cabin_1.setBedCount(3);

         Integer pk = new Integer(1);

         CabinRemote cabin_2 = home.findByPrimaryKey(pk);
         System.out.println(cabin_2.getName());
         System.out.println(cabin_2.getDeckLevel());
         System.out.println(cabin_2.getShipId());
         System.out.println(cabin_2.getBedCount());

      }
      catch (java.rmi.RemoteException re){re.printStackTrace();}
      catch (javax.naming.NamingException ne){ne.printStackTrace();}
      catch (javax.ejb.CreateException ce){ce.printStackTrace();}
      catch (javax.ejb.FinderException fe){fe.printStackTrace();}
   }

   public static Context getInitialContext()
      throws javax.naming.NamingException
   {
      return new InitialContext();
   }
}
```

The `getInitialContext()` method creates an `InitialContext` with no properties. Because no properties are set, the Java library that implements `InitialContext` searches the classpath for the file *jndi.properties*. Each example program in this workbook will have a *jndi* directory that contains a *jndi.properties* file. You will be executing all example programs through *Ant*, and it will set the classpath appropriately to refer to this properties file.

Run the *Client_1* application by invoking `ant run.client_41a` at the command prompt. Remember to set your `JBOSS_HOME` and `PATH` environment variables.

The output of *Client_1* should look something like this:

```
C:\workbook\ex04_1>ant run.client_41a
Buildfile: build.xml

prepare:

compile:

ejbjar:

run.client_41a:
     [java] Master Suite
     [java] 1
     [java] 1
     [java] 3
```

Client_1 adds a row to the database representing the Cabin bean and does not delete it at the conclusion of the program. You cannot run this program more than once unless you stop JBoss, clean the database by invoking the *Ant* task `clean.db`, and restarting JBoss. Otherwise, you will get the following error (we've tidied up the line wrapping here):

```
run.client_41a:
     [java] javax.ejb.DuplicateKeyException: Entity with primary key
            1 already exists
     [java] at
            org.jboss.ejb.plugins.cmp.jdbc.JDBCCreateEntityCommand.
            execute(JDBCCreateEntityCommand.java:160)
     [java] at
            org.jboss.ejb.plugins.cmp.jdbc.JDBCStoreManager.
            createEntity(JDBCStoreManager.java:633)
     [java] at
            org.jboss.ejb.plugins.CMPPersistenceManager.
            createEntity(CMPPersistenceManager.java:253)
     [java] at org.jboss.resource.connectionmanager.
            CachedConnectionInterceptor.createEntity(
            CachedConnectionInterce...
     [java] at org.jboss.invocation.InvokerInterceptor.
            invoke(InvokerInterceptor.java:92)
     [java] at org.jboss.proxy.TransactionInterceptor.
            invoke(TransactionInterceptor.java:77)
     [java] at
            org.jboss.proxy.SecurityInterceptor
           .invoke(SecurityInterceptor.java:80)
```

```
     [java] at
            org.jboss.proxy.ejb.HomeInterceptor.
            invoke(HomeInterceptor.java:175)
     [java] at org.jboss.proxy.ClientContainer.
            invoke(ClientContainer.java:82)
     [java] at $Proxy0.create(Unknown Source)
     [java] at com.titan.clients.Client_1.main(Client_1.java:22)
```

Run the *Client_2* application by invoking `ant run.client_41b` at the command prompt. Remember to set your `JBOSS_HOME` and `PATH` environment variables.

The output of *Client_2* should look something like this:

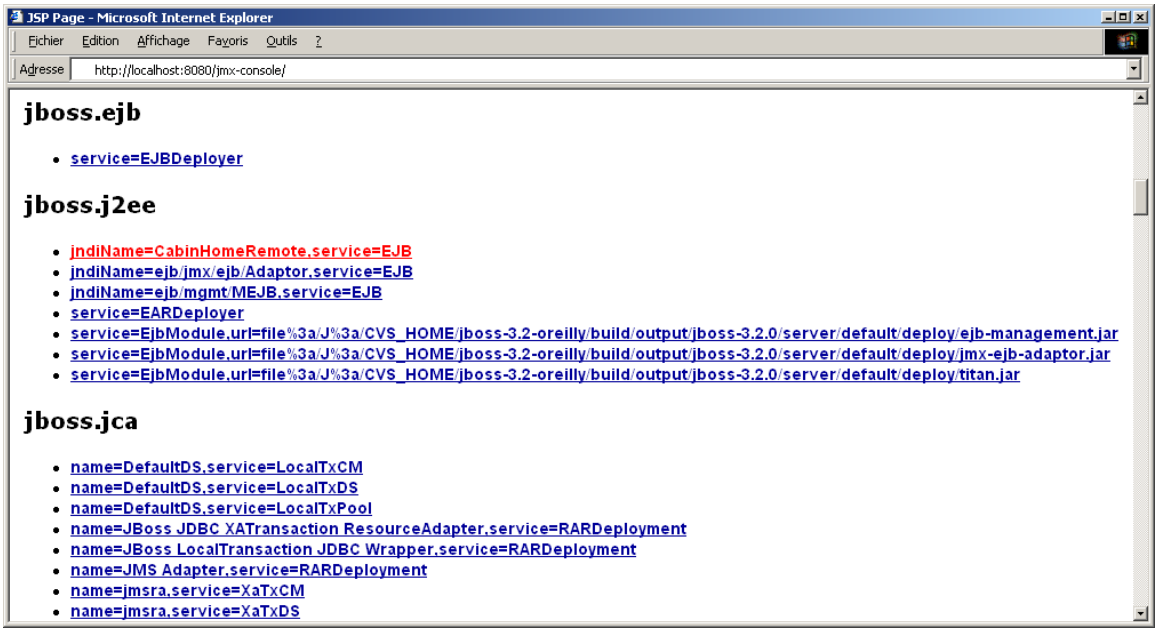
```
run.client_41b:
     [java] PK=1, Ship=1, Deck=1, BedCount=3, Name=Master Suite
     [java] PK=2, Ship=1, Deck=1, BedCount=2, Name=Suite 100
     [java] PK=3, Ship=1, Deck=1, BedCount=3, Name=Suite 101
     [java] PK=4, Ship=1, Deck=1, BedCount=2, Name=Suite 102
     [java] PK=5, Ship=1, Deck=1, BedCount=3, Name=Suite 103
     [java] PK=6, Ship=1, Deck=1, BedCount=2, Name=Suite 104
     [java] PK=7, Ship=1, Deck=1, BedCount=3, Name=Suite 105
     [java] PK=8, Ship=1, Deck=1, BedCount=2, Name=Suite 106
     ...
     [java] PK=90, Ship=3, Deck=3, BedCount=3, Name=Suite 309
     [java] PK=91, Ship=3, Deck=4, BedCount=2, Name=Suite 400
     [java] PK=92, Ship=3, Deck=4, BedCount=3, Name=Suite 401
     [java] PK=93, Ship=3, Deck=4, BedCount=2, Name=Suite 402
     [java] PK=94, Ship=3, Deck=4, BedCount=3, Name=Suite 403
     [java] PK=95, Ship=3, Deck=4, BedCount=2, Name=Suite 404
     [java] PK=96, Ship=3, Deck=4, BedCount=3, Name=Suite 405
     [java] PK=97, Ship=3, Deck=4, BedCount=2, Name=Suite 406
     [java] PK=98, Ship=3, Deck=4, BedCount=3, Name=Suite 407
     [java] PK=99, Ship=3, Deck=4, BedCount=2, Name=Suite 408
     [java] PK=100, Ship=3, Deck=4, BedCount=3, Name=Suite 409
```

Like *Client_1*, this example creates rows in the database and does not delete them when it finishes. *Client_2* can be executed only once without causing `DuplicateKey` exceptions.

Managing Entity Beans

Every EJB in JBoss is deployed and managed as a JMX MBean. You can view and manage EJBs deployed within JBoss through your web browser by accessing the JMX management console available at *http://localhost:8080/jmx-console/*.

Figure 6: The JMX management console



Click on the **jndiName=CabinHomeRemote,service=EJB** link shown above. Entity beans have two management functions. You can flush the entity bean's cache or view the number of cached objects for it. To flush, click on the **flushCache** button. To view the number of cached beans, click on the **getCacheSize** button.

Figure 7: Managing entity beans from the console

Exercise 4.2:
A Simple Session Bean

In this exercise you will create and build the TravelAgent EJB. This simple bean illustrates the use of a stateless session bean and mirrors the code shown in Chapter 4 of the EJB book.

Start Up JBoss

If you already have JBoss running there is no reason to restart it. Otherwise start it up as instructed in the *Server Installation and Configuration* chapter.

Initialize the Database

The database should contain the 100 rows created by a successful execution of the test programs from the previous exercise, *Client_1* and *Client_2*.

Build and Deploy the Example Programs

Perform the following steps:

1. Open a command prompt or shell terminal and change to the *ex04_2* directory created by the extraction process
2. Set the `JAVA_HOME` and `JBOSS_HOME` environment variables to point to where your JDK and JBoss 3.2 are installed. Examples:

Windows:

```
C:\workbook\ex04_2> set JAVA_HOME=C:\jdk1.3
C:\workbook\ex04_2> set JBOSS_HOME=C:\jboss-3.2.0
```

Unix:

```
$ export JAVA_HOME=/usr/local/jdk1.3
$ export JBOSS_HOME=/usr/local/jboss-3.2.0
```

3. Add `ant` to your execution path. *Ant* is the build utility.

Windows:

```
C:\workbook\ex04_2> set PATH=..\ant\bin;%PATH%
```

Unix:

```
$ export PATH=../ant/bin:$PATH
```

4. Perform the build by typing `ant`.

As in the last exercise, you will see *titan.jar* rebuilt, copied to the JBoss *deploy* directory, and redeployed by the application server.

Examine the JBoss-Specific Files

In this example, the *jboss.xml* deployment descriptor overrides the default JNDI binding for the deployed EJBs. `CabinEJB` is bound to `CabinHomeRemote` and `TravelAgentEJB` is bound to `TravelAgentHomeRemote`.

jboss.xml

```
<jboss>
  <enterprise-beans>
    <entity>
      <ejb-name>CabinEJB</ejb-name>
      <jndi-name>CabinHomeRemote</jndi-name>
    </entity>
    <session>
      <ejb-name>TravelAgentEJB</ejb-name>
      <jndi-name>TravelAgentHomeRemote</jndi-name>
      <ejb-ref>
        <ejb-ref-name>ejb/CabinHomeRemote</ejb-ref-name>
        <jndi-name>CabinHomeRemote</jndi-name>
      </ejb-ref>
    </session>
  </enterprise-beans>
</jboss>
```

The EJB book describes how you must use `<ejb-ref>` declarations when one EJB references another. The TravelAgent EJB references the Cabin entity bean, so the following XML is required in *ejb-jar.xml*.

ejb-jar.xml

```
    <ejb-ref>
      <ejb-ref-name>ejb/CabinHomeRemote</ejb-ref-name>
      <ejb-ref-type>Entity</ejb-ref-type>
      <home>com.titan.cabin.CabinHomeRemote</home>
      <remote>com.titan.cabin.CabinRemote</remote>
    </ejb-ref>
```

If you have a `<ejb-ref-name>` declared in your *ejb-jar.xml* file, you must have a corresponding `<ejb-ref>` declaration in your *jboss.xml* file that maps the portable JNDI name used by the TravelAgent EJB to the real JNDI name of the Cabin EJB.

jboss.xml

```
<jboss>
  <enterprise-beans>
    <entity>
      <ejb-name>CabinEJB</ejb-name>
      <jndi-name>CabinHomeRemote</jndi-name>
    </entity>
    <session>
      <ejb-name>TravelAgentEJB</ejb-name>
      <jndi-name>TravelAgentHomeRemote</jndi-name>
      <ejb-ref>
        <ejb-ref-name>ejb/CabinHomeRemote</ejb-ref-name>
        <jndi-name>CabinHomeRemote</jndi-name>
      </ejb-ref>
    </session>
  </enterprise-beans>
</jboss>
```

Examine and Run the Client Application

The example program in this section invokes the TravelAgent EJB to list cabins that meet certain criteria.

Client_3.java

```
...
Context jndiContext = getInitialContext();

Object ref = jndiContext.lookup("TravelAgentHomeRemote");
TravelAgentHomeRemote home = (TravelAgentHomeRemote)
PortableRemoteObject.narrow(ref,TravelAgentHomeRemote.class);

TravelAgentRemote travelAgent = home.create();

// Get a list of all cabins on ship 1 with a bed count of 3.
String list [] = travelAgent.listCabins(SHIP_ID,BED_COUNT);

for(int i = 0; i < list.length; i++)
{
   System.out.println(list[i]);
}
...
```

The client code does a JNDI lookup for the TravelAgent home and does a simple `create()` method invocation to obtain a reference to a TravelAgent EJB. The client then calls `listCabins()` and receives a list of cabin names that meet the provided criteria.

Let's examine a little of the code in TravelAgent EJB's `listCabins()` method to see how it works.

TravelAgentBean.java

```
public String [] listCabins(int shipID, int bedCount)
{
   try
   {
      javax.naming.Context jndiContext = new InitialContext();
      Object obj =
            jndiContext.lookup("java:comp/env/ejb/CabinHomeRemote");


      CabinHomeRemote home =
   ...
```

When a deployed EJB in JBoss wants to access JNDI all that's needed is a simple `new InitialContext()`. JBoss will automatically create an optimized, in-process reference to the JNDI server running inside the application server, to avoid the overhead of a distributed network call when accessing it. The rest of `listCabins()` is pretty straightforward, so you can just go on to running the client application.

Run the *Client_3* application by invoking `ant run.client_42` at the command prompt. Remember to set your `JBOSS_HOME` and `PATH` environment variables.

The output of *Client_3* should look something like this:

```
C:\workbook\ex04_2>ant run.client_42
Buildfile: build.xml

prepare:

compile:

ejbjar:

run.client_42:
     [java] 1,Master Suite,1
     [java] 3,Suite 101,1
     [java] 5,Suite 103,1
     [java] 7,Suite 105,1
     [java] 9,Suite 107,1
     [java] 12,Suite 201,2
     [java] 14,Suite 203,2
     [java] 16,Suite 205,2
     [java] 18,Suite 207,2
     [java] 20,Suite 209,2
     [java] 22,Suite 301,3
     [java] 24,Suite 303,3
     [java] 26,Suite 305,3
     [java] 28,Suite 307,3
     [java] 30,Suite 309,3
```

Exercises for Chapter 5

Exercise 5.1:
The Remote Component Interfaces

The example programs in Exercise 5.1 dive into some of the features of the home interface of an EJB, including the use of the `remove()` method. They also show you how to obtain and use various metadata available through an EJB's API.

Start Up JBoss

If you already have JBoss running there is no reason to restart it. Otherwise, start it up as instructed in the *Server Installation and Configuration* chapter at the beginning of this book.

Initialize the Database

The database should contain the 100 rows created by a successful execution of the test programs from Exercise 4.1.

Build and Deploy the Example Programs

Perform the following steps:

1. Open a command prompt or shell terminal and change to the *ex05_1* directory created by the extraction process.
2. Set the `JAVA_HOME` and `JBOSS_HOME` environment variables to point to where your JDK and JBoss 3.2 are installed. Examples:

Windows:

```
C:\workbook\ex05_1> set JAVA_HOME=C:\jdk1.3
C:\workbook\ex05_1> set JBOSS_HOME=C:\jboss-3.2.0
```

Unix:

```
$ export JAVA_HOME=/usr/local/jdk1.3
$ export JBOSS_HOME=/usr/local/jboss-3.2.0
```

3. Add `ant` to your execution path.

Windows:

```
C:\workbook\ex05_1> set PATH=..\ant\bin;%PATH%
```

Unix:

```
$ export PATH=../ant/bin:$PATH
```

4. Perform the build by typing `ant`.

As in the last exercise, you will see *titan.jar* rebuilt, copied to the JBoss *deploy* directory, and redeployed by the application server.

Examine the JBoss-Specific Files

There are no new JBoss configuration files or components in this exercise.

Examine and Run the Client Applications

Two example programs illustrate the concepts explained in the EJB book:

- *Client_51a.java* illustrates the use of the `remove()` method on the Cabin EJB home interface.
- *Client _51b.java* illustrates the use of bean metadata methods.

The example code for *Client_51a* and *Client_51b* is pulled directly from the EJB book. There is no need to go into this code here because the EJB book already does a very good job of that.

Run *Client_51a* by invoking `ant run.client_51a` at the command prompt. Remember to set your `JBOSS_HOME` and `PATH` environment variables. Run *Client_51b* the same way: `ant run.client_51b`.

The output of *Client_51a* should be exactly as described in the EJB book. The output of Client_51b is as follows:

```
C:\workbook\ex05_1>ant run.client_51b
Buildfile: build.xml

prepare:

compile:

run.client_51b:
     [java] com.titan.cabin.CabinHomeRemote
     [java] com.titan.cabin.CabinRemote
     [java] java.lang.Integer
     [java] false
     [java] Master Suite
```

Note that if you try to run *Client_51a* more than once an exception will tell you that the entity you're attempting to remove does not exist.

```
[java] java.rmi.NoSuchObjectException: Entity not found:
primaryKey=30
```

Exercise 5.2:
The EJBObject, Handle, and Primary Key

The example programs in Exercise 5.2 explore the APIs available through the `EJBObject` and `EJBMetaData` interfaces. They also reveal how to use `Handle` and `HomeHandle` as persistent references to EJB objects and homes.

Start Up JBoss

If you already have JBoss running there is no reason to restart it. Otherwise start it up as instructed in the *Server Installation and Configuration* chapter at the beginning of this book.

Initialize the Database

The database should contain the 100 rows created by a successful execution of the test programs from Exercise 4.1, otherwise this example will not work properly.

Build and Deploy the Example Programs

In the *ex05_2* directory, build and deploy the examples as you did for Exercise 5.1.

Examine the JBoss-Specific Files

There are no new JBoss configuration files or components in this exercise.

Examine and Run the Client Applications

Three example programs illustrate the concepts explained in the EJB book:

- *Client_52a.java* shows the use of `EJBObject` to retrieve an EJB's home interface.
- *Client_52b.java* shows you how to use `isIdentical()` to determine whether two EJB references are to the same object.
- *Client_52c.java* shows you how to use EJB handles as persistent bean references.

The example code is pulled directly from the EJB book and embellished somewhat to expand on introduced concepts. The EJB book does a pretty good job of explaining the concepts illustrated in the example programs, so further explanation of the code is not needed in this workbook.

Run *Client_52a*, *Client_52b*, and *Client_52c* by invoking the appropriate *Ant* task as you have done in previous examples: `run.client_52a`, `run.client_52b`, and `run.client_52c`. Remember to set your `JBOSS_HOME` and `PATH` environment variables.

Exercise 5.3:
The Local Component Interfaces

The example program in Exercise 5.3 explores the use of local interfaces. The Cabin entity bean you created in Exercise 4.1 will be expanded to provide a local interface for use in the TravelAgent stateless session bean. This exercise will also describe how to modify your EJB deployment descriptors to enable local interfaces.

Start Up JBoss

If you already have JBoss running there is no reason to restart it. Otherwise, start it up as instructed in the *Server Installation and Configuration* chapter.

Initialize the Database

The database should contain the 100 rows created by a successful execution of the test programs from Exercise 4.1.

Build and Deploy the Example Programs

In the *ex05_3* directory, build and deploy the examples as you did for Exercise 5.1.

Examine the JBoss-Specific Files

JBoss has a minor restriction. It requires that you use `<ejb-link>` when you want your bean to reference a local bean through an `<ejb-local-ref>` tag.

ejb-jar.xml

```
<ejb-jar>
 <enterprise-beans>
    ...
    <session>
     <ejb-name>TravelAgentEJB</ejb-name>
     <home>com.titan.travelagent.TravelAgentHomeRemote</home>
     <remote>com.titan.travelagent.TravelAgentRemote</remote>
     <ejb-class>com.titan.travelagent.TravelAgentBean</ejb-class>
     <session-type>Stateless</session-type>
     <transaction-type>Container</transaction-type>

     <ejb-local-ref>
      <ejb-ref-name>ejb/CabinHomeLocal</ejb-ref-name>
      <ejb-ref-type>Entity</ejb-ref-type>
      <local-home>com.titan.cabin.CabinHomeLocal</local-home>
      <local>com.titan.cabin.CabinLocal</local>
      <!-- ejb-link is required by jboss for local-refs. -->
      <ejb-link>CabinEJB</ejb-link>
     </ejb-local-ref>
     ...
</ejb-jar>
```

If you examine the *jboss.xml* file for Exercise 5.3, you'll see that you must also declare the JNDI binding for the remote home interface. The Cabin EJB's local home interface doesn't need a binding in *jboss.xml*, though, because the binding information is contained in the `<ejb-link>` tag instead. JBoss will register both `CabinHomeRemote` and `CabinHomeLocal` into the JNDI tree.

jboss.xml

```
<jboss>
  <enterprise-beans>
    <entity>
      <ejb-name>CabinEJB</ejb-name>
      <jndi-name>CabinHomeRemote</jndi-name>
      <local-jndi-name>CabinHomeLocal</local-jndi-name>
    </entity>
```

`TravelAgentEJB` only tells JBoss under which JNDI name it should be bound.

```
    <session>
      <ejb-name>TravelAgentEJB</ejb-name>
      <jndi-name>TravelAgentHomeRemote</jndi-name>
    </entity>
```

```
  </enterprise-beans>
</jboss>
```

Examine and Run the Client Applications

The example code for *Client_53* is exactly the same as *Client_3* from Exercise 4.2.

Run *Client_53* by invoking the appropriate *Ant* task as in previous examples: `run.client_53`. Remember to set your `JBOSS_HOME` and `PATH` environment variables.

The output should look something like this:

```
C:\workbook\ex05_3>ant run.client_53
Buildfile: build.xml

prepare:

compile:

ejbjar:

run.client_53:
     [java] 1,Master Suite,1
     [java] 3,Suite 101,1
     [java] 5,Suite 103,1
     [java] 7,Suite 105,1
     [java] 9,Suite 107,1
     [java] 12,Suite 201,2
     [java] 14,Suite 203,2
     [java] 16,Suite 205,2
     [java] 18,Suite 207,2
     [java] 20,Suite 209,2
     [java] 22,Suite 301,3
     [java] 24,Suite 303,3
     [java] 26,Suite 305,3
     [java] 28,Suite 307,3
```

Exercises for Chapter 6

Exercise 6.1:
Basic Persistence in CMP 2.0

This exercise begins walking you through the intricacies of CMP 2.0. In this chapter you will learn more detailed JBoss CMP 2.0 configuration mechanisms by creating the Customer EJB described in the EJB book.

Start Up JBoss

If you already have JBoss running there is no reason to restart it. Otherwise, start it up as instructed in the *Server Installation and Configuration* chapter.

Initialize the Database

The database table for this exercise will automatically be created in JBoss's default database, HypersonicSQL, when the EJB JAR is deployed.

Build and Deploy the Example Programs

Perform the following steps:

1. Open a command prompt or shell terminal and change to the *ex06_1* directory created by the extraction process
2. Set the `JAVA_HOME` and `JBOSS_HOME` environment variables to point to where your JDK and JBoss 3.2 are installed. Examples:

Windows:

```
C:\workbook\ex06_1> set JAVA_HOME=C:\jdk1.3
C:\workbook\ex06_1> set JBOSS_HOME=C:\jboss-3.2.0
```

Unix:

```
$ export JAVA_HOME=/usr/local/jdk1.3
$ export JBOSS_HOME=/usr/local/jboss-3.2.0
```

3. Add `ant` to your execution path.

Windows:

```
C:\workbook\ex06_1> set PATH=..\ant\bin;%PATH%
```

Unix:

```
$ export PATH=../ant/bin:$PATH
```

4. Perform the build by typing `ant`.

As in the last exercise, you will see *titan.jar* rebuilt, copied to the JBoss *deploy* directory, and redeployed by the application server.

Examine the JBoss-Specific Files

In this section we introduce a new JBoss CMP 2.0 deployment descriptor, *jbosscmp-jdbc.xml*. This file provides more detailed control of your bean's database mapping as well as more advanced performance-tuning options.

jbosscmp-jdbc.xml

```
<jbosscmp-jdbc>

   <defaults>
      <datasource>java:/DefaultDS</datasource>
      <datasource-mapping>Hypersonic SQL</datasource-mapping>
      <create-table>true</create-table>
      <remove-table>true</remove-table>
   </defaults>

   <enterprise-beans>
      <entity>
         <ejb-name>CustomerEJB</ejb-name>
         <table-name>Customer</table-name>
         <cmp-field>
            <field-name>id</field-name>
            <column-name>ID</column-name>
         </cmp-field>
         <cmp-field>
            <field-name>lastName</field-name>
            <column-name>LAST_NAME</column-name>
         </cmp-field>
         <cmp-field>
            <field-name>firstName</field-name>
            <column-name>FIRST_NAME</column-name>
         </cmp-field>
         <cmp-field>
            <field-name>hasGoodCredit</field-name>
            <column-name>HAS_GOOD_CREDIT</column-name>
         </cmp-field>
      </entity>
   </enterprise-beans>
</jbosscmp-jdbc>
```

The <defaults> section:

```
      <datasource>java:/DefaultDS</datasource>
```

The `<datasource>` configuration variable tells JBoss's CMP engine what database connection pool to use for the entity beans defined in this JAR. It is currently configured to use the default data source defined in *$JBOSS_HOME/server/default/deploy/hsqldb-service.xml*, but you can change it to your own defined data sources. Appendix A gets into more detail on how to configure your own data sources.

```
<datasource-mapping>Hypersonic SQL</datasource-mapping>
```

This variable describes the database mapping that CMP should use. Here are some other mappings you could use (this list is not exhaustive):

```
<datasource-mapping>Oracle8</datasource-mapping>
<datasource-mapping>Oracle7</datasource-mapping>
<datasource-mapping>MS SQLSERVER</datasource-mapping>
<datasource-mapping>MS SQLSERVER2000</datasource-mapping>
```

For other available supported database mappings, please review JBoss's advanced documentation on its web site at *http://www.jboss.org/*

```
<create-table>true</create-table>
```

When the `<create-table>` configuration variable is set to `true`, JBoss creates the database tables for each entity bean defined in the deployment descriptor unless these tables already exist. This create action is triggered when the EJB JAR is deployed.

```
<remove-table>true</remove-table>
```

When the `<remove-table>` configuration variable is set to `true`, JBoss drops the database tables for each entity bean defined in the deployment descriptor. This remove action is triggered when the EJB JAR is redeployed or undeployed.

The <enterprise-beans> section:

There is an XML fragment `<entity></entity>` for each entity bean defined in this EJB JAR.

```
<ejb-name>CustomerEJB</ejb-name>
```

The `<ejb-name>` variable defines the entity bean that is described in that section.

```
<table-name>Customer</table-name>
```

The `<table-name>` variable defines what database table this entity bean should map to.

```
<cmp-field>
   <field-name>id</field-name>
   <column-name>ID</column-name>
</cmp-field>
```

Each `<cmp-field>` section describes the mapping between an entity bean's fields and the corresponding columns of the database table. The `<field-name>` tag is the entity bean field's name, while the `<column-name>` defines the table column's name.

Examine and Run the Client Applications

There is only one client application for this exercise, *Client_61*. It is modeled after the example in the EJB book. It will create Customer EJBs in the database based on the command-line parameters.

To run the client, first set your `JBOSS_HOME` and `PATH` environment variables appropriately. Then invoke the provided wrapper script to execute the program. For each customer, you must supply on the command line a set of values for primary key, first name, and last name, as shown here:

```
Client_61 777 Bill Burke 888 Sacha Labourey
```

The output of this execution should be:

```
C:\workbook\ex06_1>client_61 777 Bill Burke 888 Sacha Labourey
Buildfile: build.xml

prepare:

compile:

ejbjar:

run.client_61:
     [java] 777 = Bill Burke
     [java] 888 = Sacha Labourey
```

When it finishes, the example program removes the created beans, so no data remains in the database.

Exercise 6.2:
Dependent Value Classes in CMP 2.0

The example programs in Exercise 6.2 explore using a dependent value class to combine multiple CMP fields into a single serializable object that can be passed in and out of entity-bean methods.

Start Up JBoss

If you already have JBoss running there is no reason to restart it.

Initialize the Database

No database initialization is needed.

Build and Deploy the Example Programs

Perform the following steps:

1. Open a command prompt or shell terminal and change to the *ex06_2* directory created by the extraction process
2. Set the `JAVA_HOME` and `JBOSS_HOME` environment variables to point to where your JDK and JBoss 3.2 are installed. Examples:

Windows:

```
C:\workbook\ex06_2> set JAVA_HOME=C:\jdk1.3
C:\workbook\ex06_2> set JBOSS_HOME=C:\jboss-3.2.0
```

Unix:

```
$ export JAVA_HOME=/usr/local/jdk1.3
$ export JBOSS_HOME=/usr/local/jboss-3.2.0
```

3. Add `ant` to your execution path.

Windows:

```
C:\workbook\ex06_2> set PATH=..\ant\bin;%PATH%
```

Unix:

```
$ export PATH=../ant/bin:$PATH
```

4. Perform the build by typing `ant`.

As in the last exercise, you will see *titan.jar* rebuilt, copied to the JBoss *deploy* directory, and redeployed by the application server.

Examine the JBoss-Specific Files

There are no new JBoss configuration files or components in this exercise.

Examine and Run the Client Applications

The example program, *Client_62*, shows how the `Name` dependent value class is used with the Customer EJB. The example code is pulled directly from the EJB book and embellished somewhat to expand on introduced concepts. The EJB book does a pretty good job of explaining the concepts illustrated in `Client_62`, so further explanation of the code is not needed in this workbook.

The client application uses the new `getName()` and `setName()` methods of the Customer EJB to initialize, modify, and display a newly created Customer bean using the `Name` dependent value class. This test bean is then removed from the database before the application finishes.

To run *Client_62* invoke the *Ant* task `run.client_62.` Remember to set your `JBOSS_HOME` and `PATH` environment variables.

The output should look something like this:

```
C:\workbook\ex06_2>ant run.client_62
Buildfile: build.xml

prepare:

compile:

ejbjar:

run.client_62:
     [java] 1 = Richard Monson
     [java] 1 = Richard Monson-Haefel
```

Exercise 6.3:
A Simple Relationship in CMP 2.0

The example program in Exercise 6.3 shows how to implement a simple CMP relationship between the Customer EJB and the Address EJB. The client again uses dependent value classes, to pass address information along to the Customer EJB.

Build and Deploy the Example Programs

Perform the following steps:

1. Open a command prompt or shell terminal and change to the *ex06_3* directory created by the extraction process
2. Set the `JAVA_HOME` and `JBOSS_HOME` environment variables to point to where your JDK and JBoss 3.2 are installed. Examples:

Windows:

```
C:\workbook\ex06_3> set JAVA_HOME=C:\jdk1.3
C:\workbook\ex06_3> set JBOSS_HOME=C:\jboss-3.2.0
```

Unix:

```
$ export JAVA_HOME=/usr/local/jdk1.3
$ export JBOSS_HOME=/usr/local/jboss-3.2.0
```

3. Add `ant` to your execution path.

Windows:

```
C:\workbook\ex06_3> set PATH=..\ant\bin;%PATH%
```

Unix:

```
$ export PATH=../ant/bin:$PATH
```

4. Perform the build by typing `ant`.

As in the last exercise, you will see *titan.jar* rebuilt, copied to the JBoss *deploy* directory, and redeployed by the application server.

Examine the JBoss-Specific Files

The Customer-Address relationship in this example can be mapped to a database table by defining the mapping in *jbosscmp-jdbc.xml*.

jbosscmp-jdbc.xml

```
<jbosscmp-jdbc>
...
</enterprise-beans>
<relationships>
   <ejb-relation>
      <ejb-relation-name>Customer-Address</ejb-relation-name>
      <foreign-key-mapping/>
      <ejb-relationship-role>
         <ejb-relationship-role-name>Customer-has-a-Address</ejb-
relationship-role-name>
         <key-fields/>
      </ejb-relationship-role>
      <ejb-relationship-role>
         <ejb-relationship-role-name>Address-belongs-to-
Customer</ejb-relationship-role-name>
         <key-fields>
            <key-field>
               <field-name>id</field-name>
               <column-name>HOME_ADDRESS</column-name>
            </key-field>
         </key-fields>
      </ejb-relationship-role>
   </ejb-relation>
</relationships>
</jbosscmp-jdbc>
```

To define the mapping of a relationship to a database table you must define `<key-fields>`. The `<field-name>` tag must be the primary key field of the entity bean in the relationship. Thus above, the `id` `<field-name>` corresponds to the Address EJB's primary key field. You can define the `<column-name>` field to be whatever the column name is in the database. Based on the mappings defined in this file, the Customer table would look like this:

```
CREATE TABLE CUSTOMER
(ID INTEGER NOT NULL,
LAST_NAME VARCHAR(256),
FIRST_NAME VARCHAR(256),
HAS_GOOD_CREDIT BIT NOT NULL,
HOME_ADDRESS INTEGER,
CONSTRAINT PK_CUSTOMER PRIMARY KEY (ID))
```

For details on more complex optimizations and database-to-relationship mappings, please see the JBoss CMP 2.0 documentation available at *http://www.jboss.org*.

Examine and Run the Client Applications

The example program, *Client_63*, shows how to create a Customer EJB and set the Address relation on that customer.

AddressBean.java

```
public abstract class AddressBean implements javax.ejb.EntityBean
{
   private static final int IDGEN_START =
                                  (int)System.currentTimeMillis();
   private static int idgen = IDGEN_START;

   public Integer ejbCreateAddress (String street, String city,
                                    String state,  String zip )
      Throws CreateException
   {
      setId(new Integer(idgen++));
      setStreet(street);
      setCity(city);
      setState(state);
      setZip(zip);
      return null;
   }
   ...
}
```

Automatic primary-key generation is available in JBoss only since version 3.2. Consequently, in order to be able to run these exercises in both JBoss 3.0 and 3.2, a very crude id generator has been created for this and subsequent examples. The code just takes the current time in milliseconds at the load of the bean and increments it by one at every `ejbCreate()`. Crude, workable for these examples, not recommended for real applications.

To run *Client_63* invoke the *Ant* task `run.client_63`. Remember to set your `JBOSS_HOME` and `PATH` environment variables.

The output should look something like this:

```
C:\workbook\ex06_3>ant run.client_63
Buildfile: build.xml

prepare:

compile:

ejbjar:

run.client_63:
     [java] Creating Customer 1..
     [java] Creating AddressDO data object..
     [java] Setting Address in Customer 1...
     [java] Acquiring Address data object from Customer 1...
     [java] Customer 1 Address data:
     [java] 1010 Colorado
     [java] Austin,TX 78701
     [java] Creating new AddressDO data object..
     [java] Setting new Address in Customer 1...
     [java] Customer 1 Address data:
     [java] 1600 Pennsylvania Avenue NW
     [java] DC,WA 20500
     [java] Removing Customer 1...
```

Exercises for Chapter 7

Exercise 7.1:
Entity Relationships in CMP 2.0: Part 1

This exercise walks you through implementing a complex set of interrelated entity beans defined in Chapter 7 of the EJB book.

Start Up JBoss

If JBoss is not running, start it up. If it's already running there's no reason to restart it.

Initialize the Database

The database table for this exercise will automatically be created in JBoss's default database, HypersonicSQL, when the EJB JAR is deployed.

Build and Deploy the Example Programs

Perform the following steps:

1. Open a command prompt or shell terminal and change to the *ex07_1* directory created by the extraction process
2. Set the `JAVA_HOME` and `JBOSS_HOME` environment variables to point to where your JDK and JBoss 3.2 are installed. Examples:

Windows:

```
C:\workbook\ex07_1> set JAVA_HOME=C:\jdk1.3
C:\workbook\ex07_1> set JBOSS_HOME=C:\jboss-3.2.0
```

Unix:

```
$ export JAVA_HOME=/usr/local/jdk1.3
$ export JBOSS_HOME=/usr/local/jboss-3.2.0
```

3. Add `ant` to your execution path.

Windows:

```
C:\workbook\ex07_1> set PATH=..\ant\bin;%PATH%
```

Unix:

```
$ export PATH=../ant/bin:$PATH
```

4. Perform the build by typing `ant`.

As in the last exercise, you will see *titan.jar* rebuilt, copied to the JBoss *deploy* directory, and redeployed by the application server.

Examine the JBoss-Specific Files

This chapter introduces no new features in JBoss-specific files. Please review Exercise 6.1 to understand the JBoss-specific files in this example. Also, this chapter implements non-performance-tuned entity beans and relies on the CMP 2.0 engine to create all database tables. To learn about JBoss's extensive configuration options, please review the advanced CMP 2.0 documentation at *http://www.jboss.org/*.

Examine and Run the Client Applications

From this chapter on, so that the example code matches the code illustrated in the EJB book, we will no longer use remote entity bean interfaces. Accordingly, the Customer EJB switches to local-only interfaces:

- `CustomerHomeRemote` becomes `CustomerHomeLocal`.
- `CustomerRemote` becomes `CustomerLocal`.
- Bean interface methods no longer throw `RemoteException`s.
- The *ejb-jar.xml* descriptor changes to use local interfaces. Thus:

```
<ejb-name>CustomerEJB</ejb-name>
<home>com.titan.customer.CustomerHomeRemote</home>
<remote>com.titan.customer.CustomerRemote</remote>
<ejb-class>com.titan.customer.CustomerBean</ejb-class>
```

...changes to...

```
<ejb-name>CustomerEJB</ejb-name>
<local-home>com.titan.customer.CustomerHomeLocal</local-home>
<local>com.titan.customer.CustomerLocal</local>
<ejb-class>com.titan.customer.CustomerBean</ejb-class>
```

- The JNDI binding in *jboss.xml* changes as well:

```
<entity>
  <ejb-name>CustomerEJB</ejb-name>
  <jndi-name>CustomerHomeRemote</jndi-name>
</entity>
```

...changes to ...

```
<entity>
  <ejb-name>CustomerEJB</ejb-name>
  <local-jndi-name>CustomerHomeLocal</local-jndi-name>
</entity>
```

Because interfaces are now local, the example programs no longer need to use dependent value classes to set up relationships like Customer-Address. This change simplifies the code a lot, and allows you to pass local entity beans such as Address, Credit Card, and Phone to Customer EJB methods directly.

Another consequence is that remote clients can no longer invoke business logic on the entity beans implemented in this chapter. Instead, you'll implement all example business logic in the methods of a stateless session bean. Also, EJB containers don't allow the manipulation of a relationship collection (including iteration through the collection) outside the context of a transaction. In JBoss, by default all bean methods are `Required`, so all example test code will run within a transaction. Chapter 14 in the EJB book discusses transactions in more detail.

To execute these examples from the command line, you implement separate, distinct remote clients that get a reference to the stateless test bean and invoke the appropriate test method.

Client_71a

The *Client_71a* example program reveals the unidirectional relationship between Customer and Address. The business logic for this example is implemented in `com.titan.test.Test71Bean`, in the `test71a()` method.

In `test71a()`, output is written to the `PrintWriter` created below. The method finishes by extracting a `String` from the `PrintWriter` and passing it back to the remote client for display.

```
public String test71a() throws RemoteException
{
   String output = null;
   StringWriter writer = new StringWriter();
   PrintWriter out = new PrintWriter(writer);
   try
   {
```

The first part of `test71a()` simply fetches the home interfaces of Customer and Address from JNDI. It then creates both a Customer and an Address:

```
      InitialContext jndiContext = getInitialContext();
      Object obj = jndiContext.lookup("CustomerHomeLocal");
      CustomerHomeLocal customerhome = (CustomerHomeLocal)obj;

      obj = jndiContext.lookup("AddressHomeLocal");
      AddressHomeLocal addresshome = (AddressHomeLocal)obj;

      out.println("Creating Customer 71");

      Integer primaryKey = new Integer(71);
      CustomerLocal customer = customerhome.create(primaryKey);
      customer.setName( new Name("Smith","John") );
```

```
        AddressLocal addr = customer.getHomeAddress();

        if (addr==null)
        {
           out.println("Address reference is NULL, Creating one and
                       setting in Customer..");
           addr = addresshome.createAddress("333 North Washington"
                                            ,"Minneapolis"
                                            ,"MN","55401");
```

A call to `customer.setHomeAddress()` sets up the relationship:

```
           customer.setHomeAddress(addr);
        }

        ...
```

Next you modify the address directly with new information. Calling the Address object's set methods is the correct way to modify a unidirectional relationship that has already been set up.

```
        addr.setStreet("445 East Lake Street");
        addr.setCity("Wayzata");
        addr.setState("MN");
        addr.setZip("55432");
        ...
```

The next bit of code shows the **wrong** way to modify a unidirectional relationship that's already been created. Instead of modifying the existing Address entity, it creates a new one. Passing the new one to `customer.setHomeAddress()` orphans the old one, which thereafter just sits there in the database, unused and forgotten. The result is a database "leak."

```
        addr = addresshome.createAddress("700 Main Street"
                                         ,"St. Paul","MN","55302");
        ...
        customer.setHomeAddress(addr);
```

Two different relationships can share the same entity. This code shares a single address between the Home Address and Billing Address relationships:

```
        addr = customer.getHomeAddress();
        ...
        customer.setBillingAddress(addr);

        AddressLocal billAddr = customer.getBillingAddress();
        AddressLocal homeAddr = customer.getHomeAddress();
```

The Billing Address and Home Address now refer to the same bean:

```
        if (billAddr.isIdentical(homeAddr))
        {
           out.println("Billing and Home are the same!");
        }
        else
        {
           out.println("Billing and Home are NOT the same!
                       BUG IN JBOSS!");
        }
     }
     catch (Exception ex)
     {
        ex.printStackTrace(out);
     }
```

Finally, `test71a()` closes the PrintWriter, extracts the output string, and returns it to the client for display:

```
     out.close();
     output = writer.toString();

     return output;
  }
```

To run *Client_71a* invoke the *Ant* task `run.client_71a`. Remember to set your `JBOSS_HOME` and `PATH` environment variables.

The output should look something like this:

```
C:\workbook\ex07_1>ant run.client_71a
Buildfile: build.xml

prepare:

compile:

run.client_71a:
     [java] Creating Customer 71
     [java] Address reference is NULL, Creating one and setting in
Customer..
     [java] Address Info: 333 North Washington Minneapolis, MN 55401
     [java] Modifying Address through address reference
     [java] Address Info: 445 East Lake Street Wayzata, MN 55432
     [java] Creating New Address and calling setHomeAddress
     [java] Address Info: 700 Main Street St. Paul, MN 55302
```

```
     [java] Retrieving Address reference from Customer via
getHomeAddress
     [java] Address Info: 700 Main Street St. Paul, MN 55302
     [java] Setting Billing address to be the same as Home address.
     [java] Testing that Billing and Home Address are the same
Entity.
     [java] Billing and Home are the same!
```

Client_71b

The *Client_71b* example program illustrates a simple one-to-one bidirectional relationship between a Customer bean and a Credit Card bean. The business logic for this example is implemented in `com.titan.test.Test71Bean`, in the `test71b()` method. Examine the code for this example.

You use the default JNDI context to obtain references to the local home interfaces of the Customer and Credit Card EJBs. The code also creates an instance of a Customer EJB:

```
// obtain CustomerHome
InitialContext jndiContext = getInitialContext();
Object obj = jndiContext.lookup("CustomerHomeLocal");
CustomerHomeLocal customerhome = (CustomerHomeLocal)obj;

obj = jndiContext.lookup("CreditCardHomeLocal");
CreditCardHomeLocal cardhome = (CreditCardHomeLocal)obj;
Integer primaryKey = new Integer(71);
CustomerLocal customer = customerhome.create(primaryKey);
customer.setName( new Name("Smith","John") );
```

Next, you create an instance of a Credit Card. Notice that you don't need to pass in a primary key; the crude algorithm introduced in *Chapter 6.3* will generate one automatically:

```
// set Credit Card info
Calendar now = Calendar.getInstance();
CreditCardLocal card = cardhome.create(now.getTime(),
                  "3700000000000001", "John Smith", "O'Reilly");
```

Then you establish the one-to-one bidirectional relationship between Customer and Credit Card simply by calling the Customer EJB's `setCreditCard()` method:

```
customer.setCreditCard(card);
```

The following code illustrates the bidirectional nature of the relationship by navigating from a Credit Card to a Customer and vice versa:

```
String cardname = customer.getCreditCard().getNameOnCard();
out.println("customer.getCreditCard().getNameOnCard()="
            + cardname);

Name name = card.getCustomer().getName();
String custfullname = name.getFirstName() + " " +
                      name.getLastName();
out.println("card.getCustomer().getName()="+custfullname);
```

Finally, the code illustrates how to destroy the relationship between the Customer and Credit Card beans:

```
card.setCustomer(null);

CreditCardLocal newcardref = customer.getCreditCard();
if (newcardref == null)
{
   out.println
      ("Card is properly unlinked from customer bean");
}
else
{
   out.println("Whoops, customer still thinks it has a
               card!  BUG IN JBOSS!");
}
```

To run *Client_71b* invoke the *Ant* task `run.client_71b`. Remember to set your `JBOSS_HOME` and `PATH` environment variables.

The output should look something like this:

```
C:\workbook\ex07_1>ant run.client_71b
Buildfile: build.xml

prepare:

compile:

run.client_71b:
     [java] Finding Customer 71
     [java] Creating CreditCard
     [java] Linking CreditCard and Customer
     [java] Testing both directions on relationship
     [java] customer.getCreditCard().getNameOnCard()=John Smith
```

```
[java] card.getCustomer().getName()=John Smith
[java] Unlink the beans using CreditCard, test Customer side
[java] Card is properly unlinked from customer bean
[java]
```

Client_71c

The *Client_71c* example program illustrates the proper use of a one-to-many unidirectional relationship between customers and their phone numbers. The business logic for this example is implemented in `com.titan.test.Test71Bean`, in the `test71c()` method.

First, the test code locates the Customer home interface through JNDI, then finds the Customer that needs new phone numbers:

```
// obtain CustomerHome
InitialContext jndiContext = getInitialContext();
Object obj = jndiContext.lookup("CustomerHomeLocal");
CustomerHomeLocal home = (CustomerHomeLocal)obj;

// Find Customer 71
Integer primaryKey = new Integer(71);
CustomerLocal customer = home.findByPrimaryKey(primaryKey);
```

The next bit of code invokes the Customer helper method `addPhoneNumber()` to relate two phone numbers to the customer, and outputs the contents of the customer-phone relationship after each addition:

```
// Display current phone numbers and types
out.println("Starting contents of phone list:");
ArrayList vv = customer.getPhoneList();
for (int jj=0; jj<vv.size(); jj++)
{
   String ss = (String)(vv.get(jj));
   out.println(ss);
}

// add a new phone number
out.println("Adding a new type 1 phone number..");
customer.addPhoneNumber("612-555-1212",(byte)1);

out.println("New contents of phone list:");
vv = customer.getPhoneList();
for (int jj=0; jj<vv.size(); jj++)
{
   String ss = (String)(vv.get(jj));
   out.println(ss);
}
```

```
// add a new phone number
out.println("Adding a new type 2 phone number..");
customer.addPhoneNumber("800-333-3333",(byte)2);

out.println("New contents of phone list:");
vv = customer.getPhoneList();
for (int jj=0; jj<vv.size(); jj++)
{
   String ss = (String)(vv.get(jj));
   out.println(ss);
}
```

This code uses the `updatePhoneNumber()` helper method to modify an existing phone number:

```
// update a phone number
out.println("Updating type 1 phone numbers..");
customer.updatePhoneNumber("763-555-1212",(byte)1);

out.println("New contents of phone list:");
vv = customer.getPhoneList();
for (int jj=0; jj<vv.size(); jj++)
{
   String ss = (String)(vv.get(jj));
   out.println(ss);
}
```

Finally, this code illustrates how to remove a member of a one-to-many unidirectional relationship:

```
// delete a phone number
out.println("Removing type 1 phone numbers from this
             Customer..");
customer.removePhoneNumber((byte)1);

out.println("Final contents of phone list:");
vv = customer.getPhoneList();
for (int jj=0; jj<vv.size(); jj++)
{
   String ss = (String)(vv.get(jj));
   out.println(ss);
}
```

Note that the phone entity hasn't been destroyed. It's still in the database, it's just no longer related to this customer bean.

To run *Client_71c* invoke the *Ant* task `run.client_71c`. Remember to set your `JBOSS_HOME` and `PATH` environment variables.

The output should look something like this:

```
C:\workbook\ex07_1>ant run.client_71c
Buildfile: build.xml

prepare:

compile:

run.client_71c:
     [java] Starting contents of phone list:
     [java] Adding a new type 1 phone number..
     [java] New contents of phone list:
     [java] Type=1  Number=612-555-1212
     [java] Adding a new type 2 phone number..
     [java] New contents of phone list:
     [java] Type=1  Number=612-555-1212
     [java] Type=2  Number=800-333-3333
     [java] Updating type 1 phone numbers..
     [java] New contents of phone list:
     [java] Type=1  Number=763-555-1212
     [java] Type=2  Number=800-333-3333
     [java] Removing type 1 phone numbers from this Customer..
     [java] Final contents of phone list:
     [java] Type=2  Number=800-333-3333
```

Exercise 7.2:
Entity Relationships in CMP 2.0: Part 2

The example programs in Exercise 7.2 illustrate the remaining four types of entity-bean relationship

- Many-to-one unidirectional (Cruise-Ship)
- One-to-many bidirectional (Cruise-Reservation)
- Many-to-many bidirectional (Customer-Reservation)
- Many-to-many unidirectional (Cabin-Reservation)

Start Up JBoss

If you already have JBoss running there is no reason to restart it.

Initialize the Database

No database initialization is needed; JBoss will create the needed tables at bean deployment.

Build and Deploy the Example Programs

Perform the following steps:

1. Open a command prompt or shell terminal and change to the *ex07_2* directory created by the extraction process
2. Set the `JAVA_HOME` and `JBOSS_HOME` environment variables to point to where your JDK and JBoss 3.2 are installed. Examples:

Windows:

```
C:\workbook\ex07_2> set JAVA_HOME=C:\jdk1.3
C:\workbook\ex07_2> set JBOSS_HOME=C:\jboss-3.2.0
```

Unix:

```
$ export JAVA_HOME=/usr/local/jdk1.3
$ export JBOSS_HOME=/usr/local/jboss-3.2.0
```

3. Add *ant* to your execution path.

Windows:

```
C:\workbook\ex07_2> set PATH=..\ant\bin;%PATH%
```

Unix:

```
$ export PATH=../ant/bin:$PATH
```

4. Perform the build by typing `ant`.

As in the last exercise, you will see *titan.jar* rebuilt, copied to the JBoss *deploy* directory, and redeployed by the application server.

Examine the JBoss-Specific Files

No new concepts are introduced in the JBoss-specific deployment descriptors.

Examine and Run the Client Applications

This exercise uses six example programs to demonstrate the various relationships described in the corresponding chapter of the EJB book. Note that you can rerun any of these examples as many times as you like because they clean up after themselves by removing all the entities they create.

- *Client_72a* demonstrates the many-to-one unidirectional Cruise-Ship relationship, as well as the sharing of a reference between different beans.
- *Client_72b* demonstrates the one-to-many bidirectional Cruise-Reservation relationship, and how to use set methods to modify reservations that are associated with a cruise.
- *Client_72c* expands on the Cruise-Reservation relationship, using the `addAll()` method to modify the reservations associated with a cruise.
- *Client_72d* demonstrates the many-to-many bidirectional Customer-Reservation relationship.
- *Client_72e* continues the demonstration of the Customer-Reservation relationship by showing how to use `setCustomers()` to modify the Customers for a Reservation.
- *Client_72f* demonstrates the many-to-many unidirectional Cabin-Reservation relationship.

Client_72a

The business logic for this example is implemented in `com.titan.test.Test72Bean`, in the `test72a()` method. *Client_72a* models the many-to-one unidirectional Cruise-Ship relationships shown in Figure 7-12 of the EJB book.

First, this code creates the relationships described in the top half of the figure. Cruises 1 to 3 embark on Ship A; Cruises 4 to 6 set sail on Ship B.

```
cruises[0] = cruisehome.create("Cruise 1", shipA);
cruises[1] = cruisehome.create("Cruise 2", shipA);
cruises[2] = cruisehome.create("Cruise 3", shipA);
cruises[3] = cruisehome.create("Cruise 4", shipB);
cruises[4] = cruisehome.create("Cruise 5", shipB);
cruises[5] = cruisehome.create("Cruise 6", shipB);
```

Next, the code switches Cruise 4 so that it is now handled by Ship A instead of Ship B. This relationship change is illustrated in the bottom half of Figure 7-12:

```
            ShipLocal newship = cruises[0].getShip();
            cruises[3].setShip(newship);
```

To run *Client_72a* invoke the *Ant* task `run.client_72a`. Remember to set your `JBOSS_HOME` and `PATH` environment variables.

The output should look something like this:

```
C:\workbook\ex07_2>ant run.client_72a
Buildfile: build.xml

prepare:

compile:

run.client_72a:
     [java] Creating Ships
     [java] PK=1001 name=Ship A tonnage=30000.0
     [java] PK=1002 name=Ship B tonnage=40000.0
     [java] Creating Cruises
     [java] Cruise 1 is using Ship A
     [java] Cruise 2 is using Ship A
     [java] Cruise 3 is using Ship A
     [java] Cruise 4 is using Ship B
     [java] Cruise 5 is using Ship B
     [java] Cruise 6 is using Ship B
     [java] Changing Cruise 4 to use same ship as Cruise 1
     [java] Cruise 1 is using Ship A
     [java] Cruise 2 is using Ship A
     [java] Cruise 3 is using Ship A
     [java] Cruise 4 is using Ship A
     [java] Cruise 5 is using Ship B
     [java] Cruise 6 is using Ship B
     [java] Removing created beans
```

Client_72b

The business logic for this example is implemented in `com.titan.test.Test72Bean`, in the `test72b()` method. *Client_72b* models the one-to-many bidirectional Cruise-Reservation relationships shown in Figure 7-14 of the EJB book.

First, this code creates the relationships described in the top half of the figure. Reservations 1 to 3 are for Cruise A; Reservations 4 to 6 are for Cruise B:

```
for (int i = 0; i < 6; i++)
{
   CruiseLocal cruise = (i < 3) ? cruiseA : cruiseB;
   reservations[i] = reservationhome.create
      (cruise,new ArrayList());
   reservations[i].setDate(date.getTime());
   reservations[i].setAmountPaid((i + 1) * 1000.0);
   date.add(Calendar.DAY_OF_MONTH, 7);
}
```

Next, the code sets the reservations of Cruise B to be the reservations of Cruise A. Those relationships actually move from A to B. Afterward, Cruise A and Reservations 1-3 no longer have any Cruise-Reservation relationships, as you see in the bottom half of Figure 7-14:

```
Collection a_reservations = cruiseA.getReservations();
cruiseB.setReservations( a_reservations );
```

To run *Client_72b* invoke the *Ant* task `run.client_72b`. Remember to set your `JBOSS_HOME` and `PATH` environment variables.

The output should look something like this:

```
C:\workbook\ex07_2>ant run.client_72b
Buildfile: build.xml

prepare:

compile:

run.client_72b:
     [java] Creating Cruises
     [java] name=Cruise A
     [java] name=Cruise B
     [java] Creating Reservations
     [java] Reservation date=11/01/2002 is for Cruise A
     [java] Reservation date=11/08/2002 is for Cruise A
     [java] Reservation date=11/15/2002 is for Cruise A
     [java] Reservation date=11/22/2002 is for Cruise B
     [java] Reservation date=11/29/2002 is for Cruise B
     [java] Reservation date=12/06/2002 is for Cruise B
     [java] Testing CruiseB.setReservations(
```

```
CruiseA.getReservations() )
     [java] Reservation date=11/01/2002 is for Cruise B
     [java] Reservation date=11/08/2002 is for Cruise B
     [java] Reservation date=11/15/2002 is for Cruise B
     [java] Reservation date=11/22/2002 is for No Cruise!
     [java] Reservation date=11/29/2002 is for No Cruise!
     [java] Reservation date=12/06/2002 is for No Cruise!
     [java] Removing created beans.
```

Client_72c

The business logic for this example is implemented in `com.titan.test.Test72Bean`, in the `test72c()` method. *Client_72c* explores the use of `Collection.addAll()` in the Cruise-Reservation one-to-many bidirectional relationship shown in Figure 7-15 of the EJB book.

First, this code creates the relationships described in the top half of the figure. Reservations 1 to 3 are for Cruise A; Reservations 4 to 6 are for Cruise B:

```
for (int i = 0; i < 6; i++)
{
   CruiseLocal cruise = (i < 3) ? cruiseA : cruiseB;
   reservations[i] = reservationhome.create
      (cruise,new ArrayList());
   reservations[i].setDate(date.getTime());
   reservations[i].setAmountPaid((i + 1) * 1000.0);
   date.add(Calendar.DAY_OF_MONTH, 7);
}
```

Then the code changes all reservations of Cruise A to be for Cruise B instead. The result of this action can be seen in the bottom half of Figure 7-15:

```
Collection a_reservations = cruiseA.getReservations();
Collection b_reservations = cruiseB.getReservations();
b_reservations.addAll(a_reservations);
```

To run *Client_72c* invoke the *Ant* task `run.client_72c`. Remember to set your `JBOSS_HOME` and `PATH` environment variables.

The output should look something like this:

```
C:\workbook\ex07_2>ant run.client_72c
Buildfile: build.xml

prepare:

compile:

run.client_72c:
```

```
     [java] Creating Cruises
     [java] name=Cruise A
     [java] name=Cruise B
     [java] Creating Reservations
     [java] Reservation date=11/01/2002 is for Cruise A
     [java] Reservation date=11/08/2002 is for Cruise A
     [java] Reservation date=11/15/2002 is for Cruise A
     [java] Reservation date=11/22/2002 is for Cruise B
     [java] Reservation date=11/29/2002 is for Cruise B
     [java] Reservation date=12/06/2002 is for Cruise B
     [java] Testing using b_res.addAll(a_res) to combine
reservations
     [java] Reservation date=11/01/2002 is for Cruise B
     [java] Reservation date=11/08/2002 is for Cruise B
     [java] Reservation date=11/15/2002 is for Cruise B
     [java] Reservation date=11/22/2002 is for Cruise B
     [java] Reservation date=11/29/2002 is for Cruise B
     [java] Reservation date=12/06/2002 is for Cruise B
```

Client_72d

The business logic for this example is implemented in `com.titan.test.Test72Bean`, in the `test72d()` method. *Client_72d* explores the use of `Collection.addAll()` in the Customer-Reservation many-to-many bidirectional relationship shown in Figure 7-17 of the EJB book.

First two sets of customers are created:

```
   Set lowcustomers = new HashSet();
   Set highcustomers = new HashSet();
   CustomerLocal[] allCustomers = new CustomerLocal[6];
   for (int kk=0; kk<6; kk++)
   {
      CustomerLocal cust = customerhome.create(new Integer(kk));
      allCustomers[kk] = cust;
      cust.setName(new Name("Customer "+kk,""));
      if (kk<=2)
      {
         lowcustomers.add(cust);
      }
      else
      {
         highcustomers.add(cust);
      }
      out.println(cust.getName().getLastName());
   }
```

Next, the code creates six reservations and relates them to one of the customer sets, as shown in the top half of Figure 7-17. Customers 1 to 3 have Reservation A; Customers 4 to 6 have Reservation B.

```
reservations[0] = reservationhome.create(cruiseA, lowcustomers);
reservations[0].setDate(date.getTime());
reservations[0].setAmountPaid(4000.0);
date.add(Calendar.DAY_OF_MONTH, 7);

reservations[1] = reservationhome.create(cruiseA, highcustomers);
reservations[1].setDate(date.getTime());
reservations[1].setAmountPaid(5000.0);
```

Finally, the code uses `addAll()` to relate Customers 4 to 6 with Reservation A. They now have a reservation for both Cruise A and Cruise B. The bottom half of Figure 7-17 illustrates this result:

```
Set customers_a = reservations[0].getCustomers();
Set customers_b = reservations[1].getCustomers();
customers_a.addAll(customers_b);
```

To run *Client_72d* invoke the *Ant* task `run.client_72d`. Remember to set your `JBOSS_HOME` and `PATH` environment variables.

The output should look something like this:

```
C:\workbook\ex07_2>ant run.client_72d
Buildfile: build.xml

prepare:

compile:

run.client_72d:
     [java] cruise.getName()=Cruise A
     [java] ship.getName()=Ship A
     [java] cruise.getShip().getName()=Ship A
     [java] Creating Customers 1-6
     [java] Customer 0
     [java] Customer 1
     [java] Customer 2
     [java] Customer 3
     [java] Customer 4
     [java] Customer 5
     [java] Creating Reservations 1 and 2, each with 3 customers
     [java] Reservation date=11/01/2002 is for Cruise A with
customers Customer 2 Customer 1 Customer 0
```

```
     [java] Reservation date=11/08/2002 is for Cruise A with
customers Customer 5 Customer 4 Customer 3
     [java] Performing customers_a.addAll(customers_b) test
     [java] Reservation date=11/01/2002 is for Cruise A with
customers Customer 2 Customer 1 Customer 0 Customer 5 Custo
mer 4 Customer 3
     [java] Reservation date=11/08/2002 is for Cruise A with
customers Customer 5 Customer 4 Customer 3
     [java] Removing created beans
```

Client_72e

The business logic for this example is implemented in `com.titan.test.Test72Bean`, in the `test72e()` method. *Client_72e* explores the use of `setCustomers()` to share an entire collection, in the Customer-Reservation many-to-many bidirectional relationship shown in Figure 7-18 of the EJB book.

First, four sets of customers are created:

```
Set customers13 = new HashSet();
Set customers24 = new HashSet();
Set customers35 = new HashSet();
Set customers46 = new HashSet();
CustomerLocal[] allCustomers = new CustomerLocal[6];
for (int kk=0; kk<6; kk++)
{
   CustomerLocal cust = customerhome.create(new Integer(kk));
   allCustomers[kk] = cust;
   cust.setName(new Name("Customer "+kk,""));
   if (kk<=2)          { customers13.add(cust); }
   if (kk>=1 && kk<=3) { customers24.add(cust); }
   if (kk>=2 && kk<=4) { customers35.add(cust); }
   if (kk>=3)          { customers46.add(cust); }
}
```

Next, the code sets up the relationships between Customers and Reservations shown in the top half of Figure 7-18:

```
reservations[0] = reservationhome.create(cruiseA, customers13);
reservations[0].setDate(date.getTime());
reservations[0].setAmountPaid(4000.0);
date.add(Calendar.DAY_OF_MONTH, 7);

reservations[1] = reservationhome.create(cruiseA, customers24);
reservations[1].setDate(date.getTime());
reservations[1].setAmountPaid(5000.0);
date.add(Calendar.DAY_OF_MONTH, 7);

reservations[2] = reservationhome.create(cruiseA, customers35);
reservations[2].setDate(date.getTime());
reservations[2].setAmountPaid(6000.0);
date.add(Calendar.DAY_OF_MONTH, 7);

reservations[3] = reservationhome.create(cruiseA, customers46);
reservations[3].setDate(date.getTime());
reservations[3].setAmountPaid(7000.0);
```

Finally, the code sets up the relationships shown in the bottom half of the figure:

```
Set customers_a = reservations[0].getCustomers();
reservations[3].setCustomers(customers_a);
```

To run *Client_72e* invoke the *Ant* task `run.client_72e`. Remember to set your `JBOSS_HOME` and `PATH` environment variables.

The output should look something like this:

```
C:\workbook\ex07_2>ant run.client_72e
Buildfile: build.xml

prepare:

compile:

run.client_72e:
     [java] Creating a Ship and Cruise
     [java] cruise.getName()=Cruise A
     [java] ship.getName()=Ship A
     [java] cruise.getShip().getName()=Ship A
     [java] Creating Customers 1-6
     [java] Creating Reservations 1-4 using three customers each
```

```
     [java] Reservation date=11/01/2002 is for Cruise A with
customers Customer 2 Customer 1 Customer 0
     [java] Reservation date=11/08/2002 is for Cruise A with
customers Customer 3 Customer 2 Customer 1
     [java] Reservation date=11/15/2002 is for Cruise A with
customers Customer 4 Customer 3 Customer 2
     [java] Reservation date=11/22/2002 is for Cruise A with
customers Customer 5 Customer 4 Customer 3
     [java] Performing reservationD.setCustomers(customersA) test
     [java] Reservation date=11/01/2002 is for Cruise A with
customers Customer 2 Customer 1 Customer 0
     [java] Reservation date=11/08/2002 is for Cruise A with
customers Customer 3 Customer 2 Customer 1
     [java] Reservation date=11/15/2002 is for Cruise A with
customers Customer 4 Customer 3 Customer 2
     [java] Reservation date=11/22/2002 is for Cruise A with
customers Customer 2 Customer 1 Customer 0
     [java] Removing created beans.
```

Client_72f

The business logic for this example is implemented in `com.titan.test.Test72Bean`, in the `test72f()` method. *Client_72f* demonstrates removing beans in the many-to-many unidirectional Cabin-Reservation relationship, as shown in Figure 7-20 of the EJB book.

First four sets of cabins are created:

```
Set cabins13 = new HashSet();
Set cabins24 = new HashSet();
Set cabins35 = new HashSet();
Set cabins46 = new HashSet();
CabinLocal[] allCabins = new CabinLocal[6];
for (int kk=0; kk<6; kk++)
{
   CabinLocal cabin = cabinhome.create(new Integer(kk));
   allCabins[kk] = cabin;
   cabin.setName("Cabin "+kk);
   if (kk<=2)          { cabins13.add(cabin); }
   if (kk>=1 && kk<=3) { cabins24.add(cabin); }
   if (kk>=2 && kk<=4) { cabins35.add(cabin); }
   if (kk>=3)          { cabins46.add(cabin); }
   out.println(cabin.getName());
}
```

Next, the code creates the initial relationships between Reservations and Cabins, shown in the top half of Figure 7-20:

```
reservations[0] = reservationhome.create(cruiseA, null);
reservations[0].setCabins(cabins13);
reservations[0].setDate(date.getTime());
reservations[0].setAmountPaid(4000.0);
date.add(Calendar.DAY_OF_MONTH, 7);

reservations[1] = reservationhome.create(cruiseA, null);
reservations[1].setCabins(cabins24);
reservations[1].setDate(date.getTime());
reservations[1].setAmountPaid(5000.0);
date.add(Calendar.DAY_OF_MONTH, 7);

reservations[2] = reservationhome.create(cruiseA, null);
reservations[2].setCabins(cabins35);
reservations[2].setDate(date.getTime());
reservations[2].setAmountPaid(6000.0);
date.add(Calendar.DAY_OF_MONTH, 7);

reservations[3] = reservationhome.create(cruiseA, null);
reservations[3].setCabins(cabins46);
reservations[3].setDate(date.getTime());
reservations[3].setAmountPaid(7000.0);
```

Finally, the code removes some of the relationships, as shown in the bottom half of the figure:

```
Set cabins_a = reservations[0].getCabins();
Iterator iterator = cabins_a.iterator();
while (iterator.hasNext())
{
   CabinLocal cc = (CabinLocal)iterator.next();
   out.println("Removing "+cc.getName()+" from cabins_a");
   iterator.remove();
}
```

To run *Client_72f* invoke the *Ant* task `run.client_72f`. Remember to set your `JBOSS_HOME` and `PATH` environment variables.

The output should look something like this:

```
C:\workbook\ex07_2>ant run.client_72f
Buildfile: build.xml

prepare:

compile:

run.client_72f:
     [java] Creating a Ship and Cruise
     [java] cruise.getName()=Cruise A
     [java] ship.getName()=Ship A
     [java] cruise.getShip().getName()=Ship A
     [java] Creating Cabins 1-6
     [java] Cabin 0
     [java] Cabin 1
     [java] Cabin 2
     [java] Cabin 3
     [java] Cabin 4
     [java] Cabin 5
     [java] Creating Reservations 1-4 using three cabins each
     [java] Reservation date=11/01/2002 is for Cruise A with cabins
Cabin 2 Cabin 1 Cabin 0
     [java] Reservation date=11/08/2002 is for Cruise A with cabins
Cabin 3 Cabin 2 Cabin 1
     [java] Reservation date=11/15/2002 is for Cruise A with cabins
Cabin 4 Cabin 3 Cabin 2
     [java] Reservation date=11/22/2002 is for Cruise A with cabins
Cabin 5 Cabin 4 Cabin 3
     [java] Performing cabins_a collection iterator.remove() test
     [java] Removing Cabin 2 from cabins_a
     [java] Removing Cabin 1 from cabins_a
     [java] Removing Cabin 0 from cabins_a
     [java] Reservation date=11/01/2002 is for Cruise A with cabins
     [java] Reservation date=11/08/2002 is for Cruise A with cabins
Cabin 3 Cabin 2 Cabin 1
     [java] Reservation date=11/15/2002 is for Cruise A with cabins
Cabin 4 Cabin 3 Cabin 2
     [java] Reservation date=11/22/2002 is for Cruise A with cabins
Cabin 5 Cabin 4 Cabin 3
     [java] Removing created beans
```

Exercise 7.3:
Cascade Deletes in CMP 2.0

This very short exercise demonstrates the use of the automatic cascade-delete feature of CMP 2.0 containers. It does this with an example Customer bean and some other beans related to it.

Build and Deploy the Example Programs

Perform the following steps:

1. Open a command prompt or shell terminal and change to the *ex07_3* directory created by the extraction process.
2. Set the `JAVA_HOME` and `JBOSS_HOME` environment variables to point to where your JDK and JBoss 3.2 are installed. Examples:

Windows:

```
C:\workbook\ex07_3> set JAVA_HOME=C:\jdk1.3
C:\workbook\ex07_3> set JBOSS_HOME=C:\jboss-3.2.0
```

Unix:

```
$ export JAVA_HOME=/usr/local/jdk1.3
$ export JBOSS_HOME=/usr/local/jboss-3.2.0
```

3. Add `ant` to your execution path.

Windows:

```
C:\workbook\ex07_3> set PATH=..\ant\bin;%PATH%
```

Unix:

```
$ export PATH=../ant/bin:$PATH
```

4. Perform the build by typing `ant`.

As in the last exercise, you will see *titan.jar* rebuilt, copied to the JBoss *deploy* directory, and redeployed by the application server.

Examine the JBoss-Specific Files

There are no new JBoss configuration files or components in this exercise.

Examine and Run the Client Applications

Client_73 is a simple example to demonstrate cascade-delete. The example code is pretty straightforward and needs no explanation.

To run *Client_73* invoke the *Ant* task `run.client_73`. Remember to set your `JBOSS_HOME` and `PATH` environment variables.

The output should look something like this:

```
C:\workbook\ex07_3>ant run.client_73
Buildfile: build.xml

prepare:

compile:

run.client_73:
     [java] Creating Customer 10078, Addresses, Credit Card, Phones
     [java] Creating CreditCard
     [java] customer.getCreditCard().getName()=Ringo Star
     [java] Creating Address
     [java] Address Info: 780 Main Street Beverly Hills, CA 90210
     [java] Creating Phones
     [java] Adding a new type 1 phone number..
     [java] Adding a new type 2 phone number.
     [java] New contents of phone list:
     [java] Type=1  Number=612-555-1212
     [java] Type=2  Number=888-555-1212
     [java] Removing Customer EJB only
```

Exercises for Chapter 8

Exercise 8.1:
Simple EJB QL Statements

The exercises in this section reveal some of the basic aspects of EJB QL programming and functionality. You'll explore basic finder methods, `ejbSelect` methods, and the use of the `IN` operation in EJB QL queries.

Start Up JBoss

If you already have JBoss running there is no reason to restart it.

Build and Deploy the Example Programs

Perform the following steps:

1. Open a command prompt or shell terminal and change to the *ex08_1* directory created by the extraction process
2. Set the `JAVA_HOME` and `JBOSS_HOME` environment variables to point to where your JDK and JBoss 3.2 are installed. Examples:

Windows:

```
C:\workbook\ex08_1> set JAVA_HOME=C:\jdk1.3
C:\workbook\ex08_1> set JBOSS_HOME=C:\jboss-3.2.0
```

Unix:

```
$ export JAVA_HOME=/usr/local/jdk1.3
$ export JBOSS_HOME=/usr/local/jboss-3.2.0
```

3. Add `ant` to your execution path.

Windows:

```
C:\workbook\ex08_1> set PATH=..\ant\bin;%PATH%
```

Unix:

```
$ export PATH=../ant/bin:$PATH
```

4. Perform the build by typing `ant`.

As in the last exercise, you will see *titan.jar* rebuilt, copied to the JBoss *deploy* directory, and redeployed by the application server.

Examine the JBoss-Specific Files

This exercise introduces no new features in JBoss-specific files. If you think you need to, review Exercise 6.1 of this workbook to understand the JBoss-specific files in this example.

Initialize the Database

The database tables for this exercise will automatically be created in JBoss's default database, HypersonicSQL, when the EJB JAR is deployed. To initialize all the tables in this example, though, you must perform the *Ant* task `run.initialize`:

```
C:\workbook\ex08_1>ant run.initialize
Buildfile: build.xml

prepare:

compile:

run.initialize:
     [java] added Bill Burke
     [java] added Sacha Labourey
     [java] added Marc Fleury
     [java] added Jane Swift
     [java] added Nomar Garciaparra
```

As in the preceding exercise, all business logic is implemented within a stateless session bean. If you'd like to see the database initialization code, look at `com.titan.test.Test81Bean`'s `initialize()` method, which creates all the entity beans for this exercise.

Examine and Run the Client Applications

Each example method of `Test81Bean` implements the example code fragments shown in the EJB book. Each `Test81Bean` method is invoked by a small, simple client application.

Client_81a

The *Client_81a* example program demonstrates a few simple finder methods that are exposed through the Customer home interface.

```
public interface CustomerHomeLocal extends javax.ejb.EJBLocalHome
{
   ...
   public CustomerLocal findByName(String lastName,
                                   String firstName)
      throws FinderException;

   public Collection findByGoodCredit()
      throws FinderException;
   ...
}
```

The Customer EJB's deployment descriptor defines these finder methods as follows:

```
<query>
   <query-method>
     <method-name>findByName</method-name>
     <method-params>
       <method-param>java.lang.String</method-param>
       <method-param>java.lang.String</method-param>
     </method-params>
   </query-method>
   <ejb-ql>
       SELECT OBJECT(c) FROM Customer c
       WHERE c.lastName = ?1 AND c.firstName = ?2
   </ejb-ql>
</query>
<query>
   <query-method>
     <method-name>findByGoodCredit</method-name>
     <method-params/>
   </query-method>
   <ejb-ql>
       SELECT OBJECT(c) FROM Customer c
       WHERE c.hasGoodCredit = TRUE
   </ejb-ql>
</query>
```

The example also demonstrates a few `ejbSelect` methods, defined in the Address EJB's deployment descriptor as follows:

```
<query>
   <query-method>
     <method-name>ejbSelectZipCodes</method-name>
     <method-params>
       <method-param>java.lang.String</method-param>
     </method-params>
   </query-method>
   <ejb-ql>
       SELECT a.zip FROM Address AS a
       WHERE a.state = ?1
   </ejb-ql>
</query>
```

```
<query>
   <query-method>
     <method-name>ejbSelectAll</method-name>
     <method-params/>
   </query-method>
   <ejb-ql>
       SELECT OBJECT(a) FROM Address AS a
   </ejb-ql>
</query>
<query>
   <query-method>
     <method-name>ejbSelectCustomer</method-name>
     <method-params>
       <method-param>com.titan.address.AddressLocal</method-param>
     </method-params>
   </query-method>
   <ejb-ql>
       SELECT OBJECT(C) FROM Customer AS c
       WHERE c.homeAddress = ?1
   </ejb-ql>
</query>
```

Because `ejbSelect` methods are private to the entity bean class, the Address home interface needs custom home methods to wrap and invoke the private `ejbSelect` methods.

```
public interface AddressHomeLocal extends javax.ejb.EJBLocalHome
{
   ...
   public Collection queryZipCodes(String state)
      throws FinderException;

   public Collection queryAll()
      throws FinderException;

   public CustomerLocal queryCustomer(AddressLocal addr)
      throws FinderException;
}
```

These custom home methods need corresponding `ejbHome` methods defined in the Address bean class. All they do is delegate to the `ejbSelect` methods they wrap.

```
public abstract class AddressBean implements javax.ejb.EntityBean
{
   ...
   public abstract Collection ejbSelectZipCodes(String state)
      throws FinderException;

   public abstract Collection ejbSelectAll()
      throws FinderException;

   public abstract CustomerLocal ejbSelectCustomer
      (AddressLocal addr)
      throws FinderException;

   public Collection ejbHomeQueryZipCodes(String state)
      throws FinderException
   {
      return ejbSelectZipCodes(state);
   }

   public Collection ejbHomeQueryAll()
      throws FinderException
   {
      return ejbSelectAll();
   }

   public CustomerLocal ejbHomeQueryCustomer(AddressLocal addr)
      throws FinderException
   {
      return ejbSelectCustomer(addr);
   }
   ...
}
```

Custom home methods are described briefly in Chapter 5 of the EJB book and in more detail in Chapter 11. As you can see, they are extremely useful in exposing private `ejbSelect` methods so that they can be invoked by test programs or business logic. All the workbook example programs for Chapter 8 use the custom home methods for this purpose.

Client_81a invokes these queries and displays their output. To run it invoke the *Ant* task `run.client_81a`. Remember to set your `JBOSS_HOME` and `PATH` environment variables.

The output should look something like this:

```
C:\workbook\ex08_1>ant run.client_81a
Buildfile: build.xml

prepare:

compile:

run.client_81a:
     [java] FIND METHODS
     [java] --------------------------------
     [java] SELECT OBJECT(c) FROM Customer c
     [java] WHERE c.lastName = ?1 AND c.firstName = ?2
     [java] Find Bill Burke using findByName
     [java]    Found Bill Burke
     [java]
     [java] SELECT OBJECT(c) FROM Customer c
     [java] WHERE c.hasGoodCredit = TRUE
     [java] Find all with good credit.  Sacha has bad credit!
     [java]    Bill has good credit.
     [java]    Marc has good credit.
     [java]    Jane has good credit.
     [java]    Nomar has good credit.
     [java]
     [java] SELECT METHODS
     [java] --------------------------------
     [java] SELECT a.zip FROM Address AS a
     [java] WHERE a.state = ?1
     [java] show ejbSelectZipCodes with queryZipCodes
     [java]    01821
     [java]    02115
     [java]    02116
     [java]
     [java] SELECT OBJECT(a) FROM Address AS a
     [java] show ejbSelectAll with queryAll
     [java]    123 Boston Road
     [java]    Billerica, MA 01821
     [java]
     [java]    Etwa Schweitzer Strasse
     [java]    Neuchatel, Switzerland 07711
     [java]
     [java]    Sharondale Dr.
     [java]    Atlanta, GA 06660
     [java]
     [java]    1 Beacon Street
```

```
     [java]    Boston, MA 02115
     [java]
     [java]    1 Yawkey Way
     [java]    Boston, MA 02116
     [java]
     [java]    West Broad Street
     [java]    Richmond, VA 23233
     [java]
     [java]    Somewhere
     [java]    Atlanta, GA 06660
     [java]
     [java]
     [java] SELECT OBJECT(C) FROM Customer AS c
     [java] WHERE c.homeAddress = ?1
     [java] show ejbSelectCustomer using Bill's address.
     [java] The customer is:
     [java]    Bill Burke
     [java]    123 Boston Road
     [java]    Billerica, MA 01821
```

Client_81b

The *Client_81b* example program gives you a chance to investigate some of the queries illustrated in the EJB book. For an explanation of the details of the tested queries below, please refer to the *Simple Queries with Paths* section of Chapter 8 of that book. The business logic for this example is implemented in `com.titan.test.Test81Bean`, in the `test81b()` method.

All the EJB QL queries in this example are `ejbSelect` methods. Again, these `ejbSelect` methods are wrapped by custom home methods. This example tests the following Customer EJB QL queries and home methods:

query:	`SELECT c.lastName FROM Customer AS c`
`ejbSelect` method:	`ejbSelectLastNames()`
custom home method:	`queryLastNames()`
`ejbHome` method:	`ejbHomeQueryLastNames()`

query:	`SELECT c.creditCard FROM Customer c`
`ejbSelect` method:	`ejbSelectCreditCards()`
custom home method:	`queryCreditCards()`
`ejbHome` method:	`ejbHomeQueryCreditCards()`

query:	`SELECT c.homeAddress.city FROM Customer c`
`ejbSelect` method:	`ejbSelectCities()`
custom home method:	`queryCities()`
`ejbHome` method:	`ejbHomeQueryCities()`

query:	`SELECT c.creditCard.creditCompany.address FROM Customer AS c`
`ejbSelect` method:	`ejbSelectCreditCompanyAddresses()`
custom home method:	`queryCreditCompanyAddresses()`
`ejbHome` method:	`ejbHomeQueryCreditCompanyAddresses()`
query:	`SELECT c.creditCard.creditCompany.address.city FROM Customer AS c`
`ejbSelect` method:	`ejbSelectCreditCompanyCities()`
custom home method:	`queryCreditCompanyCities()`
`ejbHome` method:	`ejbHomeQueryCreditCompanyCities()`

Client_81b invokes these queries and displays their output. To run it invoke the *Ant* task `run.client_81b`. Remember to set your `JBOSS_HOME` and `PATH` environment variables.

The output should look something like this:

```
C:\workbook\ex08_1>ant run.client_81b
Buildfile: build.xml

prepare:

compile:

run.client_81b:
     [java] SIMPLE QUERIES with PATHS
     [java] --------------------------------
     [java] SELECT c.lastName FROM Customer AS c
     [java]    Burke
     [java]    Labourey
     [java]    Fleury
     [java]    Swift
     [java]    Garciaparra
     [java]
     [java] SELECT c.creditCard FROM Customer c
     [java]    5324 9393 1010 2929
     [java]    5311 5000 1011 2333
     [java]    5310 5131 7711 2663
     [java]    5810 5881 7788 2688
     [java]    5450 5441 7448 2644
     [java]
     [java] SELECT c.homeAddress.city FROM Customer c
     [java]    Billerica
     [java]    Neuchatel
     [java]    Atlanta
     [java]    Boston
     [java]    Boston
```

```
     [java]
     [java] SELECT c.creditCard.creditCompany.address
     [java] FROM Customer AS c
     [java]    West Broad Street
     [java]    Richmond, VA 23233
     [java]
     [java]    West Broad Street
     [java]    Richmond, VA 23233
     [java]
     [java]    West Broad Street
     [java]    Richmond, VA 23233
     [java]
     [java]    Somewhere
     [java]    Atlanta, GA 06660
     [java]
     [java]    Somewhere
     [java]    Atlanta, GA 06660
     [java]
     [java]
     [java] SELECT c.creditCard.creditCompany.address.city
     [java] FROM Customer AS c
     [java]    Richmond
     [java]    Richmond
     [java]    Richmond
     [java]    Atlanta
     [java]    Atlanta
```

Client_81c

The *Client_81c* example program lets you investigate some more queries illustrated in the EJB book. For an explanation of the details of the tested queries below, please refer to the *IN Operator* section of Chapter 8 of that book. The business logic for this example is implemented in `com.titan.test.Test81Bean`, in the `test81c()` method.

All the EJB QL queries in this example are `ejbSelect` methods. Again, these `ejbSelect` methods are wrapped by custom home methods. This example tests the following Customer EJB QL queries and home methods:

query:
```
SELECT OBJECT( r )
FROM Customer AS c, IN( c.reservations ) AS r
```
`ejbSelect` method: `ejbSelectReservations()`
custom home method: `queryReservations()`
`ejbHome` method: `ejbHomeQueryReservations()`

query:
```
SELECT r.cruise
FROM Customer AS c, IN( c.reservations ) AS r
```
`ejbSelect` method: `ejbSelectCruises()`
custom home method: `queryCruises()`
`ejbHome` method: `ejbHomeQueryCruises()`

query:
```
SELECT cbn.ship
FROM Customer AS c, IN( c.reservations ) AS r,
IN( r.cabins ) AS cbn
```
`ejbSelect` method: `ejbSelectShips()`
custom home method: `queryShips()`
`ejbHome` method: `ejbHomeQueryShips()`

Client_81c invokes these queries and displays their output. To run it invoke the *Ant* task `run.client_81c`. Remember to set your `JBOSS_HOME` and `PATH` environment variables.

The output should look something like this:

```
C:\workbook\ex08_1>ant run.client_81c
Buildfile: build.xml

prepare:

compile:

run.client_81c:
     [java] THE IN OPERATOR
     [java] -------------------------------
     [java] SELECT OBJECT( r )
     [java] FROM Customer AS c, IN( c.reservations ) AS r
     [java]    Reservation for Alaskan Cruise
     [java]    Reservation for Alaskan Cruise
     [java]    Reservation for Atlantic Cruise
     [java]    Reservation for Atlantic Cruise
     [java]    Reservation for Alaskan Cruise
     [java]
     [java] SELECT r.cruise
     [java] FROM Customer AS c, IN( c.reservations ) AS r
     [java]    Cruise Alaskan Cruise
     [java]    Cruise Alaskan Cruise
     [java]    Cruise Atlantic Cruise
     [java]    Cruise Atlantic Cruise
     [java]    Cruise Alaskan Cruise
     [java]
     [java] SELECT cbn.ship
     [java] FROM Customer AS c, IN( c.reservations ) AS r,
     [java] IN( r.cabins ) AS cbn
     [java]    Ship Queen Mary
     [java]    Ship Queen Mary
     [java]    Ship Queen Mary
     [java]    Ship Queen Mary
     [java]    Ship Titanic
     [java]    Ship Titanic
     [java]    Ship Titanic
     [java]    Ship Titanic
     [java]    Ship Titanic
     [java]    Ship Titanic
     [java]    Ship Queen Mary
     [java]    Ship Queen Mary
```

Exercise 8.2:
Complex EJB QL Statements

The example programs in Exercise 8.2 delve deeper into the complexities of EJB QL. You will learn about arithmetic and logic operators in `WHERE` clauses as well as other more complex `WHERE`-clause constructs. The test programs of this section demonstrate most of the example queries provided in Chapter 8 of the EJB book.

Start Up JBoss

If you already have JBoss running there is no reason to restart it.

Build and Deploy the Example Programs

Build the examples for this exercise in the *ex08_2* directory, following the same procedure as for earlier exercises.

Examine the JBoss-Specific Files

This exercise introduces no new features in JBoss-specific files. If you think you need to, review Chapter 6.1 of this workbook to understand the JBoss-specific files in this example.

Initialize the Database

The database tables for this exercise will automatically be created in JBoss's default database, HypersonicSQL, when the EJB JAR is deployed, but to initialize all database tables in this example you must perform the *Ant* task `run.initialize`:

```
C:\workbook\ex08_2>ant run.initialize
Buildfile: build.xml

prepare:

compile:

run.initialize:
     [java] added Bill Burke
     [java] added Sacha Labourey
     [java] added Marc Fleury
     [java] added Jane Swift
     [java] added Nomar Garciaparra
     [java] added Richard Monson-Haefel
```

As in the preceding exercise, all example business logic is implemented within a stateless session bean, in this case `com.titan.test.Test82Bean`, and the database initialization code is in that bean's `initialize()` method, which creates all the entity beans for this exercise.

Examine and Run the Client Applications

Each example method of `Test82Bean` implements the example code fragments shown in the EJB book. Each `Test82Bean` method is invoked by a small, simple client application.

Client_82a

The *Client_82a* example program implements the queries illustrated in the EJB book, in the section of Chapter 8 entitled *Using DISTINCT*. The business logic for this example is implemented in `com.titan.test.Test82Bean`, in the `test82a()` method.

The code demonstrates a Customer EJB finder query that returns duplicate responses, then invokes a finder query that uses the `DISTINCT` keyword to filter out duplicates.

```
finder method:    findAllCustomersWithReservations()
query:            SELECT OBJECT( cust)
                  FROM Reservation res, IN (res.customers) cust

finder method:    findDistinctCustomersWithReservations()
query:            SELECT DISTINCT OBJECT( cust)
                  FROM Reservation res, IN (res.customers) cust
```

Client_82a invokes these queries and displays their output. To run it invoke the *Ant* task `run.client_82a`. Remember to set your `JBOSS_HOME` and `PATH` environment variables.

The output should look something like this:

```
C:\workbook\ex08_2>ant run.client_82a
Buildfile: build.xml

prepare:

compile:

run.client_82a:
     [java] USING DISTINCT
     [java] -------------------------------
     [java] Non-distinct:
     [java] SELECT OBJECT( cust)
     [java] FROM Reservation res, IN (res.customers) cust
     [java]    Bill has a reservation.
     [java]    Sacha has a reservation.
     [java]    Nomar has a reservation.
     [java]    Bill has a reservation.
```

```
     [java]    Marc has a reservation.
     [java]    Jane has a reservation.
     [java]
     [java] Distinct:
     [java] SELECT DISTINCT OBJECT( cust)
     [java] FROM Reservation res, IN (res.customers) cust
     [java]    Bill has a reservation.
     [java]    Sacha has a reservation.
     [java]    Marc has a reservation.
     [java]    Jane has a reservation.
     [java]    Nomar has a reservation.
```

Client_82b

The *Client_82b* example program implements the queries illustrated in the EJB book, in the section of Chapter 8 entitled *The WHERE Clause and Literals*. The business logic for this example is implemented in `com.titan.test.Test82Bean`, in the `test82b()` method.

Various Customer and Ship EJB finder queries show how to use string, numeric, and Boolean literals in EJB QL queries.

EJB: `Customer`
finder method: `findByAmericanExpress()`
query:
```
SELECT OBJECT( c ) FROM Customer AS c
WHERE c.creditCard.organization = 'American Express'
```

EJB: `Ship`
finder method: `findByTonnage100000 ()`
query:
```
SELECT OBJECT( s ) FROM Ship AS s
WHERE s.tonnage = 100000.0
```

EJB: `Customer`
finder method: `findByGoodCredit()`
query:
```
SELECT OBJECT( c ) FROM Customer AS c
WHERE c.hasGoodCredit = TRUE
```

Client_82b invokes these queries and displays their output. To run it invoke the *Ant* task `run.client_82b`.

The output should look something like this:

```
C:\workbook\ex08_2>ant run.client_82b
Buildfile: build.xml

prepare:

compile:
```

```
run.client_82b:
     [java] THE WHERE CLAUSE AND LITERALS
     [java] ------------------------------
     [java] SELECT OBJECT( c ) FROM Customer AS c
     [java] WHERE c.creditCard.organization = 'American Express'
     [java]    Jane has an American Express card.
     [java]    Nomar has an American Express card.
     [java]
     [java] SELECT OBJECT( s ) FROM Ship AS s
     [java] WHERE s.tonnage = 100000.0
     [java]    Ship Queen Mary as tonnage 100000.0
     [java]
     [java] SELECT OBJECT( c ) FROM Customer AS c
     [java] WHERE c.hasGoodCredit = TRUE
     [java]    Bill has good credit.
     [java]    Marc has good credit.
     [java]    Jane has good credit.
     [java]    Nomar has good credit.
     [java]    Richard has good credit.
```

Client_82c

The *Client_82c* example program implements the queries illustrated in the EJB book, in the section of Chapter 8 entitled *The WHERE Clause and Input Parameters*. The business logic for this example is implemented in `com.titan.test.Test82Bean`, in the `test82c()` method.

The code demonstrates a Customer EJB `ejbSelect` query that uses strings as input parameters to the query, and a Cruise EJB finder method that uses a Ship EJB as an input parameter. As in previous sections, the `ejbSelect` query is wrapped in a custom home method.

EJB: `Customer`
`ejbSelect` method: `ejbSelectLastNames()`
custom home method: `queryLastNames()`
`ejbHome` method: `ejbHomeQueryLastNames()`
query:
```
SELECT OBJECT( c ) FROM Customer AS c
WHERE c.homeAddress.state = ?2
AND c.homeAddress.city = ?1
```

EJB: `Cruise`
finder method: `findByShip()`
query:
```
SELECT OBJECT( crs ) FROM Cruise AS crs
WHERE crs.ship = ?1
```

Client_82c invokes these queries and displays their output. To run it invoke the *Ant* task `run.client_82c`. Remember to set your `JBOSS_HOME` and `PATH` environment variables.

The output should look something like this:

```
C:\workbook\ex08_2>ant run.client_82c
Buildfile: build.xml

prepare:

compile:

run.client_82c:
     [java] THE WHERE CLAUSE AND INPUT PARAMETERS
     [java] --------------------------------
     [java] SELECT OBJECT( c ) FROM Customer AS c
     [java] WHERE c.homeAddress.state = ?2
     [java] AND c.homeAddress.city = ?1
     [java] Get customers from Billerica, MA
     [java]    Bill is from Billerica.
     [java]
     [java] SELECT OBJECT( crs ) FROM Cruise AS crs
     [java] WHERE crs.ship = ?1
     [java] Get cruises on the Titanic
     [java]    Atlantic Cruise is a Titanic cruise.
```

Client_82d

The *Client_82d* example program implements the queries illustrated in the EJB book, in the section of Chapter 8 entitled *The WHERE Clause and CDATA Sections*. The business logic for this example is implemented in `com.titan.test.Test82Bean`, in the `test82d()` method. The code demonstrates a Reservation EJB finder method that must be enclosed in an XML CDATA section because it uses the > symbol in the query.

EJB: `Reservation`

finder method: `findWithPaymentGreaterThan()`

query:

```
<![CDATA[
OBJECT( r ) FROM Rservation r
WHERE r.amountPaid > ?1
]]>
```

Client_82d invokes this query and displays its output. To run it invoke the *Ant* task `run.client_82d`.

The output should look something like this:

```
C:\workbook\ex08_2>ant run.client_82d
Buildfile: build.xml

prepare:

compile:
```

```
run.client_82d:
     [java] THE WHERE CLAUSE AND CDATA Sections
     [java] --------------------------------
     [java] ![CDATA[
     [java] SELECT OBJECT( r ) FROM Rservation r
     [java] WHERE r.amountPaid > ?1
     [java] ]]>
     [java]    found reservation with amount paid > 20000.0: 40000.0
```

Client_82e

The *Client_82e* example program implements the queries illustrated in the EJB book, in the section of Chapter 8 entitled *The WHERE Clause and BETWEEN*. The business logic for this example is implemented in `com.titan.test.Test82Bean`, in the `test82e()` method. Two Ship EJB finder methods demonstrate how to use the `BETWEEN` keyword in a `WHERE` clause.

EJB:	`Ship`
finder method:	`findByTonnageBetween()`
query:	`SELECT OBJECT( s ) FROM Ship s` `WHERE s.tonnage BETWEEN 80000.00 and 130000.00`

EJB:	`Ship`
finder method:	`findByTonnageNotBetween()`
query:	`SELECT OBJECT( s ) FROM Ship s` `WHERE s.tonnage NOT BETWEEN 80000.00 and 130000.00`

Client_82e invokes these queries and displays their output. To run it invoke the *Ant* task `run.client_82e`.

The output should look something like this:

```
C:\workbook\ex08_2>ant run.client_82e
Buildfile: build.xml

prepare:

compile:

run.client_82e:
     [java] THE WHERE CLAUSE AND BETWEEN
     [java] -------------------------------
     [java] SELECT OBJECT( s ) FROM Ship s
     [java] WHERE s.tonnage BETWEEN 80000.00 and 130000.00
     [java]    Queen Mary has tonnage 100000.0
     [java]
     [java] SELECT OBJECT( s ) FROM Ship s
```

```
     [java] WHERE s.tonnage NOT BETWEEN 80000.00 and 130000.00
     [java]    Titanic has tonnage 200000.0
```

Client_82f

The *Client_82f* example program implements the queries illustrated in the EJB book, in the section of Chapter 8 entitled *The WHERE Clause and IN*. The business logic for this example is implemented in `com.titan.test.Test82Bean`, in the `test82f()` method.

The code uses two Customer EJB finder methods. One queries for all customers living in Georgia or Massachusetts. The other queries for all customers that do not live in these two states.

EJB: `Customer`
finder method: `findInStates()`
query:
```
SELECT OBJECT( c ) FROM Customer c
WHERE c.homeAddress.state IN ('GA', 'MA')
```

EJB: `Customer`
finder method: `findNotInStates()`
query:
```
SELECT OBJECT( c ) FROM Customer c
WHERE c.homeAddress.state NOT IN ('GA', 'MA')
```

Client_82f invokes these queries and displays their output. To run it invoke the *Ant* task `run.client_82f`.

The output should look something like this:

```
C:\workbook\ex08_2>ant run.client_82f
Buildfile: build.xml

prepare:

compile:

run.client_82f:
     [java] THE WHERE CLAUSE AND IN
     [java] -------------------------------
     [java] SELECT OBJECT( c ) FROM Customer c
     [java] WHERE c.homeAddress.state IN ('GA', 'MA')
     [java]    Bill
     [java]    Marc
     [java]    Jane
     [java]    Nomar
     [java]
     [java] SELECT OBJECT( c ) FROM Customer c
     [java] WHERE c.homeAddress.state NOT IN ('GA', 'MA')
     [java]    Sacha
```

Client_82g

The *Client_82g* example program implements the queries illustrated in the EJB book, in the section of Chapter 8 entitled *The WHERE Clause and IS NULL*. The business logic for this example is implemented in `com.titan.test.Test82Bean`, in the `test82g()` method.

There are two Customer EJB finder methods. One selects all customers that have a `null` home address. The other selects all customers that do not have a `null` address.

EJB:	`Customer`
finder method:	`findHomeAddressIsNull()`
query:	`SELECT OBJECT( c ) FROM Customer c` `WHERE c.homeAddress IS NULL`

EJB:	`Customer`
finder method:	`findHomeAddressIsNotNull()`
query:	`SELECT OBJECT( c ) FROM Customer c` `WHERE c.homeAddress IS NOT NULL`

Client_82g invokes these queries and displays their output. To run it invoke the *Ant* task `run.client_82g`.

The output should look something like this:

```
C:\workbook\ex08_2>ant run.client_82g
Buildfile: build.xml

prepare:

compile:

run.client_82g:
     [java] THE WHERE CLAUSE AND IS NULL
     [java] --------------------------------
     [java] SELECT OBJECT( c ) FROM Customer c
     [java] WHERE c.homeAddress IS NULL
     [java]    Richard
     [java]
     [java] SELECT OBJECT( c ) FROM Customer c
     [java] WHERE c.homeAddress IS NOT NULL
     [java]    Bill
     [java]    Sacha
     [java]    Marc
     [java]    Jane
     [java]    Nomar
```

Client_82h

The *Client_82h* example program implements the queries illustrated in the EJB book, in the section of Chapter 8 entitled *The WHERE Clause and IS EMPTY*. The business logic for this example is implemented in `com.titan.test.Test82Bean`, in the `test82h()` method.

The code uses two Cruise EJB finder methods to illustrate the use of `IS EMPTY`. One returns all the Cruises that do not have Reservations. The other method returns all Cruises that have Reservations.

```
EJB:            Cruise
finder method:  findEmptyReservations()
query:          SELECT OBJECT( crs ) FROM Cruise crs
                WHERE crs.reservations IS EMPTY

EJB:            Cruise
finder method:  findNotEmptyReservations()
query:          SELECT OBJECT( crs ) FROM Cruise crs
                WHERE crs.reservations IS NOT EMPTY
```

Client_82h invokes these queries and displays their output. To run it invoke the *Ant* task `run.client_82h`.

The output should look something like this:

```
C:\workbook\ex08_2>ant run.client_82h
Buildfile: build.xml

prepare:

compile:

run.client_82h:
     [java] THE WHERE CLAUSE AND IS EMPTY
     [java] -------------------------------
     [java] SELECT OBJECT( crs ) FROM Cruise crs
     [java] WHERE crs.reservations IS EMPTY
     [java]
     [java] SELECT OBJECT( crs ) FROM Cruise crs
     [java] WHERE crs.reservations IS NOT EMPTY
     [java]    Alaskan Cruise is not empty.
     [java]    Atlantic Cruise is not empty.
```

Client_82i

The *Client_82i* example program implements the queries illustrated in the EJB book, in the section of Chapter 8 entitled *The WHERE Clause and MEMBER OF*. The business logic for this example is implemented in `com.titan.test.Test82Bean`, in the `test82i()` method.

Two Cruise EJB finder methods demonstrate how to use EJB QL to find whether or not an entity is a member of a relationship.

```
EJB:            Cruise
finder method:  findMemberOf()
query:          SELECT OBJECT( crs ) FROM Cruise crs,
                IN (crs.reservations) res, Customer cust
                WHERE cust = ?1 ANT cust MEMBER OF res.customers
```

```
EJB:            Cruise
finder method:  findNotMemberOf()
query:          SELECT OBJECT( crs ) FROM Cruise crs,
                IN (crs.reservations) res, Customer cust
                WHERE cust = ?1 ANT cust NOT MEMBER OF res.customers
```

Client_82i invokes these queries and displays their output. To run it invoke the *Ant* task `run.client_82i`.

The output should look something like this:

```
C:\workbook\ex08_2>ant run.client_82i
Buildfile: build.xml

prepare:

compile:

run.client_82i:
     [java] THE WHERE CLAUSE AND MEMBER OF
     [java] -------------------------------
     [java] SELECT OBJECT( crs ) FROM Cruise crs,
     [java] IN (crs.reservations) res, Customer cust
     [java] WHERE cust = ?1 ANT cust MEMBER OF res.customers
     [java] Use Bill Burke
     [java]    Bill is member of Alaskan Cruise
     [java]    Bill is member of Atlantic Cruise
     [java]
     [java] SELECT OBJECT( crs ) FROM Cruise crs,
     [java] IN (crs.reservations) res, Customer cust
     [java] WHERE cust = ?1 ANT cust NOT MEMBER OF res.customers
     [java] Use Nomar Garciaparra
     [java]    Nomar is not member of Atlantic Cruise
```

Client_82j

The *Client_82j* example program implements the queries illustrated in the EJB book, in the section of Chapter 8 entitled *The WHERE Clause and LIKE*. The business logic for this example is implemented in `com.titan.test.Test82Bean`, in the `test82j()` method.

One Customer EJB finder method is used to query all Customers with a hyphenated name.

EJB: `Customer`
finder method: `findHyphenatedLastNames()`
query:
```
SELECT OBJECT( c ) FROM Customer c
WHERE c.lastName LIKE '%-%'
```

Client_82j invokes this query and displays its output. To run it invoke the *Ant* task `run.client_82j`.

The output should look something like this:

```
C:\workbook\ex08_2>ant run.client_82j
Buildfile: build.xml

prepare:

compile:

run.client_82j:
     [java] THE WHERE CLAUSE AND LIKE
     [java] --------------------------------
     [java] SELECT OBJECT( c ) FROM Customer c
     [java] WHERE c.lastName LIKE '%-%'
     [java]    Monson-Haefel
```

Client_82k

The *Client_82k* example program implements the queries illustrated in the EJB book, in the section of Chapter 8 entitled *The WHERE Clause and Functional Expressions*. The business logic for this example is implemented in `com.titan.test.Test82Bean`, in the `test82k()` method.

One Customer EJB finder method demonstrates the use of a couple of functional expressions.

EJB: `Customer`
finder method: `findByLastNameLength()`
query:
```
SELECT OBJECT( c ) FROM Customer c
WHERE LENGTH(c.lastName) > 6 AND
LOCATE('Monson', c.lastName) > 0
```

Client_82k invokes this query and displays its output. To run it invoke the *Ant* task `run.client_82k`.

The output should look something like this:

```
C:\workbook\ex08_2>ant run.client_82k
Buildfile: build.xml

prepare:

compile:

run.client_82k:
     [java] THE WHERE CLAUSE AND FUNCTIONAL EXPRESSIONS
     [java] -------------------------------
     [java] SELECT OBJECT( c ) FROM Customer c
     [java] WHERE LENGTH(c.lastName) > 6 AND
     [java] LOCATE('Monson', c.lastName) > 0
     [java]    Labourey
     [java]    Garciaparra
     [java]    Monson-Haefel
```

JBoss Dynamic QL

One of the features seriously lacking in EJB QL is the ability to do dynamic queries at run time. This example shows you how you can do dynamic queries on Customer EJBs with JBoss CMP 2.0.

First, you must declare an `ejbSelectGeneric()` method that will invoke your dynamic queries and an `ejbHome` wrapper method so that the test program can invoke it.

```
public abstract class CustomerBean implements javax.ejb.EntityBean
{
   public abstract Set ejbSelectGeneric(String jbossQl, Object[]
arguments)
         throws FinderException;

   public Set ejbHomeDynamicQuery(String jbossQL, Object[]
arguments)
      throws FinderException
   {
      return ejbSelectGeneric(jbossQL, arguments);
   }
```

Next, you must declare your `ejbHome` wrapper method in *CustomerHomeLocal.java*:

```
public interface CustomerHomeLocal extends javax.ejb.EJBLocalHome
{
   ...
   public Set dynamicQuery(String jbossQl, Object[] arguments)
      throws FinderException;
}
```

The `ejbSelectGeneric()` method must be defined in the *ejb-jar.xml* deployment descriptor. Notice that the `<ejb-ql>` value is empty.

```
<ejb-jar>
 <enterprise-beans>
   <entity>
      <ejb-name>CustomerEJB</ejb-name>
      ...
      <query>
        <query-method>
            <method-name>ejbSelectGeneric</method-name>
            <method-params>
               <method-param>java.lang.String</method-param>
               <method-param>java.lang.Object[]</method-param>
            </method-params>
        </query-method>
        <ejb-ql></ejb-ql>
      </query>
```

Finally, in *jbosscmp-jdbc.xml* tell JBoss that the `ejbSelectGeneric()` method is dynamic:

```
<jbosscmp-jdbc>
   <enterprise-beans>
      <entity>
         <ejb-name>CustomerEJB</ejb-name>
         <query>
            <query-method>
               <method-name>ejbSelectGeneric</method-name>
               <method-params>
                  <method-param>java.lang.String</method-param>
                  <method-param>java.lang.Object[]</method-param>
               </method-params>
            </query-method>
            <dynamic-ql/>
         </query>
      </entity>
   </enterprise-beans>
</jbosscmp-jdbc>
```

The business logic for this example is implemented in `com.titan.test.Test82Bean`, in the `test82Dynamic()` method.

```
public String test82Dynamic() throws RemoteException
{
  ...
  // obtain Home interfaces
  InitialContext jndiContext = getInitialContext();
  Object obj = jndiContext.lookup("CustomerHomeLocal");
  CustomerHomeLocal customerHome = (CustomerHomeLocal)obj;
  ...
  Object[] params = {};
  Set customers =
   customerHome.dynamicQuery("SELECT OBJECT( c ) FROM Customer c " +
                             "WHERE c.lastName LIKE 'B%'", params);
   ...
}
```

The `test82Dynamic()` method generates a dynamic query string and invokes the `dynamicQuery()` method defined in the `CustomerHomeLocal` interface.

Client_82Dynamic invokes `test82Dynamic()` and displays its output. To run it invoke the *Ant* task `run.client_82dynamic`.

The output should look something like this:

```
C:\workbook\ex08_2>ant run.client_82dynamic
Buildfile: build.xml

prepare:

compile:

run.client_82dynamic:
     [java] JBoss Dynamic Queries
     [java] -------------------------------
     [java] SELECT OBJECT( c ) FROM Customer c
     [java] WHERE c.lastName LIKE 'B%'
     [java]    Burke
```

Advanced JBoss QL

In the *Problems with EJB QL* section of Chapter 8 of the EJB book, Richard Monson-Haefel talks about some of the limitations of EJB QL. In the JBoss CMP 2.0 implementation, EJB QL is just a subset of a larger JBoss query language. Dain Sundstrom, the architect of the JBoss CMP 2.0 engine, did a great job of filling in some of the gaps in the EJB QL spec. Features like `ORDER BY` and the ability to use parameters within `IN` and `LIKE` clauses are just a few of the enhancements Dain has implemented. Please review the advanced CMP 2.0 documentation available at the JBoss web site, *http://www.jboss.org/*, for more information on these sexy features.

Exercise for Chapter 10

Exercise 10.1:
A BMP Entity Bean

In this exercise, you will build and examine a simple EJB that uses *bean-managed persistence* (BMP) to synchronize the state of the bean with a database. You will also build a client application to test this Ship BMP bean.

Start Up JBoss

If JBoss is already running there is no reason to restart it.

Initialize the Database

As in the CMP examples, the state of the entity beans will be stored in the database that is embedded in JBoss. JBoss was able to create all tables for CMP beans, but it cannot do the same for BMP beans because the deployment descriptors don't contain any persistence information (object-to-relational mapping, for example). The bean is in fact the only one that knows how to load, store, remove, and find data. The persistence mapping is not described in a configuration file, but embedded in the bean code instead.

One consequence is that the database environment for BMP must always be built explicitly. To make this task easier for the BMP Ship example, Ship's home interface defines two helpful *home methods*.

- Entity beans can define *home methods* that perform operations related to the EJB component's semantics but that are not linked to any particular bean instance. As an analogy, consider the static methods of a class: their semantics are generally closely related to the class's semantics, but they're not associated with any particular class instance. Don't worry if this is not very clear for you yet: in chapter 11 of the EJB book, you'll learn all about home methods.

Here's a partial view of the Ship EJB's home interface:

```
public interface ShipHomeRemote extends javax.ejb.EJBHome
{
    ...
    public void makeDbTable () throws RemoteException;
    public void deleteDbTable () throws RemoteException;
}
```

It defines two home methods. The first will create the table needed by the Ship EJB in the JBoss-embedded database and the second will drop it.

The implementation of the `makeDbTable()` home method is essentially a `CREATE TABLE` SQL statement:

```
public void ejbHomeMakeDbTable () throws SQLException
{
   PreparedStatement ps = null;
   Connection con = null;
   try
   {
      con = this.getConnection ();

      System.out.println("Creating table SHIP...");
      ps = con.prepareStatement ("CREATE TABLE SHIP ( " +
                                 "ID INT PRIMARY KEY, " +
                                 "NAME CHAR (30), " +
                                 "TONNAGE DECIMAL (8,2), " +
                                 "CAPACITY INT" +
                                 ")" );
      ps.execute ();
      System.out.println("...done!");
   }
   finally
   {
      try { ps.close (); } catch (Exception e) {}
      try { con.close (); } catch (Exception e) {}
   }
}
```

The `deleteDbTable()` home method differs only by the SQL statement it executes:

```
      ...
      System.out.println("Dropping table SHIP...");
      ps = con.prepareStatement ("DROP TABLE SHIP");
      ps.execute ();
      System.out.println("...done!");
      ...
```

You'll see how to call these methods in a subsequent section.

Examine the EJB Standard Files

The Ship EJB source code requires no modification to run in JBoss, so the standard EJB deployment descriptor is very simple:

ejb-jar.xml (part 1)

```
...
<enterprise-beans>
  <entity>
     <description>
           This bean represents a cruise ship.
     </description>
     <ejb-name>ShipEJB</ejb-name>
     <home>com.titan.ship.ShipHomeRemote</home>
     <remote>com.titan.ship.ShipRemote</remote>
     <ejb-class>com.titan.ship.ShipBean</ejb-class>
     <persistence-type>Bean</persistence-type>
     <prim-key-class>java.lang.Integer</prim-key-class>
     <reentrant>False</reentrant>
     <security-identity><use-caller-identity/></security-identity>
     <resource-ref>
          <description>DataSource for the Titan DB</description>
          <res-ref-name>jdbc/titanDB</res-ref-name>
          <res-type>javax.sql.DataSource</res-type>
          <res-auth>Container</res-auth>
     </resource-ref>
 </entity>
</enterprise-beans>
...
```

This first part of the deployment descriptor essentially tells the container that the Ship bean:

- is named `ShipEJB`
- has a persistence type set to `Bean` because it's a BMP bean
- declares a reference to a data source named `jdbc/titanDB`

Because the bean directly manages the persistence logic, the deployment descriptor does not contain any persistence information. In contrast, this information would have been mandatory for a CMP EJB.

The second part of the deployment descriptor declares the transactional and security attributes of the Ship bean:

ejb-jar.xml (part II)

```
...
<assembly-descriptor>

   <security-role>
      <description>
         This role represents everyone who is allowed full
         access to the Ship EJB.
      </description>
     <role-name>everyone</role-name>
   </security-role>

   <method-permission>
     <role-name>everyone</role-name>
     <method>
         <ejb-name>ShipEJB</ejb-name>
         <method-name>*</method-name>
     </method>
   </method-permission>

   <container-transaction>
     <method>
        <ejb-name>ShipEJB</ejb-name>
        <method-name>*</method-name>
     </method>
     <trans-attribute>Required</trans-attribute>
   </container-transaction>

 </assembly-descriptor>

</ejb-jar>
```

All methods of the Ship bean require a transaction. If no transaction is active when a method invocation enters the container, a new one will be started.

- In entity beans, transactions are always managed by the container and never directly by the bean. Thus, all work done on transactional resources, such as databases, will implicitly be part of the transactional context of the container.

Examine the JBoss-Specific Files

If you don't include a *jboss.xml*-specific deployment descriptor with your bean, JBoss will make the following decisions at deployment time:

- It will bind the Ship bean in the public JNDI tree under `/ShipEJB` (which is the name given to the bean in its associated *ejb-jar.xml* deployment descriptor).
- It will link the `jdbc/titanDB` data source expected by the bean to `java:/DefaultDS`, which is a default data source that represents the embedded database.

Unless you require different settings, you don't need to provide a *jboss.xml* file. While this shortcut is generally useful for quick prototyping, it will not satisfy more complex deployment situations. Furthermore, using a JBoss-specific deployment descriptor enables you to fine-tune a container for a particular situation.

If you take a look at the *$JBOSS_HOME/server/default/conf/standardjboss.xml* file, you will find all the default container settings that are predefined in JBoss (standard BMP, standard CMP, clustered BMP, etc.) In JBoss, there's a one-to-one mapping between a bean and a container, and each container can be configured independently.

  - This mapping was a design decision made by the JBoss container developers and has not been dictated by the EJB specification: other application servers may use another mapping.

When you write a JBoss-specific deployment descriptor, you have three options:

- Not specify any container configuration. JBoss will use the default configuration found in *standardjboss.xml*.
- Create a brand new container configuration. The default settings are not used at all. JBoss will configure the container only as you specify in *jboss.xml*.
- Modify an existing configuration. JBoss loads the default settings from the existing configuration found in *standardjboss.xml* and overrides them with the settings you specify in the *jboss.xml* deployment descriptor. This solution allows you to make minor modifications to the default container with minimal writing in your deployment descriptor.

The Ship bean uses the last option, to test its behavior with different commit options. As outlined below, this new configuration defines only a single setting (`<commit-option>`). All others are inherited from the `Standard BMP EntityBean` configuration declared in the *standardjboss.xml* file. We'll discuss commit options in a dedicated section at the end of this chapter.

jboss.xml

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

<jboss>
   <container-configurations>
      <container-configuration>
         <container-name>Standard BMP EntityBean</container-name>
         <commit-option>A</commit-option>
      </container-configuration>
   </container-configurations>
   ...
```

Because a single deployment descriptor may define multiple EJBs, the role of the `<ejb-name>` tag is to link the definitions from the *ejb-jar.xml* and *jboss.xml* files. You can consider this tag to be the bean's **identifier**. The `<jndi-name>` tag determines the name under which the client applications will be able to look up the EJB's home interface, in this case `ShipHomeRemote`.

You can also see how the bean refers to a specific configuration, thanks to the `<configuration-name>` tag.

```
   ...
   <enterprise-beans>
      <entity>
      <ejb-name>ShipEJB</ejb-name>
      <jndi-name>ShipHomeRemote</jndi-name>
      <resource-ref>
         <res-ref-name>jdbc/titanDB</res-ref-name>
         <jndi-name>java:/DefaultDS</jndi-name>
      </resource-ref>
      <configuration-name>Standard BMP EntityBean
      </configuration-name>
      </entity>
   </enterprise-beans>
</jboss>
```

The Ship bean BMP implementation needs to establish a database connection explicitly. It's the `getConnection()` method that manages the acquisition of this resource:

ShipBean.java

```
private Connection getConnection () throws SQLException
{
   try
   {
      Context jndiCntx = new InitialContext ();
      DataSource ds =
      (DataSource)jndiCntx.lookup ("java:comp/env/jdbc/titanDB");
      return ds.getConnection ();
      ...
```

The bean expects to find a data source bound to the `java:comp/env/jdbc/titanDB` JNDI name. That's why the *ejb-jar.xml* file contains the following declaration:

ejb-jar.xml

```
      ...
      <resource-ref>
           <description>DataSource for the Titan DB</description>
           <res-ref-name>jdbc/titanDB</res-ref-name>
           <res-type>javax.sql.DataSource</res-type>
           <res-auth>Container</res-auth>
      </resource-ref>
      ...
```

Then *jboss.xml* maps the `jdbc/titanDB` data source name to the actual name defined in JBoss:

jboss.xml

```
      ...
      <resource-ref>
        <res-ref-name>jdbc/titanDB</res-ref-name>
        <jndi-name>java:/DefaultDS</jndi-name>
      </resource-ref>
      ...
```

In any default JBoss installation, `java:/DefaultDS` represents the embedded database.

Build and Deploy the Example Programs

Perform the following steps:

1. Open a command prompt or shell terminal and change to the *ex10_1* directory created by the extraction process
2. Set the `JAVA_HOME` and `JBOSS_HOME` environment variables to point to where your JDK and JBoss 3.2 are installed. Examples:

Windows:

```
C:\workbook\ex10_1> set JAVA_HOME=C:\jdk1.3
C:\workbook\ex10_1> set JBOSS_HOME=C:\jboss-3.2.0
```

Unix:

```
$ export JAVA_HOME=/usr/local/jdk1.3
$ export JBOSS_HOME=/usr/local/jboss-3.2.0
```

3. Add `ant` to your execution path.

Windows:

```
C:\workbook\ex10_1> set PATH=..\ant\bin;%PATH%
```

Unix:

```
$ export PATH=../ant/bin:$PATH
```

4. Perform the build by typing `ant`.

As in the last exercise, you will see *titan.jar* rebuilt, copied to the JBoss *deploy* directory, and redeployed by the application server.

Examine the Client Application

In the "Initialize the Database" section, you saw how the bean implements the home methods that create and drop the table in the database. Now you'll see how the client application calls these home methods:

Client_101.java

```
public class Client_101
{
   public static void main (String [] args)
   {
      try
      {
         Context jndiContext = getInitialContext ();

         Object ref = jndiContext.lookup ("ShipHomeRemote");
         ShipHomeRemote home = (ShipHomeRemote)
         PortableRemoteObject.narrow (ref,ShipHomeRemote.class);

         // We check if we have to build the database schema...
         //
         if ( (args.length > 0) &&
              args[0].equalsIgnoreCase ("CreateDB") )
         {
            System.out.println ("Creating database table...");
            home.makeDbTable ();
         }
         // ... or if we have to drop it...
         //
         else if ( (args.length > 0) &&
                   args[0].equalsIgnoreCase ("DropDB") )
         {
            System.out.println ("Dropping database table...");
            home.deleteDbTable ();
         }
         else
         ...
```

You can see that, depending on the first argument found on the command line, either `CreateDB` or `DropDB`, the client application will call the corresponding home method.

If nothing is specified on the command line, the client will test our BMP bean:

```
...
else
{
   // ... standard behavior
   //
   System.out.println ("Creating Ship 101..");
   ShipRemote ship1 = home.create (new Integer
                      (101),"Edmund Fitzgerald");

   ship1.setTonnage (50000.0);
   ship1.setCapacity (300);

   Integer pk = new Integer (101);

   System.out.println ("Finding Ship 101 again..");
   ShipRemote ship2 = home.findByPrimaryKey (pk);

   System.out.println (ship2.getName ());
   System.out.println (ship2.getTonnage ());
   System.out.println (ship2.getCapacity ());

   System.out.println ("ship1.equals (ship2) == " +
   ship1.equals (ship2));

   System.out.println ("Removing Ship 101..");
   ship2.remove ();
}
...
```

The client application first creates a new Ship and calls some of its remote methods to set its tonnage and capacity. Then it finds the bean again by calling `findByPrimaryKey()`, and compares the bean references for equality. Because they represent the same bean instance, they must be equal. We've omitted the exception handling because it deserves no specific comments.

Run the Client Application

Testing the BMP bean is a three-step process:

1. Creating the database table
2. Testing the bean (possibly many times)
3. Dropping the database table

For each of these steps, a different *Ant* target is available.

Creating the Database Table

To create the table, use the `createdb_101` *Ant* target:

```
C:\workbook\ex10_1>ant createdb_101
Buildfile: build.xml

prepare:

compile:

createdb_101:
     [java] Creating database table...
```

On the JBoss side, the BMP bean displays the following lines:

```
...
12:31:42,584 INFO  [STDOUT] Creating table SHIP...
12:31:42,584 INFO  [STDOUT] ...done!
...
```

Once this step has been performed, the actual testing of the BMP bean can take place.

- If you're having trouble creating the database, shut down JBoss, then run the *Ant* build target `clean.db`. Doing so will remove all database files and allow you to start fresh.

Testing the BMP bean

To test the BMP bean, use the `run.client_101` *Ant* target:

```
C:\workbook\ex10_1>ant run.client_101
Buildfile: build.xml

prepare:

compile:

run.client_101:
     [java] Creating Ship 101..
     [java] Finding Ship 101 again..
     [java] Edmund Fitzgerald
     [java] 50000.0
     [java] 300
     [java] ship1.equals (ship2) == true
     [java] Removing Ship 101..
```

Analyzing the Effects of Transactions and Commit Options

Even though it's not particularly related to BMP beans, let's focus on an interesting problem that arises when the client first creates and initializes the bean:

```
ShipRemote ship1 = home.create (new Integer
                    (101),"Edmund Fitzgerald");

ship1.setTonnage (50000.0);
ship1.setCapacity (300);
```

Interestingly enough, this piece of code generates three different transactions on the server side. The client does not implicitly start any transaction in its code. The transaction starts only when the invocation enters the bean container and commits when the invocation leaves the container. Thus, when the client performs three calls, each one is executed in its own transactional context.

Look at the implications for the BMP bean:

```
14:36:31,730 INFO  [STDOUT] ejbCreate() pk=101 name=Edmund
Fitzgerald
14:36:31,780 INFO  [STDOUT] ejbStore() pk=101
14:36:31,840 INFO  [STDOUT] setTonnage()
14:36:31,840 INFO  [STDOUT] ejbStore() pk=101
14:36:31,860 INFO  [STDOUT] setCapacity()
14:36:31,860 INFO  [STDOUT] ejbStore() pk=101
```

As you can see, `ejbStore()` is called at the end of each transaction! Consequently, these three lines of code cause the bean to be stored three times. Worst of all, after any method invocation, the container has no way of knowing whether the state of the bean has been modified, and thus, to be on the safe side, it triggers storage of the bean.. Given that there is no read-only method concept in EJBs, calls to get methods also trigger calls to `ejbStore()`:

```
15:03:19,301 INFO  [STDOUT] getName()
15:03:19,311 INFO  [STDOUT] ejbStore() pk=101
15:03:19,331 INFO  [STDOUT] getTonnage()
15:03:19,331 INFO  [STDOUT] ejbStore() pk=101
15:03:19,371 INFO  [STDOUT] getCapacity()
15:03:19,371 INFO  [STDOUT] ejbStore() pk=101
```

In the execution of the test program, `ejbStore()` is called seven times.

You can see that transaction boundaries (i.e., where transactions are started and stopped) directly influence the number of callbacks from the container to the Ship bean, and consequently have a direct effect on performance. We'll now focus on another setting that also affects the set of callback methods the container will invoke on the bean: the *commit option*.

The commit option determines how an entity bean container can make use of its cache. Remember from the container configuration section that the bean is currently using commit option A. Let's examine all the options and their effects:

If you select commit option A, the entity bean container is allowed to cache any bean that it has loaded. Next time an invocation targets a bean that is already in the application server cache[1], the container will not have to make a costly database access call to load it again.

If you select commit option B or C, the entity bean container is allowed to cache a bean only if it loads that bean during the lifetime of the currently running transaction. Once the transaction commits or rolls back, the container must remove the bean from the cache. The next time an invocation targets the bean, the container will have to reload it from the database.

That extra reloading is costly – but you must use B or C[2] whenever the data represented by the container can also be modified by other means. Direct database access calls through a console, for example, will cause the container cache to become unsychronized with the database, leading to incorrect computations and other dire results. A container must not use commit option A unless it "owns" the database (or, more accurately, the specific tables it accesses).

Most of the time, this "black or white" approach isn't satisfactory: in real-world applications, commit option A can be used only very rarely, and commit options B and C will preclude useful cache optimizations. To circumvent these limitations, JBoss provides some proprietary optimizations: an additional commit option, distributed cache invalidations, and even a distributed cache in the forthcoming 4.0 release. See the JBoss web site for more information.

The JBoss-proprietary commit option D is a compromise between options A and C: The bean instance can be cached across transactions, but a configurable timeout value indicates when this cached data is stale and must be reloaded from the database. This option is very useful when you want some of the efficiency of commit option A, but want cached entities to be updated periodically to reflect modifications by an external system.

- Remember that each EJB deployed in JBoss has its own container. Consequently, you can define for each EJB the commit option that best fits its specific environment. For example, a ZIP code entity bean (whose data will most probably never change) could use commit option A whereas the Order EJB would use commit option C.

[1] Note that we are speaking about the application server cache, not the database cache. While database caches are critical to performance, application server caches can improve it even further.

[2] The difference between commit option B and C is very small: when a transaction commits, a container using commit option C must effectively throw away the bean instance while a container using commit option B may keep it and reuse it later. This distinction allows you to use commit option B for very specific container optimizations (such as checking whether the data has really been modified in the database and reusing the instance if no modification has occurred, instead of reloading the whole state).

After this introduction to what commit options are, it becomes possible to guess that the container is currently using commit option A without looking at its configuration. Two pieces of evidence lead us to this conclusion:

- The `findByPrimaryKey()` call isn't displayed in the log. The container first checks whether the cache already contains an instance for the given primary key. Because it does, there is no need to invoke the bean implementation's `ejbFindByPrimaryKey()` method.
- `ejbLoad()` isn't called for the bean. At the start of each new transaction it's already in cache and there is no need to reload it from the database.
  - Note that only direct access to a given bean (using its remote reference) or `findByPrimaryKey()` calls can be resolved in cache. All other queries (`findAll()`, `findByCapacity()`, etc.) must be resolved by the database directly (there is no way to perform queries in the container cache directly).

To see how different commit options lead to different behavior, in *jboss.xml* change the commit option from `A` to `C`:

jboss.xml

```
...
<container-configurations>
   <container-configuration>
      <container-name>Standard BMP EntityBean</container-name>
      <commit-option>C</commit-option>
   </container-configuration>
</container-configurations>
...
```

Then run the tests again. You'll see:

```
14:41:29,798 INFO  [STDOUT] ejbCreate() pk=101 name=Edmund
Fitzgerald
14:41:30,449 INFO  [STDOUT] ejbStore() pk=101
14:41:30,539 INFO  [STDOUT] ejbLoad() pk=101
14:41:30,599 INFO  [STDOUT] setTonnage()
14:41:30,609 INFO  [STDOUT] ejbStore() pk=101
14:41:30,659 INFO  [STDOUT] ejbLoad() pk=101
14:41:30,669 INFO  [STDOUT] setCapacity()
14:41:30,679 INFO  [STDOUT] ejbStore() pk=101
14:41:30,709 INFO  [STDOUT] ejbFindByPrimaryKey() primaryKey=101
14:41:30,729 INFO  [STDOUT] ejbLoad() pk=101
14:41:30,750 INFO  [STDOUT] getName()
14:41:30,750 INFO  [STDOUT] ejbStore() pk=101
14:41:30,780 INFO  [STDOUT] ejbLoad() pk=101
14:41:30,790 INFO  [STDOUT] getTonnage()
```

```
14:41:30,800 INFO  [STDOUT] ejbStore() pk=101
14:41:30,840 INFO  [STDOUT] ejbLoad() pk=101
14:41:30,850 INFO  [STDOUT] getCapacity()
14:41:30,860 INFO  [STDOUT] ejbStore() pk=101
14:41:30,880 INFO  [STDOUT] ejbLoad() pk=101
14:41:30,900 INFO  [STDOUT] ejbStore() pk=101
14:41:30,910 INFO  [STDOUT] ejbRemove() pk=101
```

Now, in addition to the `ejbStore()` calls you've already seen, you see calls to `ejbLoad()` at the start of each new transaction, and the call to `ejbFindByPrimaryKey()` as well, which reaches the bean implementation because it cannot be resolved within the cache.

Possible Optimizations

As you have seen during the execution of the client application, the Ship bean performs many `ejbLoad()` and `ejbStore()` operations. There are two reasons behind this behavior:

- Many transactions are started.
- The Ship bean BMP code is not optimized.

You can reduce the number of transactions in several ways:

- Define less fine-grained methods that return all attributes of the bean in a single data object.
- Add a new create method with many parameters, so a single call can create and initialize the bean.
- Use the Façade pattern: create a stateless session bean that starts a single transaction, then performs all the steps in that one transaction.
- Start a transaction in the client application, using a `UserTransaction` object.

BMP code optimization is a wide topic. Here are some tricks that are frequently used:

- Use an `isModified` flag in your bean. Set it to `true` each time the state of the bean changes (in set methods, for example). In the implementation of `ejbStore()`, perform the actual database call only if `isModified` is `true`. Think about the impact on the test application. All the `ejbStore()` calls resulting from invocations to get methods will detect that no data has been modified and will not try to synchronize with the database.
- Detect which fields are actually modified during a transaction and update only those particular fields in the database. This tactic is especially useful for beans with lots of fields, or with fields that contain large data. Contrast with the Ship BMP bean as it's currently written, where each `set`*`XXX`*`()` call updates all fields of the database even though only one actually changes.

Note that any decent CMP engine performs many of these optimizations automatically, by default.

Dropping the Database Table

Once you've run all the tests, you can clean the database environment associated with the BMP bean by removing the unused table. Use the `dropdb_101` target:

```
C:\workbook\ex10_1>ant dropdb_101
Buildfile: build.xml

prepare:

compile:

dropdb_101:
     [java] Dropping database table...
```

On the JBoss side, the BMP bean logs the following lines:

```
...
14:40:34,339 INFO  [STDOUT] Dropping table SHIP...
14:40:34,349 INFO  [STDOUT] ...done!
...
```

Exercises for Chapter 12

Exercise 12.1:
A Stateless Session Bean

In this exercise, you will build and examine a stateless session bean, `ProcessPaymentEJB`, that writes payment information to the database. You will also build a client application to test this ProcessPayment bean.

The bean will insert the payment information data directly into the database, without using an intermediary entity bean.

Examine the EJB

This example is based on the Customer and Address EJBs and their related data objects that you used in Exercise 6.3. The present exercise leaves these EJBs unchanged, and focuses on the ProcessPayment stateless session bean.

The ProcessPayment bean has a very simple remote interface. It offers options to process a payment by check, cash, or credit card. Each possibility is handled by a different method:

ProcessPaymentRemote.java

```
public interface ProcessPaymentRemote extends javax.ejb.EJBObject
{
   public boolean byCheck (CustomerRemote customer,
                           CheckDO        check,
                           double         amount)
   throws RemoteException, PaymentException;

   public boolean byCash (CustomerRemote customer,
                          double         amount)
   throws RemoteException, PaymentException;

   public boolean byCredit (CustomerRemote customer,
                            CreditCardDO   card,
                            double         amount)
   throws RemoteException, PaymentException;
   ...
}
```

Each method's third parameter is a simple transaction amount. The other two are more interesting.

The first is a `CustomerRemote` interface, which enables the ProcessPayment EJB to get any information it needs about the customer.

- It's possible to use EJB remote interfaces as parameters of other EJB methods because they extend `EJBObject`, which in turn extends `java.rmi.Remote`. Objects implementing either `Remote` or `Serializable` are perfectly valid RMI types. To the EJB container, this choice of parameter type makes no difference at all.

The second parameter conveys the details of the transaction in a data object whose type reflects the form of payment. A data object is a `Serializable` object that a client and a remote server can pass by value back and forth. Most of the time it is a simple data container, with minimal behavior. For example, the `CheckDO` class contains the check's number and bar code:

CheckDO.java

```
public class CheckDO implements java.io.Serializable
{
   public String checkBarCode;
   public int checkNumber;

   public CheckDO (String barCode, int number)
   {
      this.checkBarCode = barCode;
      this.checkNumber = number;
   }
```

Focus on the ProcessPayment EJB implementation for a little while. Each remote method first performs validity tests appropriate to the type of payment. Eventually all of them call the same private method: `process()`, which inserts the payment information into the database. For example, `byCredit()` implements this logic thus:

ProcessPaymentBean.java

```
public boolean byCredit (CustomerRemote customer,
                         CreditCardDO   card,
                         double         amount)
throws PaymentException
{
   if (card.expiration.before (new java.util.Date ()))
   {
      throw new PaymentException ("Expiration date has passed");
   }
```

```
    else
    {
       return
          process (getCustomerID (customer),
                   amount,
                   CREDIT,
                   null,
                   -1,
                   card.number,
                   new java.sql.Date (card.expiration.getTime ()));
    }
}
```

If the credit card has expired, the method throws an *application exception*. If not, it simply delegates to `process()` the chore of inserting the payment information into the database. Note that some parameters passed to `process()` are meaningless. For example, the fourth parameter represents the check bar code, which means nothing in a credit card payment, so `byCredit()` passes a dummy value.

The `process()` method is very similar to the `ejbCreate()` method of the BMP example in Chapter 10. It simply gets a data-source connection, creates a `PreparedStatement`, and inserts the payment information into the `PAYMENT` table:

```
...
con = getConnection ();

ps = con.prepareStatement
    ("INSERT INTO payment (customer_id, amount, " +
     "type, check_bar_code, " +
     "check_number, credit_number, " +
     "credit_exp_date) "+
     "VALUES (?,?,?,?,?,?,?)");
ps.setInt (1,customerID.intValue ());
ps.setDouble (2,amount);
ps.setString (3,type);
ps.setString (4,checkBarCode);
ps.setInt (5,checkNumber);
ps.setString (6,creditNumber);
ps.setDate (7,creditExpDate);

int retVal = ps.executeUpdate ();
if (retVal!=1)
{
   throw new EJBException ("Payment insert failed");
}
```

```
    return true;
    ...
```

Note that the returned value is not significant. The method either returns `true` or throws an application exception, so its return type could as easily be `void`.

Examine the EJB Standard Deployment Descriptor

The ProcessPayment standard deployment descriptor is very similar to one you've already seen:

ejb-jar.xml

```
...
<session>
  <description>
    A service that handles monetary payments
  </description>
  <ejb-name>ProcessPaymentEJB</ejb-name>
  <home>com.titan.processpayment.ProcessPaymentHomeRemote</home>
  <remote>com.titan.processpayment.ProcessPaymentRemote</remote>
  <ejb-class>com.titan.processpayment.ProcessPaymentBean</ejb-class>
  <session-type>Stateless</session-type>
  <transaction-type>Container</transaction-type>
  <env-entry>
    <env-entry-name>minCheckNumber</env-entry-name>
    <env-entry-type>java.lang.Integer</env-entry-type>
    <env-entry-value>2000</env-entry-value>
  </env-entry>
  <resource-ref>
    <description>DataSource for the Titan database</description>
    <res-ref-name>jdbc/titanDB</res-ref-name>
    <res-type>javax.sql.DataSource</res-type>
    <res-auth>Container</res-auth>
  </resource-ref>
</session>
...
```

Note that the `ProcessPaymentEJB`'s `<session-type>` tag is set to `Stateless` and its `<transaction-type>` tag is set to `Container`. These settings ensure that the container will automatically manage the transactions and enlist any transactional resources the bean uses. You will learn in chapter 14 of the EJB book how these tasks can be handled by the EJB itself (if it's a session bean or a message-driven bean).

The descriptor contains a reference to a data source it will use to store the payments. You use this data source the same way you did in the BMP example in chapter 10.

ProcessPaymentBean.java

```
private Connection getConnection () throws SQLException
{
   try
   {
      InitialContext jndiCntx = new InitialContext ();

      DataSource ds = (DataSource)
      jndiCntx.lookup ("java:comp/env/jdbc/titanDB");

      return ds.getConnection ();
   }
   catch(NamingException ne)
   {
      throw new EJBException (ne);
   }
}
```

The *ejb-jar.xml* file also specifies an *environment property*, `minCheckNumber`. Environment properties provide a very flexible way to parameterize a bean's behavior at deployment time. The `<env-entry>` tag for `minCheckNumber` specifies the property's type (`java.lang.Integer`) and a default value (2000). The ProcessPayment EJB will access the value of this property through its JNDI ENC:

ProcessPaymentBean.java

```
...
InitialContext jndiCntx = new InitialContext ();

Integer value = (Integer) jndiCntx.lookup
                          ("java:comp/env/minCheckNumber");
...
```

One very interesting point to note is that although the ProcessPayment bean works with Customer beans (recall that each remote method's first parameter is a Customer interface), the deployment descriptor doesn't declare any reference to the Customer EJB. No `<ejb-ref>` or `<ejb-local-ref>` tag is needed because the ProcessPayment bean won't find or create Customer beans through the `CustomerRemoteHome` interface, but instead will receive Customer beans directly from the client application. Thus, from the ProcessPayment EJB's point of view, the Customer is a standard remote Java object.

Examine the JBoss Deployment Descriptors

The JBoss-specific deployment descriptor for the ProcessPayment bean is very simple. It only maps the data source to the embedded database in JBoss:

jboss.xml

```
<session>
   <ejb-name>ProcessPaymentEJB</ejb-name>
   <jndi-name>ProcessPaymentHomeRemote</jndi-name>
   <resource-ref>
      <res-ref-name>jdbc/titanDB</res-ref-name>
      <jndi-name>java:/DefaultDS</jndi-name>
   </resource-ref>
</session>
```

The `<res-ref-name>` in *jboss.xml* maps to the same `<res-ref-name>` in *ejb-jar.xml*.

Start Up JBoss

If JBoss is already running there is no reason to restart it.

Build and Deploy the Example Programs

Perform the following steps:

1. Open a command prompt or shell terminal and change to the *ex12_1* directory created by the extraction process
2. Set the `JAVA_HOME` and `JBOSS_HOME` environment variables to point to where your JDK and JBoss 3.2 are installed. Examples:

Windows:

```
C:\workbook\ex12_1> set JAVA_HOME=C:\jdk1.3
C:\workbook\ex12_1> set JBOSS_HOME=C:\jboss-3.2.0
```

Unix:

```
$ export JAVA_HOME=/usr/local/jdk1.3
$ export JBOSS_HOME=/usr/local/jboss-3.2.0
```

3. Add `ant` to your execution path.

Windows:

```
C:\workbook\ex12_1> set PATH=..\ant\bin;%PATH%
```

Unix:

```
$ export PATH=../ant/bin:$PATH
```

4. Perform the build by typing `ant`.

As in the last exercise, you will see *titan.jar* rebuilt, copied to the JBoss *deploy* directory, and redeployed by the application server.

Initialize the Database

As in previous examples, you'll use the relational database that's embedded in JBoss to store payment information. Because the deployment descriptor of a stateless session bean does not contain any information about the database schema that the bean needs, JBoss can't automatically create the database table, as it does for CMP beans. Instead, you will have to create the database schema for the `PAYMENT` table manually through JDBC. Use the `createdb` *Ant* target:

```
C:\workbook\ex12_1>ant createdb
Buildfile: build.xml

prepare:

compile:

ejbjar:

createdb:
     [java] Looking up home interfaces..
     [java] Creating database table...
```

On the JBoss console you'll see:

```
INFO  [STDOUT] Creating table PAYMENT...
INFO  [STDOUT] ...done!
```

- If you're having trouble creating the database, shut down JBoss. Then run the *Ant* build target `clean.db`. This will remove all database files and allow you to start fresh.

A `dropdb` *Ant* target has been added as well, if you want to destroy the `PAYMENT` table:

```
C:\workbook\ex12_1>ant dropdb
Buildfile: build.xml

prepare:

compile:

dropdb:
     [java] Looking up home interfaces..
     [java] Dropping database table...

BUILD SUCCESSFUL
```

To implement the `createdb` and `dropdb` *Ant* targets, the JBoss version of the ProcessPayment bean introduced in the EJB book defines two new methods: `makeDbTable()` and `dropDbTable()`.

Here's a partial view of the ProcessPayment EJB's remote interface:

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

    public void makeDbTable () throws RemoteException;
    public void deleteDbTable () throws RemoteException;
}
```

It defines two home methods: the first one will create the table needed by the ProcessPayment EJB in the JBoss embedded database, and the second will drop it.

The implementation of `makeDbTable()` is essentially a `CREATE TABLE` SQL statement:

```
public void makeDbTable ()
{
   PreparedStatement ps = null;
   Connection con = null;

   try
   {
      con = this.getConnection ();
      System.out.println("Creating table PAYMENT...");
      ps = con.prepareStatement
         ("CREATE TABLE PAYMENT ( " + "CUSTOMER_ID INT, " +
          "AMOUNT DECIMAL (8,2), " + "TYPE CHAR (10), " +
          "CHECK_BAR_CODE CHAR (50), " + "CHECK_NUMBER INTEGER, " +
          "CREDIT_NUMBER CHAR (20), " + "CREDIT_EXP_DATE DATE" +
          ")" );
      ps.execute ();
      System.out.println("...done!");
   }
   catch (SQLException sql)
   {
      throw new EJBException (sql);
   }
   finally
   {
      try { ps.close (); } catch (Exception e) {}
      try { con.close (); } catch (Exception e) {}
   }
}
```

The `deleteDbTable()` home method differs only in the SQL statement it executes:

```
public void dropDbTable ()
{
      ...
      System.out.println("Dropping table PAYMENT...");
      ps = con.prepareStatement ("DROP TABLE PAYMENT");
      ps.execute ();
      System.out.println("...done!");
      ...
}
```

Examine the Client Applications

This exercise includes two example clients. The first simply prepares and creates a single Customer bean, which the second uses to insert data into the `PAYMENT` table.

Client_121a

Run the first test application by invoking the `run.client_121a` *Ant* target:

```
C:\workbook\ex12_1>ant run.client_121a
Buildfile: build.xml

prepare:

compile:

ejbjar:

run.client_121a:
     [java] Creating Customer 1..
     [java] Creating AddressDO data object..
     [java] Setting Address in Customer 1...
     [java] Acquiring Address data object from Customer 1...
     [java] Customer 1 Address data:
     [java] 1010 Colorado
     [java] Austin,TX 78701
```

Client_121b

The code of the client application that actually tests the PaymentProcess EJB is much more interesting.

First it acquires a reference to the remote home of the ProcessPayment EJB from a newly created JNDI context:

```
Context jndiContext = getInitialContext ();

System.out.println ("Looking up home interfaces..");
Object ref = jndiContext.lookup ("ProcessPaymentHomeRemote");

ProcessPaymentHomeRemote procpayhome = (ProcessPaymentHomeRemote)
PortableRemoteObject.narrow (ref,ProcessPaymentHomeRemote.class);
```

This home makes it possible to create a remote reference to the stateless session bean:

```
ProcessPaymentRemote procpay = procpayhome.create ();
```

Then the client acquires a remote home reference for the Customer EJB and uses it to find the Customer bean created in the preceding example:

```
ref = jndiContext.lookup ("CustomerHomeRemote");
CustomerHomeRemote custhome = (CustomerHomeRemote)
PortableRemoteObject.narrow (ref,CustomerHomeRemote.class);

CustomerRemote cust = custhome.findByPrimaryKey (new Integer (1));
```

The ProcessPayment EJB can now be tested by executing payments of all three kinds, cash, check, and credit card:

```
System.out.println ("Making a payment using byCash()..");
procpay.byCash (cust,1000.0);

System.out.println ("Making a payment using byCheck()..");
CheckDO check = new CheckDO ("010010101101010100011", 3001);
procpay.byCheck (cust,check,2000.0);

System.out.println ("Making a payment using byCredit()..");
Calendar expdate = Calendar.getInstance ();
expdate.set (2005,1,28); // month=1 is February
CreditCardDO credit = new CreditCardDO ("370000000000002",
                                        expdate.getTime (),
                                        "AMERICAN_EXPRESS");

procpay.byCredit (cust,credit,3000.0);
```

Finally, to check the validation logic, the client tries to execute a payment with a check whose number is too low. The ProcessPayment EJB should refuse the payment and raise an application exception.

```
System.out.println ("Making a payment using byCheck() with a low
                     check number..");
CheckDO check2 = new CheckDO ("111000100111010110101", 1001);
try
{
   procpay.byCheck (cust,check2,9000.0);
   System.out.println("Problem! The PaymentException has
                       not been raised!"); }
catch (PaymentException pe)
{
   System.out.println ("Caught PaymentException: "+
                       pe.getMessage ());
}

procpay.remove ();
```

You can launch this test by invoking the `run.client_121b` *Ant* target:

```
C:\workbook\ex12_1>ant run.client_121b
Buildfile: build.xml

prepare:

compile:

ejbjar:

run.client_121b:
     [java] Looking up home interfaces..
     [java] Making a payment using byCash()..
     [java] Making a payment using byCheck()..
     [java] Making a payment using byCredit()..
     [java] Making a payment using byCheck() with a low check
number..
     [java] Caught PaymentException: Check number is too low. Must
be at least 2000
```

At the same time, the JBoss console will display:

```
INFO  [STDOUT] process() with customerID=1 amount=1000.0
INFO  [STDOUT] process() with customerID=1 amount=2000.0
INFO  [STDOUT] process() with customerID=1 amount=3000.0
```

Once you've performed the tests, you can drop the table by invoking the `dropdb` *Ant* target.

```
C:\workbook\ex12_1>ant dropdb
Buildfile: build.xml

prepare:

compile:

ejbjar:

dropdb:
     [java] Looking up home interfaces..
     [java] Dropping database table...
```

The JBoss console displays:

```
INFO  [STDOUT] Dropping table PAYMENT...
INFO  [STDOUT] ...done!
```

Exercise 12.2:
A Stateful Session Bean

In this exercise, you will build and examine a stateful session bean, `TravelAgent`, that coordinates the work of booking a trip on a ship. You will also build a client application to test this EJB.

Our version of this exercise will not follow the one in the EJB book strictly. Instead of simplifying the beans and their relationships as the EJB book does, we will use the beans implemented in chapters 6 and 7 and thus take advantage of the CMP 2.0 features of JBoss.

Examine the EJB

This exercise is based on the EJBs from Exercise 7.3 and doesn't contain much material that previous sections haven't covered. Nevertheless, a few modifications have been made:

- The Customer EJB again has a remote home and bean interfaces (as in chapter 6) and exposes its relationship with the Address EJB in the remote interface through a new data object, `AddressDO`.
- The Cabin EJB has a new create method that takes several parameters.
- The Reservation EJB has a new create method that takes several parameters, and has a local reference to the Customer EJB.

The TravelAgent bean's role is to perform all activities needed to book a successful trip. Thus, as in the preceding example, this session bean acts as a coordinator between different EJBs and groups several actions on different beans in the same transaction. Here, though, the bean maintains a conversational state with the client; i.e., each client has a dedicated bean on the server.

In the previous example that featured stateless session beans, the home create method was not allowed to have parameters: providing initialization parameters would be useless, as the bean wouldn't be able to remember them for forthcoming invocations. A stateful session bean, by contrast, maintains a conversational state, so its create methods can have parameters to initialize the bean state. Indeed, the home interface can have several create methods. In this example, however, the TravelAgent home interface declares only one:

```
public interface TravelAgentHomeRemote extends javax.ejb.EJBHome
{
   public TravelAgentRemote create (CustomerRemote cust)
      throws RemoteException, CreateException;
}
```

Furthermore, if you take a look at the remote interface, you can see that methods are correlated around an identical state:

```
public interface TravelAgentRemote extends javax.ejb.EJBObject
{
   public void setCruiseID (Integer cruise)
      throws RemoteException, FinderException;

   public void setCabinID (Integer cabin)
      throws RemoteException, FinderException;

   public TicketDO bookPassage (CreditCardDO card, double price)
      throws RemoteException, IncompleteConversationalState;

   public String [] listAvailableCabins (int bedCount)
      throws RemoteException, IncompleteConversationalState;
}
```

If no conversational state between the client and the server existed, calling `setCruiseId()` would make no sense. The role of this method is simply to populate this conversational state so that future calls can use this data in their processing.

Because this exercise is based on the beans implemented in Chapters 6 and 7, it needs a database schema that includes all the relationships among them, and thus differs from the one in the EJB book. Because the `listAvailableCabins()` method performs direct SQL calls, it must be rewritten to take this new database schema into account:

```
...
Integer cruiseID = (Integer)cruise.getPrimaryKey ();
Integer shipID = (Integer)cruise.getShip ().getPrimaryKey ();
con = getConnection ();

ps = con.prepareStatement (
        "select ID, NAME, DECK_LEVEL from CABIN "+
        "where SHIP_ID = ? and BED_COUNT = ? and ID NOT IN "+
        "(SELECT RCL.CABIN_ID FROM RESERVATION_CABIN_LINK AS RCL,"+
        "RESERVATION AS R "+
        "WHERE RCL.RESERVATION_ID = R.ID " +
        "AND R.CRUISE_ID = ?)");

ps.setInt (1,shipID.intValue ());
ps.setInt (2,bedCount);
ps.setInt (3,cruiseID.intValue ());

result = ps.executeQuery ();
...
```

You may remember that in previous examples we added a method (either home or remote) to the EJB to be able to initialize the test environment. As you can guess, this example uses the same trick. The TravelAgent EJB remote interface has been extended with one method:

```
public interface TravelAgentRemote extends javax.ejb.EJBObject
{
   ...
   // Mechanism for building local beans for example programs.
   //
   public void buildSampleData () throws RemoteException;
}
```

This method removes any Customer, Cabin, Ship, Cruise, and Reservation EJBs from the database and recreates a basic environment. You can follow this initialization step by step.

First the method acquires references to the remote home of the Customer EJB, and to the local homes of the Cabin, Ship, Cruise, and Reservation EJBs:

```
public Collection buildSampleData ()
{
   Collection results = new ArrayList ();

   try
   {
      System.out.println ("TravelAgentBean::buildSampleData()");

      Object obj = jndiContext.lookup
                ("java:comp/env/ejb/CustomerHomeRemote");
      CustomerHomeRemote custhome = (CustomerHomeRemote)
      javax.rmi.PortableRemoteObject.narrow (obj,
                                    CustomerHomeRemote.class);

      CabinHomeLocal cabinhome =
      (CabinHomeLocal)jndiContext.lookup
                    ("java:comp/env/ejb/CabinHomeLocal");
      ShipHomeLocal shiphome =
      (ShipHomeLocal)jndiContext.lookup
                   ("java:comp/env/ejb/ShipHomeLocal");
      CruiseHomeLocal cruisehome =
      (CruiseHomeLocal)jndiContext.lookup
                     ("java:comp/env/ejb/CruiseHomeLocal");
      ReservationHomeLocal reshome =
      (ReservationHomeLocal)jndiContext.lookup
                     ("java:comp/env/ejb/ReservationHomeLocal");
```

Then any existing bean is deleted from the database:

```
        // we first clean the db by removing any customer, cabin,
        // ship, cruise and reservation beans.
        //
        removeBeansInCollection (custhome.findAll());
        results.add ("All customers have been removed");
        removeBeansInCollection (cabinhome.findAll());
        results.add ("All cabins have been removed");
        removeBeansInCollection (shiphome.findAll());
        results.add ("All ships have been removed");
        removeBeansInCollection (cruisehome.findAll());
        results.add ("All cruises have been removed");
        removeBeansInCollection (reshome.findAll());
        results.add ("All reservations have been removed");
```

The `removeBeansInCollection()` method is a simple one. It iterates through the specified collection and removes each `EJBObject` or `EJBLocalObject`.

Two customers and two ships are created:

```
            // We now set our new basic environment
            //
            System.out.println ("Creating Customers 1 and 2...");
            CustomerRemote customer1 =
                            custhome.create (new Integer (1));
            customer1.setName ( new Name ("Burke","Bill") );
            results.add ("Customer with ID 1 created (Burke Bill)");

            CustomerRemote customer2 =
                            custhome.create (new Integer (2));
            customer2.setName ( new Name ("Labourey","Sacha") );
            results.add("Customer with ID 2 created (Labourey Sacha)");

            System.out.println ("Creating Ships A and B...");
            ShipLocal shipA = shiphome.create (new Integer (101),
                              "Nordic Prince", 50000.0);
            results.add("Created ship with ID 101...");
            ShipLocal shipB = shiphome.create (new Integer (102),
                              "Bohemian Rhapsody", 70000.0);
            results.add("Created ship with ID 102...");
```

The `buildSampleData()` method adds a message to the `results` collection after each significant step, and ultimately returns `results` so the caller knows what's happened on the server.

It then creates 10 cabins on each ship:

```
System.out.println ("Creating Cabins on the Ships...");
ArrayList cabinsA = new ArrayList ();
ArrayList cabinsB = new ArrayList ();
for (int jj=0; jj<10; jj++)
{
   CabinLocal cabinA = cabinhome.create (new Integer
                       (100+jj),shipA,"Suite 10"+jj,1,1);
   cabinsA.add(cabinA);
   CabinLocal cabinB = cabinhome.create (new Integer
                       (200+jj),shipB,"Suite 20"+jj,2,1);
   cabinsB.add(cabinB);
}
results.add("Created cabins on Ship A with IDs 100-109");
results.add("Created cabins on Ship B with IDs 200-209");
```

The method quickly organizes some cruises for each ship:

```
CruiseLocal cruiseA1 = cruisehome.create ("Alaska Cruise",
                                          shipA);
CruiseLocal cruiseA2 = cruisehome.create ("Norwegian
                                          Fjords", shipA);
CruiseLocal cruiseA3 = cruisehome.create (
                                "Bermuda or Bust", shipA);
results.add("Created cruises on ShipA with IDs "+
            cruiseA1.getId()+", "+cruiseA2.getId()+
            ", "+cruiseA3.getId());

CruiseLocal cruiseB1 = cruisehome.create ("Indian Sea
                                          Cruise", shipB);
CruiseLocal cruiseB2 = cruisehome.create (
                              "Australian Highlights", shipB);
CruiseLocal cruiseB3 = cruisehome.create (
                              "Three-Hour Cruise", shipB);
results.add ("Created cruises on ShipB with IDs "+
             cruiseB1.getId ()+", "+cruiseB2.getId ()+
             ", "+cruiseB3.getId ());
```

Finally, some reservations are made for these cruises:

```
            ReservationLocal res =
            reshome.create (customer1, cruiseA1,
                            (CabinLocal)(cabinsA.get (3)),
                            1000.0, new Date ());
            res = reshome.create (customer1, cruiseB3,
                            (CabinLocal)(cabinsB.get (8)),
                            2000.0, new Date ());
            res = reshome.create (customer2, cruiseA2,
                            (CabinLocal)(cabinsA.get (5)),
                            2000.0, new Date ());
            res = reshome.create (customer2, cruiseB3,
                             (CabinLocal)(cabinsB.get (2)),
                            2000.0, new Date ());

            results.add ("Made reservation for Customer 1 on Cruise "+
                         cruiseA1.getId ()+" for Cabin 103");
            results.add ("Made reservation for Customer 1 on Cruise "+
                         cruiseB3.getId ()+" for Cabin 208");
            results.add ("Made reservation for Customer 2 on Cruise "+
                         cruiseA2.getId ()+" for Cabin 105");
            results.add ("Made reservation for Customer 2 on Cruise "+
                         cruiseB3.getId ()+" for Cabin 202");
        }
        ...
        return results;
    }
```

Later you'll see how to call this method to set up the environment.

Examine the EJB Standard Deployment Descriptor

Most of the *ejb-jar.xml* file comprises definitions you've seen in previous examples (entity beans, relationships, the ProcessPayment stateless session bean, etc.) Only two things have been added.

ejb-jar.xml

First, the Customer EJB now has both local and remote interfaces:

```
<entity>
  <ejb-name>CustomerEJB</ejb-name>
  <home>com.titan.customer.CustomerHomeRemote</home>
  <remote>com.titan.customer.CustomerRemote</remote>
  <local-home>com.titan.customer.CustomerHomeLocal</local-home>
  <local>com.titan.customer.CustomerLocal</local>
  <ejb-class>com.titan.customer.CustomerBean</ejb-class>
  <persistence-type>Container</persistence-type>
  <prim-key-class>java.lang.Integer</prim-key-class>
  <reentrant>False</reentrant>
  <cmp-version>2.x</cmp-version>
  <abstract-schema-name>Customer</abstract-schema-name>
  <cmp-field><field-name>id</field-name></cmp-field>
  <cmp-field><field-name>lastName</field-name></cmp-field>
  <cmp-field><field-name>firstName</field-name></cmp-field>
  <cmp-field><field-name>hasGoodCredit</field-name></cmp-field>
  <primkey-field>id</primkey-field>
  <security-identity><use-caller-identity/></security-identity>
</entity>
```

Providing the second interface enables the Customer EJB to serve local clients as well as remote ones. Note that the remote and local interfaces do not declare the same methods. For example, it's illegal for a remote interface to expose entity relationships, so they're accessible only via the local interface.

The second addition is the new TravelAgent stateful session bean that is the heart of this exercise:

```
<session>
   <ejb-name>TravelAgentEJB</ejb-name>
   <home>com.titan.travelagent.TravelAgentHomeRemote</home>
   <remote>com.titan.travelagent.TravelAgentRemote</remote>
   <ejb-class>com.titan.travelagent.TravelAgentBean</ejb-class>
   <session-type>Stateful</session-type>
   <transaction-type>Container</transaction-type>
   ...
```

As you can see, only the value of the `<session-type>` tag distinguishes the declaration of a stateful session bean from that of a stateless bean.

The deployment descriptor then declares all the beans referenced by the TravelAgent EJB:

```
...
<ejb-ref>
   <ejb-ref-name>ejb/ProcessPaymentHomeRemote</ejb-ref-name>
   <ejb-ref-type>Session</ejb-ref-type>
   <home>
      com.titan.processpayment.ProcessPaymentHomeRemote
   </home>
   <remote>
      com.titan.processpayment.ProcessPaymentRemote
   </remote>
   <ejb-link>ProcessPaymentEJB</ejb-link>
</ejb-ref>
<ejb-ref>
   <ejb-ref-name>ejb/CustomerHomeRemote</ejb-ref-name>
   <ejb-ref-type>Entity</ejb-ref-type>
   <home>
      com.titan.customer.CustomerHomeRemote
   </home>
   <remote>com.titan.customer.CustomerRemote</remote>
   <ejb-link>CustomerEJB</ejb-link>
</ejb-ref>
<ejb-local-ref>
   <ejb-ref-name>ejb/CabinHomeLocal</ejb-ref-name>
   <ejb-ref-type>Entity</ejb-ref-type>
   <local-home>
      com.titan.cabin.CabinHomeLocal
   </local-home>
   <local>com.titan.cabin.CabinLocal</local>
   <ejb-link>CabinEJB</ejb-link>
</ejb-local-ref>
<ejb-local-ref>
   <ejb-ref-name>ejb/ShipHomeLocal</ejb-ref-name>
   <ejb-ref-type>Entity</ejb-ref-type>
   <local-home>
      com.titan.cabin.ShipHomeLocal
   </local-home>
   <local>com.titan.cabin.ShipLocal</local>
   <ejb-link>ShipEJB</ejb-link>
</ejb-local-ref>
<ejb-local-ref>
   <ejb-ref-name>ejb/CruiseHomeLocal</ejb-ref-name>
   <ejb-ref-type>Entity</ejb-ref-type>
   <local-home>
      com.titan.cruise.CruiseHomeLocal
```

```
      </local-home>
      <local>com.titan.cruise.CruiseLocal</local>
      <ejb-link>CruiseEJB</ejb-link>
   </ejb-local-ref>
   <ejb-local-ref>
      <ejb-ref-name>ejb/ReservationHomeLocal</ejb-ref-name>
      <ejb-ref-type>Entity</ejb-ref-type>
      <local-home>
         com.titan.reservation.ReservationHomeLocal
      </local-home>
      <local>com.titan.reservation.ReservationLocal</local>
      <ejb-link>ReservationEJB</ejb-link>
   </ejb-local-ref>
   <resource-ref>
      <res-ref-name>jdbc/titanDB</res-ref-name>
      <res-type>javax.sql.DataSource</res-type>
      <res-auth>Container</res-auth>
   </resource-ref>
</session>
```

Examine the JBoss Deployment Descriptor

The *jboss.xml* deployment descriptor contains the JNDI name mapping found in the previous examples. The only new entry is the TravelAgent EJB definition:

jboss.xml

```
<session>
   <ejb-name>TravelAgentEJB</ejb-name>
   <jndi-name>TravelAgentHomeRemote</jndi-name>
   <resource-ref>
      <res-ref-name>jdbc/titanDB</res-ref-name>
      <jndi-name>java:/DefaultDS</jndi-name>
   </resource-ref>
</session>
```

This file defines the JNDI name for the TravelAgent, then maps the data source's JNDI ENC name to the embedded database.

The `listAvailableCabins()` method uses this mapping to execute SQL statements directly against the database, so it must know precisely the names of the tables and fields to use in each query. While *jbosscmp-jdbc.xml* already defines the field-to-column mapping of all CMP beans, it doesn't define the fields and tables used by relationships between these beans. If it doesn't have those definitions, JBoss will use arbitrary names for these tables – not good in this case. To avoid this problem, you extend *jbosscmp-jdbc.xml*, adding definitions that map the relationships into

the desired tables and columns exactly. For this exercise, we mapped only the relationships used in the SQL query: Cabin-Ship, Cabin-Reservation, and Cruise-Reservation.

jbosscmp-jdbc.xml

Cabin-Reservation is a many-to-many relationship:

```
<ejb-relation>
   <ejb-relation-name>Cabin-Reservation</ejb-relation-name>
   <relation-table-mapping>
      <table-name>RESERVATION_CABIN_LINK</table-name>
      <create-table>true</create-table>
      <remove-table>true</remove-table>
   </relation-table-mapping>
   <ejb-relationship-role>
      <ejb-relationship-role-name
       >Cabin-has-many-Reservations<
      /ejb-relationship-role-name>
      <key-fields>
         <key-field>
            <field-name>id</field-name>
            <column-name>CABIN_ID</column-name>
         </key-field>
      </key-fields>
   </ejb-relationship-role>
   <ejb-relationship-role>
      <ejb-relationship-role-name
       >Reservation-has-many-Cabins<
       /ejb-relationship-role-name>
      <key-fields>
         <key-field>
            <field-name>id</field-name>
            <column-name>RESERVATION_ID</column-name>
         </key-field>
      </key-fields>
   </ejb-relationship-role>
</ejb-relation>
...
```

Many-to-many relationships always need an intermediate table. The name of this table is defined in the `<table-name>` tag. Then, for each role of the relationship, the `<field-name>` and `<column-name>` tags do the mapping between the CMR field of the bean and the column in the table.

The last two mappings needed are for one-to-many relationships, Cabin-Ship and Cruise-Reservation:

```
...
<ejb-relation>
   <ejb-relation-name>Cabin-Ship</ejb-relation-name>
   <foreign-key-mapping/>
   <ejb-relationship-role>
      <ejb-relationship-role-name
       >Ship-has-many-Cabins<
      /ejb-relationship-role-name>
      <key-fields>
         <key-field>
            <field-name>id</field-name>
            <column-name>SHIP_ID</column-name>
         </key-field>
      </key-fields>
   </ejb-relationship-role>
   <ejb-relationship-role>
      <ejb-relationship-role-name
       >Cabin-has-a-Ship<
      /ejb-relationship-role-name>
      <key-fields/>
   </ejb-relationship-role>
</ejb-relation>

<ejb-relation>
   <ejb-relation-name>Cruise-Reservation</ejb-relation-name>
   <foreign-key-mapping/>
   <ejb-relationship-role>
      <ejb-relationship-role-name
       >Cruise-has-many-Reservations<
      /ejb-relationship-role-name>
      <key-fields>
         <key-field>
            <field-name>id</field-name>
            <column-name>CRUISE_ID</column-name>
         </key-field>
      </key-fields>
   </ejb-relationship-role>
   <ejb-relationship-role>
      <ejb-relationship-role-name
       >Reservation-has-a-Cruise<
      /ejb-relationship-role-name>
      <key-fields/>
   </ejb-relationship-role>
</ejb-relation>
```

For each relationship identified by an `<ejb-relation-name>` tag (the name must be the same as the one declared in *ejb-jar.xml*), the mapping of the CMR field to a table column is defined by the `<field-name>` and `<column-name>` tags.

Start Up JBoss

If JBoss is already running there is no reason to restart it.

Build and Deploy the Example Programs

Perform the following steps:

1. Open a command prompt or shell terminal and change to the *ex12_2* directory created by the extraction process
2. Set the `JAVA_HOME` and `JBOSS_HOME` environment variables to point to where your JDK and JBoss 3.2 are installed. Examples:

Windows:

```
C:\workbook\ex12_2> set JAVA_HOME=C:\jdk1.3
C:\workbook\ex12_2> set JBOSS_HOME=C:\jboss-3.2.0
```

Unix:

```
$ export JAVA_HOME=/usr/local/jdk1.3
$ export JBOSS_HOME=/usr/local/jboss-3.2.0
```

3. Add `ant` to your execution path.

Windows:

```
C:\workbook\ex12_2> set PATH=..\ant\bin;%PATH%
```

Unix:

```
$ export PATH=../ant/bin:$PATH
```

4. Perform the build by typing `ant`.

As in the last exercise, you will see *titan.jar* rebuilt, copied to the JBoss *deploy* directory, and redeployed by the application server.

Initialize the Database

Because the exercise uses the ProcessPayment EJB from the previous example, the database must contain the `PAYMENT` table. The `createdb` and `dropdb` *Ant* targets, Java code, and clients here have been borrowed from Exercise 12.1.

If you have dropped the `PAYMENT` table after running the examples in Exercise 12.1, re-create it now by running the `createdb` *Ant* target.

```
C:\workbook\ex12_2>ant createdb
Buildfile: build.xml

prepare:

compile:

ejbjar:

createdb:
     [java] Looking up home interfaces..
     [java] Creating database table...
```

On the JBoss console, you'll see:

```
INFO  [STDOUT] Creating table PAYMENT...
INFO  [STDOUT] ...done!
```

- If you're having trouble creating the database, shut down JBoss. Then run the *Ant* build target `clean.db`. This will remove all database files and allow you to start fresh.

The container manages the persistence of all other entity beans used in this exercise, so it will create the needed tables for them automatically.

Examine the Client Applications

This exercise includes three example client applications.

Client_122a

The first client simply calls the TravelAgent bean's `buildSampleData()` method. To run this application, invoke the *Ant* target `run.client_122a`:

```
C:\workbook\ex12_2>ant run.client_122a
Buildfile: build.xml

prepare:

compile:

ejbjar:

run.client_122a:
  [java] Calling TravelAgentBean to create sample data..
  [java] All customers have been removed
  [java] All cabins have been removed
  [java] All ships have been removed
  [java] All cruises have been removed
  [java] All reservations have been removed
  [java] Customer with ID 1 created (Burke Bill)
  [java] Customer with ID 2 created (Labourey Sacha)
  [java] Created ship with ID 101...
  [java] Created ship with ID 102...
  [java] Created cabins on Ship A with IDs 100-109
  [java] Created cabins on Ship B with IDs 200-209
  [java] Created Alaska Cruise with ID 0 on ShipA
  [java] Created Norwegian Fjords Cruise with ID 1 on ShipA
  [java] Created Bermuda or Bust Cruise with ID 2 on ShipA
  [java] Created Indian Sea Cruise with ID 3 on ShipB
  [java] Created Australian Highlights Cruise with ID 4 on ShipB
  [java] Created Three-Hour Cruise with ID 5 on ShipB
  [java] Made reservation for Customer 1 on Cruise 0 for Cabin 103
  [java] Made reservation for Customer 1 on Cruise 5 for Cabin 208
  [java] Made reservation for Customer 2 on Cruise 1 for Cabin 105
  [java] Made reservation for Customer 2 on Cruise 5 for Cabin 202
```

Now that you've prepared the environment, you can use the other two client applications. *Client_122b* allows you to book a passage, while *Client_122c* gives you a list of the Cabins for a specific Cruise that have a specified number of beds.

Client_122b

The second client starts by getting remote home interfaces to the TravelAgent and Customer EJBs:

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

   if (args.length != 4)
   {
      System.out.println
         ("Usage: java " +
          "com.titan.clients.Client_122b" +
          "<customerID> <cruiseID> <cabinID> <price>");
      System.exit(-1);
   }

   Integer customerID = new Integer(args[0]);
   Integer cruiseID = new Integer(args[1]);
   Integer cabinID = new Integer(args[2]);
   double price = new Double(args[3]).doubleValue();

   Context jndiContext = getInitialContext();
   Object obj = jndiContext.lookup("TravelAgentHomeRemote");
   TravelAgentHomeRemote tahome = (TravelAgentHomeRemote)
      javax.rmi.PortableRemoteObject.narrow(obj,
                TravelAgentHomeRemote.class);

   obj = jndiContext.lookup("CustomerHomeRemote");
   CustomerHomeRemote custhome = (CustomerHomeRemote)
      javax.rmi.PortableRemoteObject.narrow(obj,
                CustomerHomeRemote.class);
```

With the home references in hand, it can now get a reference to the customer whose ID was given on the command line. If no customer with this ID exists, an exception is thrown.

```
   // Find a reference to the Customer for which to book a cruise
   System.out.println("Finding reference to Customer "+customerID);
   CustomerRemote cust = custhome.findByPrimaryKey(customerID);
```

The application then creates a TravelAgent stateful session bean, and gives it, as part of the transactional state, the reference to the customer, the cruise ID, and the Cabin ID.

```
    // Start the Stateful session bean
    System.out.println("Starting TravelAgent Session...");
    TravelAgentRemote tagent = tahome.create(cust);

    // Set the other bean parameters in agent bean
    System.out.println("Setting Cruise and Cabin information in
TravelAgent..");
    tagent.setCruiseID(cruiseID);
    tagent.setCabinID(cabinID);
```

It can then book the passage, thanks to a dummy credit card:

```
    // Create a dummy CreditCard for this
    //
    Calendar expdate = Calendar.getInstance();
    expdate.set(2005,1,5);
    CreditCardDO card = new CreditCardDO("370000000000002",
                                         expdate.getTime(),
                                         "AMERICAN EXPRESS");

    // Book the passage
    //
    System.out.println("Booking the passage on the Cruise!");
    TicketDO ticket = tagent.bookPassage(card,price);

    System.out.println("Ending TravelAgent Session...");
    tagent.remove();

    System.out.println("Result of bookPassage:");
    System.out.println(ticket.description);

  }
```

Test this client application by booking Suite 201 for Mr. Bill Burke on the Three-Hour Cruise aboard the "Bohemian Rhapsody."

Ant doesn't make it particularly easy to pass command-line parameters through to the client. To make this task easier, use one of the scripts that accept command-line parameters in a more customary fashion, available in the *ex12_2* directory.

To book a passage, use the BookPassage.bat (Windows) or the BookPassage script (Unix):

```
BookPassage.bat <customerID> <cruiseID> <cabinID> <price>
Or
./BookPassage <customerID> <cruiseID> <cabinID> <price>
```

```
C:\workbook\ex12_2>BookPassage 1 5 201 2000.0
Buildfile: build.xml

prepare:

compile:

ejbjar:

run.client_122b:
     [java] Finding reference to Customer 1
     [java] Starting TravelAgent Session...
     [java] Setting Cruise and Cabin information in TravelAgent..
     [java] Booking the passage on the Cruise!
     [java] Ending TravelAgent Session...
     [java] Result of bookPassage:
     [java] Bill Burke has been booked for the Three-Hour Cruise
cruise on ship Bohemian Rhapsody.
     [java]  Your accommodations include Suite 201 a 2 bed cabin on
deck level 1.
     [java]  Total charge = 2000.0

BUILD SUCCESSFUL
```

Client_122c

The last application gives you a list of available cabins for a specific cruise that have a desired number of beds. First the application verifies that it's been given the correct number of command-line arguments and gets a remote home reference to the TravelAgent EJB:

```
public static void main(String [] args) throws Exception
{
   if (args.length != 2)
   {
      System.out.println("Usage: java " +
                         "com.titan.clients.Client_122c" +
                         " <cruiseID> <bedCount>");
      System.exit(-1);
   }

   Integer cruiseID = new Integer(args[0]);
   int bedCount = new Integer(args[1]).intValue();
   Context jndiContext = getInitialContext();
   Object obj = jndiContext.lookup("TravelAgentHomeRemote");
   TravelAgentHomeRemote tahome = (TravelAgentHomeRemote)
      javax.rmi.PortableRemoteObject.narrow(obj,
                TravelAgentHomeRemote.class);
```

Because the session bean is not really dedicated to a specific instance of Customer, but is instead making an SQL query in the database, the client creates a TravelAgent bean with a dummy Customer reference, which will never be used. Then it supplies the Cruise ID:

```
   // Start the Stateful session bean
   System.out.println("Starting TravelAgent Session...");
   TravelAgentRemote tagent = tahome.create(null);

   // Set the other bean parameters in agent bean
   System.out.println
      ("Setting Cruise information in TravelAgent..");
   tagent.setCruiseID(cruiseID);
```

Finally, the application asks for a list of all available cabins with a desired number of beds on a particular cruise and displays the result, if any:

```
   String[] results = tagent.listAvailableCabins(bedCount);

   System.out.println("Ending TravelAgent Session...");
   tagent.remove();

   System.out.println("Result of listAvailableCabins:");
   for (int kk=0; kk<results.length; kk++)
   {
      System.out.println(results[kk]);
   }

 }
```

To launch this application, you can use the ListCabins.bat (Windows) or ListCabins (Unix) script:

```
ListCabins.bat <cruiseID> <bedCount>
Or
./ListCabins <cruiseID> <bedCount>
```

Ask the system for a list of the two-bed cabins that are available on the Three-Hour Cruise, the one Mr. Bill Burke chose:

```
C:\workbook\ex12_2>ListCabins 5 2
Buildfile: build.xml

prepare:

compile:

ejbjar:

run.client_122c:
     [java] Starting TravelAgent Session...
     [java] Setting Cruise information in TravelAgent..
     [java] Ending TravelAgent Session...
     [java] Result of listAvailableCabins:
     [java] 200,Suite 200,1
     [java] 203,Suite 203,1
     [java] 204,Suite 204,1
     [java] 205,Suite 205,1
     [java] 206,Suite 206,1
     [java] 207,Suite 207,1
     [java] 209,Suite 209,1

BUILD SUCCESSFUL
```

Suite 201 has two beds but is not shown as available. This omission is correct, because Mr. Bill Burke has booked that suite.

Exercises for Chapter 13

Exercise 13.1:
JMS as a Resource

This exercise is entirely based on the beans implemented in Exercise 12.2. You'll modify the TravelAgent EJB so it publishes a text message to a JMS topic when it completes a reservation.

You'll learn how to create a new JMS topic in JBoss, and configure your bean to use JMS as a resource. You'll also build a client application that will subscribe to this topic and display any published message. To complete new reservations, you'll use one of the client applications created for the preceding example.

Start Up JBoss

If JBoss is already running there is no reason to restart it.

Initialize the Database

Because the exercise uses the ProcessPayment EJB used in recent exercises, the database must contain the `PAYMENT` table. The `createdb` and `dropdb` *Ant* targets, Java code, and clients here have been borrowed from Exercise 12.1.

If you haven't already dropped the `PAYMENT` table after running the examples in Exercise 12.2, do so now by running the `dropdb` *Ant* target.

```
C:\workbook\ex13_1>ant dropdb
Buildfile: build.xml

prepare:

compile:

dropdb:
     [java] Looking up home interfaces..
     [java] Dropping database table...

BUILD SUCCESSFUL
```

Then re-create the `PAYMENT` database table by running the `createdb` *Ant* target

```
C:\workbook\ex13_1>ant createdb
Buildfile: build.xml

prepare:

compile:
```

```
ejbjar:

createdb:
     [java] Looking up home interfaces..
     [java] Creating database table...
```

On the JBoss console, the following lines are displayed:

```
INFO  [STDOUT] Creating table PAYMENT...
INFO  [STDOUT] ...done!
```

- If you're having trouble creating the database, shut down JBoss. Then run the *Ant* build target `clean.db`. This will remove all database files and allow you to start fresh.

The persistence of all other entity beans used in this exercise is managed by the container (CMP), so JBoss will create the needed tables for them automatically.

Create a New JMS Topic

Because the TravelAgent EJB will publish messages in a JMS topic, you'll have to create this new topic in JBoss. This exercise will walk you through two different ways to create a new JMS topic: through an XML configuration file, and through the JBoss JMX HTTP connector.

Adding a JMS Topic Through a Configuration File

The most common way to set up a JMS topic is to use an XML configuration file. As you learned in the installation chapter, every component in JBoss is a JMX MBean that can be hot-deployed. This part of the exercise shows you how to write a JMX MBean definition for a new JMS topic.

You can find the JMX configuration file in the *ex13_1/src/resources/services* directory:

jbossmq-titantopic-service.xml

```
<server>
  <mbean code="org.jboss.mq.server.jmx.Topic"
         name="jboss.mq.destination:service=Topic,
               name=titan-TicketTopic">
    <depends optional-attribute-name="DestinationManager"
     >jboss.mq:service=DestinationManager</depends>
  </mbean>
</server>
```

Each set of MBeans in a JMX configuration file must be defined within a `<server>` tag. An MBean itself is declared in an `<mbean>` tag. The only MBean declaration in this file defines the actual JMS topic you'll use for the example code in this chapter. Each MBean is uniquely identified by its name, called an *ObjectName*. JMX ObjectNames can include any number of key-value parameters to describe the MBean further. In our case, the MBean class representing the JMS topic is declared first (`org.jboss.mq.server.jmx.Topic`), along with its JMX ObjectName (`jboss.mq.destination:service=Topic, name=titan-TicketTopic`). For JMS topic MBeans, a single parameter is useful: `name`. This is where the name of the JMS topic is defined (`titan-TicketTopic`).

One thing to note is that the application server must deploy the `DestinationManager` MBean before any queue or topic is deployed. This dependency is declared in *jbossmq-titantopic-service.xml*'s `depends` tag. JBoss will take care of satisfying this dependency and make sure the `titan-TicketTopic` isn't started until the `DestinationManager` MBean has finished initializing and is ready to provide services to new queues and topics. Copying this file into the JBoss deploy directory will hot-deploy the JMS topic and make it ready for use.

We've defined a `make-topic` *Ant* target for deploying the topic bean. Run this target to copy *jbossmq-titantopic-service.xml* into JBoss's deploy directory:

```
C:\workbook\ex13_1>ant make-topic
Buildfile: build.xml

make-topic:
     [copy] Copying 1 file to C:\jboss-3.2.0\server\default\deploy
```

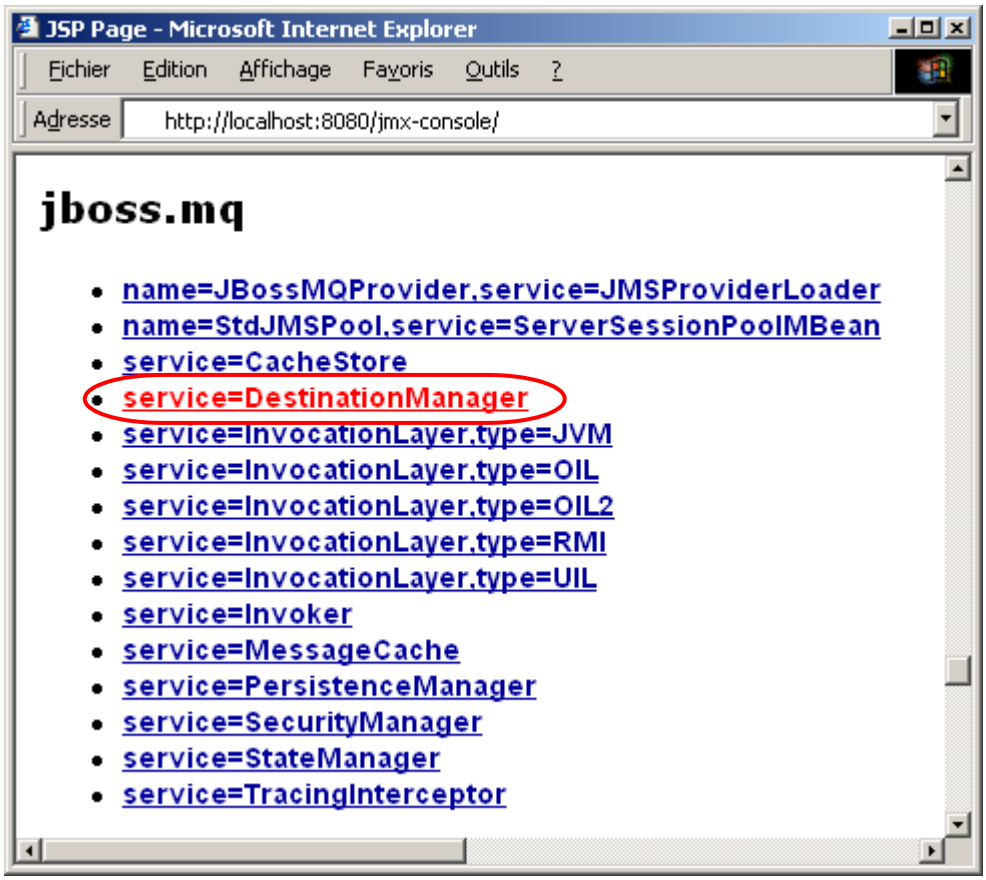
On the server side, the following lines are displayed:

```
[MainDeployer] Starting deployment of package: file:/C:/jboss-
3.2.0/server/default/deploy/jbossmq-titantopic-service.xml
[SARDeployer] Looking for nested deployments in : file:/C:/jboss-
3.2.0/server/default/deploy/jbossmq-titantopic-service.xml
[titan-TicketTopic] Creating
[titan-TicketTopic] Created
[titan-TicketTopic] Starting
[titan-TicketTopic] Bound to JNDI name: topic/titan-TicketTopic
[titan-TicketTopic] Started
[MainDeployer] Deployed package: file:/C:/jboss-
3.2.0/server/default/deploy/jbossmq-titantopic-service.xml
```

Adding a JMS Topic Through the JMX HTTP Connector

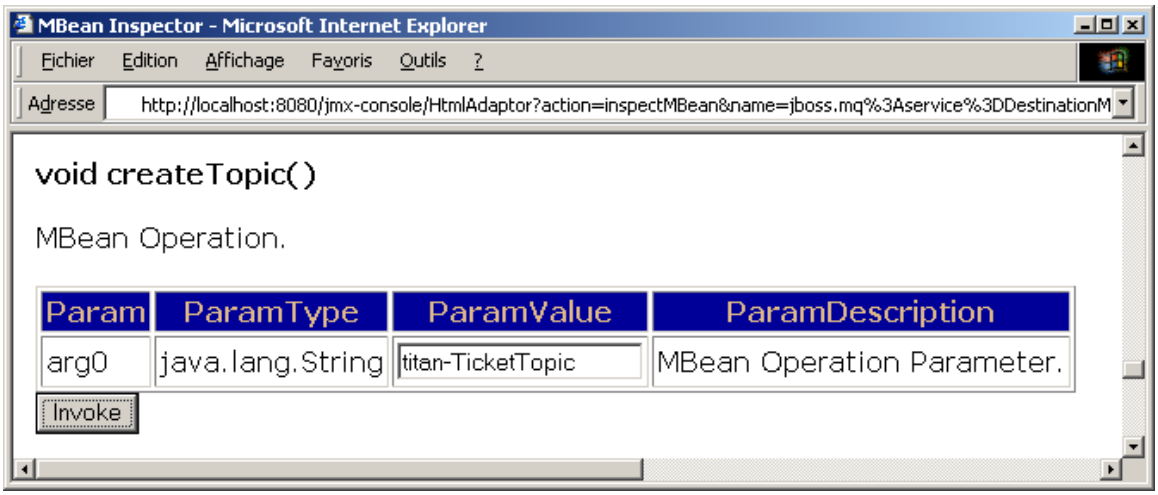
An XML configuration file is the preferred means to deploy a JMS topic permanently, but for quick tests and such an alternative approach that uses JBoss's JMX HTTP connector and the `DestinationManager` is sometimes better, because the topic lives in JBoss only until the application server is shut down. First open your browser and go to *http://localhost:8080/jmx-console/*, where you can browse through all deployed JBoss JMX MBeans. Scroll down to the `jboss.mq` section and find in it the MBean service `DestinationManager`:

Figure 8: Finding the `DestinationManager`



Click on the `service=DestinationManager` link and you get a list of the MBean's attributes and operations. One of the operations, `createTopic()`, allows you to create a new JMS topic:

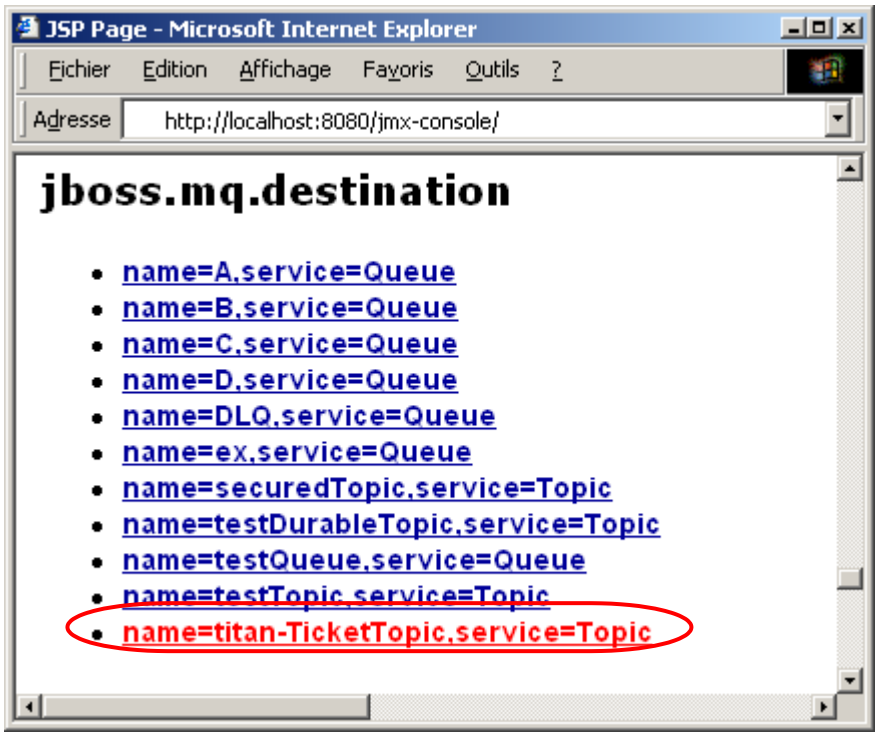
Figure 9: Naming a new JMS topic



Type the name of the new JMS topic in the text area, and click on the **Invoke** button associated with the `createTopic()` operation. The Destination Manager will create the JMS topic and display a status message.

To see your new JMS topic MBean, go back to the home page of the JMX HTTP connector and search for the `jboss.mq.destination` domain. You should be able to see your new topic MBean:

Figure 10: Finding the new topic



Note that you can use the JMX HTTP connector to see the status of your topics and queues even if you create then in an XML configuration file.

Examine the EJB Standard Files

The *ejb-jar.xml* deployment descriptor is equivalent to the one for Exercise 12.2 except for the TravelAgent EJB. The definition for this bean has been extended to reference the JMS topics you just created.

ejb-jar.xml

```
<session>
  <ejb-name>TravelAgentEJB</ejb-name>
  <home>com.titan.travelagent.TravelAgentHomeRemote</home>
  <remote>com.titan.travelagent.TravelAgentRemote</remote>
  <ejb-class>com.titan.travelagent.TravelAgentBean</ejb-class>
  <session-type>Stateful</session-type>
  <transaction-type>Container</transaction-type>
  ...
  <resource-ref>
    <res-ref-name>jdbc/titanDB</res-ref-name>
    <res-type>javax.sql.DataSource</res-type>
    <res-auth>Container</res-auth>
  </resource-ref>
  <resource-ref>
    <res-ref-name>jms/TopicFactory</res-ref-name>
    <res-type>javax.jms.TopicConnectionFactory</res-type>
    <res-auth>Container</res-auth>
  </resource-ref>
  <resource-env-ref>
    <resource-env-ref-name>jms/TicketTopic</resource-env-ref-name>
    <resource-env-ref-type>javax.jms.Topic</resource-env-ref-type>
  </resource-env-ref>
</session>
```

A reference to a `TopicConnectionFactory` is declared in the same way as a reference to a `DataSource`. The definition contains the name of the resource (`jms/TopicFactory`), the class of the resource (`javax.jms.TopicConnectionFactory`), and whether the container or the bean performs the authentication.

Examine the JBoss-Specific Files

The TravelAgentEJB definition in *jboss.xml* must be modified as well, to describe the JMS topic references declared in *ejb-jar.xml*.

jboss.xml

```
...
<session>
  <ejb-name>TravelAgentEJB</ejb-name>
  <jndi-name>TravelAgentHomeRemote</jndi-name>
  <resource-ref>
    <res-ref-name>jdbc/titanDB</res-ref-name>
    <jndi-name>java:/DefaultDS</jndi-name>
  </resource-ref>
  <resource-ref>
    <res-ref-name>jms/TopicFactory</res-ref-name>
    <jndi-name>java:/JmsXA</jndi-name>
  </resource-ref>
  <resource-env-ref>
    <resource-env-ref-name>jms/TicketTopic</resource-env-ref-name>
    <jndi-name>topic/titan-TicketTopic</jndi-name>
  </resource-env-ref>
</session>
...
```

The `<resource-ref>` entry from *ejb-jar.xml* is mapped in the *jboss.xml* file to the JNDI name `java:/JmsXA`. If you take a look at the JBossMQ default configuration file in *$JBOSS_HOME/server/default/deploy/jms-service.xml*, you'll see that the XA connection manager is bound to this name by default.

The last part of the TravelAgent EJB descriptor in *jboss.xml* maps the `jms/TicketTopic` name from the JNDI ENC of the bean to the `topic/titan-TicketTopic` JNDI name. This name corresponds to the JMS topic you just created.

Build and Deploy the Example Programs

Perform the following steps:

1. Open a command prompt or shell terminal and change to the *ex13_1* directory created by the extraction process
2. Set the `JAVA_HOME` and `JBOSS_HOME` environment variables to point to where your JDK and JBoss 3.2 are installed. Examples:

Windows:

```
C:\workbook\ex13_1> set JAVA_HOME=C:\jdk1.3
C:\workbook\ex13_1> set JBOSS_HOME=C:\jboss-3.2.0
```

Unix:

```
$ export JAVA_HOME=/usr/local/jdk1.3
$ export JBOSS_HOME=/usr/local/jboss-3.2.0
```

3. Add `ant` to your execution path.

Windows:

```
C:\workbook\ex13_1> set PATH=..\ant\bin;%PATH%
```

Unix:

```
$ export PATH=../ant/bin:$PATH
```

4. Perform the build by typing `ant`.

You will see *titan.jar* rebuilt, copied to the JBoss *deploy* directory, and redeployed by the application server.

Examine the Client Applications

This exercise includes two client applications. You can find the code for them in the *ex13_1/src/main/com/titan/clients* directory.

The first application is the one used in Exercise 12.2 to make a reservation. The *Ant* target `run.client_122b` hasn't changed, and needs no review.

The second application is new. *JmsClient_1* subscribes to the `titan-TicketTopic` JMS topic and displays all messages published on it.

The application first gets an `InitialContext`, and looks up its `TopicConnectionFactory` and `Topic`:

JmsClient_1.java

```
...
Context jndiContext = getInitialContext();

TopicConnectionFactory factory = (TopicConnectionFactory)
   jndiContext.lookup("ConnectionFactory");

Topic topic = (Topic)
   jndiContext.lookup("topic/titan-TicketTopic");
```

The name of the JMS topic is the same as the one you created in Exercise 12.1, but the name of the `TopicConnectionFactory` is **not** the same as the one used by the TravelAgent EJB.

Remember that the `java:/JmsXA` connection factory used by the EJB was in the **private** JNDI space of the JBoss JVM (indicated by the `java:` prefix). Thus, the client application cannot look up this name from its JVM. For external applications, JBoss binds a set of connection factories within the public JNDI tree, each dedicated to a particular message transport protocol.

JBossMQ supports several different kinds of message invocation layers. Each layer has its own `ConnectionFactory` that is bound in JNDI:

Table 3: JBossMQ message invocation layers

Invocation Layer	JNDI name
JVM Hyper-efficient invocation layer using standard Java method invocation, used for in-JVM JMS clients. External clients cannot use this invocation layer.	`java:/ConnectionFactory` and `java:/XAConnectionFactory` (with XA support)
RMI RMI-based invocation layer.	`RMIConnectionFactory` and `RMIXAConnectionFactory` (with XA support)
OIL (Optimized Invocation Layer) Uses custom TCP/IP sockets to obtain good network performance with a small memory footprint.	`ConnectionFactory` and `XAConnectionFactory` (with XA support)
UIL Used by client applications that cannot accept network connections originating from the server.	`UILConnectionFactory` and `UILXAConnectionFactory` (with XA support)

```
TopicConnection connect = factory.createTopicConnection();

TopicSession session =
   connect.createTopicSession(false, Session.AUTO_ACKNOWLEDGE);

TopicSubscriber subscriber = session.createSubscriber(topic);

subscriber.setMessageListener(this);

System.out.println
   ("Listening for messages on topic/titan-TicketTopic...");
connect.start();
```

The end of the client application code is the same as in the EJB book.

Run the Client Applications

When you redeployed *titan.jar*, JBoss dropped and recreated the database tables, destroying any existing content. For this reason, you must have *Ant* execute the `run.client_122a` target to repopulate the database.

- The `run.client_122a` target originated in Exercise 12.2, but we've duplicated it in the *ex13_1* directory to facilitate your work.

Output:

```
C:\workbook\ex13_1>ant run.client_122a
Buildfile: build.xml

prepare:

compile:

ejbjar:

run.client_122a:
  [java] Calling TravelAgentBean to create sample data..
  [java] All customers have been removed
  [java] All cabins have been removed
  [java] All ships have been removed
  [java] All cruises have been removed
  [java] All reservations have been removed
  [java] Customer with ID 1 created (Burke Bill)
  [java] Customer with ID 2 created (Labourey Sacha)
  [java] Created ship with ID 101...
  [java] Created ship with ID 102...
  [java] Created cabins on Ship A with IDs 100-109
  [java] Created cabins on Ship B with IDs 200-209
  [java] Created Alaska Cruise with ID 0 on ShipA
  [java] Created Norwegian Fjords Cruise with ID 1 on ShipA
  [java] Created Bermuda or Bust Cruise with ID 2 on ShipA
  [java] Created Indian Sea Cruise with ID 3 on ShipB
  [java] Created Australian Highlights Cruise with ID 4 on ShipB
  [java] Created Three-Hour Cruise with ID 5 on ShipB
  [java] Made reservation for Customer 1 on Cruise 0 for Cabin 103
  [java] Made reservation for Customer 1 on Cruise 5 for Cabin 208
  [java] Made reservation for Customer 2 on Cruise 1 for Cabin 105
  [java] Made reservation for Customer 2 on Cruise 5 for Cabin 202

BUILD SUCCESSFUL
```

For your new application to receive the message published on the JMS topic, you have to start it first:

```
C:\workbook\ex13_1>ant run.client_131
Buildfile: build.xml

prepare:

compile:

ejbjar:

client_131:
     [java] Listening for messages on topic/titan-TicketTopic...
```

The last line of the output confirms that the client application has successfully subscribed to the topic and is waiting for messages.

Now you need to make some reservations exactly as you did in Exercise 12.2. Open a new shell and use the *BookPassage* script to make a reservation for Bill Burke on the Three-Hour Cruise for cabin 101 at $3000.00:

```
C:\workbook\ex13_1>BookPassage 1 5 101 3000.0
Buildfile: build.xml

prepare:

compile:

ejbjar:

run.client_122b:
     [java] Finding reference to Customer 1
     [java] Starting TravelAgent Session...
     [java] Setting Cruise and Cabin information in TravelAgent..
     [java] Booking the passage on the Cruise!
     [java] Ending TravelAgent Session...
     [java] Result of bookPassage:
     [java] Bill Burke has been booked for the Three-Hour Cruise
cruise on ship Bohemian Rhapsody.
     [java]  Your accommodations include Suite 101 a 1 bed cabin on
deck level 1.
     [java]  Total charge = 3000.0
```

In the JMS subscriber window you started, the following lines should appear:

```
[java] Listening for messages on topic/titan-TicketTopic...
[java]
[java]  RESERVATION RECEIVED:
[java] Bill Burke has been booked for the Three-Hour Cruise cruise
on ship Bohemian Rhapsody.
[java]  Your accommodations include Suite 101 a 1 bed cabin on deck
level 1.
[java]  Total charge = 3000.0
```

Remember from the EJB Book that our client application uses a non-durable subscription to the topic. Consequently, all the messages sent while the subscriber client application is not running are lost. That would not be the case if we had used a durable subscription to the topic.

To see the "many-to-many" nature of JMS topics, you can launch several JMS listener applications at the same time. They will all receive the messages sent to the topic.

Exercise 13.2:
The Message-Driven Bean

This exercise is an extension of the preceding one. You'll add a message-driven bean (MDB), `ReservationProcessor`, that will play the same role as the TravelAgent EJB but will receive its booking orders through a JMS queue instead of synchronous RMI invocations.

To test the MDB, you'll build a new client application that will make multiple reservations in batch, using a JMS queue that's bound to the MDB. You'll also build a second client application that will listen on another queue to receive booking confirmations.

Along the way, you'll learn how to create a new JMS queue in JBoss and configure a message-driven bean (MDB).

Start Up JBoss

If JBoss is already running there is no reason to restart it.

Initialize the Database

Because the exercise uses the ProcessPayment EJB used in recent exercises, the database must contain the `PAYMENT` table. The `createdb` and `dropdb` *Ant* targets, Java code, and clients here have been borrowed from exercise 13_1.

If you haven't already dropped the `PAYMENT` table after running the examples in Exercise 13.1, do so now by running the `dropdb` *Ant* target.

```
C:\workbook\ex13_2>ant dropdb
Buildfile: build.xml

prepare:

compile:

dropdb:
     [java] Looking up home interfaces..
     [java] Dropping database table...

BUILD SUCCESSFUL
```

Then re-create the `PAYMENT` database table by running the `createdb` *Ant* target

```
C:\workbook\ex13_2>ant createdb
Buildfile: build.xml

prepare:

compile:

ejbjar:

createdb:
     [java] Looking up home interfaces..
     [java] Creating database table...
```

On the JBoss console, the following lines are displayed:

```
INFO  [STDOUT] Creating table PAYMENT...
INFO  [STDOUT] ...done!
```

- If you're having trouble creating the database, shut down JBoss. Then run the `Ant` build target `clean.db`. This will remove all database files and allow you to start fresh.

The persistence of all other entity beans used in this exercise is managed by the container, so it will create the needed tables for them automatically.

Create a New JMS Queue

This exercise requires two different JMS queues, one for the ReservationProcessor MDB and one to receive booking confirmations.

Adding new JMS queues to JBoss is much like adding new JMS topics. As in the preceding exercise, you have two options, one involving a configuration file, the other the JMX HTTP connector.

Adding a JMS Queue Through a Configuration File

The most common way to set up a JMS queue is to use an XML configuration file. This part of the exercise shows you how to write a JMX MBean definition for a new JMS queue. You can find the JMX configuration file in *ex13_2/src/resources/services*:

jbossmq-titanqueues-service.xml

```
<server>
  <mbean code="org.jboss.mq.server.jmx.Queue"
         name="jboss.mq.destination:service=Queue,
               name=titan-ReservationQueue">
    <depends optional-attribute-name="DestinationManager"
     >jboss.mq:service=DestinationManager</depends>
  </mbean>

  <mbean code="org.jboss.mq.server.jmx.Queue"
         name="jboss.mq.destination:service=Queue,
               name=titan-TicketQueue">
    <depends optional-attribute-name="DestinationManager"
     >jboss.mq:service=DestinationManager</depends>
  </mbean>
</server>
```

Recall that each set of MBeans must be defined within a `<server>` tag and each MBean declared in an `<mbean>` tag. Because this exercise requires two different queues, we've defined two MBeans. The MBean class that represents a JMS queue is `org.jboss.mq.server.jmx.Queue`. Its `name` property specifies the name of the JMS queue to be created, such as `titan-ReservationQueue` and `titan-TicketQueue`.

Remember also that the application server must deploy the `DestinationManager` MBean before any queue or topic is deployed. This dependency is declared within the `<depends>` tag in *jbossmq-titanqueues-service.xml*. JBoss will take care of satisfying this dependency and make sure the `titan-ReservationQueue` and `titan-TicketQueue` will not be started until the `DestinationManager` MBean has finished initializing and is ready to provide services to new queues and topics. Copying this file into the JBoss deploy directory will hot-deploy these JMS queues and make them ready for use.

To deploy *jbossmq-titanqueues-service.xml*, run the `make-queues` *Ant* target:

```
C:\workbook\ex13_2>ant make-queues
Buildfile: build.xml

make-queues:
     [copy] Copying 1 file to C:\jboss-3.2.0\server\default\deploy
```

On the server side, the following lines are displayed:

```
[MainDeployer] Starting deployment of package: file:/C:/jboss-
3.2.0/server/default/deploy/jbossmq-titanqueues-service.xml
[titan-ReservationQueue] Creating
[titan-ReservationQueue] Created
[titan-TicketQueue] Creating
```

```
[titan-TicketQueue] Created
[titan-ReservationQueue] Starting
[titan-ReservationQueue] Bound to JNDI name:
                         queue/titan-ReservationQueue
[titan-ReservationQueue] Started
[titan-TicketQueue] Starting
[titan-TicketQueue] Bound to JNDI name: queue/titan-TicketQueue
[titan-TicketQueue] Started
[MainDeployer] Successfully completed deployment of package:
file:/C:/jboss-3.2.0/server/default/deploy/jbossmq-titanqueues-
service.xml
```

- You must deploy the XML file containing the queues **before** you deploy the JAR containing your beans (see below). If you deploy your EJB JAR first, JBoss will detect that the MDB's expected queue does not exist and will create it dynamically. Then, when you try to deploy the XML file that contains the queues, an exception will arise, and you'll be told you're trying to create a queue that already exists.

Adding a JMS Queue Through the JMX HTTP Connector

Add each of the new JMS queues through the JMX HTTP connector the same way you added the JMS topic in the preceding exercise, with one obvious difference: instead of using the `createTopic()` operation of the JBossMQ server, use the `createQueue()` operation.

Remember that queues and topics created in the JMX HTTP Connector live only until the application server is shut down.

Examine the EJB Standard Files

The *ejb-jar.xml* file for this exercise is based on the one for Exercise 13.1. The only notable difference is the addition of the new ReservationProcessor MDB:

ejb-jar.xml

```
<message-driven>
  <ejb-name>ReservationProcessorEJB</ejb-name>
  <ejb-class
   >com.titan.reservationprocessor.ReservationProcessorBean<
  /ejb-class>
  <transaction-type>Container</transaction-type>
  <message-selector>MessageFormat = 'Version 3.4'</message-selector>
  <acknowledge-mode>auto-acknowledge</acknowledge-mode>
  <message-driven-destination>
    <destination-type>javax.jms.Queue</destination-type>
  </message-driven-destination>
```

The MDB descriptor specifies container-managed transactions and automatic acknowledgement of messages, and that messages will be received from a queue rather than a topic. The descriptor also contains a `<message-selector>` tag that allows the MDB to receive only those messages that conform to a specified format.

Then a set of `<ejb-ref>` entries identifies all the beans that ReservationProcessor beans will use during their execution:

```
<ejb-ref>
  <ejb-ref-name>ejb/ProcessPaymentHomeRemote</ejb-ref-name>
  <ejb-ref-type>Session</ejb-ref-type>
  <home>
    com.titan.processpayment.ProcessPaymentHomeRemote
  </home>
  <remote>
    com.titan.processpayment.ProcessPaymentRemote
  </remote>
  <ejb-link>ProcessPaymentEJB</ejb-link>
</ejb-ref>
<ejb-ref>
  <ejb-ref-name>ejb/CustomerHomeRemote</ejb-ref-name>
  <ejb-ref-type>Entity</ejb-ref-type>
  <home>
      com.titan.customer.CustomerHomeRemote
  </home>
  <remote>com.titan.customer.CustomerRemote</remote>
  <ejb-link>CustomerEJB</ejb-link>
</ejb-ref>
<ejb-local-ref>
  <ejb-ref-name>ejb/CruiseHomeLocal</ejb-ref-name>
  <ejb-ref-type>Entity</ejb-ref-type>
  <local-home>
      com.titan.cruise.CruiseHomeLocal
  </local-home>
  <local>com.titan.cruise.CruiseLocal</local>
  <ejb-link>CruiseEJB</ejb-link>
</ejb-local-ref>
<ejb-local-ref>
  <ejb-ref-name>ejb/CabinHomeLocal</ejb-ref-name>
  <ejb-ref-type>Entity</ejb-ref-type>
  <local-home>
      com.titan.cabin.CabinHomeLocal
  </local-home>
  <local>com.titan.cabin.CabinLocal</local>
  <ejb-link>CabinEJB</ejb-link>
</ejb-local-ref>
<ejb-local-ref>
```

```
      <ejb-ref-name>ejb/ReservationHomeLocal</ejb-ref-name>
      <ejb-ref-type>Entity</ejb-ref-type>
      <local-home>
          com.titan.reservation.ReservationHomeLocal
      </local-home>
      <local>com.titan.reservation.ReservationLocal</local>
      <ejb-link>ReservationEJB</ejb-link>
    </ejb-local-ref>
    <security-identity>
      <run-as><role-name>everyone</role-name></run-as>
    </security-identity>
```

Because the MDB will send a confirmation message to a queue once the booking has been successful, it needs a reference to a `javax.jms.QueueConnectionFactory`, specified in the `<resource-ref>` at the end of the MDB descriptor:

```
    <resource-ref>
      <res-ref-name>jms/QueueFactory</res-ref-name>
      <res-type>javax.jms.QueueConnectionFactory</res-type>
      <res-auth>Container</res-auth>
    </resource-ref>
  </message-driven>
```

Note this difference from the preceding exercise: While this bean does send messages to a queue, its descriptor does not contain a `<resource-env-ref>` entry that refers to the destination queue. Why not? In Exercise 13.1 the destination was fixed at deployment time, but in this exercise the destination is not fixed and not even known by the MDB. It is the client application that knows the destination, and transmits it to the MDB by serializing the JMS queue object as part of the JMS message.

Examine the JBoss-Specific Files

No modifications have been made to the CMP entity beans, so the *jbosscmp-jdbc.xml* file is unchanged.

The *jboss.xml* file does need modification, to take into account the new ReservationProcessor EJB:

jboss.xml

```
<message-driven>
  <ejb-name>ReservationProcessorEJB</ejb-name>
  <destination-jndi-name
    >queue/titan-ReservationQueue<
    /destination-jndi-name>
  <resource-ref>
    <res-ref-name>jms/QueueFactory</res-ref-name>
    <jndi-name>java:/JmsXA</jndi-name>
  </resource-ref>
</message-driven>
```

The `<destination-jndi-name>` tag maps the MDB to an existing JMS destination in the deployment environment. You should recognize the name of one of the two JMS queues you just created: `titan-ReservationQueue`.

- By default, each MDB EJB deployed in JBoss can serve up to 15 concurrent messages.

The `<resource-ref>` tag maps the ConnectionFactory name used by the ReservationProcessor EJB to an actual factory in the deployment environment. This mapping is identical to the one in the exercise for the TravelAgent EJB.

Build and Deploy the Example Programs

Perform the following steps:

1. Open a command prompt or shell terminal and change to the *ex13_2* directory created by the extraction process
2. Set the `JAVA_HOME` and `JBOSS_HOME` environment variables to point to where your JDK and JBoss 3.2 are installed. Examples:

Windows:

```
C:\workbook\ex13_2> set JAVA_HOME=C:\jdk1.3
C:\workbook\ex13_2> set JBOSS_HOME=C:\jboss-3.2.0
```

Unix:

```
$ export JAVA_HOME=/usr/local/jdk1.3
$ export JBOSS_HOME=/usr/local/jboss-3.2.0
```

3. Add `ant` to your execution path.

Windows:

```
C:\workbook\ex13_2> set PATH=..\ant\bin;%PATH%
```

Unix:

```
$ export PATH=../ant/bin:$PATH
```

4. Perform the build by typing `ant`.

As in the last exercise, you will see *titan.jar* rebuilt, copied to the JBoss *deploy* directory, and redeployed by the application server.

Examine the Client Applications

In this exercise, you'll use two client applications at the same time. The producer will generate large numbers of JMS messages reporting passage bookings, destined for the ReservationProcessor MDB EJB. The consumer will listen to a JMS queue for messages confirming the bookings, and display them as they come in.

The producer first gets from the command line the cruise ID and the number of bookings:

JmsClient_ReservationProducer.java

```
public static void main (String [] args) throws Exception
{
   if (args.length != 2)
      throw new Exception
   ("Usage: java JmsClient_ReservationProducer <CruiseID> <count>");

   Integer cruiseID = new Integer (args[0]);
   int count = new Integer (args[1]).intValue ();
```

It then looks up a `QueueConnectionFactory` and two JMS queues from the JBoss naming service. The first queue is the one bound to the ReservationProcessor MDB, to which passage booking messages will be sent. The second is not used directly, as you'll see later.

```
   QueueConnectionFactory factory = (QueueConnectionFactory)
      jndiContext.lookup ("ConnectionFactory");
   Queue reservationQueue = (Queue)
      jndiContext.lookup ("queue/titan-ReservationQueue");
   Queue ticketQueue = (Queue)
      jndiContext.lookup ("queue/titan-TicketQueue");
   QueueConnection connect = factory.createQueueConnection ();
   QueueSession session = connect.createQueueSession
      (false,Session.AUTO_ACKNOWLEDGE);
   QueueSender sender = session.createSender (reservationQueue);
```

The client application is now ready to send `count` booking messages in batch. Among other chores, it has looked up the ticket queue, the JMS queue that the ReservationProcessor MDB will use to send confirmation messages.

For each booking, it then creates a JMS `MapMessage`, assigns the ticket queue into the message's `JMSReplyTo` property, and sets the booking data:Cruise ID, Customer ID, Cabin ID, price, credit card number and expiration date, etc. Note that only basic data types such as `String` and `int` can be stored in a `MapMessage`:

```
    for (int i = 0; i < count; i++)
    {
       MapMessage message = session.createMapMessage ();

       // Used in ReservationProcessor to send Tickets back out
       message.setJMSReplyTo (ticketQueue);

       message.setStringProperty ("MessageFormat", "Version 3.4");

       message.setInt ("CruiseID", cruiseID.intValue ());
       // either Customer 1 or 2, all we've got in database
       message.setInt ("CustomerID", i%2 + 1);
       // cabins 100-109 only
       message.setInt ("CabinID", i%10 + 100);
       message.setDouble ("Price", (double)1000 + i);

       // the card expires in about 30 days
       Date expDate = new Date (System.currentTimeMillis () +
                                30*24*60*60*1000L);

       message.setString ("CreditCardNum", "5549861006051975");
       message.setLong ("CreditCardExpDate", expDate.getTime ());
       message.setString ("CreditCardType",
                          CreditCardDO.MASTER_CARD);

       System.out.println ("Sending reservation message #" + i);
       sender.send (message);
    }

    connect.close ();
}
```

One interesting property that's set in the JMS message header is `MessageFormat`. Recall that the `<message-selector>` tag in the MDB deployment descriptor used this property to specify a constraint on the messages the MDB is to receive.

Once all messages are sent, the application closes the connection and terminates. Because messages are sent asynchronously, the application may terminate before the ReservationProcessor EJB has processed all of the messages in the batch.

The consumer application is very similar to the client application you implemented in Exercise 13.1. This time, though, it will subscribe not to a topic but to a queue.

JmsClient_TicketConsumer.java

To receive JMS messages, the client application class implements the `javax.jms.MessageListener` interface, which defines the `onMessage()` method. The main method simply creates an instance of the class and uses a trick to make the main thread wait indefinitely:

```
public class JmsClient_TicketConsumer
   implements javax.jms.MessageListener
{

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

      while(true) { Thread.sleep (10000); }
   }
```

The constructor is very simple JMS code that subscribes the client application to the JMS queue and waits for incoming messages:

```
   public JmsClient_TicketConsumer () throws Exception
   {
      Context jndiContext = getInitialContext ();

      QueueConnectionFactory factory = (QueueConnectionFactory)
         jndiContext.lookup ("ConnectionFactory");
      Queue ticketQueue = (Queue)
         jndiContext.lookup ("queue/titan-TicketQueue");
      QueueConnection connect = factory.createQueueConnection ();
      QueueSession session =
        connect.createQueueSession (false,Session.AUTO_ACKNOWLEDGE);
      QueueReceiver receiver = session.createReceiver (ticketQueue);
      receiver.setMessageListener (this);

      System.out.println ("Listening for messages on titan-
                           TicketQueue...");
      connect.start ();
   }
```

When a message arrives in the queue, the consumer's `onMessage()` method is called. The method simply displays the content of the ticket:

```
public void onMessage (Message message)
{
   try
   {
      ObjectMessage objMsg = (ObjectMessage)message;
      TicketDO ticket = (TicketDO)objMsg.getObject ();
      System.out.println ("*******************************");
      System.out.println (ticket);
      System.out.println ("*******************************");

   }
   catch (JMSException displayed)
   {
      displayed.printStackTrace ();
   }
}
```

Run the Client Applications

When you redeployed *titan.jar*, JBoss dropped and recreated the database tables, destroying any existing content, so you must repopulate the database. Have *Ant* execute the `run.client_122a` target.

The `run.client_122a` target originated in Exercise 12.2, but we've duplicated it in the *ex13_2* directory for your convenience.

```
C:\workbook\ex13_1>ant run.client_122a
Buildfile: build.xml

prepare:

compile:

ejbjar:

run.client_122a:
  [java] Calling TravelAgentBean to create sample data..
  [java] All customers have been removed
  [java] All cabins have been removed
  [java] All ships have been removed
  [java] All cruises have been removed
  [java] All reservations have been removed
  [java] Customer with ID 1 created (Burke Bill)
```

```
    [java] Customer with ID 2 created (Labourey Sacha)
    [java] Created ship with ID 101...
    [java] Created ship with ID 102...
    [java] Created cabins on Ship A with IDs 100-109
    [java] Created cabins on Ship B with IDs 200-209
    [java] Created Alaska Cruise with ID 0 on ShipA
    [java] Created Norwegian Fjords Cruise with ID 1 on ShipA
    [java] Created Bermuda or Bust Cruise with ID 2 on ShipA
    [java] Created Indian Sea Cruise with ID 3 on ShipB
    [java] Created Australian Highlights Cruise with ID 4 on ShipB
    [java] Created Three-Hour Cruise with ID 5 on ShipB
    [java] Made reservation for Customer 1 on Cruise 0 for Cabin 103
    [java] Made reservation for Customer 1 on Cruise 5 for Cabin 208
    [java] Made reservation for Customer 2 on Cruise 1 for Cabin 105
    [java] Made reservation for Customer 2 on Cruise 5 for Cabin 202

BUILD SUCCESSFUL
```

At this point you're going to launch both the client that sends booking messages and the client that receives the tickets as passage confirmations. Launch the consumer **first**, by invoking the *Ant* target `run.client_132`:

```
C:\workbook\ex13_2>ant run.client_132
Buildfile: build.xml

prepare:

compile:

ejbjar:

run.client_132:
     [java] Listening for messages on titan-TicketQueue...
```

Now start the producer, adhering to the following usage:

```
BookInBatch <cruiseID> <count>
```

...where `cruiseID` is the ID of a Cruise in the database (created when you invoked the `run.client_122a` *Ant* target) and `count` is the number of passages to book.

Book 100 passages on the Alaskan Cruise:

```
C:\workbook\ex13_2>BookInBatch  0 100
Buildfile: build.xml

prepare:

compile:

ejbjar:

run.bookinbatch:
     [java] Sending reservation message #0
     [java] Sending reservation message #1
     [java] Sending reservation message #2
     [java] Sending reservation message #3
     ...
     [java] Sending reservation message #98
     [java] Sending reservation message #99
```

Shortly after the producer starts, the consumer, which has been patiently listening to its JMS queue for booking confirmations, will display:

```
run.client_132:
     [java] Listening for messages on titan-TicketQueue...
     [java] ********************************
     [java] Bob Smith has been booked for the Alaska Cruise cruise
            on ship Nordic Prince.
     [java]  Your accommodations include Suite 100 a 1 bed cabin on
             deck level 1.
     [java]  Total charge = 1000.0
     [java] ********************************
     [java] ********************************
     [java] Joseph Stalin has been booked for the Alaska Cruise
            cruise on ship Nordic Prince.
     [java]  Your accommodations include Suite 101 a 1 bed cabin on
             deck level 1.
     [java]  Total charge = 1001.0
     [java] ********************************
     [java] ********************************
     [java] Bob Smith has been booked for the Alaska Cruise cruise
            on ship Nordic Prince.
     [java]  Your accommodations include Suite 102 a 1 bed cabin on
             deck level 1.
     [java]  Total charge = 1002.0
     [java] ********************************
     ...
     [java] ********************************
     [java] Joseph Stalin has been booked for the Alaska Cruise
            cruise on ship Nordic Prince.
     [java]  Your accommodations include Suite 109 a 1 bed cabin on
             deck level 1.
     [java]  Total charge = 1099.0
     [java] ********************************
```

Note that, because the booking confirmation messages are queued, you could start the consumer much later than the producer, rather than before. The confirmation messages sent by the ReservationProcessor MDB would then be stored on the server until the client application starts, and begins to listen to the queue.

Here we are, back at the dock, our "EJB on JBoss" cruise complete! We really hope you've enjoyed the voyage and that we'll soon meet you on JBoss's forums for some more exciting adventures...

Appendix

Appendix A:
Database Configuration

This appendix describes how to set up database pools for data sources other than the default database embedded in JBoss, Hypersonic SQL. It also shows you how to set up your EJBs to use these database pools. For illustration we've modified Exercise 6.1 to configure and use an Oracle connection pool with JBoss.

Set Up the Database

To deploy a database connection pool, JBoss requires a configuration file. In JBoss 3.0, your only option is to create a quite complex file that allows for very fine-grained settings. The result is called "the complete setup." Users of JBoss 3.2 have a second option, "the simple setup." The configuration file is very simple, yet can be used for almost all standard datapool setups.

Here you'll see both methods: the complete setup, for JBoss 3.0 and 3.2, and the simple setup, for JBoss 3.2 only. Pick the one that works for you, and skip over the setup section of the one you don't use.

Complete Setup

First of all you must download the JDBC driver classes for your database. Copy your database's JDBC JAR file to *$JBOSS_HOME/server/default/lib*. For example, the Oracle JDBC class files are contained in *classes12.zip*.

The JBoss distribution includes example database connection-pool files, in the directory *$JBOSS_HOME/docs/examples/jca*. The name of each "complete setup" file ends in *-service.xml*. For this exercise, we've copied the *oracle-service.xml* configuration file to *exAppendixA/titandb-service.xml* and modified it accordingly.

To deploy this connection pool, you must copy *titandb-service.xml* to the *$JBOSS_HOME/service/default/deploy* directory. Note that the name of this config file must end with *-service.xml*, or JBoss will not deploy it.

- Database connection pools are among the many things that can be hot-deployed in JBoss, simply by plopping the pool's XML configuration file into the *deploy* directory.

Examine some of the configuration parameters this file defines:

titandb-service.xml

```
<mbean
code="org.jboss.resource.connectionmanager.LocalTxConnectionManager"
name="jboss.jca:service=LocalTxCM,name=OracleDS">

...

<depends optional-attribute-name="ManagedConnectionFactoryName">
  <!--embedded mbean-->
  <mbean code="org.jboss.resource.connectionmanager.RARDeployment"
    name="jboss.jca:service=LocalTxDS,name=OracleDS">

  <attribute name="JndiName">OracleDS</attribute>
```

The `JndiName` attribute identifies the connection pool within JNDI. You can look up this pool in JNDI with the `java:/OracleDS` string. The class of this bound object is `javax.sql.DataSource`.

```
  <attribute name="ManagedConnectionFactoryProperties">
  <properties>
    <config-property
        name="ConnectionURL"type="java.lang.String">
            jdbc:oracle:thin:@localhost:1521:JBOSSDB
    </config-property>
```

The `ConnectionURL` attribute tells the Oracle JDBC driver how to connect to the database. It varies depending on the database you use, so consult your database JDBC manuals to find out how to obtain the appropriate URL.

```
    <config-property name="DriverClass" type="java.lang.String">
            oracle.jdbc.driver.OracleDriver
    </config-property>
```

The `DriverClass` attribute tells JBoss and the base JDBC classes the name of Oracle's JDBC driver class they need to instantiate and use.

```
    <config-property name="UserName" type="java.lang.String">
            scott
    </config-property>
    <config-property name="Password" type="java.lang.String">
            tiger
    </config-property>
  </properties>
  </attribute>
```

Finally, the `UserName` and `Password` attributes are used when connecting to the Oracle database.

Okay, you've seen that the first part of this file describes the JDBC driver and how to connect to it. The second part of *titandb-service.xml* describes the attributes of the connection pool.

```
...
<depends optional-attribute-name="ManagedConnectionPool">
  <!--embedded mbean-->
  <mbean  code=
  "org.jboss.resource.connectionmanager.JBossManagedConnectionPool"
  name="jboss.jca:service=LocalTxPool,name=OracleDS">
    <attribute name="MinSize">0</attribute>
    <attribute name="MaxSize">50</attribute>
```

The `MinSize` attribute tells JBoss how many JDBC connections to have in the connection pool initially, and the `MaxSize` attribute specifies the maximum number of connections allowed.

```
    <attribute name="BlockingTimeoutMillis">5000</attribute>
```

Whenever all available connections are in use, a thread requesting a connection will block until a connection is released back into the pool. `BlockingTimeoutMillis` specifies the maximum time a thread will wait for a connection before it aborts and throws an exception.

```
    <attribute name="IdleTimeoutMinutes">15</attribute>
```

When a connection has been idle in the connection pool for `IdleTimeoutMinutes` it will be closed and released from the connection pool.

Simple Setup

The configuration file for the simple setup is much less complex than for the complete setup, but the procedures are much the same. For example, the first step is to download the JDBC driver classes for your database. Copy your database's JDBC JAR file to *$JBOSS_HOME/server/default/lib*. For example, the Oracle JDBC class files are contained in *classes12.zip*.

The JBoss distribution includes example database connection-pool files, in the directory *$JBOSS_HOME/docs/examples/jca*. The name of each "simple setup" file ends in *-ds.xml*. For this exercise, we've copied the *oracle-ds.xml* configuration file to *exAppendixA/titandb-ds.xml* and modified it accordingly.

To deploy this connection pool, you must copy *titandb-ds.xml* to the *$JBOSS_HOME/service/default/deploy* directory. Note that the name of this config file must end with *-ds.xml*, or JBoss will not deploy it.

- Database connection pools are among the many things that can be hot-deployed in JBoss, simply by plopping the pool's XML configuration file into the *deploy* directory.

Examine some of the configuration parameters this file defines.

titandb-ds.xml

```
<datasources>
  <local-tx-datasource>
    <jndi-name>OracleDS</jndi-name>
```

The `<jndi-name>` tag identifies the connection pool within JNDI. You can look up this pool in JNDI with the `java:/OracleDS` string. The class of this bound object is `javax.sql.DataSource`.

```
    <connection-url
     >jdbc:oracle:thin:@localhost:1521:JBOSSDB</connection-url>
```

The `<connection-url>` tag tells the Oracle JDBC driver how to connect to the database. The URL varies depending on the database you use, so consult your database JDBC manuals to find out how to obtain the appropriate address.

```
    <driver-class>oracle.jdbc.driver.OracleDriver</driver-class>
```

The `<driver-class>` tag tells JBoss and the base JDBC classes the name of Oracle's JDBC driver class they need to instantiate and use.

```
    <user-name>scott</user-name>
    <password>tiger</password>
  </local-tx-datasource>
</datasources>
```

Finally, the `<user-name>` and `<password>` tags are used when connecting to the Oracle database.

If you compare *titandb-ds.xml* to *titandb-service.xml*, you'll see that the file for the simple setup is much smaller, and contains only the information required to deploy a new database pool.

Examine the JBoss-Specific Files

The example code for this appendix has been borrowed from Exercise 6.1 of this workbook. It is fairly easy to configure the EJBs from this exercise to use the Oracle connection pool you created above. Simply point the data source to `java:/OracleDS` and use the Oracle8 database mapping.

jbosscmp-jdbc.xml

```
<jbosscmp-jdbc>

  <defaults>
    <datasource>java:/OracleDS</datasource>
    <datasource-mapping>Oracle8</datasource-mapping>
    <create-table>true</create-table>
    <remove-table>true</remove-table>
  </defaults>

  <enterprise-beans>
    <entity>
      <ejb-name>CustomerEJB</ejb-name>
      <table-name>Customer</table-name>
      <cmp-field>
        <field-name>id</field-name>
        <column-name>ID</column-name>
      </cmp-field>
      <cmp-field>
        <field-name>lastName</field-name>
        <column-name>LAST_NAME</column-name>
      </cmp-field>
      <cmp-field>
        <field-name>firstName</field-name>
        <column-name>FIRST_NAME</column-name>
      </cmp-field>
      <cmp-field>
        <field-name>hasGoodCredit</field-name>
        <column-name>HAS_GOOD_CREDIT</column-name>
      </cmp-field>
    </entity>
  </enterprise-beans>

</jbosscmp-jdbc>
```

Start Up JBoss

In this variation of Exercise 6.1, you must restart JBoss, so it will recognize the JDBC JAR file you copied into the *lib* directory. Please review the *Server Installation and Configuration* chapter at the beginning of this workbook if you don't remember how to start JBoss.

Build and Deploy the Example Programs

To build and deploy the example for this chapter, you must configure one of the files described above, *titandb-service.xml* or *titandb-ds.xml*, to conform to the database you're using.

Perform the following steps:

1. Open a command prompt or shell terminal and change to the *exAppendixA* directory created by the extraction process.
2. Set the `JAVA_HOME` and `JBOSS_HOME` environment variables to point to where your JDK and JBoss 3.2 are installed. Examples:

Windows:

```
C:\workbook\exAppendixA> set JAVA_HOME=C:\jdk1.3
C:\workbook\exAppendixA> set JBOSS_HOME=C:\jboss-3.2.0
```

Unix:

```
$ export JAVA_HOME=/usr/local/jdk1.3
$ export JBOSS_HOME=/usr/local/jboss-3.2.0
```

3. Add `ant` to your execution path.

Windows:

```
C:\workbook\exAppendixA> set PATH=..\ant\bin;%PATH%
```

Unix:

```
$ export PATH=../ant/bin:$PATH
```

4. Perform the build by typing `ant` for the complete setup, or `ant simple` for the simple setup.

You will see *titan.jar* rebuilt, copied to the JBoss *deploy* directory, and redeployed by the application server. The build script copies *titandb-service.xml* or *titandb-ds.xml*, as appropriate, to the *deploy* directoy as well, which triggers deployment of the customer database pool.

Examine and Run the Client Applications

There is only one client application for this exercise, *Client_61*. It's modeled after the example in Chapter 6 of the EJB book. It will use information you supply in the command-line parameters to create Customer EJBs in the database.

To run the client, first set your `JBOSS_HOME` and `PATH` environment variables appropriately. Then invoke the provided wrapper script. You must supply data on the command line, specifiying a primary key, first name, and last name for each Customer; for example:

```
Client_61 777 Bill Burke 888 Sacha Labourey
```

For the sample command, the output should be:

```
777 = Bill Burke
888 = Sacha Labourey
```

The example program removes the created beans at the conclusion of operation, so there will be no data left in the database.

About the Authors

Bill Burke is the Chief Architect of JBossGroup, LLC. Bill has more than 10 years experience implementing and using middleware in the industry. He was one of the primary developers of Iona Technology's Orbix 2000 CORBA product and has also designed and implemented J2EE applications at Mercantec, Dow Jones, and Eigner Corporation. Besides hanging with his wonderful wife, you can find Bill cheering for the New England Patriots at Gillette Stadium with his dad.

Sacha Labourey is one of the core developers of JBoss Clustering and the General Manager of JBoss Group Europe. He holds a master's degree in computer science from the Swiss Federal Institute of Technology and is the founder of Cogito Informatique. To prove to himself he is not a computer addict, he regularly goes on trips to the Alps to extend his Rumantsch vocabulary.

Colophon

This book is set in the legible and attractive Georgia font. Manuscripts were composed and edited in Microsoft Word, and converted to PDF format for download and printing with Adobe Acrobat.