Exercises for Chapter 8

Exercise 8.1: Simple EJB QL Statements

This exercise demonstrates some basic EJB Query Language (EJB QL) statements. This functionality is available as a standard mechanism in CMP 2.0. The EJB QL is typically used in `finder` methods and `ejbSelect` methods that are new to CMP 2.0. This exercise will walk through some simple examples. The following exercise will explore some complex queries and operations on EJBs.

Building the application for Exercise 8.1

Download the *ex08-1.jar* file from the download site. Downloads may be available for the complete bundle of Exercises.

Copy the downloaded file and extract the file using *WinZip* or *jar* with *xvf* flags to *$EXAMPLES_HOME*. Example *c:\> EJBBook* directory.

Change directory to *$EXAMPLES_HOME\src\ex08_1*, referred to as *$EX08_1_HOME*.

Open a command prompt and type `Ant` or `Ant -buildfile build.xml`. This should create *$EXAMPLES_HOME\build\ex08_1* directory and compile the required files.

$EXAMPLES_HOME\pre-built\ex08_1 directory contains pre-packaged Application archive.

Readers are encouraged to use the pre-packaged *jar* to first understand the concepts before creating their own application archives. This chapter will use the same EJBs as chapter 7 and will contain the `finder` and `ejbSelect` methods to test the EJB QL facilities.

Database schema for the Cabin EJB

This exercise will create the **CustomerBeanTable, AddressBeanTable, PhoneBeanTable, CreditCardBeanTable, ReservationBeanTable, CruiseBeanTable, CabinBeanTable, ShipBeanTable, CustomerBean_phoneNumbers_PhoneBean_Table, CustomerBean_reservations_ReservationBean_customersTable, CruiseBean_reservations_ReservationBean_cruiseTable, CruiseBean_ship_ShipBean_Table** and **ReservationBean_cabins_CabinBean_Table** tables as part of the deployment process.

Note that the **CustomerBeanTable** has **_homeAddress_id, _creditCard_id** columns for one-one relationship. Similarly **AddressBeanTable** contains the foreign key **__reverse_homeAddress_id** and **CreditCardBeanTable** contains the **_customer_id** as the foreign Key.

- Make sure that the Cloudscape server is running when compiling the classes using *Ant*. **SequenceTable** from Chapter 7 will be used for the rest of the exercises. A Cloudscape DDL file is also provided as part of the exercises called *sequencer.sql*. Run this file

```
Cloudscape -isql
```

```
Isql> …connect to TitanDB
Isql> run 'sequencer.sql'
isql> exit;
```

The code examples assume that the Entity Bean Primary Key is generated in most of the cases other than the CustomerEJB, CabinEJB and ShipEJB. There are many solutions for obtaining a Primary Key. J2EE SDK does not provide the facility for in-built sequence generation. Sequence table is another option to obtain a Primary Key. In this workbook we approach it by using the database to generate the Primary Key and as part of the `ejbCreate(..)` method invocation we set the ID for the Entity Bean.

Building the Application Archive for Ex08_1

1. Make sure the jdbc/TitanUID datasource is created and is bound to the J2EE server.
2. Run the *Deploytool* by typing in at command prompt `c:\>deploytool`
3. Create a new application and call it *Titan08_1App.ear*.

A pre-built application is available in *$EXAMPLES_HOME/ears* directory.

4. If starting from scratch create a new EJB JAR and give it a name *EX08_1EJBs*. As explained in Chapter 4 create all the EJBs by mapping the container-managed fields. Do not map the cmr-fields.

   - Do not map the relationship fields. Relationship fields are mapped by defining the cardinality of the relationship and the reference directionality.

5. Select the **EX08_1EJBs** node and then click on the **Relationships** tab. Define all the relationships as mentioned in Chapter 7.
6. Select **File→Add to Application→Web War** from the menu. Select from the *$EXAMPLES_HOME/pre-built/ex08_1/Titan08_1Web.war* and add it to the application. You can also create the web client using **File→New→Web War** option. If using the pre-built *EAR* file there is no need for this step.
7. Generate the default SQL and run the application through the verifier.

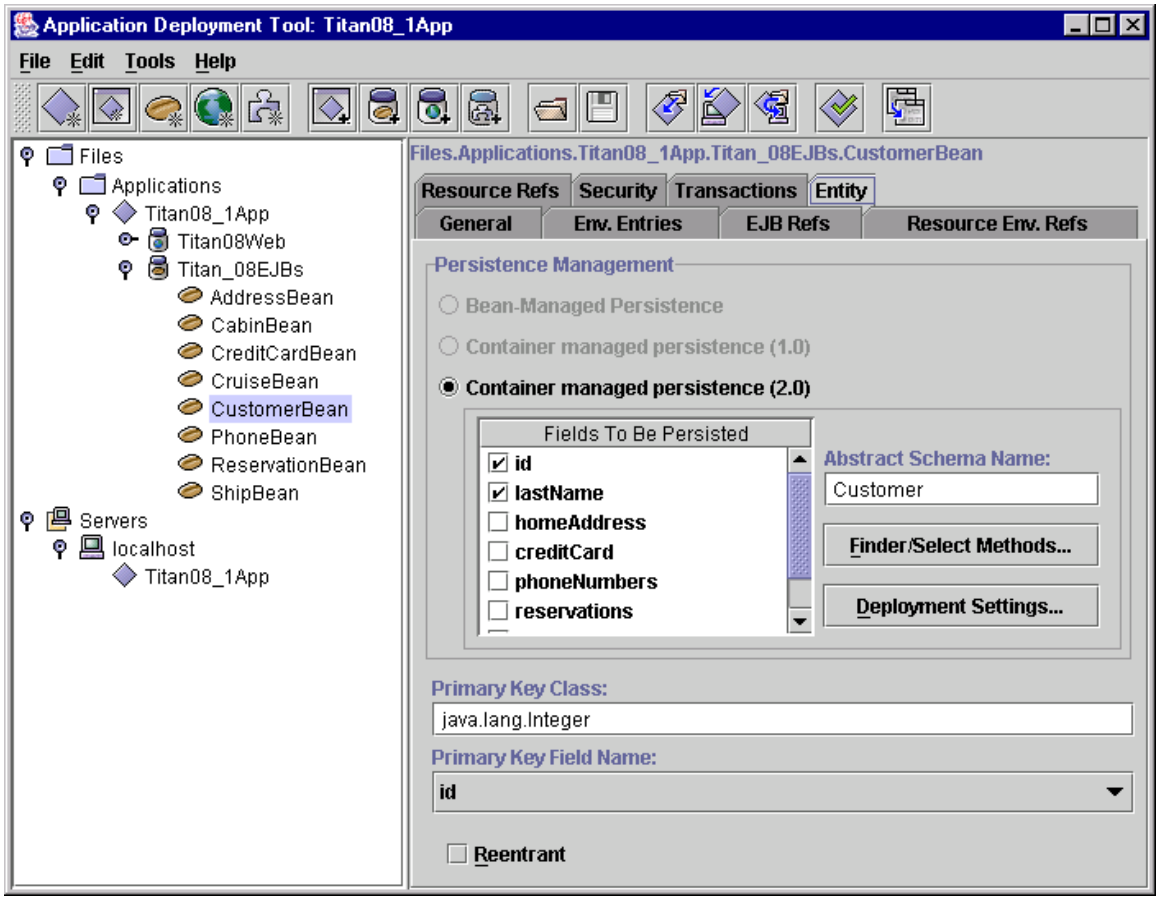
   - Sometimes the verifier complains about transaction demarcation errors. It seems that the *xml* descriptors don't get updated. Simply go to the **Transactions** tab of the EJB and set the transaction attribute to **required** for those methods that the verifier complains even though it shows **required**. This triggers the update of the descriptor files needed. Also select **Tools→Update Files** from the menu or **Save** option from the **File** Menu.

We are not yet ready to deploy the application. The following section covers the concepts of the EJB QL for the exercise and then walks through creating the EJB QL statements through the *deploytool*.

Examine the EJB QL Statements in Deploytool and ejb-jar.xml file

The *ejb-jar.xml* descriptor file contains the EJB QL-related statements as EJB QL is part of the standard specification of EJB 2.0. We have consistently given names through all the chapters but have not come across the usage yet. Abstract Schema Name is the name used in the EJB QL to refer to the Entity Bean. For example the Customer EJB Entity tab looks like the following:

Figure 1: Abstract Schema Name definition for Customer EJB



For each of the EntityBeans, click on the **Entity** tab and notice the Abstract Schema Name. These names are also defined in the *ejb-jar.xml* file that is generated using *Deploytool*.

```
<abstract-schema-name>Customer</abstract-schema-name>
```

The *ejb-jar.xml* file contains all the query elements for all the `finder` and `ejbSelect` methods. Let us examine the EJB QL definitions creation in the *Deploytool* before we examine the Queries themselves.

For the entity beans that have defined local finders and select methods the methods can be viewed in the *Deploytool*.

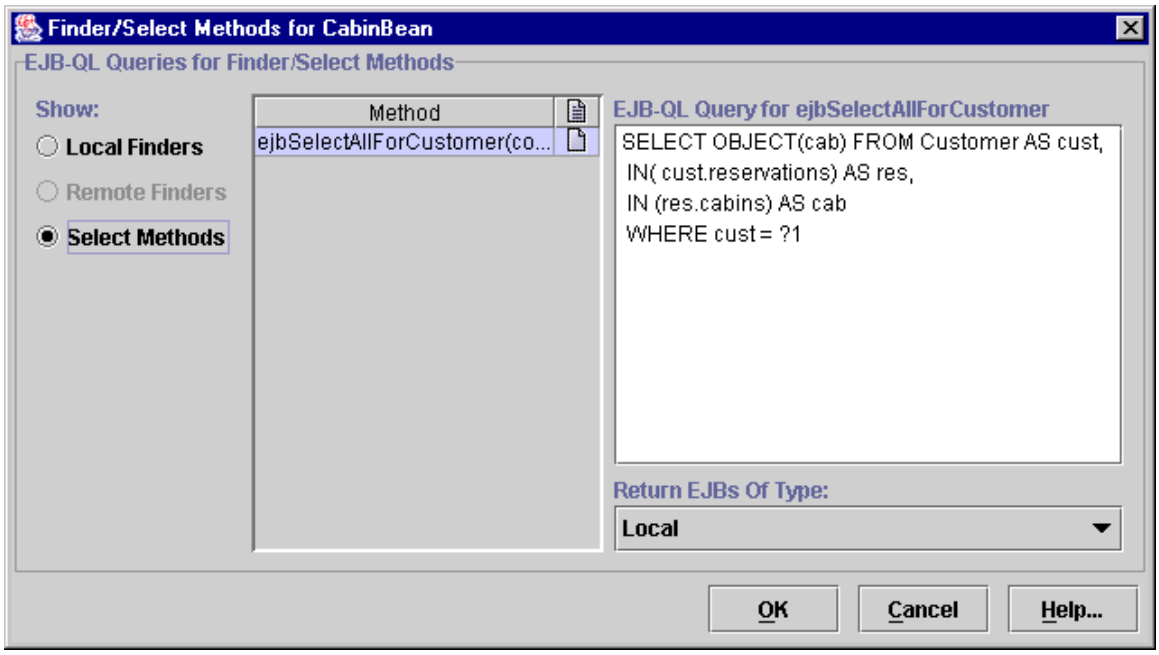
We will walk through the CabinEJB finder/select method creation here to clarify the process. Cabin EJB defines the following methods:

- Finder method: `findAllOnDeck(..)`
- EjbSelect method: `ejbSelectAllForCustomers(..)`

To define the EJB QL for these methods follow these steps:

1. Select CabinEJB node from the *Titano8_EJBS jar*.
2. Click on the **Finder/Selector** methods button.

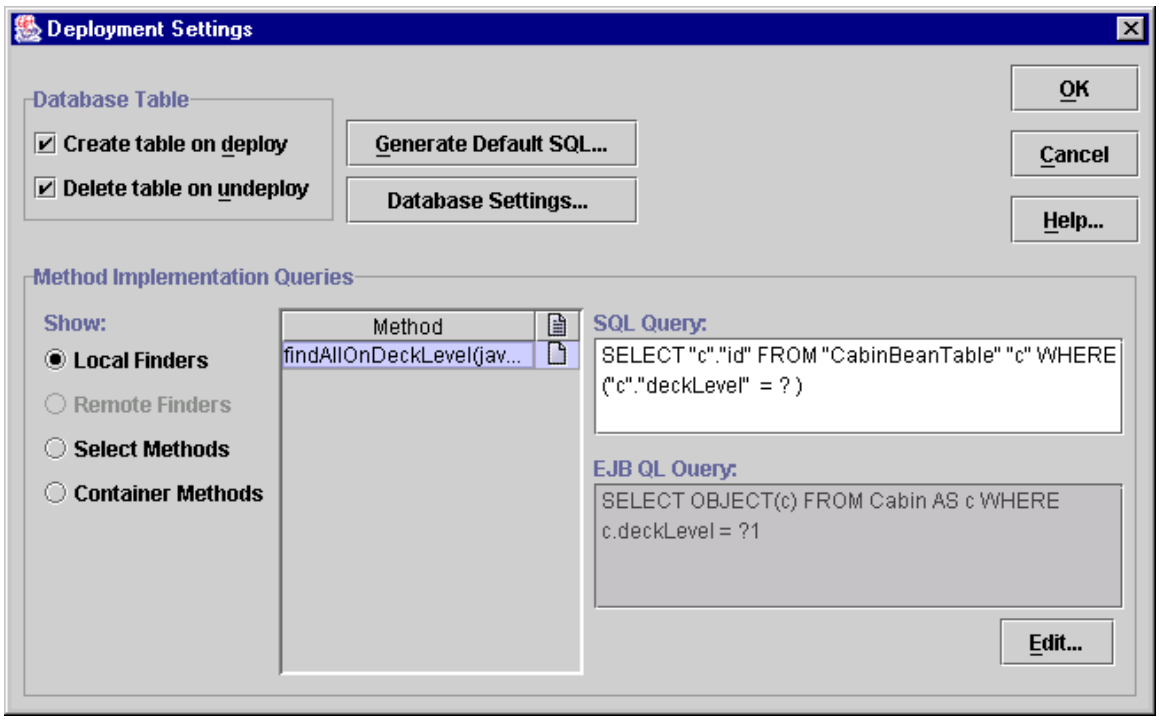
Figure 2: EJB QL Statements window for an EJB



This opens another window with radio buttons for local finders and select methods.

3. Type in the EJB QL statement for each of the finder and select methods and click on the **OK** button.

Figure 3: Corresponding SQL for EJBQL after default generation



- When the default SQL is generated the *Deploytool* also generates the corresponding SQL for the EJB QL statements.

Now let us examine the descriptor file that contains the query elements. There are multiple `<query>` elements in the *ejb-jar.xml* file, each corresponding to one EJB QL statement. Customer EJB has three local finder methods.

CustomerEJB : EJB QL semantics

```
<query>
   <description/>
   <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>
   <description/>
   <query-method>
         <method-name>findByCity</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.homeAddress.city = ?1 AND c.homeAddress.state = ?2 )
   </ejb-ql>
</query>
<query>
   <description/>
   <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>
```

All the finder methods return zero or more Customer EJBs that satisfy the selection criteria. Finders must return the EJBs for the corresponding Home Interface. Finder methods must not return objects of types other than the corresponding EJB or a collection of EJBs. The placeholders for the query parameters are identified similarly to `PreparedStatement` using JDBC. These placeholders follow a pattern of "?x", x being the positional value.

Also notice that the Object query language semantics are not typically flat two dimensional. In the above example of `findByCity` query, the execution of the method navigates through the object lattice, traversing a single-valued relationship `homeAddress`, rather than the database-represented foreign key that refers to the `AddressBeanTable`.

All the finders are defined in the home Interface for the EJB. *CustomerHomeLocal.java* defines the following finder methods.

CustomerHomeLocal.java

```
public CustomerLocal findByName(String lastName, String firstName)
         throws FinderException;

   public Collection findByGoodCredit()
         throws FinderException;
   public Collection findByCity(String city, String state)
         throws FinderException;
```

The method names and the parameters need to match exactly in the *ejb-jar.xml* as defined in the signature of the methods in the EJBHome interface. The parameter names are irrelevant.

- As mentioned earlier finder methods can return the EJBObject or a collection of EJBObjects that the EJBHome interface is responsible. It is recommended to use a collection unless the criteria are distinct enough to identify a single EJB. The container semantics can vary depending on the implementation to either return the first in the list or based on the underlying indexing mechanism.

Local select methods should be conceptually considered as private methods for the EJB. The `ejbSelect<name>` naming convention is used for all the local select methods. For an external client to access the EJB object, it has to provide some business methods that in the Bean class can make a call to the local ejbSelect method. For this reason the ejbSelect methods do not have a corresponding select method defined in the `EJBLocalObject` interface for the container callbacks. Methods defined in the local interface are considered business methods from the container perspective.

The following sections show the `select` method definition for the AddressBean EJB.

AddressEJB EJB QL Semantics

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

```
    <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>
```

The corresponding `ejbSelect` method and a convenience method defined in the home interface are shown below. The convenience methods can be either in local interface or local home interface. We follow the pattern of treating the select method accessors to be similar to a finder method. In this exercise we follow the semantics of defining the accessor select methods in the Home Interface.

AddressBean.java

```
public abstract Set ejbSelectZipCodes(String state)
   throws FinderException;

public abstract Collection ejbSelectAll()
   throws FinderException;

public abstract CustomerLocal ejbSelectCustomer(AddressLocal addr)
   throws FinderException;

// Public Home method required to test the private ejbSelectZipCodes
method
public Collection ejbHomeSelectZipCodes(String state)
   throws FinderException
{
   return this.ejbSelectZipCodes(state);
}

// Public Home method required to test the private ejbSelectCustomer
method
public CustomerLocal ejbHomeSelectCustomer(AddressLocal addr)
   throws FinderException
{
   return (CustomerLocal)(this.ejbSelectCustomer(addr));
}
```

The corresponding public method defined in the *AddressHomeLocal.java* to access the pass through method in the EJB is:

```
 // Home method, requires ejbHomeSelectZipCodes method in bean class
```

```
  public Collection selectZipCodes(String state)
                      throws javax.ejb.FinderException;

 // Home method, requires ejbHomeSelectCustomer method in bean class
  public CustomerLocal selectCustomer(AddressLocal addr)
                      throws javax.ejb.FinderException;
```

CruiseBean EJB QL Semantics

```
<query>
   <description/>
   <query-method>
         <method-name>findByName</method-name>
         <method-params>
               <method-param>java.lang.String</method-param>
         </method-params>
   </query-method>
   <ejb-ql>SELECT OBJECT(c) FROM Cruise c WHERE c.name=?1</ejb-ql>
</query>
```

CruiseBean has a simple finder method and contains no select methods. This finder method retrieves all the *CruiseBean* EJBs by name as provided in the search criteria.

The corresponding finder definition in the home interface is shown below:

CruiseHomeLocal.java

```
public CruiseLocal findByName(String name) throws FinderException;
```

CabinEJB EJB QL semantics

```
<query>
   <description/>
   <query-method>
         <method-name>ejbSelectAllForCustomer</method-name>
         <method-params>
               <method-param>
                  com.titan.customer.CustomerLocal
               </method-param>
         </method-params>
   </query-method>
   <result-type-mapping>Local</result-type-mapping>
   <ejb-ql>SELECT OBJECT(cab) FROM Customer AS cust,
           IN( cust.reservations) AS res,
           IN (res.cabins) AS cab
           WHERE cust = ?1
   </ejb-ql>
</query>
<query>
```

```
    <description/>
    <query-method>
          <method-name>findAllOnDeckLevel</method-name>
          <method-params>
                <method-param>java.lang.Integer</method-param>
          </method-params>
    </query-method>
    <ejb-ql>SELECT OBJECT(c) FROM Cabin AS c WHERE c.deckLevel =
              ?1
    </ejb-ql>
</query>
```

CruiseBean defines one `finder` and one `select` method. The `finder` method is very simple and retrieves all the Cabin EJBs based on the deck level. The `ejbSelect` method on the other hand uses the `IN` operator to identify the cabins based on the customer-reservation relationship and reservation-cabin relationship traversal for a given Customer. There are two many-to-many relationships between *CustomerBean* to *ReservationBean* and *ReservationBean* to *CabinBean*. To obtain a distinct list of Cabins `java.util.Set` is returned instead `of java.util.Collection`.

The corresponding methods in the Home interface and the `CabinBean` class are as follows:

CabinBean.java

```
public abstract Set ejbSelectAllForCustomer(CustomerLocal cust)
         throws FinderException;
// Public Home method required to test the private
ejbSelectAllForCustomer method
public Set ejbHomeSelectAllForCustomer(CustomerLocal cust)
 throws FinderException
{
   return this.ejbSelectAllForCustomer(cust);
}
```

CabinHomeLocal.java

```
public abstract Collection findAllOnDeckLevel(Integer level)
         throws FinderException;
// Home method, requires ejbHomeSelectAllForCustomer method in bean
class
public Set selectAllForCustomer(CustomerLocal cust)
        throws FinderException;
```

These EJB QL statements can be entered into the *Deploytool* EJB QL windows. Once all the EJB QL statements are typed in, use the deployment settings to generate the default SQL. Notice the SQL statements for each of the EJB QL statement that will be executed on the persistence store.

Run the application through the verifier and deploy the application.

Examine and Run the Client JSP Pages

This exercise consists of multiple JSPs that explore different simple EJB QL features in EJB 2.0.

Client_81.jsp

The *Client_81.jsp* page demonstrates the `findByName` finder method in the CustomerBean's Home Interface. This example creates a set of *CustomerBean* EJBs and then performs the `findByName` operation on the CustomerHomeLocal Interface to retrieve the EJBs that match the criteria specified. See the previous section in this exercise where we examined the *ejb-jar.xml* file and the methods for *CustomerBean* EJB and its interfaces.

The JSP performs the call to the `findByName` finder method to instantiate the *CustomerBean* with the desired name:

```
CustomerLocal cust85 = customerhome.findByName("Smith85", "John");
```

Open the browser and change the URL to *http://localhost:8000/Titan08_1*. You should see a listing of all the JSPs for the application. Click on the **<href>** that shows *Client_81.jsp* and you should see the output as shown below.

```
Client_81 Example Showing Simple Customer EJB-QL
Creating Customers 80-99
pk=80 John Smith80 1080 Elm Street Minneapolis, MN 55401
pk=81 John Smith81 1081 Elm Street St. Paul, CA 55402
pk=82 John Smith82 1082 Elm Street Rochester, MN 55403
pk=83 John Smith83 1083 Elm Street Winona, CA 55404
pk=84 John Smith84 1084 Elm Street Wayzata, MN 55405
pk=85 John Smith85 1085 Elm Street Eagan, CA 55401
pk=86 John Smith86 1086 Elm Street Minneapolis, MN 55402
pk=87 John Smith87 1087 Elm Street St. Paul, CA 55403
pk=88 John Smith88 1088 Elm Street Rochester, MN 55404
pk=89 John Smith89 1089 Elm Street Winona, CA 55405
pk=90 John Smith90 1090 Elm Street Wayzata, MN 55401
pk=91 John Smith91 1091 Elm Street Eagan, CA 55402
pk=92 John Smith92 1092 Elm Street Minneapolis, MN 55403
pk=93 John Smith93 1093 Elm Street St. Paul, CA 55404
pk=94 John Smith94 1094 Elm Street Rochester, MN 55405
pk=95 John Smith95 1095 Elm Street Winona, CA 55401
pk=96 John Smith96 1096 Elm Street Wayzata, MN 55402
pk=97 John Smith97 1097 Elm Street Eagan, CA 55403
pk=98 John Smith98 1098 Elm Street Minneapolis, MN 55404
pk=99 John Smith99 1099 Elm Street St. Paul, CA 55405
```

```
Finding Customer having name 'John Smith85'
John Smith85 1085 Elm Street Eagan, CA 55401
.....
```

You do not need to delete the contents of the tables for this exercise. The subsequent JSPs *Client_82.jsp* and *Client_83.jsp* use the customers created in Client_81.

Client_82.jsp

The *Client_82.jsp* demonstrates the finder method in the Customer Home Interface. The JSP invokes the Customer's `findByCity` method and retrieves a collection of matching Customer Bean references.

```
Collection mplscustomers =
customerhome.findByCity("Minneapolis","MN");
```

Then the JSP iterates through the collection and displays the information for each Customer that matches the criteria specified. The search parameters are case sensitive. With J2EE SDK the SQL generated for the EJB QL statement can be modified in the *Deploytool* to perform case-insensitive searches if desired. The output of the JSP is shown below:

```
Client_82 Example Showing Customer EJB-QL Using findByCity()
Finding Customers having City = Minneapolis
John Smith80 1080 Elm Street Minneapolis, MN 55401
John Smith86 1086 Elm Street Minneapolis, MN 55402
John Smith92 1092 Elm Street Minneapolis, MN 55403
John Smith98 1098 Elm Street Minneapolis, MN 55404
```

Client_83.jsp

This JSP demonstrates the `select` method by performing the `ejbSelectZipCodes` operation on the *AddressBean* EJB. Since this method is **private** the public accessor created in the Home Interface is executed by the JSP, which in turn in the *AddressBean* executes the local private `ejbSelectZipCodes`. The JSP iterates through the Set and displays the information. The following code in the JSP performs the operations explained.

```
// use the Home method "selectZipCodes" to call the private
"ejbSelectZipCodes" method on the bean
Collection mnzips = addresshome.selectZipCodes("MN");

Iterator iterator = mnzips.iterator();
while (iterator.hasNext())
{
   String zip = (String)(iterator.next());
   out.print(zip+"<br>");
}
```

The output of the JSP is shown below:

```
Client_83 Example Showing Address ejbSelectZipCodes()
Finding zip codes in MN
```

```
55405
55404
55403
55402
55401
.....
```

Client_83b.jsp

This JSP demonstrates the Customer to Address relationship by executing `ejbSelectCustomer` method on the *AddressBean* EJB. The JSP executes the public convenience method on the Home Interface that in turn performs the execution of the private `ejbSelectCustomer` operation.

```
CustomerLocal cust80 = customerhome.findByPrimaryKey(new
Integer(80));  // known key
AddressLocal addr = cust80.getHomeAddress();
CustomerLocal customer = addresshome.selectCustomer(addr);
```

The output of the JSP is asfollows:

```
Client_83b Example Showing Address ejbSelectCustomer()
Finding an Address bean to use for the test
Calling selectCustomer Home method on address
Displaying data for returned Customer bean
John Smith80 1080 Elm Street Minneapolis, MN 55401
```

- **Before running the next JSPs empty out the tables as they will be using a different set of data.**

Client_84a.jsp

This example creates data that will be used by the subsequent JSPs. This JSP creates six *CustomerBean* EJBs with two *ReservationBean* EJBs and two *CabinBean* EJBs associations.

```
Client_84a Setup Tables for Example Demonstrating EJB-QL
Creating a Ship and Cruise
cruise.getName()=Cruise A
ship.getName()=Ship A
cruise.getShip().getName()=Ship A
Creating Customers 1-6, each with 2 reservations for 2 cabins
Customer 1
  2002-11-01 3000.0
    Cabin 1001
    Cabin 1002
  2002-11-08 3100.0
    Cabin 1011
    Cabin 1012
Customer 2
  2002-11-15 4000.0
    Cabin 2001
```

```
    Cabin 2002
  2002-11-22 4100.0
    Cabin 2011
    Cabin 2012
Customer 3
  2002-11-29 5000.0
    Cabin 3001
    Cabin 3002
  2002-12-06 5100.0
    Cabin 3011
    Cabin 3012
Customer 4
  2002-12-13 6000.0
    Cabin 4001
    Cabin 4002
  2002-12-20 6100.0
    Cabin 4011
    Cabin 4012
Customer 5
  2002-12-27 7000.0
    Cabin 5001
    Cabin 5002
  2003-01-03 7100.0
    Cabin 5011
    Cabin 5012
Customer 6
  2003-01-10 8000.0
    Cabin 6001
    Cabin 6002
  2003-01-17 8100.0
    Cabin 6011
    Cabin 6012
```

Client_84b.jsp

This client JSP page demonstrates the `findAllOnDeckLevel` finder method in the CabinEJB. This is a very straightforward execution. The JSP receives the collection, iterates through the collection and displays the information.

```
Collection cabins = cabinhome.findAllOnDeckLevel(new Integer(3));
Iterator iter = cabins.iterator();
while (iter.hasNext())
{
   CabinLocal cabin = (CabinLocal)(iter.next());
   out.print(cabin.getName()+" on deck "+
               cabin.getDeckLevel()+"<br>");
}
```

Also note the container does not allow manipulation of the relationship collection. Explicit transaction demarcation is needed to perform operations on collections. In this case we do not need a transactional context. Primarily this is because the Home Interface in this case returns the collection of EJBObjects rather than a collection of relationships between the EJBs. For this specific reason we can perform operations on the collection outside the context of a transaction.

The output of the JSP is as follows:

```
Client_84b Example Demonstrating EJB-QL finder methods
Retrieve a collection of all cabins on deck 3
Cabin 3001 on deck 3
Cabin 3002 on deck 3
Cabin 3011 on deck 3
Cabin 3012 on deck 3
```

Client_84c.jsp

This JSP demonstrates the `ejbSelectAllForCustomers` select method in the *Cabin* bean. Notice as mentioned earlier that this query uses the `IN` operator to navigate the object graph of *Customer-Reservation-Cabin* relationships to select distinct Cabins for a Customer. Again as this `ejbSelect` method is private we use the defined `ejbHomeSelect` method to make a call to the `ejbSelect` method within the context of the entity bean class.

```
CustomerLocal customer1 = customerhome.findByPrimaryKey(new
Integer(1));

// use the Home method "selectAllForCustomer" to call the private
"ejbSelectAllForCustomer" method on the bean
Collection cabins_for_1 = cabinhome.selectAllForCustomer(customer1);

Iterator iter = cabins_for_1.iterator();
while (iter.hasNext())
{
   CabinLocal cabin = (CabinLocal)(iter.next());
   out.print(cabin.getName()+" on deck "+
   cabin.getDeckLevel()+"<br>");
}
```

This select method could have been replaced by a simple finder method in the *Cabin* EJB that performs the complex query of navigating through the graph. That would have eliminated the method definitions in the Bean class for the private `ejbSelect` method and the public accessible `ejbHomeSelect` method. Home Interface could have simply declared the finder method.

The output for this JSP is as shown below:

```
Client_84c Example Demonstrating EJB-QL IN Operations
Retrieve a collection of all cabins for Customer 1
Cabin 1001 on deck 1
Cabin 1002 on deck 1
```

```
Cabin 1011 on deck 1
Cabin 1012 on deck 1
```

This completes the Exercise 8.1. Undeploy the application so that the tables are removed from the database other than the sequence table for the next exercise.

Exercise 8.2: Complex EJB QL Statements

This exercise further demonstrates some more complex EJB QL statements. The ORDERBY operation is not available in the EJB QL and application servers such as Weblogic Server provide specific extension. A way around this problem is provided in this exercise. This exercise explores the following EJB QL statements

- *ShipBean*: `findByTonnage` – Finder method(s) – method overloading
- *CustomerBean*: `findByExactName` - Finder method.
- *CustomerBean*: `findByName` – Finder method using LIKE operator
- *CustomerBean*: `findByNameAndState` – Finder method using `AND` operator in `WHERE` clause.
- *CustomerBean*: `findInHotStates` – Finder method using `IN` operator in `WHERE` clause.
- *CustomerBean*: `findWithNoReservations` – Finder method using `IS EMPTY`.
- *CustomerBean*: `findOnCruise` – Finder method using `MEMBER Of` operator.
- *CustomerBean*: `findByState` – Finder method using `ORDER BY` in generated SQL.

Building the application for Exercise 8.2

1. Download the *ex08-2.jar* file from the download site. Downloads may be available for the complete bundle of Exercises.
2. Copy the downloaded file and extract the file using *WinZip* or *jar* with xvf flags to *$EXAMPLES_HOME*. Example *c:\> EJBBook* directory.
3. Change directory to *$EXAMPLES_HOME\src\ex08_2*, referred to as *$EX08_2_HOME*.
4. Open a command prompt and type `Ant` or `Ant -buildfile build.xml`

This should create *$EXAMPLES_HOME\build\ex08_2* directory and compile the required files.

$EXAMPLES_HOME\pre-built\ex08_2 directory contains pre-packaged client application *jars* for this exercise. As shown in the exercise 7.1 create the *Titan08_2App* application components.

Database schema for the Ex08_2

This exercise will use the **SequenceTable** as well as the other tables generated by the *Deploytool*.

Building the Application Archive for Ex08_2

1. Run the Deploytool by typing in at command prompt `c:\>deploytool`
2. Create a new application and call it *Titan8_2App.ear*.

A pre-built application is available in *$EXAMPLES_HOME/ears* directory.

In this exercise we will walk through the pre-built application. Readers are encouraged to understand different settings in the *Deploytool* for this application.

3. Select any EJB and click on the **Entity** tab. Make sure the database connection parameters are set and generate the default SQL.
4. Verify the application by selecting the application node in the left pane and selecting **Tools→Verifier**. If the application doesn't have any failures deploy the application with a web Context of **titan7_2.**
5. Open the browser and go to *http://localhost:8000/Titan_82*

Are we are ready to run the example? Not yet! Let us explore the application components and understand them better.

Examine the ejb-jar descriptor file and the corresponding EJB components

The *ejb-jar.xml* descriptor file contains the information about the EJB QL in the `<query>` element for each query.

Ship Bean query elements in standard descriptor file

```
<query>
 <description></description>
   <query-method>
     <method-name>findByTonnage</method-name>
     <method-params>
       <method-param>java.lang.Double</method-param>
     </method-params>
   </query-method>
   <ejb-ql>SELECT OBJECT(S) FROM Ship s where s.tonnage =
           ?1
   </ejb-ql>
 </query>
 <query>
```

```
    <description></description>
    <query-method>
      <method-name>findByTonnage</method-name>
      <method-params>
        <method-param>java.lang.Double</method-param>
        <method-param>java.lang.Double</method-param>
      </method-params>
    </query-method>
    <ejb-ql>SELECT OBJECT(s) FROM Ship s where s.tonnage BETWEEN ?1
            AND ?2
    </ejb-ql>
</query>
```

These two finder methods in the *ShipBean* have the same name. It is legal to overload the finder methods in EJBs similar to any methods in Java. The appropriate version will be called and the method will execute based on the parameters. The first finder selects *ShipBean* EJBs based on the tonnage. The second method that overloads `findByTonnage` takes in two parameters and selects only those *ShipBean* EJBs that fall within the range specified.

The ShipHomeLocal defines both the finder methods as follows:

ShipHomeLocal.java

```
public Collection findByTonnage(Double tonnage)
                    throws javax.ejb.FinderException;

  public Collection findByTonnage(Double tonnage1, Double tonnage2)
                    throws javax.ejb.FinderException;
```

CustomerBean query elements in standard descriptor file

Customer EJB adds many finder methods in this exercise. The following is the standard description of the query elements in the *ejb-jar.xml* descriptor file.

```
    <query>
        <description></description>
        <query-method>
          <method-name>findByExactName</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>
        <description></description>
```

```
        <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
                LIKE 'S%' AND  c.firstName LIKE 'Jo%')
        </ejb-ql>
    </query>
    <query>
        <description></description>
        <query-method>
          <method-name>findByNameAndState</method-name>
          <method-params>
            <method-param>java.lang.String</method-param>
            <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
                LIKE   '%' AND c.firstName LIKE '%'  AND
                c.homeAddress.state = ?3
        </ejb-ql>
    </query>
    <query>
        <description></description>
        <query-method>
          <method-name>findInHotStates</method-name>
          <method-params />
        </query-method>
        <ejb-ql>SELECT OBJECT(c) FROM Customer c WHERE
                c.homeAddress.state IN ('FL', 'TX', 'AX',
                'CA')
        </ejb-ql>
    </query>
    <query>
        <description></description>
        <query-method>
          <method-name>findWithNoReservations</method-name>
          <method-params />
        </query-method>
        <ejb-ql>SELECT OBJECT(c) FROM Customer c where
                 c.reservations is EMPTY
        </ejb-ql>
```

```
    </query>
    <query>
         <description></description>
         <query-method>
           <method-name>findOnCruise</method-name>
           <method-params>
             <method-param>
                 com.titan.cruise.CruiseLocal
             </method-param>
           </method-params>
         </query-method>
         <ejb-ql>SELECT OBJECT(cust) FROM Customer cust, Cruise cr,
                 IN (cr.reservations ) as res where cr = ?1 AND cust
                 MEMBER OF res.customers
         </ejb-ql>
    </query>
    <query>
         <description></description>
         <query-method>
           <method-name>findByState</method-name>
           <method-params>
             <method-param>java.lang.String</method-param>
           </method-params>
         </query-method>
         <ejb-ql>SELECT OBJECT(c) FROM Customer c WHERE
                 c.homeAddress.state  = ?1
         </ejb-ql>
    </query>
```

Exercise 8.1 defines `findByName` finder method that takes in `firstName` and `lastName` parameters to find an exact match. In this exercise we rename the method to `findByExactName`. The query semantics are identical.

CustomerBean defines a finder using the `LIKE` operator. This finder `findByName` uses `LIKE` comparison to retrieve a collection of Customers. The EJB Book on page 249 clearly explains the operator usage. The `LIKE` operator cannot be used with input parameters. The patterns for `LIKE` comparison have to be provided at deployment time rather than runtime. For this reason the query statement **cannot** be written as follows :

```
SELECT OBJECT(c) FROM Customer c WHERE  (c.lastName
             LIKE ? 1 AND  c.firstName LIKE  ? 2 )
```

Instead the patterns have to be provided at deployment time. The descriptor file contains the patterns with wildcard characters “%” provided for this purpose. The correct form of the query is shown below:

```
SELECT OBJECT(c) FROM Customer c WHERE  (c.lastName
             LIKE 'S%' AND  c.firstName LIKE 'Jo%')
```

This query would result in retrieving a collection of Customers whose `lastName` starts with 'S' and `firstName` starts with "Jo".

The next query defines an additional finder that demonstrates the use of the `AND` operator in a `WHERE` clause. The `findByNameAndState` finder method uses wildcard semantics for the `lastName`, `firstName` values and uses the `homeAddress.state` as the third parameter.

```
SELECT OBJECT (c) FROM Customer c WHERE c.lastName
            LIKE   '%' AND c.firstName LIKE '%'  AND
            c.homeAddress.state = ?3
```

Notice that the parameters passed in for positional values 1 and 2 are not used. These exercises were also based on Weblogic Server workbook. To keep the code base as close as possible , we have not changed the signatures of the methods. Weblogic Server seems to allow input parameters to the `LIKE` operator that can contain a pattern. This can be runtime bound in BEA Weblogic Server, but not in J2EE™ SDK.

The next query demonstrates the use of the `IN` operator in a `WHERE` clause. This query returns only those customers having `homeAddress` in one of the States specified for in the `IN` operator. The `IN` operator in this `WHERE` clause takes in a set of String literals.

```
SELECT OBJECT(c) FROM Customer c WHERE
  c.homeAddress.state IN ('FL', 'TX', 'AX','CA')
```

The next query demonstrates the use of `IS EMPTY` operator to identify the Customers who do not have a Reservation.

```
SELECT OBJECT(c) FROM Customer c where c.reservations is EMPTY
```

The following query demonstrates the usage of the `MEMBER OF` operator to identify a set of Customers that have reservations on a given Cruise.

```
SELECT OBJECT(cust) FROM Customer cust, Cruise cr,
IN (cr.reservations ) as res where cr = ?1 AND cust
MEMBER OF res.customers
```

This query uses the single CruiseLocal parameter to identify the cruise for which the query is performed. It retrieves the list of Customers that have reservations for the Cruise specified using the `MEMBER OF` operator on the Reservation-Customer relationship.

The last query looks very simple. It retrieves a list of Customers who reside in the specified State. The list returned is not ordered. EJB QL does not specify `ORDER BY` semantics. After the default SQL is generated we modify the SQL statement for the EJB QL so that proper ordering semantics are applied in the SQL. The application is saved with the custom SQL. The query looks like:

```
SELECT "c"."id", "c"."firstName", "c"."lastName" FROM
"CustomerBeanTable" "c" , "AddressBeanTable" "@tmp1" WHERE
("@tmp1"."state"  = ? ) AND (("c"."_homeAddress_id" = "@tmp1"."id"))
ORDER BY "c"."firstName", "c"."lastName"
```

All the finder methods are defined in the CustomerHomeLocal interface.

CustomerHomeLocal.java

```
public CustomerLocal findByExactName(String lastName, String
firstName)throws FinderException;

public Collection findByName(String lastName, String firstName)
        throws FinderException;

public Collection findByNameAndState(String lastName, String
firstName, String state)
        throws FinderException;

public Collection findByCity(String city, String state)
        throws FinderException;

public Collection findInHotStates()
        throws FinderException;

public Collection findWithNoReservations()
        throws FinderException;

public Collection findOnCruise(CruiseLocal cruise)
        throws FinderException;

public Collection findByState(String state)
        throws FinderException;
// Excluded the other finders from previous exercise.
```

Examine and Run the Client JSPs

Save the application and deploy it with a web-context of Titan08_2. This exercise contains a set of JSPs that provide information based on the queries that we examined earlier.

Client_85.jsp

This JSP demonstrates the `findByTonnage` methods on the Ship home interface. This client program creates a few *ShipBean* EJBs and then calls the overloaded `findByTonnage` methods on the home to retrieve the *ShipBean* EJBs.

```
Collection ships100k = shiphome.findByTonnage(new Double(100000.0));
Iterator iterator = ships100k.iterator();
while (iterator.hasNext())
{
ShipLocal ship = (ShipLocal)(iterator.next());
out.print("pk="+ship.getId()+" Name="+ship.getName()+"
tonnage="+ship.getTonnage()+"<br>");
}
```

```
Collection ships50110k = shiphome.findByTonnage(new Double(50000.0),
new Double(110000.0));
iterator = ships50110k.iterator();
while (iterator.hasNext())
{
ShipLocal ship = (ShipLocal)(iterator.next());
out.print("pk="+ship.getId()+" Name="+ship.getName()+"
tonnage="+ship.getTonnage()+"<br>");
}
```

The output of the program is shown below:

```
Client_85 Example Using Ship findByTonnage variants
Creating Ship Beans with Various Tonnage Values
pk=1 Name=Ship 1 tonnage=40000.0
pk=2 Name=Ship 2 tonnage=50000.0
pk=3 Name=Ship 3 tonnage=60000.0
pk=4 Name=Ship 4 tonnage=70000.0
pk=5 Name=Ship 5 tonnage=80000.0
pk=6 Name=Ship 6 tonnage=90000.0
pk=7 Name=Ship 7 tonnage=100000.0
pk=8 Name=Ship 8 tonnage=110000.0
pk=9 Name=Ship 9 tonnage=120000.0
pk=10 Name=Ship 10 tonnage=130000.0
Finding Ships with Exactly 100K Tonnage
pk=7 Name=Ship 7 tonnage=100000.0
Finding Ships with Tonnage between 50K and 110K
pk=2 Name=Ship 2 tonnage=50000.0
pk=3 Name=Ship 3 tonnage=60000.0
pk=4 Name=Ship 4 tonnage=70000.0
pk=5 Name=Ship 5 tonnage=80000.0
pk=6 Name=Ship 6 tonnage=90000.0
pk=7 Name=Ship 7 tonnage=100000.0
pk=8 Name=Ship 8 tonnage=110000.0
```

The *ShipBean* EJBs created will be used in the subsequent JSPs. So do not delete the rows in the tables.

Client_86a.jsp

This JSP creates a set of Customers, Addresses, Phone Numbers and the Credit card information for the subsequent examples.

Client_86b.jsp

This JSP demonstrates the usage of two finder methods in CustomerHome. The `findbyName` and `findByExactName` are the finder methods that are executed. It then displays the information. Note that the arguments passed to `findByName` are not really used. This method

can be a `no-arg` method. The descriptor contains the pattern for the `LIKE` operator. The output of the program is:

```
Client_86b Example Using Customer findByName variants
Finding Customer having a name exactly matching 'Joe Star'
Joe Star 1088 Elm Street Rochester, MN 55404
Finding Customers having a name like 'Jo S' (no wildcards)
John Smith 1080 Elm Street Minneapolis, MN 55401
Joe Smith 1083 Elm Street Winona, CA 55404
John Star 1085 Elm Street Eagan, CA 55401
Joe Star 1088 Elm Street Rochester, MN 55404
John Smith 1095 Elm Street Winona, CA 55401
Joe Smith 1098 Elm Street Minneapolis, MN 55404
Finding Customers having a name like 'Jo% S%' (with wildcards)
John Smith 1080 Elm Street Minneapolis, MN 55401
Joe Smith 1083 Elm Street Winona, CA 55404
John Star 1085 Elm Street Eagan, CA 55401
Joe Star 1088 Elm Street Rochester, MN 55404
John Smith 1095 Elm Street Winona, CA 55401
Joe Smith 1098 Elm Street Minneapolis, MN 55404
Finding Customers having a name like 'Jo% S%' and living in MN
John Smith 1080 Elm Street Minneapolis, MN 55401
Ringo Star 1082 Elm Street Rochester, MN 55403
Roger Johnson 1084 Elm Street Wayzata, MN 55405
Paul Smith 1086 Elm Street Minneapolis, MN 55402
Joe Star 1088 Elm Street Rochester, MN 55404
John Johnson 1090 Elm Street Wayzata, MN 55401
Ringo Smith 1092 Elm Street Minneapolis, MN 55403
Roger Star 1094 Elm Street Rochester, MN 55405
Paul Johnson 1096 Elm Street Wayzata, MN 55402
Joe Smith 1098 Elm Street Minneapolis, MN 55404
```

Client_87.jsp

This example is a straightforward finder execution and performs the `findInHotStates` operation.

The output of the program is asfollows:

```
Client_87 Example Using Customer findInHotStates()
Finding Customers Living in Warm Climates
Paul Johnson 1081 Elm Street St. Paul, CA 55402
Joe Smith 1083 Elm Street Winona, CA 55404
John Star 1085 Elm Street Eagan, CA 55401
Ringo Johnson 1087 Elm Street St. Paul, CA 55403
Roger Smith 1089 Elm Street Winona, CA 55405
Paul Star 1091 Elm Street Eagan, CA 55402
Joe Johnson 1093 Elm Street St. Paul, CA 55404
```

```
John Smith 1095 Elm Street Winona, CA 55401
Ringo Star 1097 Elm Street Eagan, CA 55403
Roger Johnson 1099 Elm Street St. Paul, CA 55405
```

Client_88.jsp

This JSP demonstrates the usage of `findWithNoReservations` as well as `findOnCruise` methods on the CustomerHomeLocal interface.

The first call uses the `<cmr-field>` element referencing the reservations that are null for the Customers. This provides the list of Customers with no Reservations. The JSP then displays the information about the Customers.

```
Collection poorcustomers = customerhome.findWithNoReservations();
Iterator iterator = poorcustomers.iterator();
while (iterator.hasNext())
{
   CustomerLocal customer = (CustomerLocal)(iterator.next());
   AddressLocal addr = customer.getHomeAddress();
   out.print(customer.getPrimaryKey()+"
   "+customer.getName().getFirstName()+"
   "+customer.getName().getLastName()+" "+addr.getStreet()+"
   "+addr.getCity()+", "+addr.getState()+" "+addr.getZip()+"<br>");
}
```

The same JSP then demonstrates the use of `findOnCruise` method. This operation is performed by retrieving the desired *CruiseBean* EJB by name and then is passed on as an argument to the `findOnCruise` finder method to retrieve a list of Customers that are on the Cruise with Reservations.

```
CruiseLocal cruiseA = cruisehome.findByName("Alaska Cruise");
Collection alaskacustomers = customerhome.findOnCruise(cruiseA);
iterator = alaskacustomers.iterator();
while (iterator.hasNext()) {
CustomerLocal customer = (CustomerLocal)(iterator.next());
   AddressLocal addr = customer.getHomeAddress();
   out.print(customer.getPrimaryKey()+"
"+customer.getName().getFirstName()+"
"+customer.getName().getLastName()
   +" "+addr.getStreet()+" "+addr.getCity()+", "+addr.getState()+"
"+addr.getZip()+"<br>");
}
```

This finder method uses the `MEMBER OF` operator to navigate through the object lattice to retrieve the corresponding customers associated with the Cruise and having reservations.

The output of the program is as follows:

```
Client_88 Example Using Customer Reservation/Cruise Finders
Finding Customers With No Reservations
```

```
81 Paul Johnson 1081 Elm Street St. Paul, CA 55402
84 Roger Johnson 1084 Elm Street Wayzata, MN 55405
87 Ringo Johnson 1087 Elm Street St. Paul, CA 55403
90 John Johnson 1090 Elm Street Wayzata, MN 55401
93 Joe Johnson 1093 Elm Street St. Paul, CA 55404
96 Paul Johnson 1096 Elm Street Wayzata, MN 55402
99 Roger Johnson 1099 Elm Street St. Paul, CA 55405
Finding Customers On Alaska Cruise
82 Ringo Star 1082 Elm Street Rochester, MN 55403
85 John Star 1085 Elm Street Eagan, CA 55401
88 Joe Star 1088 Elm Street Rochester, MN 55404
91 Paul Star 1091 Elm Street Eagan, CA 55402
94 Roger Star 1094 Elm Street Rochester, MN 55405
97 Ringo Star 1097 Elm Street Eagan, CA 55403
```

Client_89.jsp

This example is very simple. This provides the solution for not having the `ORDERBY` operator as part of the EJB QL. Recall that we customized the generated SQL to perform the `ORDERBY` since EJB 2.0 doesn't support `ORDERBY` as a standard feature. The `findByState` finder method returns the list of customers ordered by `firstName` and `lastName`. The output of the program is as shown below:

```
Client_89 Example Using Customer findByState() with ORDERBY clause
Finding Customers In MN using findByState() with ORDERBY clause
Joe Smith 1098 Elm Street Minneapolis, MN 55404
Joe Star 1088 Elm Street Rochester, MN 55404
John Johnson 1090 Elm Street Wayzata, MN 55401
John Smith 1080 Elm Street Minneapolis, MN 55401
Paul Johnson 1096 Elm Street Wayzata, MN 55402
Paul Smith 1086 Elm Street Minneapolis, MN 55402
Ringo Smith 1092 Elm Street Minneapolis, MN 55403
Ringo Star 1082 Elm Street Rochester, MN 55403
Roger Johnson 1084 Elm Street Wayzata, MN 55405
Roger Star 1094 Elm Street Rochester, MN 55405
```

Undeploy the application after having fun testing these JSPs before proceeding to the next chapter.